UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

Programming Contest
Fall 2001

P. N. Hilfinger

2001 Programming Problems

To set up your account, execute

```
source ~ctest/bin/setup
```

in all shells that you are using. (This is for those of you using the C-shell. Others will have to examine this file and do the equivalent for their shells.)

This booklet should contain eight problems on 17 pages. You have 5 hours in which to solve as many of them as possible. Put each complete C solution into a file N`.c`, each complete C++ solution into a file N`.cc`, and each complete Java program into a file N`.java`, where N is the number of the problem. Each program must reside entirely in a single file. In Java, the class containing the main program for problem N must be named `P`N. Each C/C++ file should start with the line

```
#include "contest.h"
```

and must contain no other `#include` directives, except as indicated below. Upon completion, each program *must* terminate by calling `exit(0)` (or `System.exit(0)` in Java).

Aside from files in the standard system libraries and those we supply, you may not use any pre-existing computer-readable files to supply source or object code; you must type in everything yourself. Selected portions of the standard `g++` class library are included among of the standard libraries you may use: specifically, the headers `string`, `vector`, `iostream`, `iomanip`, `strstream`, `fstream`, `algorithms`, `hash_map`, and `hash_set`. Likewise, you can use the standard C I/O libraries (in either C or C++), and the math library (header `math.h`). In Java, you may use the standard packages `java.lang`, `java.io`, `java.text`, and `java.util`. You may not use utilities such as `yacc`, `bison`, `lex`, or `flex` to produce programs. Your programs may not create other processes (as with the `system`, `popen`, `fork`, or `exec` series of calls). You may use any inanimate reference materials you desire, but no people. You can be disqualified for breaking these rules.

When you have a solution to problem number N that you wish to submit, use the command

`submit` *N*

from the directory containing *N*`.c`, *N*`.cc`, or *N*`.java`. Before actually submitting your program, `submit` will first compile it and run it on one sample input file. No submission that is sent after the end of the contest will count. You should be aware that `submit` takes some time before it actually sends a program. In an emergency, you can use

`submit -f` *N*

which submits problem *N* without any checks.

You will be penalized for incorrect submissions that get past the simple test administered by `submit`, so be sure to test your programs (if you get a message from `submit` saying that it failed, you will *not* be penalized). All tests will use the compilation command

`contest-gcc` *N*

followed by one or more execution tests of the form (Bourne shell):

`./`*N* `<` *test-input-file* `>` *test-output-file* `2>` *junk-file*

which sends normal output to *test-output-file* and error output to *junk-file*. The output from running each input file is then compared with a standard output file, or tested by a program in cases where the output is not unique. In this comparison, leading and trailing blanks are ignored and sequences of blanks are compressed to single blanks. Otherwise, the comparison is literal; be sure to follow the output formats *exactly*. It will do no good to argue about how trivially your program's output differs from what is expected; you'd be arguing with a program. Make sure that the last line of output ends with a newline. Your program must not send any output to `stderr`; that is, the temporary file *junk-file* must be empty at the end of execution. Each test is subject to a time limit of about 45 seconds. You will be advised by mail whether your submissions pass.

The command `contest-gcc` *N*, where *N* is the number of a problem, is available to you for developing and testing your solutions. For C and C++ programs, it is equivalent to

`gcc -Wall -o` *N* `-O -g -I`*our-includes* *N*`.* -lstdc++ -lm`

For Java programs, it is equivalent to

`javac -g` *N* *N*`.java`

followed by a command that creates an executable file called *N* that runs the command

`java P`*N*

when executed (so that it makes the execution of Java programs look the same as execution of C/C++ programs). The *our-includes* directory contains `contest.h` for C/C++, which also supplies the standard header files. The files in `~ctest/submission-tests/`N, where N is a problem number, contain the input files and standard output files that `submit` uses for its simple tests.

All input will be placed in `stdin`. You may assume that the input conforms to any restrictions in the problem statement; you need not check the input for correctness. Consequently, you are free to use `scanf` to read in numbers and strings and `gets` to read in lines.

Terminology. The term *free-form input* indicates that input numbers, words, or tokens are separated from each other by arbitrary whitespace characters. By standard C/UNIX convention, a whitespace character is a space, tab, return, newline, formfeed, or vertical tab character. A *word* or *token,* accordingly, is a sequence of non-whitespace characters delimited on each side by either whitespace or the beginning or end of the input file.

Scoring. Scoring will be according to the ACM Contest Rules. You will be ranked by the number of problems solved. Where two or more contestants complete the same number of problems, they will be ranked by the *total time* required for the problems solved. The total time is defined as the sum of the *time consumed* for each of the problems solved. The time consumed on a problem is the time elapsed between the start of the contest and successful submission, plus 20 minutes for each unsuccessful submission, and minus the time spent judging your entries. Unsuccessful submissions of problems that are not solved do not count. As a matter of strategy, you can derive from these rules that it is best to work on the problems in order of increasing expected completion time.

Protests. Should you disagree with the rejection of one of your problems, first prepare a file containing the explanation for your protest, and then use the `protest` command (without arguments). It will ask you for the problem number, the submission number (submission 1 is your first submission of a problem, 2 the second, etc.), and the name of the file containing your explanation. Do not protest without first checking carefully; groundless protests will be result in a 5-minute penalty (see Scoring above). The Judge will *not* answer technical questions about C, C++, Java, the compilers, the editor, the debugger, the shell, or the operating system.

Notices. During the contest, the Web page at URL

`http://http.cs.berkeley.edu/~hilfingr/programming-contest/announce.html`

will contain any urgent announcements, plus a running scoreboard showing who has solved what problems. Sometimes, it is useful to see what problems others are solving, to give you a clue as to what is easy.

1. [M. Dynin, from a contest in St. Petersburg] Given a rectangle of letters (such as

```
AB
BC
```

), a starting position within that rectangle (such as the character in the upper-left corner), and a string (such as `"ABBC"`), consider the question of finding paths through the letters that match the given string, begin at the starting position, and at each step move one square in one of the eight compass directions (north, south, east, west, northeast, northwest, southeast, southwest). For the given example, there are two such paths. They are (`E-SW-E`) and (`S-NE-S`)—that is, "one step east, one step southwest, one step east" and "one step south, one step northeast, and one step south." For the rectangle

```
CBB
BBA
```

and the string `"BBCBBA"`, starting at square in the middle of the top row, there are 12 paths. Paths are allowed to visit the same position twice.

You are to write a program that, given such a rectangle, string, and starting position, reports the *number* of distinct paths that match the string, according to the definitions above. The input will consist of four positive integers (call them M, N, r, and c), a string (S), and M strings consisting of upper-case letters `A-Z`, each of which is N characters long. These inputs are all separated from each other by whitespace. The M strings are the rows of the rectangle, from top to bottom. The pair (r, c) are the coordinates of the starting position, with $0 \leq r < M$, $0 \leq c < N$. Row 0 is the top row; column 0 is the left column. The output will be a line reporting the number of paths in the format shown in the examples.

You may assume that the number of paths is less than 2^{31}, $M \leq 80$, $N \leq 80$. Nevertheless, be aware that the time limit is less than a minute.

Example 1:

Input	Output

`2 2 0 0`
`ABBC AB`
`BC`
	`There are 2 paths.`

Example 2:

Input	Output
`2 3 0 1 BBCBBA CBB BBA`	`There are 12 paths.`

2. [M. Dynin] A simple method of *compressing* a message is to replace sections of its text with references back to previous portions. For example, given the input text

`how␣much␣wood␣could␣a␣woodchuck␣chuck␣if␣a␣woodchuck␣could␣chuck␣wood`

we could encode it as

`how␣much␣wood␣could␣a<12,5>chuck␣<5,6>if<20,14><38,5><11,6><21,4>`

The interpretation of each <p, n> in the output (we will call it a *back reference*) is that it is replaced by a copy of n characters of the output, starting p characters to the left of the last output character. For example, `<5,6>` is replaced by six characters ("`chuck␣`"), beginning at preceding character #5 (the preceding '␣' is 0, '`k`' is 1, '`c`' is 2, etc.). As printed, the encoding saves little space, but if we assume that each '<$\cdot, \cdot$>' can be stored in the same space as two characters, then we do save 28 characters.

It is legal for a back reference to indicate a sequence of characters produced by previous back references, or even the same back reference. For example, '`ab<1,10>`' encodes '`abababababab`'. The p values always refer to the expanded description. For example, "`abcdef<3,8>x<14,3>`" expands to "`abcdefcdefcdefxabc`."

Your program is to create a compressed encoding of an input string, in the format illustrated above. The input will consist of positive integers M_p, M_n, and C in free format on one line, followed by a single line of text whose length is at most 4096 characters. You are to create an optimal (minimal-length) encoding of the string, under the following constraints. In the final encoded string:

1. Every p value must be in the range $0 \leq p \leq M_p$.
2. Every n value must be in the range $C < n \leq M_n$.
3. Given a choice between two different possible encodings of the same string, consider the first (leftmost) point at which they differ. Prefer the encoding that contains a back reference at that point. If both have back references there, prefer the one with the larger n value. If both have the same n value, prefer the one with the smaller p value.

There are examples of input and output on the next page.

Example 1.
Input:

```
64 16 2
how much wood could a woodchuck chuck if a woodchuck could chuck wood
```

Output:

```
how much wood could a<12,5>chuck <5,6>if<20,14><38,5><11,6><21,4>
```

Example 2.
Input:

```
64 64 8
how much wood could a woodchuck chuck if a woodchuck could chuck wood
```

Output:

```
how much wood could a woodchuck chuck if<20,14>ould chuck wood
```

Example 3.
Input:

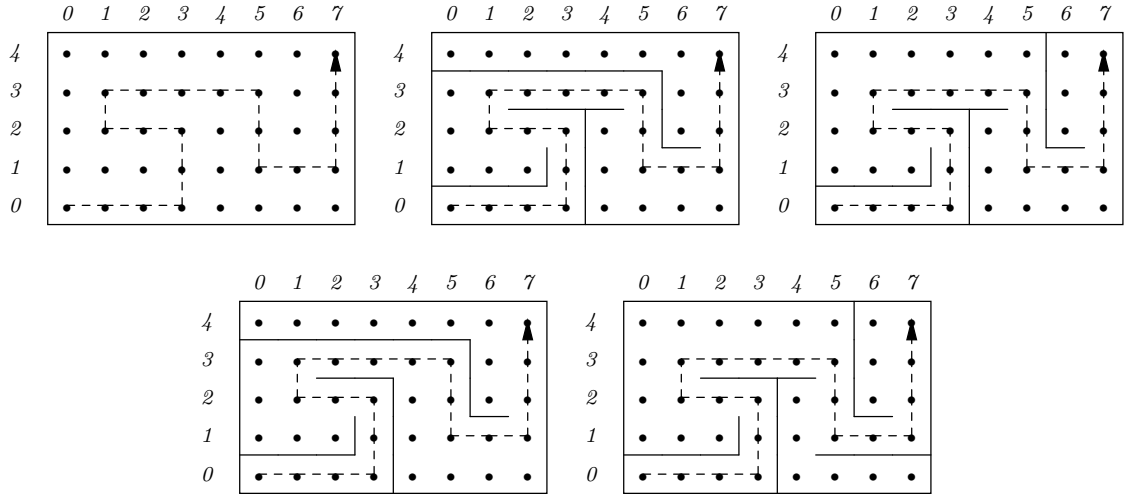
```
16 16 2
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Output:

```
x<0,16><0,15>
```

3. In these contests, you'll often see questions such as "find the shortest path out of this maze." Let's turn this on its head and ask "given a path, find a maze for which it is the shortest path out." Our paths will run vertically and horizontally between the regularly spaced points of a rectangular grid. The problem is compute a set of baffles (walls) separating grid points that forces the given path to be the *unique* shortest path from its starting point to the end point. To make things interesting, furthermore, we will require that there be no extraneous walls constructed in the sense that it should not be possible to remove any wall and still have the given path be the unique shortest path.

For example, consider the path through the 8×5 grid on the left below. The wall placements in the two mazes to its right make that path unique. The two lower mazes are faulty. The path is not unique in the one to the left, and there are redundant walls on the right.



WRONG: Not unique WRONG: unnecessary walls

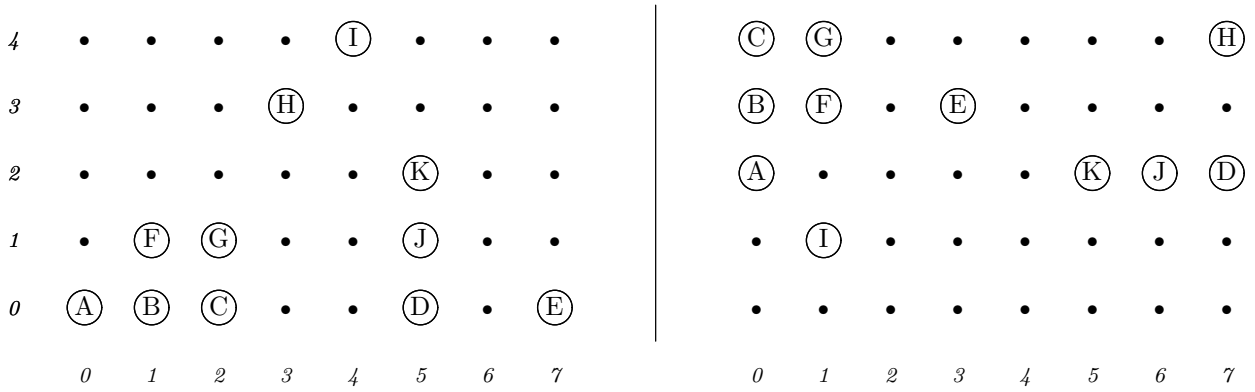
The input to your program (in free format) will consist of two positive integers, $W \leq 100$ and $H \leq 100$, giving the number of grid points horizontally and vertically, respectively, followed by a path. The path always starts in the lower-left corner, (0,0). It is specified as a string of `U` (up), `D` (down), `L` (left), and `R` (right) characters (with no embedded whitespace). You may assume that the path remains within the bounds of the maze and does not intersect itself. It may end anywhere in the maze (i.e., not necessarily in a corner or against a wall).

The output will consist of a sequence of zero or more wall specifications, in the format shown in the example on the next page. Each wall specification is in the form of two pairs of (x, y) coordinates specifying adjacent grid points ($0 \leq x < W$, $0 \leq y < H$), using the coordinate system illustrated in the diagrams above. Each such pair specifies that a segment of wall blocks passage between the given grid points. The possible output is not unique.

Example: This example shows the maze used in the examples above and the first (middle) solution shown.

Input	Output
`8 5`	`Wall between 0,0 and 0,1`
`RRRUULLURRRRDDRRUUU`	`Wall between 1,0 and 1,1`
	`Wall between 2,0 and 2,1`
	`Wall between 2,1 and 3,1`
	`Wall between 3,0 and 4,0`
	`Wall between 3,1 and 4,1`
	`Wall between 3,2 and 4,2`
	`Wall between 3,2 and 3,3`
	`Wall between 2,2 and 2,3`
	`Wall between 4,2 and 4,3`
	`Wall between 0,3 and 0,4`
	`Wall between 1,3 and 1,4`
	`Wall between 2,3 and 2,4`
	`Wall between 3,3 and 3,4`
	`Wall between 4,3 and 4,4`
	`Wall between 5,3 and 5,4`
	`Wall between 5,3 and 6,3`
	`Wall between 5,2 and 6,2`
	`Wall between 6,1 and 6,2`

4. [D. Garcia] The game of 2D-Nim is played on a rectangular grid, with pieces on the grid points. On each move, a player may remove any non-zero number of *contiguous* pieces in any row or column. The player who removes the last piece wins. In the left position, for example:



the player on move may remove (A), (B), (C), (A, B), (A, B, C), (B, C), (B,F), etc., but may *not* remove (A, C), (D, E), or (H, I).

For purposes of writing 2D-Nim-playing software, a certain programmer wants to be able to tell whether a certain position has ever been analyzed previously. Because of the rules of 2D-Nim, it should be clear that the board on the right is essentially equivalent to that on the right, and might as well be counted as the same position. That is, if there is a winning strategy for the left board, the same one must apply to the right. The fact that the contiguous groups of pieces appear in different places and orientations is clearly irrelevant. All that matters is that the same clusters of pieces (a *cluster* being a set of contiguous pieces that can be reached from each other by a sequence of one-square vertical or horizontal moves) appear in each. For example, the cluster of pieces (A, B, C, F, G) appears on both boards, but it has been reflected (swapping left and right), rotated, and moved. Your task is to determine whether two board positions are equivalent in this sense.

Each set of input, in free format, consists of a pair of integers, W and H, giving the number of grid points horizontally (W) and vertically (H), followed by a sequence of pairs of integers $x_i \;\; y_i$, giving the coordinates of the pieces of the first board, followed by a pair of -1s, followed by more pairs $x_i \;\; y_i$ of coordinates for pieces on the second board, again followed by two -1s. Here, $0 \le x_i < W$ and $0 \le y_i < H$. You may assume that $0 < W, H \le 100$. There may be more than one set.

You are to output indications of whether each pair is or is not equivalent, using the format shown in the example on the next page.

Example: The first set is the example above. In the second, piece K has been moved to (6,1) from (5,2) in the first board.

Input	Output

8 5
0 0 1 0 2 0
5 0 7 0 1 1 2 1
5 1 3 3 5 2
4 4
-1 -1
0 4 0 3 0 2
1 1
1 4 1 3
3 3
5 2 6 2 7 2
7 4
-1 -1
 	
8 5
0 0 1 0 2 0
5 0 7 0 1 1 2 1
5 1 3 3 6 1
4 4
-1 -1
0 4 0 3 0 2
1 1
1 4 1 3
3 3
5 2 6 2 7 2
7 4
-1 -1
	Set 1.
	  Boards are equivalent.
	
	Set 2.
	  Boards are not equivalent.

5. The Color Lines computer game is played on a 9x9 board, each square of which can either be empty or can contain a piece having one of seven colors: red, orange, yellow, green, dark blue, light blue, purple. Each move starts with having the computer place three pieces with random colors on random empty squares. The human player may then move any one piece to any empty square that can be reached by moving the piece through a path of empty squares, each of which is one square up, down, left, or right from the preceding (no diagonal moves allowed). After each of these moves—the computer's and the human's—any adjacent sequence of at least five pieces with the same color is removed. The removal of these sequences is simultaneous, so that if one piece is part of two such sequences, both are removed. The game continues as long as the board is not filled, with the human accumulating points for the pieces he manages to remove.

This problem is concerned only with the human's move. Given a position that results after a computer has made its move and removed any pieces, you are to choose the "best" next move, defined here naïvely to mean the move that removes the most pieces. When there are two such moves, prefer moving the pieces nearer the top row, then (if that's not sufficient to decide), prefer moving pieces nearer the left column, then prefer moving to the square nearest the top row, then prefer moving to the square nearest the left column. As an example, the proper move in the diagram on the left is (8,8) to (7,4) (that is, row 8, column 8 to row 7, column 4, as shown in the diagrams).

	1	2	3	4	5	6	7	8	9
1					O				
2	O				O		G	G	
3					O		P		
4	B				O		P		
5	B	B					P		
6			B		O		R		
7	B	B	B		B		G		
8					B		G	B	B
9	B						G		Y

	1	2	3	4	5	6	7	8	9
1					O				
2	O				O		G	G	
3					O		P		
4					O		P		
5	B						P		
6					O		R		
7							G		
8							G		B
9	B						G		Y

Input will consist of picture of the board: nine lines of nine characters apiece. Each character is either '-' (blank), or one of the color codes ('Y', 'G', 'P', 'O' (capital Oh), 'R', 'B', and 'L' (light blue)). The output will consist of a picture of the resulting board, in the format shown in the examples. When no move results in removing pieces, however, the output will instead consist of a single line:

```
No removals possible.
```

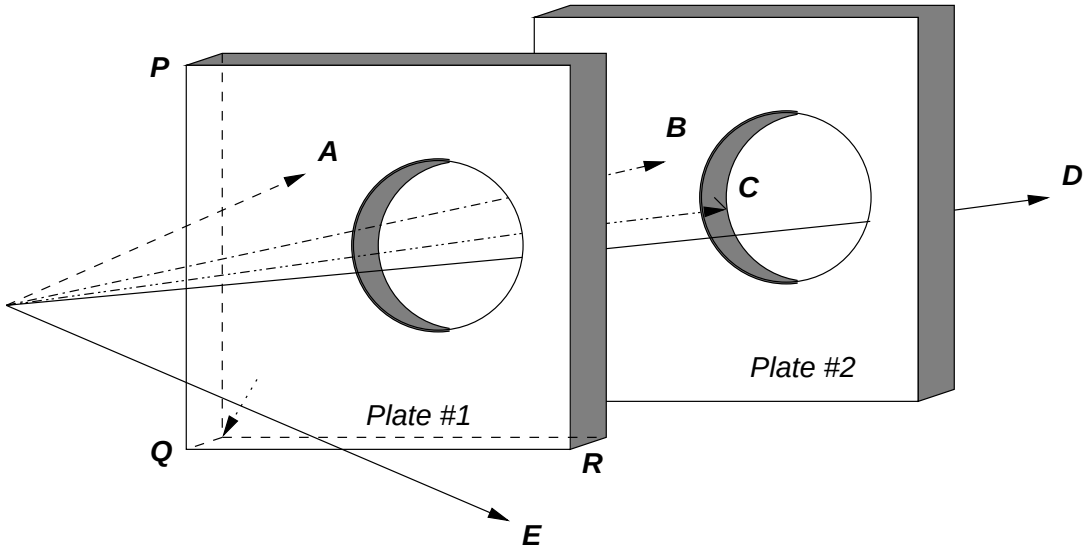
Example 1: This is the example shown above.

Input	Output
----O----	----O----
O---O-GG-	O---O-GG-
----O-P--	----O-P--
B---O-P--	----O-P--
BB----P--	B-----P--
--B-O-R--	----O-R--
BBB-B-G--	------G--
----B-GBB	------G-B
B-----G-Y	B-----G-Y

Example 2:

Input	Output
----O----	No removals possible.
O---OYGG-	
----O-P--	
B---O-P--	
BB--Y-P--	
--B-O-R--	
BBBOB-G--	
YYYYB-GBB	
------G-Y	

6. A room contains a set of thick, rectangular steel plates with round holes of various diameters drilled through them. A pellet (whose dimensions are so small that we will treat it as a point) is fired in a straight line. The task is to discover which plate, if any, it collides with and whether it collides with a face of the plate or with the interior of the plate—that is, with the wall of one of the holes (we will assume that the path of the pellet, extended to infinity, never intersects the outer edges of the plate, but only the (curved) inside surface of a hole or the front or back surface of the plate). For example, in the diagram below, pellet A collides with the surface of plate #1, B collides with the surface of plate #2, C collides with the interior surface of the hole in plate #2, and D and E miss the plates entirely.



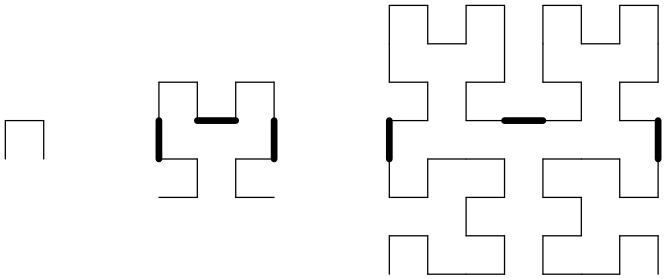
The input data, in free form, consists of an integer, $K \geq 0$, followed by K descriptions of plates, followed by descriptions of zero or more pellets. Each plate's description starts with four points, each of which is given as a triple of floating-point numbers (the x, y, and z coordinates) in free format. The first three points (call them P, Q, and R) are three corners of one surface of a plate, so that $\angle PQR$ is a right angle, as illustrated in the diagram above. The fourth point (call it Q') is the corner of the other surface of closest to Q (so that $\overline{QQ'}$ is perpendicular to the surfaces of the plate and is as long as the plate is thick). These data entirely determine the opposite surface and the thickness of the plate. Next follows an integer, $N \geq 0$, giving the number of holes in the plate, followed by N sets of four numbers x, y, z, r giving the center of the hole on the first surface of the plate (the one containing P, Q, and R) and its radius. The holes go straight through, at right angles to the surface, do not intersect each other, and do not intersect the edges of their plate. Plates do not intersect. Next come any number of descriptions of pellets. Each pellet is fired from the coordinate origin, (0,0,0). The pellets' directions are given by triples of the floating-point numbers, which give the x, y, and z coordinates of a direction vector (whose length is irrelevant). Output is in the form given in the example on the next page.

Example 1: This is roughly example shown above, seen from behind, above, and to the right of the origin. The positive z direction is into the paper. The plates are both 3×3 and 0.25 units thick.

Input	Output

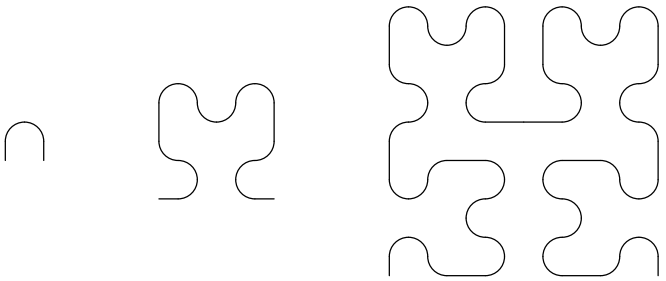
`2`
 	
`-2 1.5 3  -2 -1.5 3  1 -1.5 3`
`  -2 -1.5 3.25`
`1`
`0 0 3 0.5`
 	
`-2 1.5 6  -2 -1.5 6  1 -1.5 6`
`  -2 -1.5 6.25`
`1`
`0 0 6 0.5`
 	
`-0.5 0.5 1.5`
`-0.4 0.1 3.25`
`-0.25 0 3.0625`
`0.01666 -0.01666 1`
`0.333 -0.666 1`
	`Pellet #1 hits surface of plate #1.`
	`Pellet #2 hits surface of plate #2.`
	`Pellet #3 hits interior of plate #2.`
	`Pellet #4 misses all plates.`
	`Pellet #5 misses all plates.`

7. [D. Garcia] A *Hilbert curve* is a recursively defined figure, defined as follows. Each curve is a continuous path contained within a square bounding box and has two ends at the lower-left and lower-right corners. The *level-1 Hilbert curve* is the left figure in the diagram below. Each of its sides is one unit long. For $k > 1$, the level-k Hilbert curve is composed of four level-$(k-1)$ curves connected by three one-unit segments, laid out as shown in the other figures in the diagram below (which show the level-2 and level-3 curves). The three thicker line segments in the diagrams show the connecting segments that join the four instances of lower-order curves.



In a true Hilbert curve, one carries this process out indefinitely, scaling the entire picture each time to have the same bounds. In the limit, one gets a space-filling curve. Unfortunately, our programs won't have enough time to go all the way, so for this problem, we are only interested in finite levels of the curve.

We can make the curve a little more... well... curvaceous by replacing all right angles in the finished curve with quarter arcs (90°) with radius 1/2 unit, as shown below:



Such a curve can be described as a sequence of commands to draw line segments 1/2 unit long in the up, down, right, or left direction, and to draw quarter arcs in which arc is initially directed up, down, right or left, and curves around to the left, right, up, or down direction through 90°. Your program is to input a single integer, N, indicating the level of curve desired, and to output a sequence of such commands in the format illustrated in the examples on the next page (`down`, `left`, etc. draw 1/2-unit lines in the indicated directions; `arcupright` draws a quarter arc that proceeds upward and then to the right; etc.). The sequence of commands must draw a path starting in the lower-left corner and proceeding in a sequence of connected, non-overlapping arcs and line segments. This determines a unique solution.

Example 1: Level 1 curve

Input	Output

`1`	`1 hilbert`
	`up`
	`arcupright`
	`arcrightdown`
	`down`

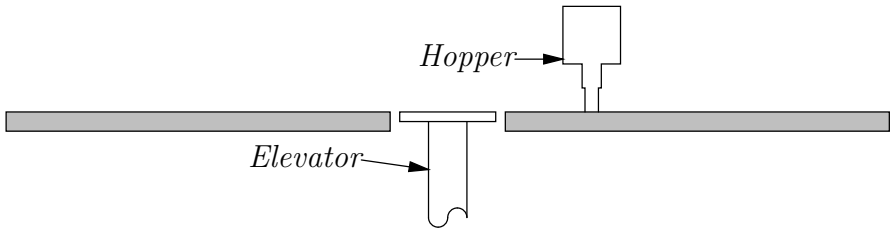
Example 2: Level 2 curve

Input	Output

`2`	`2 hilbert`
	`right`
	`arcrightup`
	`arcupleft`
	`arcleftup`
	`up`
	`up`
	`arcupright`
	`arcrightdown`
	`arcdownright`
	`arcrightup`
	`arcupright`
	`arcrightdown`
	`down`
	`down`
	`arcdownleft`
	`arcleftdown`
	`arcdownright`
	`right`

Note: The output you are asked to generate for this problem consists of Postscript commands. To help you test your solutions, you can paste these lines into the indicated place in the file `~ctest/lib/template.pro`. The resulting file can be printed on a Postscript printer or viewed using `ghostview`, to show the curve you have drawn.

8. As a field engineer of Peculiar Tasks, Inc., you are asked to move a hopping machine down an elevator. The initial situation is as illustrated below:



The hopper is capable of hopping left or right, but the distance covered by one hop can only be set to one of a fixed, finite set of integral values at a time. You are to determine a sequence of distance settings and numbers of hops (left or right) that will place the machine on the elevator, given that it starts out one unit of distance to the right of the elevator. The input will consist of a positive number, N, followed by N possible hop distance settings, all positive, in strictly increasing order. The output is to have the format shown in the examples, with the numbers of hops of each hop distance listed in increasing order of hop distance, listing only distances with non-zero numbers of hops.

You may assume there is at least one solution, that $N <= 8$, and that the total distance hopped back and forth will not exceed $2^{31} - 1$ units of distance.

Example 1:

Input	Output
`2 7 11`	`Hop right 3 times by 7 units.` `Hop left 2 times by 11 units.`

Example 2:

Input	Output

`4`
`6 10 15 30`
	`Hop right 4 times by 6 units.`
	`Hop left 4 times by 10 units.`
	`Hop right 1 times by 15 units.`