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

Programming Contest **P. N. Hilfinger**
Fall 2002

2002 Programming Problems

If you are participating in the contest, please send me e-mail as soon as possible from the account you use to submit solutions. Use

```
Contest registration
```

(written just like that) as the subject line. Include in the body of the message one line that starts with your name. If your usual e-mail address differs from the login you use for the contest, also include that e-mail address.

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
```

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. 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 -Iour-includes N.* -lstdc++ -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 PN
```

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.

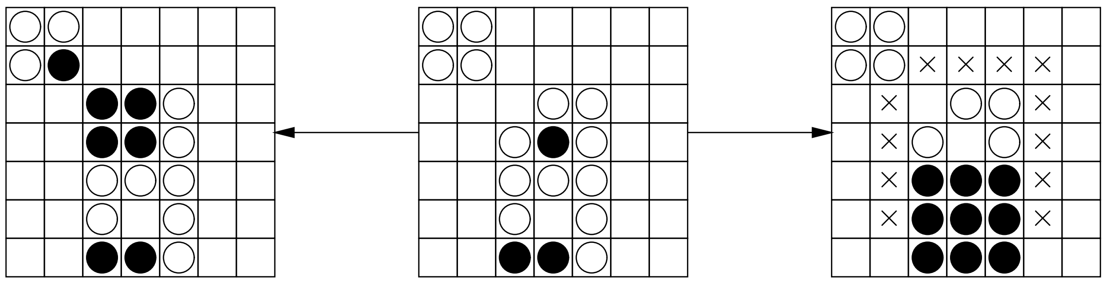
Protests. Should you disagree with the rejection of one of your problems, first prepare a file containing the explanation for your protest, and then use the `protest` command (without arguments). It will ask you for the problem number, the submission number (submission 1 is your first submission of a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

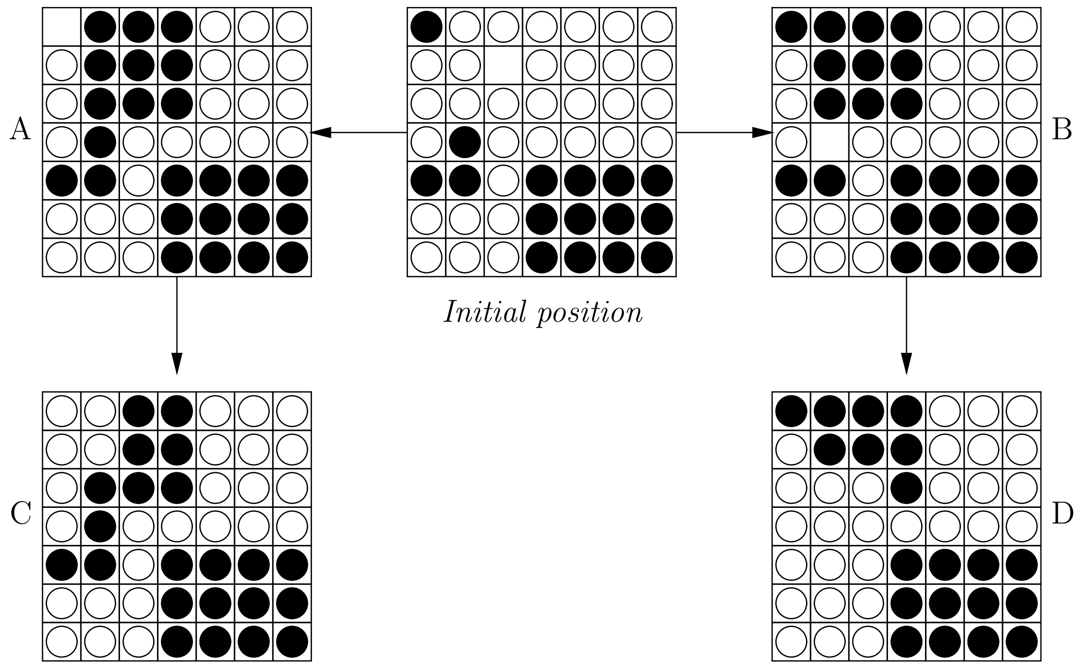
will contain any urgent announcements, plus a running scoreboard showing who has solved what problems. Sometimes, it is useful to see what problems others are solving, to give you a clue as to what is easy.

1. Ataxx is a two-person game played with blue and red pieces on a 7-by-7 board (we, however, will use white and black). As illustrated below, there are two possible kinds of move: you can *extend* from a piece of your own color by laying down a new piece of your color in an empty square next to that existing piece (horizontally, vertically, or diagonally), or you can *jump:* move a piece of your own color to an empty, non-adjacent square that is no more than two rows and no more than two columns distant. In either case, all opposing pieces that are next to the previously empty destination square are replaced by pieces of your color. Here is an example of an initial position (in the middle) and two possible moves with the same piece. Squares marked × show black's other possible jumps with that piece:



After black move to adjacent square *Initial position* *After black jump*

You are asked to implement a simple strategy to choose a best move for black in a given situation. The problem is to find a black move that results in black having the most pieces after white's best next move (that is, after white makes a move resulting in the fewest black pieces, or after white does nothing if he has no legal move). For example, starting from the position in the middle, move A is best for black, since after white's best move (C), black will have 22 pieces rather than 20:



Initial position

The input to your program will consist of a sequence of board positions, each of which is in the form of seven seven-character strings of w's (white squares), b's (black squares), and hyphens (empty squares) in free form, as illustrated below. You may assume that black has at least one move in each situation. Resolve ties as follows:

- Prefer to jump or extend *from* the piece that comes first when scanning the board left-to-right, top-to-bottom. If two jumps or extensions are possible from a single piece,

- Prefer to extend or jump *to* the empty square that comes first when scanning the board left-to-right, top-to-bottom.

For each input board, print out the board number, the input board, and the board as it would appear after black's best move (that is, you do not show the results of white's responding moves). Again use the format shown. Put a blank line between input sets. End of file terminates the input.

Example.

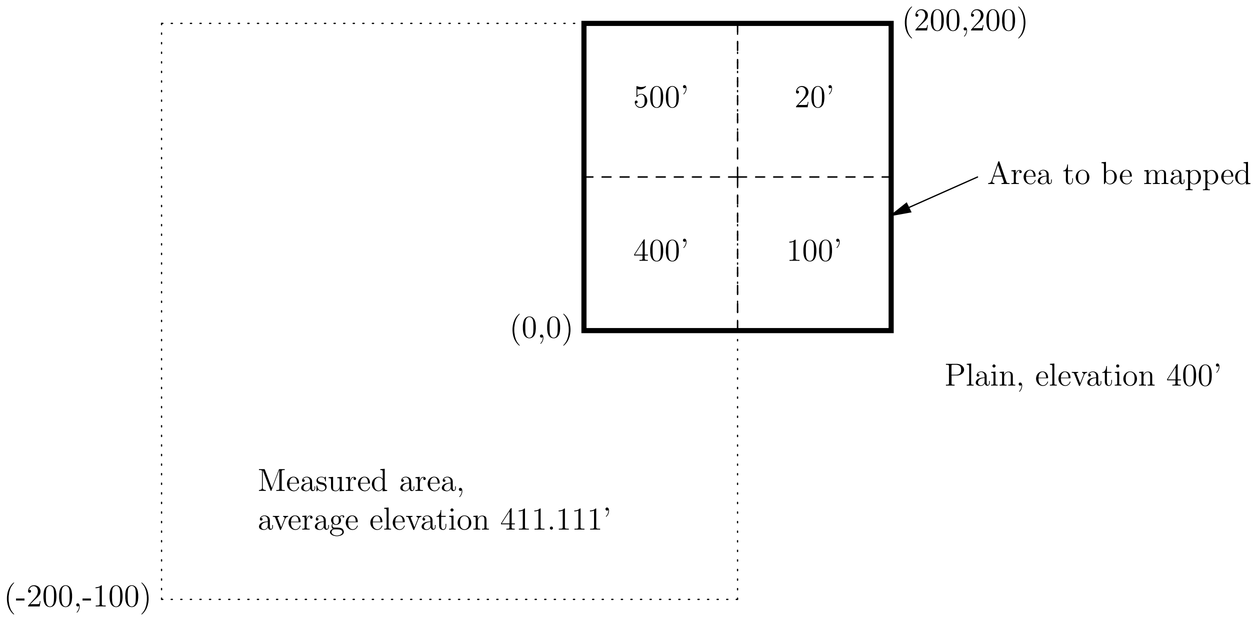
Input

```
bwwwwww ww-wwww wwwwwww wbwwwww
bbwbbbb
wwwbbbb
wwwbbbb

------- ------- ------- ------- -------
w------ wwbb---
```

Output

```
Board 1.
bwwwwww     -bbbwww
ww-wwww     wbbbwww
wwwwwww     wbbbwww
wbwwwww     wbwwwww
bbwbbbb     bbwbbbb
wwwbbbb     wwwbbbb
wwwbbbb     wwwbbbb

Board 2.
-------     -------
-------     -------
-------     -------
-------     -------
-------     -------
w------     bb-----
wwbb---     bbbb---
```

2. A cartographer is trying to find the elevations of points within a rectangular patch of land. The terrain within this patch is very hilly, but the surrounding land is essentially flat, and all at the same elevation. Unfortunately, the cartographer's instruments, while fast, are a little crude. Basically, he has a simple radar device mounted beneath an airplane that can fly over the land, look downward at a 300'-square (that's 300 feet) patch beneath it, and report the average elevation of the area it is looking at. He wants to divide the land into 100'-square patches, and to find the average elevation within each patch, given a large number of average-elevation readings from the plane and the positions at which they were taken. You are to process these data into the information the cartographer wants. So for example, if the territory in question is a $200' \times 200'$ patch consisting of four $100' \times 100'$ squares with the elevations shown below, and the plane measures the $300' \times 300'$ dashed square, it will get an average elevation reading of 411.111', the average of a 500'-high square, a 400'-high square, and seven other squares from the plain, all of which are 400' high.



The input to your program (in free format) consists first of two integers L and B, giving respectively the east-west and north-south span of the hilly rectangle, in feet. Both will be evenly divisible by 100. Next, you will get a series of triples representing readings in the form

$$X \quad Y \quad E$$

where (X, Y) are the coordinates in feet of the southeast corner of a 300×300 square, and E (a floating-point number) is the average elevation in that square. Here, $(0, 0)$ represents the southeast corner of the hilly rectangle, with X increasing to the east, and Y to the north. The pairs (X, Y) will include all those pairs for which $X = 100k_x$ and $Y = 100k_y$ for integers k_x and k_y and $-200 \le X < L$ and $-200 \le Y < B$, plus at least one calibration reading that is entirely within the surrounding plain and does not include any of the rectangle to be mapped (its coordinates will also be divisible by 100). Input ends at the end of file.

The output will consist of the average elevations (to the nearest foot) of each 100×100 square in the hilly region (only) in the format shown below.

Example.

Input

```
200 200
-300 -300 400
-200 -100 411.111
-100 -200 366.667
-200 0 411.111
-100 -100 335.556
0 -200 366.667
-200 100 411.111
-100 0 335.556
0 -100 335.556
100 -200 366.667
-100 100 368.889
0 0 335.556
100 -100 324.444
0 100 368.889
100 0 324.444
100 100 357.778
```

Output

```
500   20
400  100
```

3. [D. Garcia] Two players play the following game. They start with a list containing an even number of integers. Each player in turn removes either the first or last remaining item in the list until all are removed. Each player attempts to get the largest possible sum (think of the numbers as representing numbers of gold coins). Your problem is to find a sequence of moves that would be made if both players played optimally (that is, so as to get the largest possible sum).

For example, given the sequence

$$6, 12, 0, 8$$

the first player's best move is to remove the 8. Regardless of whether the second player chooses 6 or 0, the first player will get the 12, so as to give the first player a total of 20 and the second player 6.

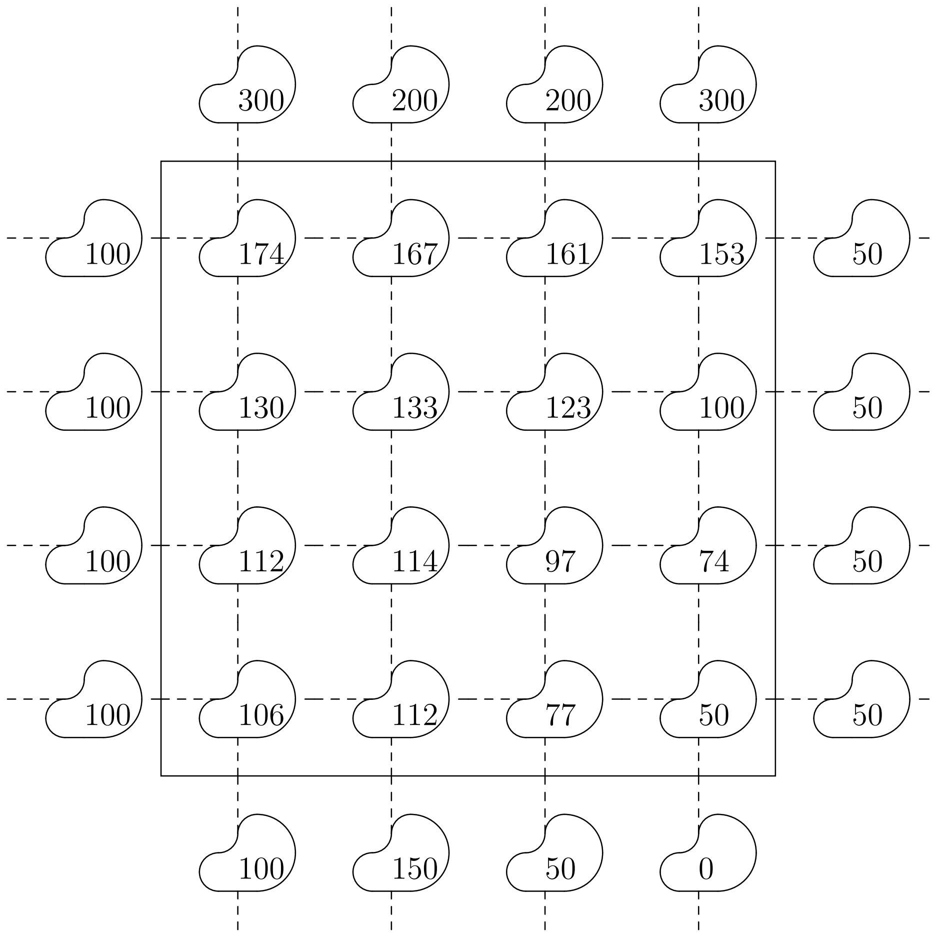
The input to your program will consist of a sequence of data sets in free format. Each set begins with an even integer value $0 \le N < 80$ giving the size of the list, followed by the N non-negative integers in the list (free format). A value of $N = 0$ indicates the end of the input.

The output is an optimal sequence of moves in the format shown in the example below—a string of 'F's and 'L's, where 'F' means "choose the first item," and 'L' means "choose the last item." Begin each set with a sequence number as shown, and separate sets with a blank line. When choosing either end leads to an optimal sequence, prefer choosing the first (therefore, the last choice is always 'F').

Example.

Input	**Output**
`4 6 12 0 8`	`Game 1: LFFF`
`6`	
`0 7 2 3 4 5`	`Game 2: LFFLLF`
`0`	

4. A system of ponds is laid out in a checkerboard pattern. Pipes connect each pond to its four neighbors (east, west, north, south). Water flows through the pipe between two ponds at a rate that is proportional to the difference in the heights of the water in those ponds. The ponds along the outer edge are constantly filled or drained as fast as needed to keep them at constant, pre-set heights. Your problem is to compute an estimate of the heights in all the interior ponds—those not on the edge—at equilibrium (that is, when the total rate at which water flows into each interior pond from its higher neighbors is equal to the rate flowing out to its lower neighbors, so that the pond's height remains constant). (The equilibrium height does not depend on the constant of proportionality that relates differences in heights to rate of flow.)



In the diagram above, the interior comprises the ponds inside the square. The numbers inside the ponds indicate the water's equilibrium height above ground level to the nearest centimeter. The dashed lines are pipes. The unconnected pipes leading from the ponds along the exterior are connected to sources (or sinks) that maintain the external ponds at the heights shown.

The input to your program (in free format) consists first of two integers, W and H, giving the number of ponds in the interior in the east-west and north-south directions, respectively. These are followed by $2(W + H)$ integers giving the constant exterior ponds' heights in centimeters in the following order: W integers giving the heights along the bottom from left to right (west to east), W integers giving the heights along the top from left to right, H integers giving the heights along the left (west) side from bottom to top (south to north), and H integers giving the heights along the right side from bottom to top.

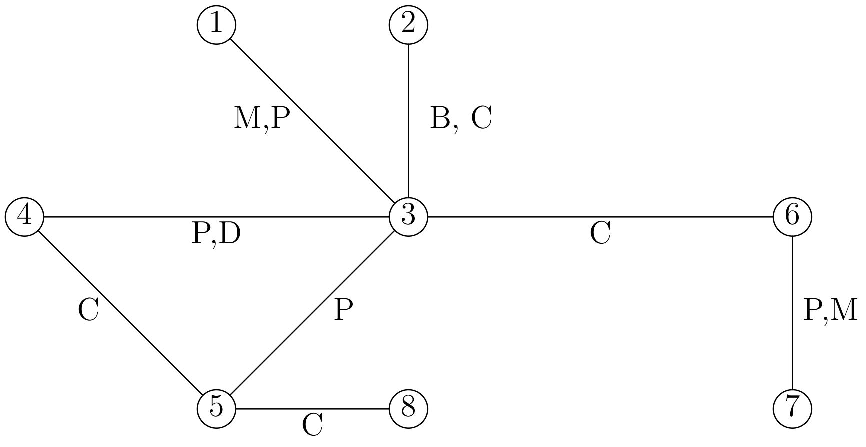
The output should give all the pond heights, including the external ponds, in the format shown in the example. All numbers are to the nearest integer number of centimeters. You may assume that the actual heights are at least 0.05 centimeters away from being halfway between two integers (e.g., we will select the data so that the real height is never something like 95.46, but might be 95.42). We will not be fussy about spacing.

Example.

Input	**Output**
`4 4`	`     300   200   200   300`
`100 150 50 0`	`100  174   167   161   153   50`
`300 200 200 300`	`100  130   133   123   100   50`
`100 100 100 100`	`100  112   114    97    74   50`
`50 50 50 50`	`100  106   112    77    50   50`
	`     100   150    50     0`

5. The city of East Murk is notorious for its haphazardly arranged roads and its complete lack of street signs, a condition that has become a source of perverse pride with its inhabitants. Visitors are forever getting lost and then calling their hosts with questions such as "I'm at the corner of a couple of streets, there's a pastry shop on the street that goes northeast from here, and I got here by traveling west from another corner, passing a coffee shop on the way. Where am I?" They don't necessarily give all the details when they do this (e.g., they might not mention the other streets leading out of the intersection or the other shops they saw). Also, they might get confused as to direction, so that the northeast-running street in the example above might really be running north, and the westbound street leading to it would, in that case, actually be going southwest. This confusion, however, is consistent, in the sense that a visitor who mistakes west for north also mistakes south for west. You have been asked to produce a database system that can determine the possible locations of hapless visitors from this sort of incomplete and inaccurate information.

For example, given the road system below (where 'P' stands for "pastry shop" and 'C' for "coffee shop," 'D' for "car dealership," and 'M' for "movie theater"), the sample query above might be answered "You could be at intersections #3, #4, or #5."



The input (in free format) consists first of an integer, $N > 1$, giving the number of intersections (intersections are identified by numbers ranging from 1 to N). An "intersection," here may be a dead end (if there is only one road coming to it). Next, there follow descriptions of road segments, zero or more, of the form

$$N_1 \;\; N_2 \;\; D \;\; S_1 \; S_2 \cdots S_k \; ;$$

Here,

- N_1 and N_2 are intersection numbers, with $N_1 < N_2$;
- D is a direction, which can be 0 (for north), 45 (for northeast), 90 (for east), etc., up to 315 (for northwest);
- The S_i identify types of shops (single, upper-case letters), and $k \geq 0$.
- All items (including the semicolons) are separated from each other by whitespace.

If there is a road in direction D from N_1 to N_2, that implies that there is also a road from N_2 to N_1 in the opposite direction ($D + 180 \bmod 360$) with the same shops. Since $N_1 < N_2$ in the input, these implied connections are not explicitly listed. The list of connections ends with $N_1 = N_2 = D = 0$ followed by a semicolon, after which come one or more queries. A query has the same form as the first part of the data—a number of intersections, followed by descriptions of road segments. The intersection numbers, of course, will generally have no relationship to the real intersection numbers. The queries are terminated by a single N value of 0. The roads listed in a query will always connect all the intersections in the query; that is, using just the listed roads in any given query, you can always get from any intersection in the query to any other.

Output the possible proper intersection numbers that intersection 1 in the query might actually be, in ascending order of intersection number, in the format shown for the example. Separate the output for one query from another with a blank line.

Example.

Input

```
8
2 3 180 B C ; 1 3 135 M P ;
3 4 270 P D ;
3  6 90 C ;
6 7 180 P M ;
4 5 135 C ;
3 5 225 P ;
5 8 90 C ;
0 0 0 ;
3
1 2 90 C ;
1 3 45 P ;
0 0 0 ;
3
1 2 180 P ;
1 3 90 D ;
0 0 0 ;
0
```

Output

```
Query 1: 3 4 5

Query 2:
```

6. The Java language has a *definite assignment rule,* which guarantees that each local variable is assigned to before its value is first used. The rule guarantees this by enforcing a more stringent condition. For example, consider the following block,

```
x = true ;
if ( x ) {
   y = true ;
}
while ( x ) {
   x = false ;
   z = true ;
}
q = z ;  // ERROR
if ( y ) { // ERROR
   q = x ;
}
```

According to the definite assignment rule, the last two assignments are illegal because neither `z` nor `y` has been "definitely assigned" to before their value is used (in an assignment or test). The rule, in other words, assumes that each **if** test might go either way, and each **while** loop might or might not execute each time around, regardless of what the tests are. It then checks that regardless of which way the tests go, all variables are assigned to before being used. (In the assignment `x = x&&y`, `x` is used on the right-hand side *before* it is assigned to.)

You are to write a program that tests that the definite assignment rule is followed. We use a vastly simplified Java syntax for input:

- No declarations, all variables are single-letter, lower-case, and boolean and are assumed to be already declared and to start out unassigned.
- No comments.
- All adjacent tokens (variables, punctuation marks, keywords, operators) are separated by whitespace.
- All **if** statements have `else`s.
- All expressions are either **true**, **false**, or a sequence of variables separated by `&&` operators (e.g., `x&&y&&z`).
- The bodies of **while** statements, and the two branches of each **if** are blocks (i.e., have `{}` around them).
- The only other kinds of statements besides **if** and **while** are assignments to a simple variable.

Your output will be either the single statement "`OK`" or "`Definite assignment error`."

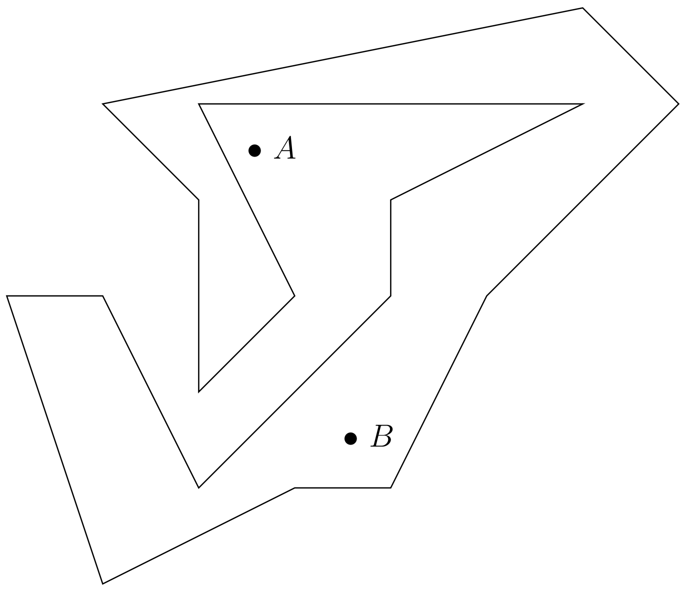
Example 1.

Input	Output
`z = true ;`	`Definite assignment error`
`if ( z ) {`	
`  x = true ; y = false ; r = true ;`	
`} else {`	
`  y = true ;`	
`  if ( z ) { x = false ; } else {  }`	
`}`	
`while ( y ) { q = true ; y = false ; }`	
`if ( z ) { q = y ; }`	
`else { r = r && y ; q = q && x ; }`	

Example 2.

Input	Output
`z = true ;`	`OK`
`if ( z ) {`	
`  x = true ; y = false ; r = true ;`	
`} else {`	
`  y = true ;`	
`  if ( z ) { x = false ; }`	
`  else { x = y && z ; }`	
`}`	
`q = true ;`	
`if ( z ) { q = y ;`	
`} else { r = r && y ; q = q && x ; }`	

7. The infamous prisons on Haunted Simplex Island are all built out of unscalable walls in the shape of immensely elaborate simple (non-self-intersecting) polygons of enormous extent. The outsides of the walls look just like the insides, and the sadistic jailors often delight in placing the prisoner *outside* the jail wall (without telling him, of course), next to an immovable source of water. Not wanting to risk dying of thirst (it is very hot on the island), prisoners will generally stick close to the water rather than exploring to see what side of the wall they are on. If a prisoner *did* know that he was outside, he would probably risk searching for the exit. For example, prisoner A below is actually free, and B is not.



Given a description of the wall and a prisoner's position, determine whether the prisoner is actually free. The input to your program will consist of a pair of integer coordinates P_x and P_y, giving the prisoner's position, an integer, $3 \leq N < 200$, giving the number of walls, and N sets of coordinates, X_i and Y_i, of the corners where adjacent sections of the wall meet, where each coordinate may range from 0 to 100000. The corners are listed in clockwise order around the wall from an arbitrary starting point. There is an implicit wall section between the last corner listed and the first. You may assume the prisoner is not on the wall, that the wall segments do not intersect except at the corners (and that only two wall segments join there), and that (although it is really strong) the wall is infinitely thin.

Print out either "The prisoner is free" or "The prisoner is confined", as shown in the examples on the next page.

Example 1.

Input	Output
175 225	The prisoner is free
17	
100 250 350 300 400 250	
300 150 250 50 200 50	
100 0 50 150 100 150	
150 50 250 150 250 200	
350 250 150 250 200 150	
150 100 150 200	

Example 2.

Input	Output
225 75	The prisoner is confined
17	
100 250 350 300 400 250	
300 150 250 50 200 50	
100 0 50 150 100 150	
150 50	
250 150	
250 200	
350 250 150 250 200 150	
150 100 150 200	

8. The `procmail` program is a UNIX tool in the war against spam. Here, we'll consider a much-simplified variant. Essentially, it is a filter that distributes incoming e-mail messages into files (including `/dev/null`, the UNIX bit bucket, and the only appropriate file for spam), as determined by a configuration file. For this problem, we use a vastly simplified version of this configuration file. Here is an example:

```
:0
*Content-Type:.*html
*Precedence: bulk
/dev/null

:0
*.*PRCS
$HOME/Mail/PRCS
```

Each group of lines beginning with a line ":0" starts a *rule.* The one or more lines beginning with '`*`' that follow are *patterns.* If each of them matches some line in the message being filtered (not necessarily the same line), then the message is directed to the file named by the line that follows that last pattern of the rule. If more than one rule applies to a message, the first rule is used. If no rule applies, the message is directed to a special file called `$DEFAULT`. Any blank lines among the rules are ignored.

A pattern matches a line if, after stripping off the leading asterisk, the pattern matches some prefix of the line, given that '`.`' matches any character, '`.*`' matches any string of 0 or more characters, and all other characters (including `*`, if it does not follow a period) match only themselves. For example, the pattern line

```
    *Content-Type:.*html
```

will match either of the lines

```
    Content-Type:html, charset=us-ascii
    Content-Type: text/html, charset=us-ascii
```

but not

```
    The Content-Type field contains "html"
```

since what it does match is not a prefix of the line.

The input to your program will consist of sequence of rules like these, followed by a line

```
    ::
```

and then by a single message. The output should be the name of the file to which the message should be directed, as shown in the example on the next page.

Example:

Input

```
:0
*Content-Type:.*html
*Precedence: bulk
/dev/null

:0
*.*PRCS
$HOME/Mail/PRCS
::
Date: Tue, 2 Oct 2001 17:02:03 -0700 (PDT)
From: John Doe <doe2048@hotmail.com>
To: hilfingr@syracuse.mckusick.com
Subject: Source code control?
Content-Type: text

Hello,

I understand you have something to do with
a program for doing source-code control
called PRCS.  What can you tell me about
it?

J. Doe
```

Output

```
$HOME/Mail/PRCS
```