2002 Stanford Local Programming Contest

October 5, 2002

Read these guidelines carefully!

Rules

1. You may use resource materials such as books, manuals and program listings. You may *not* use any machine-readable versions of software, 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.

2. Students may not collaborate in any way with each other or anybody else, including people contacted via Internet.

3. You are expected to adhere to the honor code. You are still expected to conduct yourself according to the rules, even if you are not in Gates B02.

Guidelines for submitted programs

1. All programs must be written in C, C++, or Java. To judge the programs, I will compile programs in the following way:

   - `.c`: using `gcc -lm`
   - `.cc`: using `g++ -lm`
   - `.java`: using `javac`

   I will use the Leland system to do this, so if your programs compile on a Leland host, they should compile for me. Please use only the listed filename extensions.

2. **Java users: Please place your `public static void main()` function in `public class Main`.** I want to be able to run your program with the command `java Main`.

3. Your solution may have as many files as necessary. However, please use the same language (C, C++, or Java) for a given run. Make sure each problem's code files are in a separate directory.

4. 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.

5. Be careful to follow the output format described in the problem. I will be judging programs based on a `diff` of your output with the correct solution.

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; you won't be registered until you do this. This will also make sure the submission script is working for you.

2. At 1:00 the contestants will gather in the **outside 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.* The time limit for a run is **one minute**.

5. Watch your **email on Leland**. I will be sending out any necessary messages as well as notification if your programs are accepted or rejected via email, and the emails will go to

   `your_leland_id@leland.stanford.edu`.

6. 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.

7. 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 time used for correct solutions (the time elapsed since 1:00 plus 20 minutes per rejected solution).

8. The top six contestants will advance to the regionals, subject to the constraint of a maximum of one graduate student per three person team.

If you have any questions, come talk to me in **Gates B02**, or send email to **cooperb@leland.**

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 leland user id, 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.
C/C++ users: Make sure only one of your C/C++ code files has a `main()` method.
Java users: Make sure there is a `public static void main()` method in `public class Main`.

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 of a problem you eventually get right). 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 leland user id, 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,Brian Cooper,Graduate
```

1 baa baa (or is that aab aab?)

1.1 Description

Let us now turn our attention to modified L-Systems. As a model, let us have words of length n, over a two letter alphabet $\{a, b\}$. The words are cyclic, which means that we can write one word in any of n forms we receive by cyclic shift, whereby the first and the last letters in the word are considered to be neighbors.

Rewriting rules rewrite a letter at a position i, depending on letters at the positions $i-2, i, i+1$. We rewrite all letters of the word in one step. When we have a given starting word and a set of rewriting rules a natural question is: how does the word look after s rewriting steps?

1.2 Input

There are several blocks in the input, each describing one system. There is an integer number n, $2 < n < 16$, the length of the input word in the first line. There is a word in the next line. The words contains only lowercase letters "a" and "b". There are four characters c_1, c_2, c_3, c_4 in the next eight lines. Each quadruple represents on rewriting rule with the following meaning: when the letter at the position $i-2$ is c_1 and the letter at the position i is c_2 and the letter at the position $i+1$ is c_3 then the letter at the position i after rewriting will be c_4. Rewriting rules are correct and complete. There is an integer number s, $0 \leq s \leq 2000000000$, in the last line of the block. The input is terminated by EOF. For example:

```
5
aaaaa
aaab
aabb
abab
abbb
baab
babb
bbab
bbbb
1
```

1.3 Output

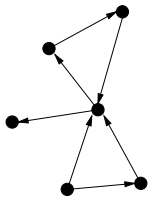
There is one line corresponding to each block of the input file. The line contains a word which we receive after s rewriting steps from the corresponding starting word using given rewriting rules. As we mentioned above, the word can be written in any of n cyclic shifted forms. The output contains the lexicographically smallest word, assuming that $a < b$, for each block of the input. For example:

```
bbbbb
```

2 How strong is your network?

2.1 Description

Computer networks are often modeled as graphs. A graph to a theoretician is a set of vertices and edges connecting those vertices. A vertex might represent, for example, a computer on a network, while an edge may represent a physical or logical connection between two computers. If a graph is *directed*, then the edges have a specific direction. An example of a directed graph is:



It is often useful to study the properties of graphs so that we can understand the properties of the real world systems (e.g. networks) that the graph models. One property that is interesting to study is whether a graph is *connected*, which means that there is a path between every pair of nodes. In a directed graph, we must be more precise in what we man by "connected." A *strongly connected* directed graph is a graph in which there is a directed path between every pair of vertices in the graph. A directed path follows edges by respecting the direction of those edges. An undirected path is like a directed path, except that we ignore the direction of edges. We say that a graph is *weakly connected* if there is an undirected path between every pair of vertices in the graph. A graph is *not connected* if it is not weakly connected.

2.2 Input

Your program will read in a set of graphs. Each graph will be in the following form:

n
L
X_0,Y_0
X_1,Y_1
...
X_{L-1},Y_{L-1}

where n is the number of vertices in the graph, L is the number of edges, and "X_i,Y_i" is a directed edge $X_i \rightarrow Y_i$. The final graph will be followed by a zero ('0') on its own line. For example:

```
4
3
0,1
1,2
2,3
4
2
0,1
2,3
0
```

2.3 Output

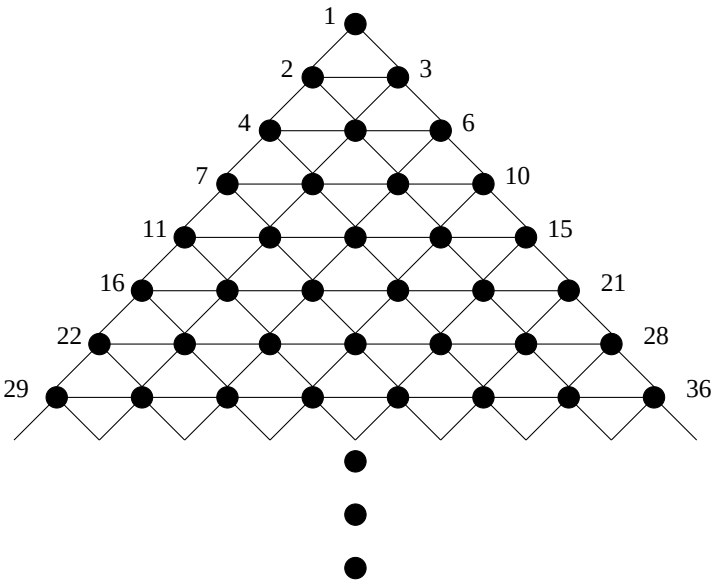
For each graph in the input, your program should output "Strongly connected" if the graph is strongly connected, "Weakly connected" if the graph is weakly connected and not strongly connected, and "Not connected" if the graph is not connected. The result for each graph should be on its own line. For example:

```
Weakly connected
Not connected
```

3 Triaparallelogons

3.1 Description

Consider the points on an infinite grid of equilateral triangles as shown below:



Note that if we number the points from left to right and top to bottom, then groups of these points form the vertices of certain geometric shapes. For example, the sets of points 1,2,3 and 7,9,18 are the vertices of triangles, the sets 11,13,26,24 and 2,7,9,18 are the vertices of parallelograms, and the sets 4,5,9,13,12,7 and 8,10,17,21,32,34 are the vertices of hexagons.

Your job is to write a program which will repeatedly accept a set of points on this triangular grid, analyze it, and determine whether the points are the vertices of one of the following "acceptable" figures: triangle, parallelogram, or hexagon. In order for a figure to be acceptable, it must meet the following two conditions:

1. Each side of the figure must coincide with an edge in the grid.
2. All sides of the figure must be of the same length.

3.2 Input

The input will consist of an unknown number of point sets. Each point set will appear on a separate line in the file. There are at most six points in a set and the points are limited to the range 1..32767.
For example:

```
1 2 3
11 13 29 31
26 11 13 24
4 5 9 13 12 7
1 2 3 4 5
47
11 13 23 25
```

3.3 Output

For each point set in the input file, your program should deduce from the number of points in the set which geometric figure the set potentially represents; e.g., six points can only represent a hexagon, etc. The output must be a series of lines listing each point set followed by the results of your analysis. For example:

```
1 2 3 are the vertices of a triangle
11 13 29 31 are the vertices of a parallelogram
26 11 13 24 are the vertices of a parallelogram
4 5 9 13 12 7 are the vertices of a hexagon
1 2 3 4 5 are not the vertices of an acceptable figure
47 are not the vertices of an acceptable figure
11 13 23 25 are not the vertices of an acceptable figure
```

4 The magic bus

4.1 Description

Smallville has a well developed bus system. Since most of the important buildings in town are located along the Main Street, almost all the buses pass through the street. All the stops along the Main Street, and there are very many of them, are numbered with numbers: 0, 1, 2 and so on. All the buses enter the Main Street at the stop numbered 0 and continue towards the higher numbered stops. For each bus line (which is also identified by a number), there is a schedule posted at stop 0 giving the minutes within an hour when a bus of this line leaves the stop.

Brian arrives at stop 0 at m minutes ($0 \leq m < 60$) after the hour and takes the first bus that leaves stop 0 along the Main Street. If Brian arrives at stop 0 the same minute when a bus leaves, Brian manages to board the bus. He leaves the bus at the next stop and takes the next bus that arrives and continues to the next stop where he leaves. Brian continues in this fashion until he reaches stop n, $0 < n < 1000000000$. By which bus will Brian arrive at stop n?

4.2 Input

Input contains multiple cases. The first line of a case gives t, the number of bus lines, $0 < t < 30$. The next t lines each contain a number of a bus line followed by a colon and then a list (separated by spaces) of minutes after a full hour when the bus of this line leaves stop 0. Each list of departure times is terminated by -1. No bus leaves stop 0 more than 20 times in an hour, no two buses leave station 0 at the same time, all the buses have the same speed and they never meet at a stop. A line with m and n then follows. The input ends with a line where $t = 0$. This line should not be processed. For example:

```
1
11: 10 20 30 -1
3 2
2
11: 10 20 30 -1
134: 16 25 35 45 58 -1
48 783
0
```

4.3 Output

For each case of input, output in a format shown below the number of the bus line by which Brian arrives at stop n. For example:

```
Case 1: Brian arrives at stop 2 by bus 11.
Case 2: Brian arrives at stop 783 by bus 134.
```

5 Wet

5.1 Description

To enable homebuyers to estimate the cost of flood insurance, a real-estate firm provides clients with the elevation of each 10-meter by 10-meter square of land in regions where homes may be purchased. Water from rain, melting snow and burst water mains will collect first in those squares with the lowest elevations, since water from squares of higher elevation will run downhill. For simplicity, we also assume that storm sewers enable water from high-elevation squares in valleys (completely enclosed by still higher elevation squares) to drain to lower elevation squares, and that water will not be absorbed by the land.

From weather data archives, we know the typical volume of water that collects in a region. As prospective homebuyers, we wish to know the elevation of the water after it has collected in low-lying squares, and also the percentage of the region's area that is completely submerged (that is, the percentage of 10-meter squares whose elevation is strictly less than the water level). You are to write the program that provides these results.

5.2 Input

The input consists of a sequence of region descriptions. Each begins with a pair of integers, m and n, each less than 30, giving the dimensions of the rectangular region in 10-meter units. Immediately following are m lines of n integers giving the elevations of the squares in row-major order. Elevations are giving in meters, with positive and negative numbers representing elevations above and below sea level, respectively. The final value in each region description is an integer that indicates the number of cubic meters of water that will collect in the region. A pair of zeroes follows the description of the last region. For example:

```
3 3
25 37 45
51 12 34
94 83 27
10000
0 0
```

5.3 Output

For each region, display the region number (1, 2, ...), the water level (in meters above or below sea level) and the percentage of the region's area under water, each on a separate line. The water level and percentage of the region's area under water are to be displayed accurate to two fractional digits. Follow the output for each region with a blank line. For example:

```
Region 1
Water level is 46.67 meters.
66.67 percent of the region is under water.
```