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

Programming Contest **P. N. Hilfinger**
Fall 1994

Programming Problems

To set up your account, execute

```
source ˜ctest/contest/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.)

You have 5 hours in which to solve as many of the attached eight problems as possible. Put each complete solution into a single N`.c` file (for C) or N`.C` file (for C++), where N is the number of the problem. Each program must reside entirely in a single file. Each file should start with the line

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

and must contain no other `#include` directives. Upon completion, each program must terminate by calling `exit(0)`.

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. The standard system libraries do *not* include the `gcc` class library. 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` or N`.C`. 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. 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 2> junk-file
```

The output of each input file is then compared with a standard output file. 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.* Make sure that the last line of output ends with a newline. Your program must not send any output to `stderr`; the temporary file *junk-file* must be empty at the end of execution. Each test is subject to a time limit of about 15 seconds. You will be advised by mail whether your submissions pass.

The command `contest-gcc [-g]` N, where N is the number of a problem, is available to you for developing and testing your solutions (as usual, the optional `-g` is for debugging information). It is equivalent to

```
gcc -Wall -o N -O [-g] -Iour-includes N.[cC] our-library -lg++ -lm
```

The *our-includes* directory contains `contest.h`, which also supplies the standard header files. The *our-library* file contains an additional runtime library routine that you may use, which are described below.

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.

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).

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.

Additional library routine. The following routine is available for you to call.

```
    extern int linSolve(const double A[], const double b[], double x[],
                        int N, int C);
        /* Treats A as an N-by-N matrix, and b and x as N-element column */
        /* vectors. Normally sets x to the solution of Ax=b and returns 0. */
        /* If any 0 pivots are encountered (indicating a singular system) */
        /* returns 1 and leaves x undefined (in any case, A and b are */
        /* unchanged).  The vector x may not overlap A or b. */
        /*    The parameter C>=N controls where the rows of A are assumed */
        /* to be stored.  The kth row of A (0<=k<N) is stored in elements */
        /* A[kC] through A[kC + N - 1].  This allows one to declare the */
        /* actual parameter for A as, e.g., */
        /*    double Q[100][100]; */
        /* and then store a 10x10 matrix in the first 10 elements of the */
        /* first 10 rows of Q (so N=10, C=100), or as */
        /*    double R[200]; */
        /* and store a 10x10 matrix in the first 100 elements of R (so */
        /* N=10, C=10). */
```

1. The game `xrobots` is played on a rectangular grid of squares. At any time, each square is either unoccupied, occupied by a robot, occupied by the human player, or occupied by a pile of robotic debris. On each turn, the human player may either stay put or move to an adjacent square in any direction, as long as that square is unoccupied and is not adjacent to a square containing a robot. After the player moves, each robot moves to the square adjacent to it that minimizes that robot's horizontal and vertical distance to the player. If multiple robots move to the same square, or if a robot moves to a square containing debris, then all robots moving to that square are destroyed, leaving debris.

These rules make it possible for there to be no square to which a player can legally move in his turn (including the square he is on). When this happens, the player can teleport to a random unoccupied square, losing if he lands next to a robot. To maximize the chances of winning, you hit upon the "greedy" strategy of trying to move so as to maximize the fraction of resulting unoccupied squares that are *not* next to any robot (after the robots move). In this problem, you are to read in a board containing a configuration in which it is the player's turn to move. You must determine the move ("stay put" is a move) that implements this strategy, and display the board that results after you and then the robots move.

The input consists of a sequence of configurations. Each configuration begins with two integers in free form, giving the width, W, and height, H, of a board (in number of squares). These are followed by H strings, each W characters long, separated by white space. Each string gives the contents of the squares on one row of the board, starting with the top ("northernmost") row. A '`-`' character denotes an unoccupied square; '`P`' denotes the player (there is exactly one per board); '`R`' denotes a robot; and '`D`' denotes debris. There will always be at least one unoccupied square.

The output should consist of a sequence of reports displaying the number of the configuration (0 for the first, 1 for the second, etc.), echoing the board, and then showing the board after an optimal move by the player. If the player has no move, the program should print

```
The player has no move.
```

in lieu of showing this second board. When more than one move is optimal, the player should prefer first to stay put (if that is an optimal move), then to move north, then to move northeast, and so forth clockwise around the compass. Put a blank line after each configuration report, as shown in the example.

For example, given the following inputs, the desired outputs are as shown.

Input

```
6 5
--RR--
R----R
R-P--R
----D-
-RR--R

10 3
----R-R-R-
---P-R-R--
----------

4 3
R--R
--P-
R--R

8 5
--------
--------
-P---RRR
--------
--------
```

Output

```
Configuration 0.
--RR--
R----R
R-P--R
----D-
-RR--R

Best result:
------
--D---
-DP-D-
--D-D-
------

Configuration 1.
----R-R-R-
---P-R-R--
----------

Best result:
--PRRRRR--
----------
----------

Configuration 2.

R--R
--P-
R--R

The player has no move.
```

More Output

```
Configuration 3.

--------
--------
-P---RRR
--------
--------

Best result:
--------
--------
-P--RRR-
--------
--------
```

2. This problem concerns a restricted form of the *predicate calculus* (logic with "for all" and "there exists"). In this restricted language, formulas are written in postfix form: operands first, followed by an operator. Each formula is a string of

- *Variables*, written as lower-case letters a–z.
- The *relational operators* '$>$' (greater than), '$<$' (less than), and '`=`' (equals).
- The *logical operators* '&' (and), '`|`' (or), '_' (implies), and '`~`' (not).
- The *constants*, optionally signed integer numerals.

Variables are implicitly *universally quantified*; that is, a formula is true iff it is true for all values of any variables in it.

For example the formula

```
i j > j i < _
```

denotes the true assertion "$i > j$ implies $j < i$ for all i and j." (In logic, A implies B iff A is false or B is true). The relational operators must be applied to variables and integer constants only; they yield true or false. The logical operators must be applied to logical values only. The formula must yield a logical value. You may assume all formulas obey this rule.

You are to write a program to read formulas and print whether each is true or false. Each formula is on a separate line, preceded by two integer constants, L and U, which indicate the lower and upper bounds over which the variables may range. You may assume $L \leq U$. The formula follows L and U, with its individual operators and operands separated by one or more blanks.

The output should echo the input and then say whether the formula is true, in the format shown in the examples. Sample input and output follows.

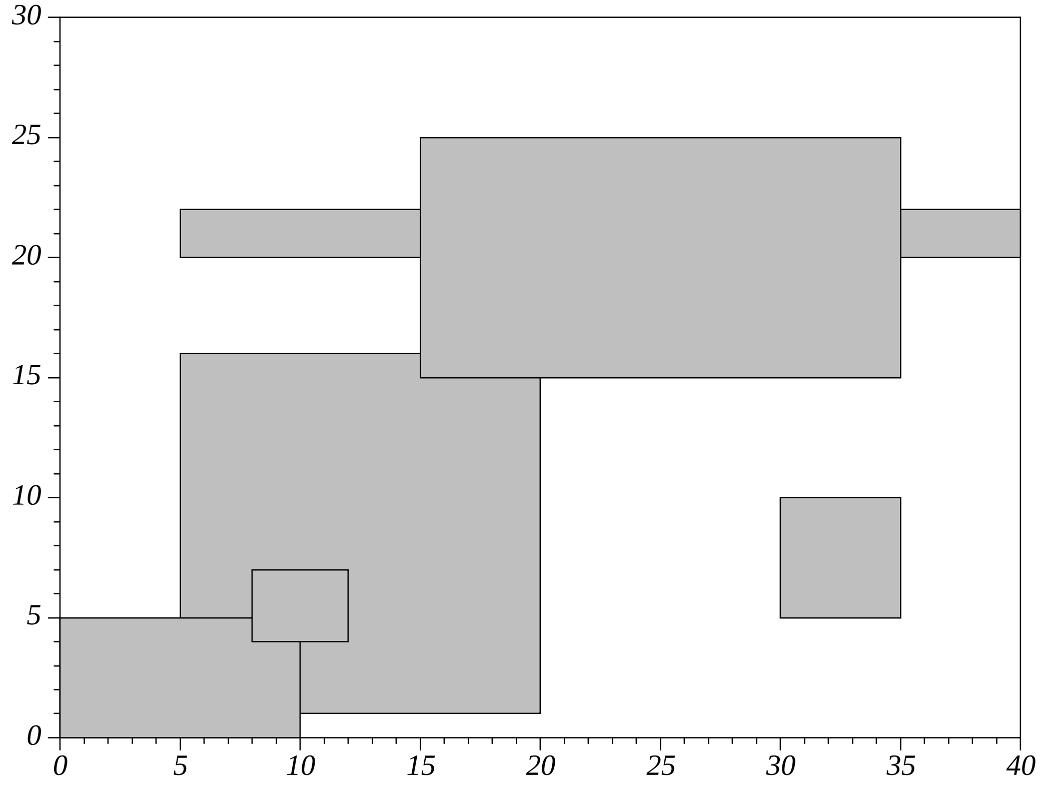
INPUT:

```
-1 1 i j > k j < & k i < _
 0 1 i j = ~ k i = ~ k j = ~ & & 0 1 = _
 0 2 i j = ~ k i = ~ k j = ~ & & 0 1 = _
```

OUTPUT:

```
i j > k j < & k i < _ is true in [-1..1]
i j = ~ k i = ~ k j = ~ & & 0 1 = _ is true in [0..1]
i j = ~ k i = ~ k j = ~ & & 0 1 = _ is false in [0..2]
```

3. A set of rectangular tiles are laid upon a rectangular floor, all with one edge parallel to one of the walls. The tiles may overlap. You are to write a program that determines how much of the floor is covered, given the sizes and positions of the rectangles. The problem, naturally, is in making sure that you count the areas where tiles overlap only once. For example, in the figure below, six tiles cover 505 square units of area, while the total area of the tiles is 582 square units.



The input to your program will consist of a sequence of sets of data in free format. Each set consists of two positive integers L and W, respectively giving the length and width of the floor, followed by quadruples of non-negative integers L_i, W_i, x_i, and y_i, each giving the length and width (L_i, W_i) of a rectangle, and the position of its lower-left corner (x_i, y_i). These numbers satisfy the following constraints.

$$0 \le x_i \le L - L_i, \;\; 0 \le y_i \le W - W_i, \;\; L_i, W_i > 0.$$

A quadruple of all 0's marks the end of a set of data.

For each set of data, the output is to consist of a single message of the form

“`Set` k `covers` M `square units`”

where k is the number of this set of data (numbering from 0), and M is the amount of floor covered. Sample input and the resulting output follow.

Input

```
40 30
15 15 5 1 35 2 5 20
10 5 0 0
4  3 8 4
20 10 15 15
5 5 30 5
0 0 0 0

10 10
5 4 0 0
5 6 5 0
5 4 0 0
5 6 5 0
0 0 0 0
```

Output

```
Set 0 covers 505 square units
Set 1 covers 50 square units
```

4. The set of *merges* of two strings of length N is the set of all strings whose ith character for all $0 \leq i < N$ is the ith character of one or the other string. You are to write a program that reads in pairs of strings of equal length and lists the elements of the set of merges, without duplicates.

The input will consist of a sequence of strings of non-whitespace, printable characters, separated by whitespace. You may assume the strings are at most 128 characters long. The output is to echo each two strings and then list the members of the set of merges in lexicographic order using the standard ASCII character ordering, without duplicates, in the format shown following.

Input

```
lust
left
graft brain
```

Output

```

Set 0: "lust" and "left"
left
lest
luft
lust

Set 1: "graft" and "brain"
brafn
braft
brain
brait
grafn
graft
grain
grait
```

5. [Due to E. W. Dijkstra] Consider decimal numerals containing only the digits 1–3. A numeral is considered “good” if no two adjacent non-empty substrings of it are equal; otherwise it is “bad.” Hence, the numerals ‘1’, ‘12’, and ‘1213’ are good, while ‘11’, ‘32121’, and ‘121321312’ are bad.

You are to write a program that, given $n > 0$, finds the smallest good n-digit numeral. The input consists of a sequence of positive integers. For each of these integers, n, the output is to contain a line of the form

`The smallest good numeral of length` n `is` s`.`

where s is the answer. For example,

Input	Output
`1 4`	`The smallest good numeral of length 1 is 1.`
`7`	`The smallest good numeral of length 4 is 1213.`
`9`	`The smallest good numeral of length 7 is 1213121.`
	`The smallest good numeral of length 9 is 121312313.`

6. [Due to Geoff Pike] An ant wanders about an anthill. At each room, it chooses randomly to which room it will wander next (including back to the one it just came from). Given the room at which it starts, we can ask how many rooms it will visit (on average) before reaching another given room.

The input for this problem will consist of sets of input in free format. Each set consists of a positive number N, the number of rooms, followed by a list of connections between rooms. Each connection is denoted by a pair of positive numbers between 1 and N, identifying the rooms at the ends of that connection. A pair '0 0' ends the list. There may be self-loops—connections that proceed from an room back to that room.

Room 1 is the starting point for the ant, and room N is the ending point. The ant follows connections from room to room, choosing among the possible connections to take next room with equal probability (the connection it just took being one of the possibilities), until it (first) reaches room N. The length of its path from start to finish is the number of connections it follows. (As a result of these definitions, the average length of the ant's path when $N = 1$ is 0.) You may assume that N is reachable from all other rooms.

For each set of input, your program is to print out the number of the set (starting at 0) followed by a message of the form

```
    Average path length = ddd.dd
```

(that is, display the result rounded to two decimal places). Use the format given in the following examples.

Input

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

Output

```
Set 0.
    Average path length = 1.00

Set 1.
    Average path length = 2.00

Set 2.
    Average path length = 4.00

Set 3.
    Average path length = 0.00

Set 4.
    Average path length = 0.00

Set 5.
    Average path length = 7.00
```

7. There are numerous applications these days for modular arithmetic involving very large integers. In this problem, we consider one particular modular arithmetic operation—the calculation of the remainder of a divided by M for $a \geq 0$ an arbitrarily large integer and $0 < M < 2^{32}$.

The input will consist of sets of two positive integer numerals (a and M in that order) separated by whitespace. In each case, $M < 2^{32}$ (so that M fits in an `unsigned long int` quantity,) but there is no limit on a or b. For each set, the output consists of an echo of the input and the result, in the format shown below. You may *not* use `gcc`'s "long long" integer data types for this problem.

Input:

```
48 12
10000000000 101
1234567890987654321
3000000000
```

Output:

```
  48
= 0 mod 12

  10000000000
= 100 mod 101

  1234567890987654321
= 987654321 mod 3000000000
```

8. A robot wanders through an obstacle course. Each time it encounters an obstacle, it makes a 90° left turn. You are to write a program to determine its path up to the point where it either reaches a designated destination or begins to repeat its path.

In this problem, we will approximate the course with a rectangular grid of squares. Each square may either be empty, be filled with an obstacle, or be a destination square (there may be more than one). The robot begins at a given empty square, moving in one of the four directions parallel to a side of the rectangle. At each step, the robot either moves to the adjacent square in the direction it is moving (or off the board) if there is no obstacle in the way, or changes its direction 90° counter-clockwise. A path ends when the robot reaches a destination square, falls off the edge of the region, or reaches a configuration it has already encountered (a configuration is a combination of position and direction).

The input will consist of sets of data. Each set will consist of two integers, L and W, giving the length and width of the rectangular region (the length is displayed going left to right, the width goes up and down the page), followed by W rows of L characters (uppermost first), with the rows separated from each other by whitespace. The possible characters in each row are '`-`' for an empty square, '`X`' for an obstacle, '`D`' for a destination, or one of the characters '$<$', '$>$', '`^`', or '`V`' for a square containing a robot moving (respectively) left, right, up or down. There will be exactly one robot on the board. You may assume $0 < L < 100$ and $0 < W < 100$. Within these constraints there may be any number of obstacles and destinations.

The output is to consist of a depiction of the board that was input with the robot's path other than the final square indicated with periods. In addition, if the robot has not fallen off the board, its final position is marked with '$<$', '$>$', '`^`', or '`V`' to indicate its final direction. See the examples on the next page.

Input

```
10 8
XXXXXX----
X-------XX
X--DDˆ----
---------X
-XX------X
----------
----------
XXXXXXXXXX

10 8
XXXXXX---- ---------X
X--DDˆ--X- ---------X
--X------X
----------
----------
XXXXXXXXXX

4 3
XXXX
X>-X
XXXX
```

Output

```
Board 0:
XXXXXX----
X.....--XX
X.-D<....-
-........X
-XX------X
----------
----------
XXXXXXXXXX

Board 1:
XXXXXX----
......---X
X--DD.--X-
---------X
--X------X
----------
----------
XXXXXXXXXX

Board 2:
XXXX
X>.X
XXXX
```