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

Programming Contest
Fall 2004

P. N. Hilfinger

2004 Programming Problems, 3rd edition

Please make sure your electronic registration is up to date, and that it contains the correct account you are going to be using to submit solutions (we connect names with accounts using the registration data).

To set up your account, execute

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

in all shells that you are using. (This is for those of you using csh-like shells. Those using `bash` should instead type

```
. ~ctest/bin/setup.bash
```

Others will have to examine this file and do the equivalent for their shells.)

This booklet should contain eight problems on 20 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* (yes, it is OK to have a Java source file whose base name consists of a number, even though it doesn't match the name of the class). 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`, and `algorithms`. 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 roughly equivalent to

`gcc -Wall -o` N `-O2 -g -I`*our-includes* N`.*  -lm`

For Java programs, it is equivalent to

`javac -g` 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.

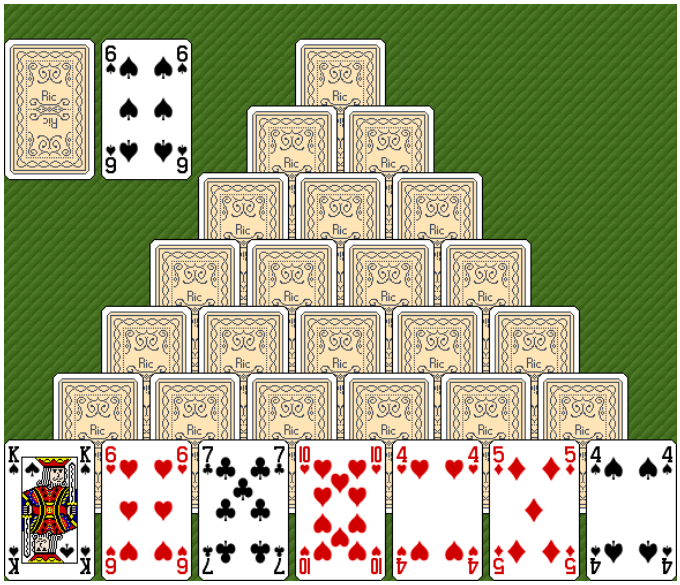
Protests. Should you disagree with the rejection of one of your problems, first prepare a file containing the explanation for your protest, and then use the `protest` command (without arguments). It will ask you for the problem number, the submission number (submission 1 is your first submission of a 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.

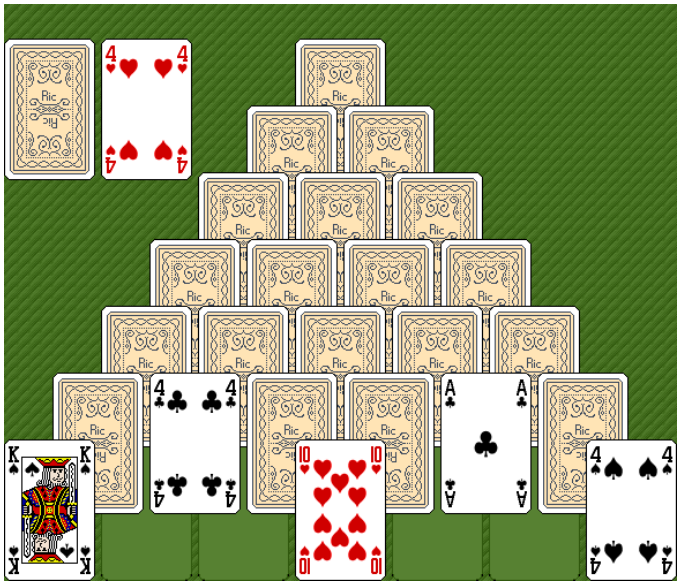
Notices. During the contest, the Web page at URL

`http://inst.cs.berkeley.edu/~ctest/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. The object of the solitaire game Elevator (found on a GNU Linux distribution) is to move all cards from a triangular tableau to a waste pile. Initially the tableau is arranged in rows, as shown below in the diagram on the left, with the remaining cards (the *stock*) face-down at the top left of the diagram. Play proceeds by turning over a card from the stock to the waste pile (immediately to its right), and then playing exposed cards from the tableau onto the waste pile, as permitted. When all cards that were partially covering a tableau card are moved to the waste pile, that tableau card is turned over (exposed). (Since the lowest row of the tableau has no cards on top of it, it is therefore initially face-up, as shown.) These moves continue until the stock is exhausted and no more moves are possible (a loss), or the tableau is empty (a win). A card from the tableau may be moved to the top of the waste pile if its rank is one greater than or one less than the uppermost card on the waste pile. The card ranks are Ace, 2, 3, ..., 10, Jack, Queen, King, Ace, 2, etc. (i.e., Ace is after King and before 2). Suits are ignored. In the illustration below the left configuration shows an initial tableau, stock (face down in the upper left), and waste pile (face-up to its right) just after the top stock card (6♠) has been played onto the waste pile. We can now play the 7♣, 6♡ (exposing the 4♣), the 5♢, and the 4♡ (exposing the A♣) in that order, leaving the 4♡ on top of the waste pile. Since there are no exposed cards that are legal to place on the 4♡, it's time to turn over another stock card.





You are to write a program that simulates any number of games of elevator, printing out the sequence of cards played to the waste pile. The input for each game consists of cards in free format. A card has the form RS with no embedded whitespace, where the rank, R, is 'A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', or 'K' and the suit, S, is 'C', 'D', 'H', 'S'. The first 28 cards represent the contents of the tableau, from top to bottom, left to right. The remaining cards represent the contents of the stock, with the 29th input being the bottom card and the last input being the top card (in the example below, the top card of the stock is the 6♠, which has been turned over onto the waste pile). The last

set of cards is followed by a single digit 0.

For each game, output the game number followed by the sequence of cards played to the waste (from either the stock or the tableau). Play cards from the tableau as long as there is a legal play before playing another card from the stock. When there is a choice of two or more cards that are legal to play from the tableau, play the leftmost feasible card from the lowest feasible row of the tableau. Print the cards played in rows, with the first card in each row being from the stock and the rest (if any) from the tableau. After each set, print a summary saying either "Player wins" if the tableau is exhausted, or "Player leaves N cards" if the stock is exhausted and there are $N > 0$ cards left on the tableau. Separate games with a blank line. Use the output format shown in this example:

Example.

Input	Output
9D 2C 3C 6D 5C 6C 8C 9C 10C JC QC KC AD 2D 3D 4D 4C 7D 8D AC 10D KS 6H 7C 10H 4H 5D 4S JD QD KD AH 2H 3H 5H 7H 8H 9H JH QH KH AS 2S 3S 5S 7S 8S 9S 10S JS QS 6S KC JD QC 8D 9D 10D 4D 5D 6D 7D KD AD 2D 3D JH 6S 7H 8H 9H 10H 10C KS QS JS 10S 9S 8S 7S 2C 3C 4C 5C 6C 7C 8C 9C QD 3H AH 2H 4H 5H 6H KH AS 2S 3S 4S 5S QH JC AC 0	Game 1: 6S 7C 6H 5D 4H QS KS AC JS 10H 10S 9S 8D 7D 8S 7S 5S 4S 3S 4D 2S AD 2D AS KH QH JH 10D 9H 10C 8H 7H 5H 4C 3D 3H 2H AH KC QC JC KD QD JD Player leaves 8 cards Game 2: AC KS QS JS 10S 9S 8S 7S 6S 7H 8H 9H 10H JC 10C JH QH KD AD 2D 3D 4D 5D 6D 7D 8D 9D 10D JD QC KC Player wins

2. Morse code is an encoding of the alphabet, digits, and some punctuation marks into dots, dashes, and pauses. For the purposes of this problem, here is the encoding:

A	.-	N	-.	0	-----	period (.)	.-.-.-
B	-...	O	---	1	.----	comma (,)	--..--
C	-.-.	P	.--.	2	..---	?	..--..
D	-..	Q	--.-	3	...--		
E	.	R	.-.	4	....-		
F	..-.	S	...	5	.....		
G	--.	T	-	6	-....		
H	....	U	..-	7	--...		
I	..	V	...-	8	---..		
J	.---	W	.--	9	----.		
K	-.-	X	-..-				
L	.-..	Y	-.--				
M	--	Z	--..				

Normally, characters are separated by a pause approximately as long as a dash, and words are separated by a longer pause. Unfortunately, we have a large collection of messages from an inexperienced operator who has his dots and dashes down pat, but is very unreliable about pauses. Your task, therefore is to take a stream of dots and dashes and figure out possible messages that might correspond to them. To make this feasible, we'll assume that every word (where "words" are sequences of non-blank characters separated by blanks) must be one of the following:

(1) A non-empty string of letters that is found in a given dictionary.

(2) A string of one or more digits (0–9).

(4) One of (1) or (2) followed by a single period, comma, or question mark.

For example, if the dictionary contains the words *age, ail, an, bat, deem, man, manager,* and *rail,* and the input is

```
--.--..---...-..-...-..-....--.-.-.-
```

then the legal interpretations would be

```
man␣age␣rail␣bat.
man␣age␣rail␣deem.
manager␣ail␣bat.
manager␣ail␣deem.
```

(The character ␣ is a blank.)

The input will consist of a sequence of lower-case, free-format words for the dictionary in dictionary order, followed by a single slash (/) separated from the preceding words by whitespace, followed by one or more messages separated by whitespace, and terminated by a message consisting of a single slash. Each message (other than the terminating slash) will consist only of periods and hyphens.

For each message, print a sequence number followed by a repeat of the message, followed by all possible valid interpretations, each on one line, as shown in the example below. Print the messages in dictionary order according to the standard ASCII collating sequence (i.e., space, comma, period, digits 0–9, question mark, letters a–z). As shown in the example, if no interpretation is possible, print the single message "?" (which is not a valid message, according to the rules above).

Example.

Input

```
age ail  an bat deem
man manager rail  /
 --.--..---...-..-...-..-....--.-.-.-
 .-.-.-   .-..-...-......-..--- /
```

Output

```
Message 1: --.--..---...-..-...-..-....--.-.-.-
  man age rail bat.
  man age rail deem.
  manager ail bat.
  manager ail deem.

Message 2: .-.-.-
  ?

Message 3: .-..-...-......-..---
  rail 4 2
  rail 42
```

3. An N-story building contains a single elevator, which you are to simulate. Each floor has a single *call button* to request the elevator to come to that floor (i.e., there aren't separate "up" and "down" buttons). The elevator itself has N *floor buttons.* Buttons become "active" once they are pressed and stay that way until the system "resets" them. All action happens at discrete "ticks" separated by one "time unit."

When the elevator stops on a floor at some time t, its door becomes completely open at time $t+1$. People waiting to get off the elevator at this floor do so at time $t+1$. Likewise, people waiting to get on when the elevator arrives do so at time $t+1$. Before the elevator door next starts closing, people arriving at the elevator enter it immediately (at the time they arrive). The elevator doors start closing at time t when (a) some button (floor button or call button) was last activated at time $t-1$ or before and is still active, and (b) nobody entered or left the elevator at times $t-1$ or $t-2$ (assume nobody has entered the elevator at any time $t < 0$). People press their floor button at the moment they enter the elevator. If the door does start to close at time t, nobody arriving at time t can enter. Doors take one tick to close, at which point the elevator starts moving.

We say the elevator is in "up mode" if the last direction it moved was up, and "down mode" otherwise. When the doors fully close, the elevator starts moving up if

- it is in up mode and a (call or floor) button is active for a floor above it, or
- it is in down mode, a button is active for a higher floor, and no button is active for a lower floor.

Contrariwise for moving down. Once it starts moving, the elevator takes three ticks to go from one floor to the next. If the elevator arrives at a floor at time t, it stops at that floor if that floor's call or floor button is active and was last activated at or before time $t-2$. The door begins to open when the elevator arrives. A button is reset when the doors start to open on the corresponding floor. The button continues to be reset immediately (i.e., pressing it does not activate it) until the elevator next starts to move. Initially (at time 0), the elevator is in up mode on floor 1 with its doors completely open, and no buttons are active.

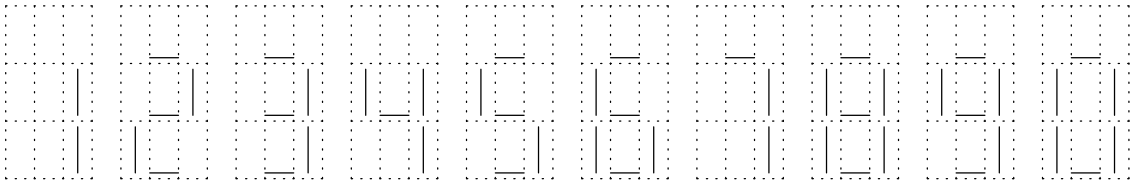
You are to simulate this system, reporting when each person starts to enter and leave the elevator. The input is in groups, one for each simulation. Each group starts with an integer $N > 1$, giving the total number of floors. This is followed by a sequence of arrival records in free format, each consisting of five pieces of information: a name (a string of 100 or fewer characters and containing no embedded whitespace), an arrival time (a non-negative integer number of ticks from the start of the simulation) indicating when the person arrives at the elevator, a starting floor (an integer in the range 1–N), and a destination floor (different from the starting floor). The last arrival record is followed by delimiting record in which all five fields are the single digit '0'. Arrival records are ordered by arrival times. The last group of input is followed by the integer 0.

The output should report entry and exit times corresponding to all arrival records, in the format shown in the example, in the same order that the arrival records were input. Follow the output for each group with a blank line.

Example.

Input	Output
7	Simulation 1:
Kathy 0 3 7	Kathy enters on 3 at time 9 and exits on 7 at time 31
Brian 0 4 7	Brian enters on 4 at time 17 and exits on 7 at time 31
John 40 4 6	John enters on 4 at time 52 and exits on 6 at time 80
Jack 41 4 2	Jack enters on 4 at time 52 and exits on 2 at time 63
0 0 0 0	
10	Simulation 2:
Peter 0 5 10	Peter enters on 5 at time 15 and exits on 10 at time 35
Paul 4 2 10	Paul enters on 2 at time 64 and exits on 10 at time 93
0 0 0 0	
0	

4. [From the Universidad de Valladolid's on-line problem collection] A bank wishes to use optical character recognition (OCR) to read account numbers from checks. These particular account numbers are written using seven-segment digits. Some image-processing software recognizes horizontal and vertical segments on a scanned check, and converts them to ASCII bars '|' and underscores '_'. When correctly printed, the ten digits look like this:



A bank account has a 9-digit account number. Not all 9-digit numbers are valid; for purposes of error detection and correction, a correct account number $d_9 d_8 \cdots d_1$ satisfies the following checksum condition:

$$(d_1 + 2d_2 + 3d_3 + \ldots + 9d_9) \bmod 11 = 0.$$

Unfortunately, the scanner sometimes makes mistakes: some line segments may be missing. Your task is to write a program that deduces the original number, assuming that:

- when the input represents a valid account number, it is the original number;
- at most one digit is garbled;
- the scanned image contains no extra segments.

The scanner outputs ASCII bars ('|' and '_') for segments it sees, and periods for spaces that contain no segment. For example, the following input

```
....._..._.....___..._..___..___.
..|._|._||_||_|.|_...||_||_|
..|._..._|..|._||_|..||_|._|
```

used to be `123456789`. Making the second digit into a '3' instead of '2' would violate the checksum condition. Other modifications (such as changing '5' into '6') would violate the "only one garbled digit" criterion.

The input consists of a sequence of account numbers rendered in ASCII. Each account number takes up three lines, and each digit takes up three columns, giving 27 characters per line. There are no blank lines or extra blank columns anywhere.

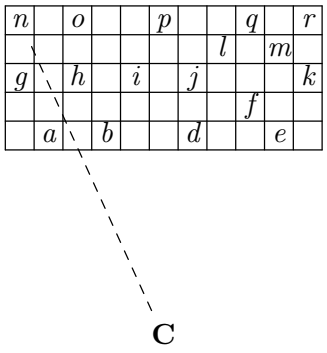
For each test case, the output contains one line giving the sequence number of the input and nine digits if the correct account number can be determined, the string "failure" if no solutions were found and "ambiguous" if more than one solution was found. Use the format shown in the example below.

Example.

Input	Output
....._..._..._..._..._. ..|._|._||_||_.|_...||_||_| ..|._..._|..|._||_|..||_|._| ....._..._..._..._..._. |_||_||.||.||_...|..|..||_. ..|._||_||_||_|..|..|..|._| ._..._..._..._..._..._. |_||_||_||_||_||_||_||_||_
_		_		_		_		_		_		_		_		_	 ._..._..._..._..._..._. 	_	..		_		_		_		_		_		_
_|..||_||_||_||_||_||_|	Input 1: 123456789 Input 2: ambiguous Input 3: failure Input 4: 878888888

5. Large choirs that perform in numerous venues have a chronic problem: how to place everyone so that they can see the conductor. Naturally, this placement depends on the choristers' heights, the conductor's position and height, and the available places for standing (we will simplify matters here and consider only performances on flat areas, so that you won't have to deal with different elevations for different parts of the floor). Your job is to write a program that detects sight problems in a given arrangement of choristers, given the choristers' heights, their positions, and the conductor's location.

We'll consider a very simplified arrangement in which the possible positions are points on a regular grid. Here, for example, is a small ensemble:



Each lower-case italic letter represents a singer; the capital boldface 'C' represents the conductor; empty boxes are vacant. Choristers' x and y coordinates are non-negative integers. Seen from the top, with the conductor "below" the chorus, the floor space occupied by a chorister at location (x, y) is a square whose lower-left corner is at $(x - \frac{1}{2}, y - \frac{1}{2})$ and whose upper-right corner is at $(x + \frac{1}{2}, y + \frac{1}{2})$. The center of the lower-left grid square in the diagram is at coordinates (0,0); the conductor in this example is at coordinates $(5, -7)$; the conductor will always have a negative y coordinate.

The problem is to check if a particular group of choristers is properly arranged so that none none *blocks* another's sight-line to the conductor. For simplicity, we'll take the conductor (actually, the conductor's eyes) to be located at a certain point in space, and choristers to be boxes (rectangular solids) with various heights and identical square bases that fit exactly in one grid space. Define *eye level* (somewhat arbitrarily) as 0.95 of height (because you block even some sight lines that are slightly higher than your eyes). We'll say that chorister Q blocks chorister P if a line segment drawn from the center of chorister P's box at eye level to the conductor's eyes intersects chorister Q's box.

In the example above, no other chorister can block chorister n, because the line from n to the conductor (shown dashed in the figure) does not intersect any other grid square with a chorister in it. On the other hand, chorister d might block j, depending on height. For example, if the conductor's eyes are at height 105, and choristers d and j both have height 100, then chorister d *does* block j, because a line from j's eyes (at height 95) to the conductor intersects d's box (height 100). If, on the other hand, d's height were 95, then d would not block j.

The input to your program begins with a positive number, $M < 300$ of choristers. Next come three integer numbers giving the x, y, and z coordinates of the conductor's eyes where one unit is the length of one side of a grid square, the origin (0,0,0) corresponds to floor level in the middle of the leftmost front grid square (leftmost bottommost in the diagram above), and coordinates increase left-to-right, front-to-back, and floor to ceiling. Finally come M 4-tuples of data, one for each chorister, giving a name, an integer height, and x and y grid coordinates (also integers). (It may occur to you that the proportions used in the data below are from unhumanly thin choristers. Let's just say that we're using different units for height than for x and y positions; it makes no difference in the calculations). All names are no more than 30 characters long.

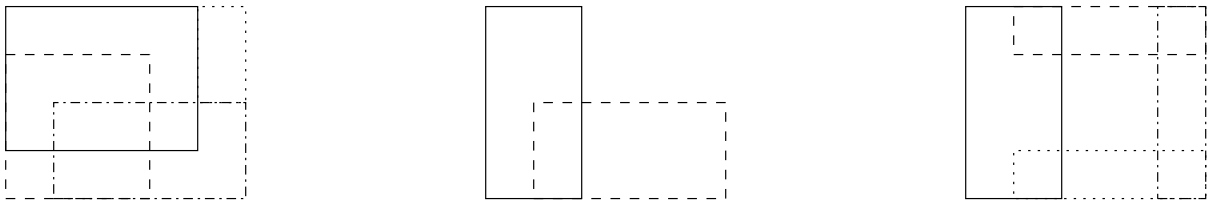
Print out a list of all conflicts in the format shown below, or the message "No conflicts." List "A blocks B" before "C blocks D" if B occurs before D in the input list of choristers. If B and D are the same (two choristers block the same chorister), then list "A blocks B" first if A precedes C in the input list.

Examples

Input	Output
17 5 -7 80 Able 70 1 0 Baker 67 3 0 Donner 68 6 0 Ernst 70 9 0 Fowler 65 8 1 Gantry 76 0 2 Halpern 70 2 2 Irving 76 4 2 Jacobs 77 6 2 Kenner 75 10 2 Lane 75 7 3 Muller 70 9 3 Norman 70 0 4 O'Brian 75 2 4 Philips 65 5 4 Quimby 70 8 4 Roberts 75 10 4	No conflicts.

Input	Output
3 5 -7 80 Fowler 70 8 1 Muller 70 9 3 Roberts 71 10 4	Fowler blocks Muller Muller blocks Roberts

6. Given a set of rectangular areas in the plane, possibly overlapping, is the area they cover (their union) itself a complete rectangular area? For example, the four rectangles on the left below do form a rectangle, while the rectangles in the middle and on the right do not (the middle figure is not rectangular and the one on the right has a hole in it).



The input for this problem will consist of one or more sets of data in free format. Each set will begin with an integer $N < 1000$ giving a number of rectangles to follow. Next will follow N rectangles, each specified as four integers, X_l, Y_l, X_r, and Y_r. These specify coordinates for the lower left corner (X_l, Y_l) and upper right corner (X_r, Y_r) of the rectangle (whose sides are always aligned parallel to the x and y axes). Coordinates may be any integer in the range -2^{31} to $2^{31} - 1$. The last set of input is followed by a single integer 0.

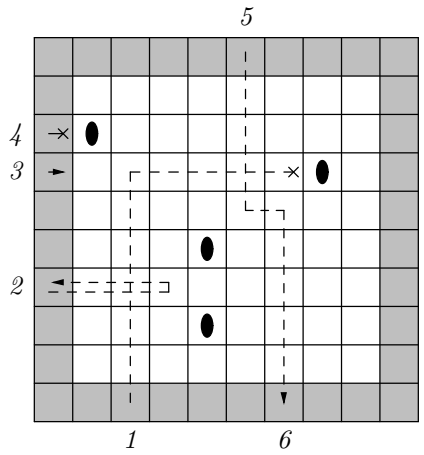
The output will consist of one line for each set, giving the number of the set and either the lower left and upper right corners of the rectangle formed by the union of the input rectangles, or a message that the area covered is not rectangular. Use the format shown in the example.

Example.

Input	Output

`4`
`0 0 3 3  0 1 4 4`
`1 0 5 2  4 2 5 4`
 	
`2  -1 -1 1 3 0 -1 4 1`
 	
`4`
`0 0 2 4  1 0 5 1  1 3 5 4`
`4 0 5 4`
`0`
	`Set 1: Covers (0,0), (5,4)`
	`Set 2: Not rectangular`
	`Set 3: Not rectangular`

7. My newly installed operating system came with a logic puzzle that requires that you guess the locations of hidden markers on a 10×10 grid, such as the one pictured below. The markers are never placed in squares around the edges of the grid (shaded in the diagram). A player can get additional information about placement of markers by *probing:* choosing a square on the edge of the grid (other than the corners), sending out a particle inward along the chosen row or column, and observing the results. The particle either eventually (re-)enters an edge square, leaves the board, or is absorbed by a marker. The player is informed of the result of each probe, and after making as few probes as possible, is supposed to guess where the markers are. In this problem, however, we are only concerned with computing the results of probes.



The particle moves from grid square to grid square according to the following rules:

- If there is a marker one square directly in front of the particle, the particle is absorbed and stops. In the diagrams above, this is marked with an $\times$.
- Otherwise, if there is a marker immediately ahead of the particle and one square to its right, the particle stops if it is on the edge (i.e., has just started), and otherwise makes a 90° turn to the left without leaving the square.
- Otherwise, if there is a marker immediately ahead of the particle and one square to its left, the particle stops if it is on the edge, and otherwise makes a 90° turn to the right without leaving the square.
- Otherwise, the particle moves forward one square. If this takes it onto an edge square, it stops.

The steps above repeat until the probe particle stops. For example, in the diagram above, particles starting from *1* and *4* are absorbed, particles starting from *2* and *3* end up on the square they started from, and the particle starting from particle *5* is deflected twice to end up at *6*.

You are to write a program that, given the placement of a set of markers, determines the results of a sequence of probes. The input data consist of a sequence of puzzles. Each

puzzle begins with two integers integer, $M, P \geq 0$, indicating the number of markers ($0 \leq M \leq 64$) and the number of probes $P > 0$. These are followed by M pairs of integers $1 \leq c, r \leq 8$, each indicating the column and row at which a marker is placed. This is followed by P pairs, $0 \leq c, r \leq 9$, giving the column and row at which a probe starts (always on the edge, never in the corners). The last puzzle is followed by two integers -1.

For each puzzle, the output lists the sequence number of the puzzle and a list of results of the probes. Each probe result shows the starting position (on the edge) and either an ending position or an 'X', depending on whether the probe stops on the edge or is absorbed. Use the format shown in the example below.

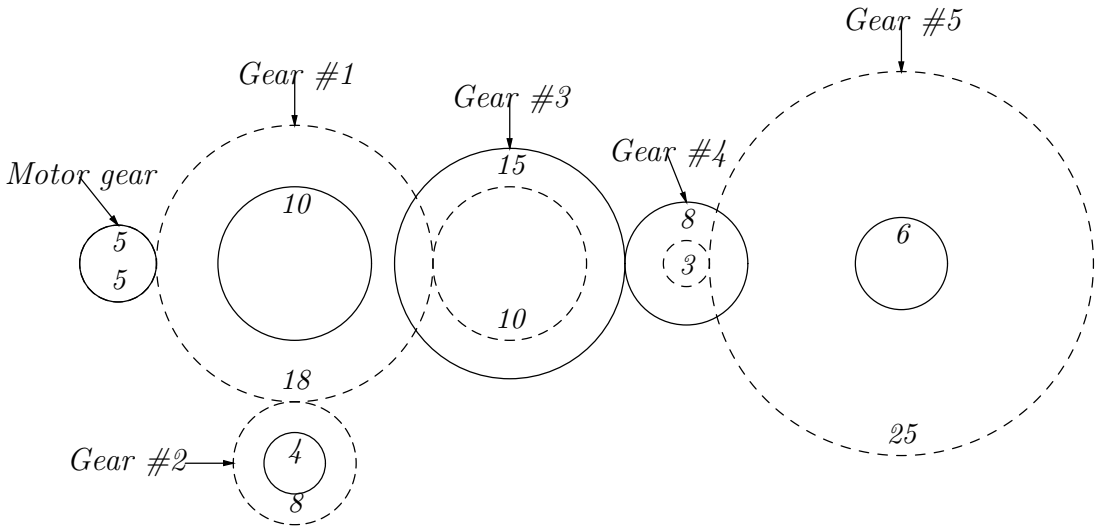
Example.

Input	Output

`4 5`
`1 7  4 2  4 4  7 6`
`2 0  0 3  0 6  0 7`
`5 9`
`-1 -1`
	`Puzzle #1:`
	`  1. (2,0) -> X`
	`  2. (0,3) -> (0,3)`
	`  3. (0,6) -> (0,6)`
	`  4. (0,7) -> X`
	`  5. (5,9) -> (6,0)`

8. [Miguel Revilla, from the Universidad de Valladolid's on-line problem collection] An engineering firm called "Gears R Us" needs a program that can evaluate the operation of gears on a board. The board is a two- dimensional mounting plane that allows gears with two levels of teeth (a lower radius closest to the board and an upper radius away from the board). Each gear rotates around its center and interacts by turning an adjacent gear that it touches (either on the lower or upper level), so that at the point of contact, both of the touching edges are moving at the same velocity (with one gear rotating clockwise and the other counter-clockwise). Upper gears mesh only with other upper gears; lower gears only with other lower gears. The upper and lower parts of a gear rotate in lock step at the same angular velocity (the same number of RPM). Gears that are not driven (directly or indirectly) by the motor gear are "idle" and do not rotate.

"Gears R Us" uses square mounting boards that have a 300×300 grid of mounting holes. When viewing the board, the lower left of the board is position $x = 1, y = 1$; the upper right of the board is position $x = 300, y = 300$. The gears mount only on mounting holes and are available with integer lower and upper radii between 1 and 100. The "motor" gear is the only original source of energy of any board and is actually a gear (powered from behind the board) subject to all the restrictions described for gears above. The following diagram shows a sample configuration. The bottom part of each gear is shown with a dashed circle, and the upper part with a solid circle. The radii of the parts are shown just inside each circle (both parts of the motor gear in this example happen to have the same radius).



If the motor gear in this example is rotating counter-clockwise at 300 RPM, and the gears contact each other as shown (without overlapping), then it is possible to compute that Gear #1 rotates to the right (clockwise) at 83.33 RPM; Gear #2 rotates to the left (counter-clockwise) at 187.50 RPM; Gear #3 rotates to the left at 150.00 RPM; Gear #4 rotates to the right at 281.25 RPM; and Gear #5 rotates to the left at 33.75 RPM.

It is illegal for the upper or lower parts of two gears to overlap rather than touch. It is also illegal for a gear to be driven at two or more different speeds. It is valid for two or more gears to drive another gear at the same speed (and in the same direction).

Input to your program consists of an undetermined number of configurations to be analyzed in free format. Each configuration starts with two integers, N and v, giving the number of gears (including the motor gear) in the set and the the rotational velocity of the motor in RPM (negative representing counter-clockwise rotation and positive representing clockwise rotation). For each of the N gears, there are four integers—x, y, r_l and r_u—giving the (x,y) coordinates of the center ($1 \leq x, y \leq 300$) and the radii of the lower and upper parts of the gear ($1 \leq r_l, r_u \leq 100$). The first gear listed is the motor gear. The data for the last configuration is followed by two integer 0's.

For each input set, your program should output the sequence number of the simulation and either a list of the gears and the rotation value for each or one of two error messages. Use the format shown in the example below. Express rotation values as a letter 'L' (for left or counter-clockwise) or 'R' (for right or clockwise) followed by the rotation rate as a positive number rounded to two decimal places. For idle gears, output "Idle" instead. Print the rotation rates for the gears in the same order they appeared in the input, leaving off the motor gear. If any gears overlap in their upper or lower portions, print "Error: overlapping gears" and no list of rotation rates. Otherwise, if any gear is being driven at two or more speeds, print "Error: conflicting rotations" and no list of rotation rates. Separate output sets by blank lines.

Example.

Input	Output

`6 -300`
`20 100 5 5`
`43 100 18 10`
`43 74 8 4`
`71 100 10 15`
`94 100 3 8`
`122 100 25 6`
 	
`6 -300`
`20 100 5 5`
`43 100 18 10`
`43 74 8 4`
`71 100 10 10`
`89 100 3 8`
`105 100 25 6`
 	
`6 -300`
`20 100 5 5  43 100 18 10`
`43 74 8 4  71 100 10 10`
`89 100 3 8  125 100 25 6`
 	
`3 10`
`100 100 3 4  106 100 3 1`
`106 92  5 6`
`0 0`
	`Simulation #1`
	` 1: R 83.33`
	` 2: L 187.50`
	` 3: L 150.00`
	` 4: R 281.25`
	` 5: L 33.75`
	
	`Simulation #2`
	`Error: overlapping gears`
	
	`Simulation #3`
	` 1: R 83.33`
	` 2: L 187.50`
	` 3: L 150.00`
	` 4: R 187.50`
	` 5: Idle`
	
	`Simulation #4`
	`Error: conflicting rotations`