1999 Stanford Local Programming Contest

October 9, 1999

Welcome to the 1999 Stanford Local Programming Contest. Good luck and program well!

Contest information. Read carefully!

Rules

1. Each contestant has three hours to complete the problems.
2. Contestants may use resource materials such as books, manuals and program listings. Contestants may *not* use any machine-readable versions of software or data or existing electronic code. All software submitted during the contest *must* be typed in during the contest. No cutting and pasting of code is allowed.
3. Students may not collaborate in any way with each other or anybody else. Do not contact other people via the internet to help you.
4. Contestants are expected to adhere to the honor code. Although some students may not be in the main contest room during the contest, they are still expected to conduct themselves according to the rules.

How will the contest work?

1. From 12:00 to 1:00 students will pick a computer, set up their workspace and complete a test problem. The purpose of this test problem is to send me information about who you are. I won't consider you registered unless you send me this information via the test problem. The test problem will also make sure the submission script is working for you.
2. At 1:00 the contestants will gather in the main contest room (**Gates B02**) and I will hand out the problems. You should then return to your workspace and begin solving the problems.
3. The test problem description explains how to submit your solutions. **Make sure that you run the submission script from the correct directory, and that only the files for your solution are in that directory!**
4. For every run, your solution will either be accepted or rejected for one of the following reasons: compiler error, run time error, time limit exceeded, wrong answer, presentation error. I will try to be lenient on presentation (putting things in the right order, the right whitespace, etc) but you should follow the output format for each problem. The time limit for a run is **one minute**; all of these problems can be solved with programs that terminate in less than 60 seconds.
5. If you want to follow the progress of the contest, go to the **standings web page**. A link to this page exists on the main contest page, http://www.db.stanford.edu/ cooperb/acm. I will update these standings as frequently as I can.

6. At 4:00 the contest will end. No more submissions will be accepted. The contestants will be ranked by the number of solved problems. Ties will be broken based on the total number of time used. This is time elapsed since 1:00 for correct solutions, and 20 minutes per rejected run for incorrect solutions. The top four undergraduates and top two graduates will form the teams that advance to the regionals (unless 5 or 6 undergraduates beat all the graduates; then the undergrads will advance).
7. If you have any questions, come talk to me in Gates B02, or send email to **cooperb@leland**. (Do not send it to cooperb@stanford, @DB or at any other address).

Guidelines for submitted programs

1. All programs must be written in C or C++. To judge the programs, I will compile programs ending in `.c` with `gcc` and programs ending in `.cc` with `g++`. I will use the Leland system to do this, so if your programs compile on a Leland host, they should compile for me.
2. Your solution may have as many files as necessary. However, please use all `.c` or all `.cc` for a given run. Make sure each problem's code files are in a separate directory.
3. All programs should accept their input on stdin and produce their output on stdout. They should be batch programs in the sense that they do not require human input other than what is piped into stdin.
4. I will be judging programs based on a `diff` with the correct solution, so be very careful about the output format. The output should be clearly described in the problems, so follow the description.

0 Test Problem

0.1 Description

This is a test problem to make sure you can compile and submit programs. Your program for this section will print your email address, name and whether you are a graduate student or an undergraduate.

You must submit your solutions via the Leland system and the submission script I have created. You can develop your programs on whatever platform you want, but the submission must be via Leland.

Your solution to a given problem should be in its own directory, separate from the other problems. For example, you may have a `p0` directory, a `p1` directory, and so on. When you submit your solution, everything in this directory will be tarred and sent to the judge, so be careful that the right things are in the directory.

To submit a solution, run the `cooperb/submit` script with a single argument: the problem you are submitting. For example, to submit problem 0, type:

```
>  cooperb/submit 0
```

Make sure you are in the directory with the solution for that program. Make sure only one of your C/C++ code files has a `main()` method. You may want to create a symbolic link to the submit script to make your life easier during the contest.

When you submit your solution, the judge will receive an email and will judge your solution as soon as he can (be patient!) Once this is finished, you will receive an email stating that you have completed the problem or that you have not, and a reason why not. You can resubmit rejected solutions as many times as you like (though incurring a 20 minute penalty for each rejected run). Once you have submitted a correct solution, future submissions of that problem will be disregarded.

0.2 Input

This program will take no input.

0.3 Output

The program should output your email address, name and student status. This information should be separated by commas, and all on the same line. The line should be terminated with a newline. For example:

```
cooperb@leland,Brian Cooper,Graduate
```

1 Numerical Frequency

1.1 Description

The National Security Agency (NSA) has invented a new cryptography algorithm. This algorithm is supposed to be completely unbreakable, and is based on several interacting components. One important component is a key based on the date on which a message is sent. The NSA plans to take the factorial of the day of the year, and count the occurrences of each digit.

Remember that the factorial $n!$ is just $n \times (n-1) \times (n-2) ... \times 1$. Thus, the factoral 4! is 24. In this example, there is one occurrence of the digit 2, one occurrence of the digit 4, and zero occurrences of every other digit. Your program will be given a list of numbers as input and should output, for each number, the count of each digit, including digits for which the count is zero.

1.2 Input

Your program will take as input a series of numbers, separated by spaces, and terminated by a 0. All numbers will be integers x such that $1 \leq x \leq 366$. Do not process the 0; it is a terminator only. For example:

```
3 100 0
```

1.3 Output

Your program should process each number and output the number of occurrences for each digit. Each input number should be printed, followed by an ! and a newline. Then, each digit 0-9 should be printed on a separate line, followed by a colon ":", a single space, and the number of occurrences. For the sample input above:

```
3!
0:  0
1:  0
2:  0
3:  0
4:  0
5:  0
6:  1
7:  0
8:  0
9:  0
100!
0:  30
1:  15
2:  19
3:  10
4:  10
5:  14
6:  19
7:  7
8:  14
9:  20
```

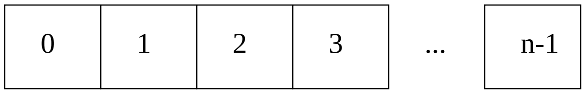

Figure 1: The blocks world.

2 Building Blocks

2.1 Description

Artificial Intelligence is a mysterious and glamorous field. Little do most people know that AI can often be like child's play. Consider robotics, where research projects focus on making robots play with toys. Although any four year old child can do this easily, it can be very hard for a robot.

In this problem your program has to interpret and execute a series of commands in a very simple robot planning language. This language is designed to deal with a *blocks world*, consisting of n blocks numbered 0 to $n-1$, initially laid side by side. Such a world is shown in Figure 1.

The planning language is designed to instruct a robot to move the blocks around, using the following commands (a and b are block numbers such that $0 \leq a, b < n$):

- `move` *a* `onto` *b*
  Puts block a on top of block b, after returning any blocks that were stacked on a and b to their original positions.

- `move` *a* `over` *b*
  Puts block a on top of the pile on block b, after returning any blocks that were stacked on a to their original positions. Retains the order of the b pile.

- `pile` *a* `onto` *b*
  Puts the pile of blocks on top of (and including) block a on top of block b, after returning any blocks that were stacked on b to their original positions. Retains the order of the a pile.

- `pile` *a* `over` *b*
  Puts the pile of blocks on top of (and including) block a on top of the pile on block b. Retains the order of the a and b piles.

- `quit`
  Terminates the planning program.

Your job is to execute these commands in order and display the resulting positions of all of the blocks. Any command in which $a = b$ or in which a and b are in the same stack of blocks is an illegal command and should be ignored. Illegal commands should not affect the final block positions.

2.2 Input

The input begins with an integer n on a line by itself representing the number of blocks. You may assume that $0 < n < 25$. The number of blocks is followed by a sequence of block commands, one command per line. Your program should process all commands until the `quit` command is encountered. For example:

```
10
move 9 onto 1
move 8 over 1
```

```
move 7 over 1
move 6 over 1
pile 8 over 6
pile 8 over 5
move 2 over 1
move 4 over 9
quit
```

You may assume that all commands will be of the form specified above and that there will be no syntactically incorrect commands.

2.3 Output

The output should consist of the final state of the blocks world. Each original block position numbered $i (0 \leq i < n$ where n is the number of blocks) should appear followed immediately by a colon, a single space, and a list of blocks that appear stacked in that position with each block number separated from other block numbers by a single space. There should be one line of output for each block position. For the sample above:

```
0:  0
1:  1 9 2 4
2:
3:  3
4:
5:  5 8 7 6
6:
7:
8:
9:
```

3 The $3n+1$ problem

3.1 Description

Problems in Computer Science are often classified as belonging to a certain class of problems (e.g., NP, Unsolvable, Recursive). In this problem you will be analyzing a property of an algorithm whose classification is not known for all possible inputs.

Consider the following algorithm:

1. input n
2. print n
3. if $n = 1$ then STOP
4.   if n is odd then $n \longleftarrow 3n + 1$
5.   else $n \longleftarrow n/2$
6. GOTO 2

Given the input 22, the following sequence of numbers will be printed

22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

It is conjectured that the algorithm above will terminate (when a 1 is printed) for any integral input value. Despite the simplicity of the algorithm, it is unknown whether this conjecture is true. It has been verified, however, for all integers n such that $0 < n < 1,000,000$ (and, in fact, for many more numbers than this.)

Given an input n, it is possible to determine the number of numbers printed before and including the 1 is printed. For a given n this is called the *cycle-length* of n. In the example above, the cycle length of 22 is 16.

For any two numbers i and j you are to determine the maximum cycle length over all numbers between and including both i and j.

3.2 Input

The input will consist of a series of pairs of integers i and j, one pair of integers per line. All integers will be less than 10,000 and greater than 0. You should process all pairs of integers and for each pair determine the maximum cycle length over all integers between and including i and j.

```
1 10
100 200
201 210
900 1000
```

3.3 Output

For each pair of input integers i and j you should output i, j, and the maximum cycle length for integers between and including i and j. These three numbers should be separated by one space with all three numbers on one line and with one line of output for each line of input. The integers i and j must appear in the output in the same order in which they appeared in the input and should be followed by the maximum cycle length (on the same line).

```
1 10 20
100 200 125
201 210 89
900 1000 174
```

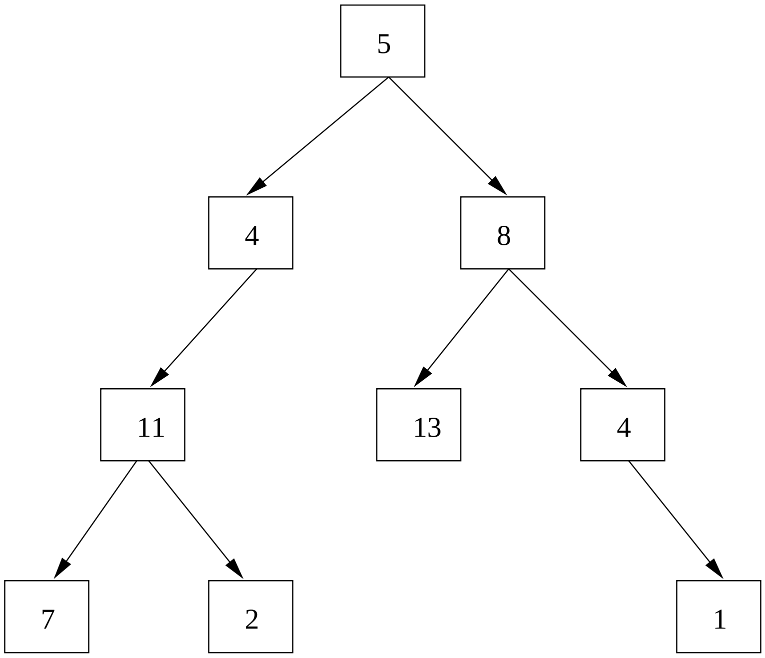

Figure 2: A tree.

4 Trees and Levels

4.1 Description

Trees are everywhere in Computer Science. Traditionally, trees are traversed from the root down, but in this problem you will be concerned with a level-by-level traversal of the tree. A *level-order* traversal of a tree prints the contents of each node in a given level in left to right order, and all nodes at level k are printed before any nodes at level $k+1$. For example, for the tree shown in Figure 2, the level-order traversal is 5, 4, 8, 11, 13, 4, 7, 2, 1.

Your job is to read a specification of a binary tree and print out the level-order traversal. The specification is a series of nodes of the form (n, s) where n is a node number and s is a string representing the path from the root node. s will consist of a series of L's and R's, where L is for a left branch and R is for a right branch. For example, in the example above the node containing 13 is (13,RL) and the node containing 2 is (2,LLR). The root node would be (5,); there is no path string from the root since it is the root.

A binary tree is *completely specified* if every node on all root-to-node paths in the tree is given a value exactly once. If the specification does not specify a complete tree, then your program should display `not complete` instead of the level-order traversal.

4.2 Input

The input is a sequence of binary trees specified as described above. Each tree in a sequence consists of several pairs (n, s) as described above, separated by whitespace. The last entry in the tree is (). No whitespace appears between left and right parentheses. All nodes contain a positive integer. Every tree in the input will consist of at least one node and no more than 256 nodes. For example:

```
(11,LL) (7,LLL) (8,R)
(5,) (4,L) (13,RL) (2,LLR) (1,RRR) (4,RR) ()
```

```
(3,L) (4,R) ()
```

4.3 Output

For each completely specified binary tree in the input, the level order traversal of that tree should be output. If the tree is not completely specified, i.e. some node in the tree is not given a value or a node is given more than one value, then the string `not complete` should be printed. Each traversal or `not complete` should appear on a separate line, corresponding to the order of the trees in the input. For the example above, the output should be:

```
5 4 8 11 13 4 7 2 1
not complete
```

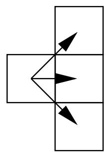

Figure 3: Legal moves.

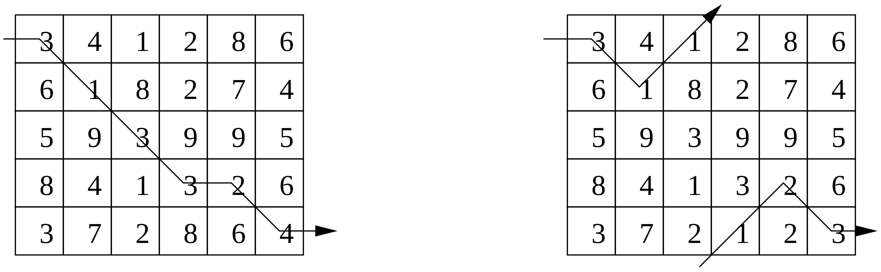

Figure 4: Two examples.

5 The Salesman Still Travels

5.1 Description

The Travelling Salesman Problem (TSP) is a legendary problem in computer science, an example of an NP-complete problem. Here, you will deal with a much simplified version.

Given an $m \times n$ matrix of integers, you are to write a program that computes a path of minimal weight. A path starts anywhere in column 1 (the first column) and consists of a sequence of steps terminating in column n (the last column). A step consists of travelling from clumn i to column $i+1$ in an adjacent (horizontal or diagonal) row. The first and last rows (rows 1 and m) of a matrix are considered adjacent, i.e. the matrix "wraps" so that it represents a horizontal cylinder. Legal steps are illustrated in Figure 3

The *weight* of a path is the sum of the integers in each of the n cells of the matrix that are visited. For example, two slightly different 5×6 matrices are shown in Figure 4 (the only difference is the numbers in the bottom row). The minimal path is illustrated for each matrix. Note that the path for the matrix on the right takes advantage of the adjacency property of the first and last rows.

5.2 Input

The input consists of a series of matrix specifications. Each matrix specification consists of the row and column dimensions in that order on a line followed by $m \times n$ integers where m is the row dimension and n is the column dimension. The integers appear in the input in row major order, i.e. the first n integers constitute the first row of the matrix, the second n integers constitute the second row and so on. The integers on a line will be separated from other integers by one or more spaces. Note: integers are not restricted to being positive. There will be one or more matrix specifications in the input.

For each specification the number of rows will be between 1 and 10 inclusive; the number of columns will be betwen 1 and 100 inclusive. No path's weight will exceed 30 bit integer values. For example:

```
5 6
3 4 1 2 8 6
```

```
6 1 8 2 7 4
5 9 3 9 9 5
8 4 1 3 2 6
3 7 2 8 6 4
5 6
3 4 1 2 8 6
6 1 8 2 7 4
5 9 3 9 9 5
8 4 1 3 2 6
3 7 2 1 2 3
2 2
9 10 9 10
```

5.3 Output

Two lines should be output for each matrix specification in the input file. The first line represents a minimal-weight path, and the second line is the cost of a minimal path. The path consists of a sequence of n integers (separated by one space) representing the rows that constitute the minimal path. If there is more than one path of minimal weight the path that is lexicographically smallest should be output. For the example above:

```
1 2 3 4 4 5
16
1 2 1 5 4 5
11
1 1
19
```