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

Programming Contest **P. N. Hilfinger**
Fall 2006

2006 Programming Problems (revised)[*]

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

```
source ~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 16 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). Do not make class `P`*N* public, or the Java compiler will complain. 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`, `sstream`, `fstream`, `map`, and `algorithms`. Likewise, you can use the

[*]I have modified these problems slightly since the contest to correct errors in problem statements or examples.

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`, `java.math`, and `java.util` and their subpackages. 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 or their Java-library equivalents). 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 compiling or running it.

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 (for any language) 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.

In the actual ACM contests, you will not be given nearly as much information about errors in your submissions as you receive here. Indeed, it may occur to you to simply take the results you get back from our automated judge and rewrite your program to print them out verbatim when your program receives the corresponding input. Be warned that I will feel free to fail any submission in which I find this sort of hanky-panky going on (retroactively, if need be).

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 -Iour-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 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 (typically `~ctest/include`) 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 C/C++ programmers are free to use `scanf` to read in numbers and strings and `gets` to read in lines.

Terminology. The terms *free format* and *free-format input* indicate 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/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. [Misha Dynin] Your goal is to implement a spelling checker. You should correct words by finding words in the dictionary that are no more than two edits away from the input. Here, an *edit* is either:

- Inserting a single letter, or
- Deleting a single letter

with the restriction that

- If the edits are both insertions or both deletions, they may not be of adjacent characters.

The input will consist of a dictionary followed by a sequence of possibly misspelt words. Both contain words (strings of letters) of up to 50 characters. The dictionary, in free format, is followed by a line containing just the string '`===`'. After this, there will be zero or more *text lines* containing words followed again by a line containing the string '`===`'. The input is case-insensitive; print corrections from the dictionary in the case they appear in the dictionary and unchanged words from the text lines in their original case as well.

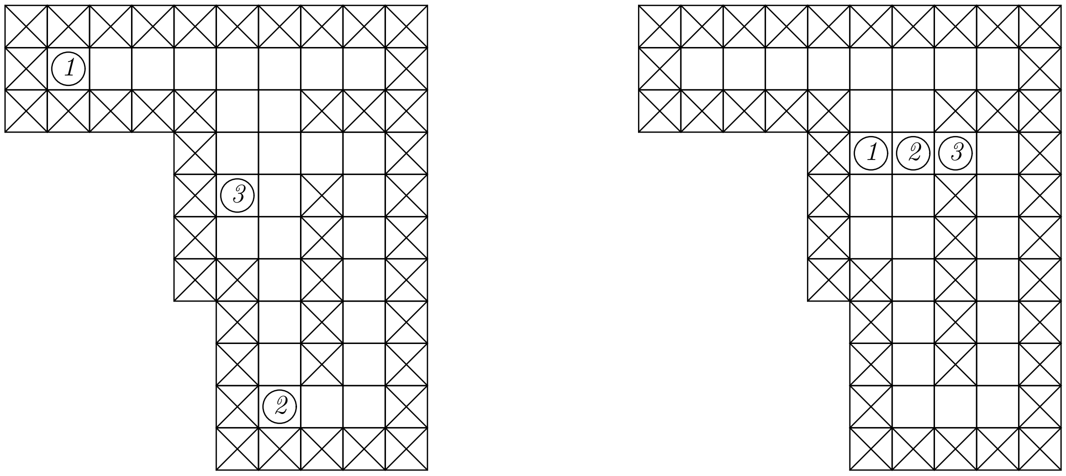
As output, you should print the text lines with whitespace intact, with the following changes on each word, W:

- If W is in the dictionary, print it as is.
- Otherwise, if W is not in the dictionary,
  - If no corrections can be found, print "$\{W?\}$".
  - Ignore any corrections that require two edits if there is at least one that requires only one edit; then ...
  - If exactly one correction is left, print that word.
  - If more than one possible correction is left, print the set of corrections as $\{W_1\ \ W_2 \cdots\}$, in the order they appear in the dictionary.

Example.

Input	Output
`rain spain plain plaint pain main mainly`	`the {rame?} in pain falls`
`the in on fall falls his was`	`{main mainly}    on the plain`
`===`	`was {hints?} plaint`
`hte rame in pain fells`	
`mainy    oon teh lain`	
`was hints pliant`	
`===`	

2. The Marbles Puzzle Game involves moving a set of marbles around a restricted set of paths so as to achieve a certain configuration. A typical puzzle instance looks like this:



The problem is to move the marbles (labeled 1, 2, and 3 for this instance) around the grid so as to convert the initial configuration on the left into the position on the right. A move consists of moving one marble in one of four possible directions (north, south, west, or east) as far as it will go without entering a square that is occupied. A square may be occupied either by another marble or by a section of wall (shown as a square with an 'X' in it). The piece must move as far as possible each time; it may not stop while there is still an unoccupied square in the direction of motion. For example, the puzzle above may be solved with the sequence of moves

```
2 north, 1 east, 1 south, 3 east, 2 south, 3 south, 3 east, 3 north, 3 west
```

The input to your program will consist of a sequence of problems. Each problem begins with four integers, W, H, M, and N in free format, where $3 \leq W < 16$, $3 \leq H < 16$, $0 < M \leq 4$, and $0 \leq N \leq 10$. Next come H lines of W characters each, giving the initial configuration (northernmost line first) Each character in these lines represents the contents of a grid cell. A hyphen ('-') indicates an unoccupied cell, and an 'X' indicates a wall. Next come $2M$ pairs of integers $0 \leq x_i < W$ and $0 \leq y_i < H$ giving the starting positions of M marbles, $1 \leq i \leq M$. The coordinates run from the top left, at which $x = y = 0$, with x running left to right and y running from top to bottom. The first M pairs give the initial positions of the marbles labeled $1, \ldots, M$ respectively and the second M pairs give the final positions of these same marbles. You may assume that there are enough wall squares to prevent any marble from moving off the grid. The last problem in the input will be followed by four integer 0's.

For each problem in the input, you are to output either a single line giving the smallest number of moves needed to solve the puzzle, or saying that no solution is possible in $\leq N$ moves. Use the format shown in the examples.

Example.

Input

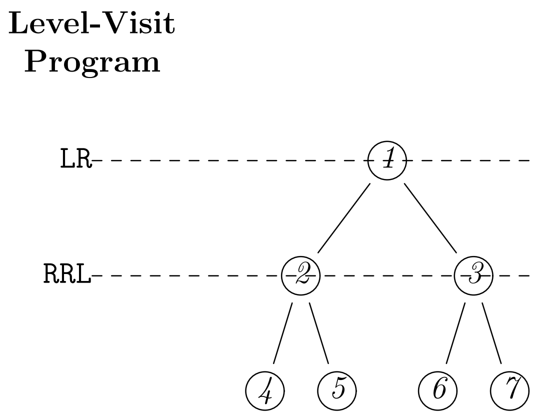
```
10 11 3 12
XXXXXXXXXX
X--------X
XXXXX--XXX
----X----X
----X--X-X
----X--X-X
----XX-X-X
-----X-X-X
-----X-X-X
-----X---X
-----XXXXX
1 1 6 9 5 4
5 3 6 3 7 3
10 11 3 3
XXXXXXXXXX
X--------X
XXXXX--XXX
----X----X
----X--X-X
----X--X-X
----XX-X-X
-----X-X-X
-----X-X-X
-----X---X
-----XXXXX
1 1 6 9 5 4
5 3 6 3 7 3
0 0 0 0
```

Output

```
Puzzle 1. Solvable in 9 steps
Puzzle 2. Not solvable within 3 steps
```

3. [Miguel Revilla, adapted] A number of balls are dropped one by one from the root of a full binary tree (in which all nodes have two children or no children and all leaves are at the same level). Each time a ball visits an internal node, it moves down either to the left child or to the right, depending on the number of balls that have visited that node, and continues until it reaches a leaf. The internal nodes at each level above the bottom of the tree share the same *level-visit program,* represented as a non-empty string from the alphabet $\{L, R\}$. The program `LRLLR`, for example, means that the first ball to visit a particular node goes to the left (L) child, the second goes to the right (R), the third and fourth to the left, and the fifth to the right, after which the pattern repeats. (The program is the same for all nodes on one level, but each individual node executes its copy independently, so what counts is the number of balls that have visited that particular node.)

For example, given the tree below, where the visit program is `LR`, at the top level and `RRL` on the second, ball #1 travels from node $1 \Rightarrow 2 \Rightarrow 5$, #2 from $1 \Rightarrow 3 \Rightarrow 7$, #3 from $1 \Rightarrow 2 \Rightarrow 5$, #4 from $1 \Rightarrow 3 \Rightarrow 7$, #5 from $1 \Rightarrow 2 \Rightarrow 4$, #6 from $1 \Rightarrow 3 \Rightarrow 6$, #7 from $1 \Rightarrow 2 \Rightarrow 5$, etc.



The problem is determine where (at what leaf) a particular ball will end up.

Your input will consist of a number of test cases in free format. Each case starts with two integers $0 < H < 30$ and $0 < K < 2^{62}$, indicating that we are dealing with a tree with height (longest distance to the root) H and want the final destination of ball number K. In the sample tree above, $H = 2$. Next come H strings representing the level-visit programs for the root node, the level below the root, and so forth. Each string is composed only of capital letters 'L' and 'R'. The last set of inputs is followed by two integer 0s.

The output consists of one line for each case giving the number of the leaf node at which ball K^{th} ball ends up (for the first ball, $K = 1$). Nodes are numbered in level order as shown in the example so that the first leaf node is numbered 2^H. Use the format shown in the example below. K can be quite large; you will need to use Java type `long` or G++ type `long long int` to represent it.

Example.

Input	Output

`2 8 LR RRL`
`10 10000000000 L L L L L L L L L L`
`0 0`
	`Case 1. Ball 8 ends at node 7`
	`Case 2. Ball 10000000000 ends at node 1024`

4. [Misha Dynin] Given a set of n strings $s_1, \ldots, s_n$, a *longest common subsequence* of the strings is a string s that is a subsequence of each s_i individually such that no longer string has the same property. We say that s is a subsequence of s_i if s_i can be transformed into s by removing characters. Thus, a longest common subsequence of "aboveboard", "beauregard," and "fiberboard" is "beard." Given a set of strings, you are to compute the length of the longest common subsequence.

The input will consist of sets of strings. Each set consists of a sequence of one or more non-empty strings, one per line, followed by one empty line (containing no characters except for the newline at the end). The strings do not include the newlines at the end of each. The input is terminated by the end of file. You may assume that there are at most five strings, and that the product of their lengths is less than 1,000,000.

For each set, output a report of the length of the longest common subsequence in the format shown in the example.

Example.

Input

```
aboveboard
beauregard
fiberboard

apple
bind
cumulus
hog

```

Output

```
Set 1. Length of longest common subsequence = 5
Set 2. Length of longest common subsequence = 0
```

5. You are in the process of building a proof checker for a logical language with quantifiers. This language is fully parenthesized (all operator expressions are surrounded in parentheses), so that substituting any well-formed formula (wff), F, for a variable in any other wff does not change the grouping (and therefore the meaning) of F. Saying that "Expression E is balanced for parentheses" means that it has equal numbers of left and right parentheses (possibly 0) and that reading E from left to right, there is never a surplus of right parentheses.

At the moment you are helping to write the part of the checker that determines whether a given wff is an instance of one of the following *axiom schemata* (basically, patterns):

1. `((A`V`.`Z`)->`$Z_{V=T}$`)`
2. `(`$Z_{V=T}$`->(E`V`.`Z`))`
3. `((A`V`.(`W`->`Z`))->(`W`->(A`V`.`Z`)))`
4. `((A`V`.(`Z`->`W`))->((E`V`.`Z`)->`W`))`

The notation `(A`V`.`Q`)` is supposed to mean "For all V, Q is true" and `(E`V`.`Q`)` is supposed to mean "There is a V for which Q is true." We say that one of the schemata *matches* a wff F if substituting some variable for V and some wffs for Z, T, and W (subject to constraints below) makes the schema identical to F. All variables are identifiers having the form `v`N, where N is a decimal numeral; the letter `v` is used only for variable names. The period is used only in the `(A`V`.`Q`)` and `(E`V`.`Q`)` constructs. The substring `->` is used only as an operator in fully parenthesized expressions as shown.

We say that an occurrence of a variable, V, in a wff is *free* if it is not inside a construct of the form `(A`V`.`E`)` or `(E`V`.`E`)` (that is, not inside a quantifier that *binds* that same variable V). For example, the first appearance of `v1` in `(P(v1)|(Av1.Q(v1)))` is free and the rest are not. In the axiom schemata above, the variable V must not occur free in W, and the notation $Z_{V=T}$ means "the formula Z with all free occurrences of V replaced by T."

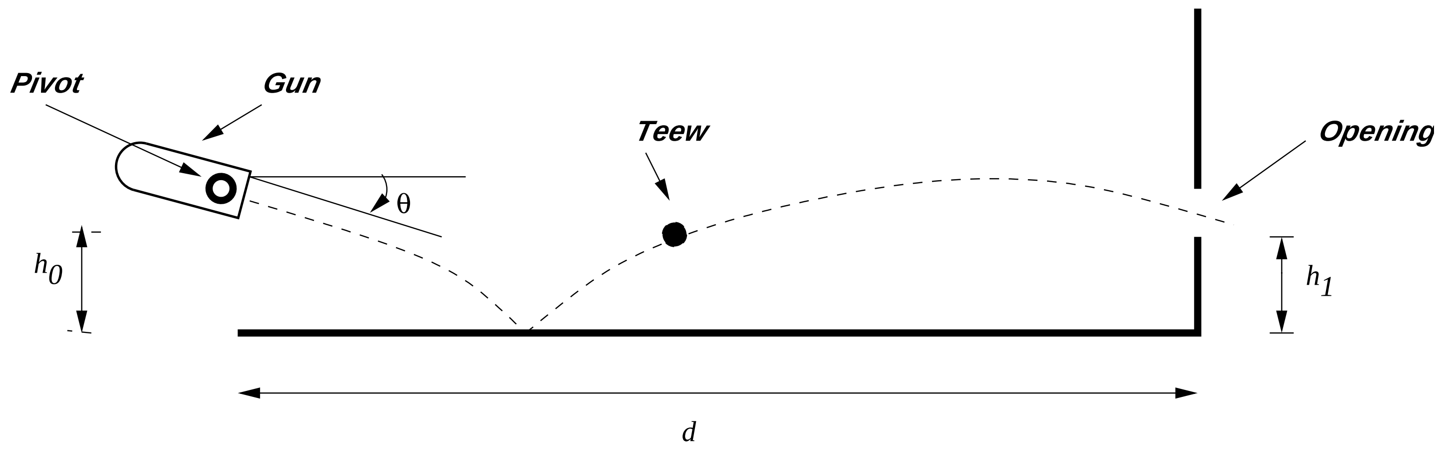
The input to your program will consist of a sequence of wffs in free form. Assume that each wff contains no embedded spaces.

The output will consist of a sequence of lines that tell which schema, if any, matches that wff. Use the format shown in the examples.

Example.

Input	Output
`((Av1.(v1>v2))->(f(a,v1)>v2))`	`Formula 1. Matches schema 1`
`((Av2.(v1>v2))->(f(a,v1)>v2))`	`Formula 2. No match`
`((Av1.(f(a,v2)->g(v1,c)))->(f(a,v2)->(Av1.g(v1,c))))`	`Formula 3. Matches schema 3`
`((Av1.(f(a,v1)->g(v1,c)))->(f(a,v1)->(Av1.g(v1,c))))`	`Formula 4. No match`

6. A space-faring race known as the Gliqx play a game known as Bounce the Teew, in which one player adjusts the local force of gravity and the velocity with which a small gun ejects a teew (a small ball, actually) and challenges his opponent to aim the gun correctly so as to make the teew go through a teew-size opening in the opposite wall. For this problem, we'll simplify to two dimensions, as shown in the illustration below:



The opponent chooses the angle θ (measured in degrees counterclockwise from the horizontal, so that an angle of 45.0 is pointing up and to the right and 0.0 is pointing horizontally to the right). The gun's pivot allows it to assume any angle between -80 to 80 degrees. The teew must bounce exactly once and go through the opening. For the purposes of this problem, we'll count the shot as successful if the bottom edge of the teew is at height $h_1 \pm 0.01$ at the point that its center is at horizontal distance d from the starting point (treat the wall as if the teew could pass right through it; we are interested only in where the bottom goes, not about other parts of the teew that might hit the wall). The teew starts with its bottom edge at a height of h_0 and at some total velocity v. When it bounces, the teew bounces ideally: it retains all its horizontal velocity, and its vertical velocity simply changes sign. For those of you who have forgotten your physics, if at $t = 0$ the vertical position of an object is y_0 and its vertical velocity of $v_0^{(y)}$ (positive is up), then its vertical position at each later time $t > 0$ is given by $y(t) = y_0 + t \cdot v_0^{(y)} - g \cdot t^2/2$, and its velocity is $v^{(y)}(t) = v_0^{(y)} + g \cdot t$, where g is the acceleration of gravity. This formula, of course, applies only between bounces.

The input to your program will consist of multiple problems in free form. Each problem is specified by 5 positive floating-point numbers, $h0$, h_1, d, v_0, and g, giving the heights of the gun and bottom edge of the opening, the length of the room, the initial velocity of the ball (total velocity, not just vertical), and the acceleration of gravity, all in some compatible units. (Never mind exactly what those units are; they're Gliqxian after all, and not terribly familiar to us). The last input set will be followed by 5 constant 0's.

Print your the values of $-80 < \theta < 80$ rounded to the nearest integer degree in the format shown in the examples below. When more than one solution exists, display the largest (i.e., where the gun is most elevated). Assume in all cases that there is a solution (setting g and v_0 so that the opponent can't hit the opening in ≤ 1 bounce is considered cheating).

Example.

Input	Output
`10.0 10.0 20.0 30.0 10.0`	`Shot 1. Angle is -42`
`10.0 5.0 20.0 30.0 10.0`	`Shot 2. Angle is -32`
`10.0 10.0 20.0 20.0 10.0`	`Shot 3. Angle is 78`
`0 0 0 0 0`	

7. [Miguel Revilla, adapted] Every now and then, the management at MakeWork, Inc. reshuffles the inhabitants of the offices in its huge complex. The problem, I guess, is that some offices are better than average, and some worse, and so they try to rotate people through them to reduce envy and strife. Unfortunately, they have only one office-moving crew, and that crew can pack up only one office at a time, which they can then move to a vacant office (whose former inhabitants, if any, must already have been moved out). The process is long and arduous, therefore. Fortunately, there is always at least one vacant office; otherwise, the task would be impossible with one crew. Even as it is, some people may have to move twice (once to a vacant office and then to their final destination). Your problem is to schedule a minimal sequence of moves, given a description of the current office occupants and the new offices to which they are moving.

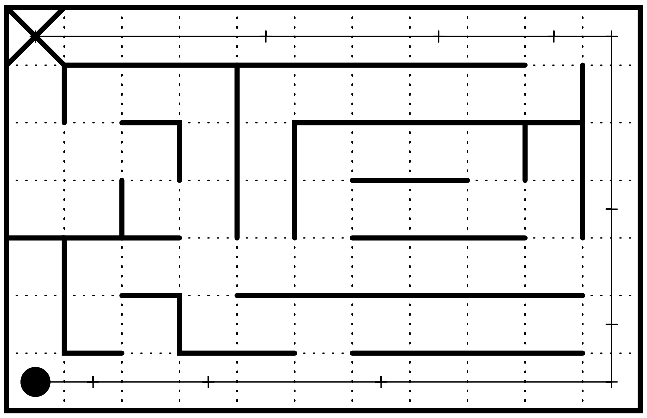
The input consists of a number of problem sets in free format. Each set begins with two integers, M and N with $0 < N < M$, giving the total number of offices and the number of occupied offices, respectively. Offices are numbered 1–M. There will then follow a list of N pairs of integers, F_i and T_i, all in the range $[1..M]$. Each pair indicates that the occupants of office F_i must end up in office T_i. It is possible for $F_i = T_i$, which means that the occupants of F_i do not move. The last set of inputs is followed by two integer 0s.

The output should be a minimal schedule (i.e., having fewest possible moves) that, if performed in the sequence listed, will put everyone in the desired offices. You may assume that such a schedule exists in each case. Use the format shown in the example below. Each move must be from an occupied office to an empty office (empty either because it was unoccupied before the moving began or because a previous move vacated it).

Example.

Input	Output
`4 3`	`Set 1. 1->2, 4->1`
`1 2 4 1 3 3`	`Set 2. No moves`
	`Set 3. 3->4, 1->3, 2->1, 3->2`
`3 2`	
`1`	
`1`	
`3 3`	
`4 3`	
`1 2 2 1 3 4`	
`0 0`	

8. [Misha Dynin] You've probably seen those problems that ask you to find your way out of a maze. In this variation, you are equipped with a scooter that is capable of modest acceleration. The maze is laid out on a grid, as shown in the illustration below. On each move, you may travel north, south, east, or west. On any given move in which you move in the same direction as the previous move, you may move $\leq k+1$ squares if you moved k squares on the previous move. On other moves, you may move one square. The problem is to determine the minimum number of moves from the starting position to the designated exit square. In the example, the minimum number is 11, as indicated by the small crosses along the solution path shown (the black circle marks the start and the X marks the exit).



The input to your program will consist of a sequence of problems in free format. Each starts with integers $1 < W$, $1 < H$, $0 \leq X_s < W$, $0 \leq Y_s < H$, $0 \leq X_e < W$, and $0 \leq Y_e < H$, respectively giving the width and height of the grid in cells, the starting position—where (0,0) is the upper-left grid cell—and the exit's position. Next come a sequence of H strings of $2W$ characters each, giving the walls to the left and below each grid cell, top row to bottom row. Each cell is represented by two characters. The first is either vertical bar '`|`' (vertical wall on the west) or '.' (no vertical wall), and the second is either underscore '`_`' (horizontal wall on the south) or '.' (no horizontal wall). The leftmost character in each row will be `|`, since there is a wall to the left of the grid, and the second characters along the south (last) row of cells will all be '`_`'. The east and north walls are implicit and not represented. The last problem in the input will be followed by 6 integer 0s.

The output will consist of a single line giving the problem number and either a message giving the number of steps needed, or the message saying "`No escape possible`," in the format shown in the example.

Example.

Input	Output

`11 7 0 6 0 0`
`|.._._._._._._._....`
`|.|.._..|.._._._._._|.`
`|.....|.|.|.._._..|.|.`
`|_._|_..|.|.._._._..|.`
`|.|.._..._._._._._._..`
`|.|_..|_._..._._._._..`
`|_._._._._._._._._._._`
`3 2 0 1 0 0`
`|_._|.`
`|_._._`
`0 0 0 0 0 0`
	`Maze 1. 11 steps to the exit`
	`Maze 2. No escape possible`