2017 CCSC Eastern Conference Programming Competition

October 21st, 2017

Muhlenberg College, Allentown, Pennsylvania

This page is intentionally left blank.

Problem 1 — Secret Code

You can't be too careful with your secrets! To enable you to communicate securely with your network of agents, you write a program to decrypt secret messages.

The program decrypts using a *key*, which is a sequence of 26 distinct upper-case letters. When decrypting an encrypted message, 'A' is replaced by whatever letter is in the first position in the key, 'B' is replaced by whatever letter is in the second position, and so forth.

Program input: The input of the program is a series of pairs of lines of text. The first line in each pair is a decryption key as described above. The second line in each pair is an encrypted message. As a special case, if the first line in a pair is the text `quit`, then the program should exit immediately without further output. Example input:

```
FSEOABLITURZKXNYDCQHGVWJPM
WTCO HO QDJFI, JBC FKJIC ADKRC.
IYUCAXFEKWGVBOMHQJRSPNDTZL
HAXPHS BNC CTH DNOUCXHST NS BNC DEV'X WN XPH RNM.
OCVUJDMFBHWIGQTXYRASZELPNK
LH QAD KSYO OA MA TAGVKJVRV, MAOA LT OJV IVTO KSQ OA MVO OJVRV.
quit
```

Program output: For each input pair (decryption key and encrypted message), the program should output a single line of text containing the decrypted text. All letters (A–Z) in the encrypted message should be replaced using the substitution algorithm described above. All other characters (e.g., spaces and punctuation) should be left unmodified. Example output (corresponding to the example input above):

```
WHEN IN DOUBT, USE BRUTE FORCE.
EITHER YOU USE COMPUTERS OR YOU CAN'T DO THE JOB.
IF YOU WANT TO GO SOMEWHERE, GOTO IS THE BEST WAY TO GET THERE.
```

This page is intentionally left blank.

Problem 2 — Whee!

You are one of the world's premier designers of two-dimensional roller coasters. You live by three principles:

1. The ride must be fun
2. The car must reach the end of the track
3. The riders must survive

You uphold the first principle through years of experience and an unwavering devotion to your art. Upholding the second and third principles, however, is a bit trickier!

Fortunately, through the magic of computation, you can simulate two-dimensional roller coaster physics, and be sure that (1) your riders reach the end of the track, and (2) reach it alive. You have developed your own notation for describing a 2-D coaster. An example:

```
\\_/_\\\_/\
```

Each \ character indicates a downward slope, each _ character indicates a level grade, and each / character indicates an incline. The example above corresponds to the following roller coaster:

```
\    _
 \_/ \
      \
       \_/\
```

The car moves from left to right, starting at the leftmost position. Its velocity is initially 0. Each downward slope increases the car's velocity by 1. Each incline decreases the car's velocity by 1. Level grades do not change the car's velocity. The car only moves forward if its velocity is positive.

The simulation tracks the car's velocity and location. If the car moves past the last position, then it is considered to reach the end of the track—that's good! However, if the car's final velocity is 10 or greater, the car is considered to have crashed, dooming its riders—that's not good.

Program input: Each line of input contains a 2-D roller coaster specification using the notation described above. As a special case, a line of input consisting of the text `quit` indicates the end of input, and the program should terminate immediately without any further output. Example input:

```
\\_/_\\\_/\
_\\\_/_\
\\\_\_//_//_
\\\_/\\\\\\_//_\\\\\\_\\_
```

```
\\\_/\\\\\\_//_\\\\\\_\\_/////
\\\_/\\\\\\_//_\\\\\\_\\_////
\\_\__\\_/__//_//
\_//_\\\\\
quit
```

Program output: For each input (other than `quit`), the program should print a single line of output with one of the following responses:

- `Whee!` if the car reached the end of the track and did not crash
- `Splat!` if the car reached the end of the track and crashed
- `Stopped at` *N* if the car did not reach the end of the track, where *N* is the position where the car stopped (the leftmost position being 0)

Example output (corresponding to the example input above):

```
Whee!
Stopped at 0
Stopped at 10
Splat!
Whee!
Splat!
Stopped at 16
Stopped at 2
```

Problem 3 — Here comes the flood

You are one of the world's most celebrated ASCII artists! Your medium is text, and letters, numbers, and symbols are your palette.

To help you create your compositions more efficiently, you decide to write a program to do *flood fill*, which allows you to replace all occurrences of a selected character in a contiguous region with a *fill* character. It works like this:

- A composition is a grid of text characters
- You select a particular character in the grid (specifying its row and column)
- All reachable occurrences of the selected character are replaced by a chosen *fill* character
- An occurrence is "reachable" if it can be reached from the location of the selected character by some sequence of up, down, right, and/or left moves without ever encountering a character other than the selected character

Program input: The program input consists of a series of *specifications*. Each specification starts with a line specifying integer width, height, x, and y values, followed by a single character, which is the *fill* character. Then, a number of input lines equal to the height value follows, with each line specifying the characters forming one row of the composition grid. As a special case, if a specification starts with an input line consisting of the text `quit`, then the program should exit immediately without any further output. Example input:

```
16 7 8 5 o
................
................
....#######.....
...#......#.....
....#......####.
...#............
..#.............
16 7 8 5 o
................
................
....#######.....
...#......#.....
....#......#####
...#............
..#.............
quit
```

Program output: For each specification in the input the program should print an output grid showing the results of performing a flood fill at the location specified by the x and y coordinates. Note that coordinate value 0 (zero) indicates the leftmost column (x) or topmost row (y). Starting from the location of the flood fill, all reachable occurrences of the selected character should be replaced with the fill character. Each output grid should be separated with a single blank line. Example output (corresponding to the input above):

```
oooooooooooooooo
oooooooooooooooo
oooo#######ooooo
ooo#oooooo#ooooo
oooo#oooooo####o
ooo#ooooooooooo
oo#ooooooooooooo

................
................
....#######.....
...#oooooo#.....
....#oooooo#####
...#ooooooooooo
..#ooooooooooooo
```

Problem 4 — On Task

There's so much to do these days! Homework, sports, hobbies, socializing, watching over the flock, shearing, taking the wool to market—it seems like it never ends! To further complicate matters, sometimes one task must be done *before* another can be done, such as making sure that the sheep dip is on hand *before* the sheep are treated for parasites. Also, in some cases, tasks will need to be done simultaneously. For example, Sheila likes to listen to music as she dips sheep, so you'll need to schedule Bruce the DJ to raise the roof while the sheep dip is taking place.

To help you manage your busy life, you decide to write a program to schedule your tasks. Each task is identified by a capital letter (A, B, C, etc.) Ordering constraints can be specified: A<B means that task A must be done before task B, and A=B means that tasks A and B must be done simultaneously.

Program input: Each line of input specifies a list of tasks (separated by commas, no spaces), followed by a series of zero or more constraints (separated by spaces) of the form X<Y or X=Y, where X and Y are tasks. As a special case, a line containing the text `quit` indicates the end of input, and when encountered the program should exit without any further output. Example input:

```
A,B,C,D A<C B<D
A,B,C,D A<C B<D A=B
A,B,C,D A<C B<D A=B C=D
A,B,C,D D<A D<B C=D
A,B,C,D D<A A=B B=C
A,B,C,D C<B
A,B,C A<B B<A
A,B,C A<B B<C C<A
quit
```

Program output: For each input to the program (other than `quit`), the program should output a line of text with a *schedule*. A schedule consists of the text `Order:`, followed by a space, followed by a sequence of steps separated by spaces. Each step consists of one or more comma-separated tasks in alphabetical order. A step will have multiple tasks only if there are constraints specifying that tasks must be scheduled simultaneously. The schedule must honor all of the constraints specified by the input, and must include all of the tasks. In the absence of an explicit ordering constraint, steps should be ordered alphabetically by the "smallest" (alphabetically earliest) task in each step. As a special case, if there is no possible schedule that satisfies the ordering constraints, the output line should consist of the text `No schedule`. Example output (corresponding to the example input above):

```
Order: A B C D
```

```
Order: A,B C D
Order: A,B C,D
Order: C,D A B
Order: D A,B,C
Order: A C B D
No schedule
No schedule
```

Problem 5 — C'est très logique

Disjunctive Normal Form, or DNF, is a way of writing a boolean expression in a canonical form.

Consider a language for writing arbitrary boolean expressions, where `^` means "and", `+` means "or", `!` means "not", the letters `a`, `b`, `c`, etc. indicate propositions (statements that are true or false), and parentheses may be used to explicitly control the order of evaluation. Here is an example expression (not in DNF):

```
a^(b+!c)
```

This expression means "`a` and (`b` or not `c`)", meaning that the overall expression is true if and only if both

- `a` is true, and
- either `b` is true or `c` is false (or both)

In DNF, this example expression would be written

```
(a^!b^!c)+(a^b^!c)+(a^b^c)
```

An expression written in DNF is some number of terms connected by `+` (or). Each term specifies, for all propositions in the original expression, one combination of values (true or false) which, if assigned to the propositions, will make the original expression true. For example, in the DNF expression above, the first term indicates that one case where the original expression is true occurs when `a` is true, `b` is false, and `c` is false.

Your program should read arbitrary boolean expressions and convert them to DNF. The input expressions use the syntax described above. The `^` operator ("and") has a higher precedence than the `+` operator ("or"). The `!` operator ("not") is a unary prefix operator. Parentheses may be used arbitrarily for grouping. Input expressions will not have any whitespace characters (space, tab, etc.)

Requirements for the DNF output are:

- Each term is parenthesized
- The propositions in each term are in alphabetical order
- The terms are ordered such that the assignments of truth values to propositions are ordered lexicographically (with false being considered less than true); for example, all terms where the first proposition is false appear earlier than all terms where the first proposition is true

Program input: The input to the program will be a sequence of boolean expressions, one per line. As a special case, a line of input consisting of the text `quit` terminates the input, and should cause the program to exit without generating any further output. Example input:

```
a^(b+!c)
(a^b)+!c
a^b+!c
a+!((b)^c)
a^(b+!c)^d
quit
```

Program output: For each input expression, the program should print a single line containing the DNF form of the corresponding input expression. Example output (corresponding to the example input shown above):

```
(a^!b^!c)+(a^b^!c)+(a^b^c)
(!a^!b^!c)+(!a^b^!c)+(a^!b^!c)+(a^b^!c)+(a^b^c)
(!a^!b^!c)+(!a^b^!c)+(a^!b^!c)+(a^b^!c)+(a^b^c)
(!a^!b^!c)+(!a^!b^c)+(!a^b^!c)+(a^!b^!c)+(a^!b^c)+(a^b^!c)+(a^b^c)
(a^!b^!c^d)+(a^b^!c^d)+(a^b^c^d)
```

Problem 6 — Interleavings

The Milliard Gargantubrain, one of the most powerful computers ever conceived, nonetheless has a very simple instruction set:

- `ld` *reg*, *var*
- `st` *var*, *reg*
- `inc` *reg*

The `ld` instruction loads an integer value from a memory location named *var* into a CPU register *reg*. The `st` instruction stores an integer value in a CPU register *reg* to a memory location named *var*. The `inc` instruction increments (adds one to) a CPU register *reg*.

Why is this computer so powerful? The answer is that the Milliard Gargantubrain can execute programs consisting of multiple *threads*. Each thread consists of a sequence of instructions. Each thread has its own private registers, but shares memory locations with all other threads in the program. All memory locations and registers initially contain the value 0 (zero). Threads run in *parallel*. Because instructions in different threads aren't synchronized, in general there can be more than one outcome of running a program with multiple threads. An outcome is determined by an *interleaving* of instructions from the threads in the program. An interleaving is an ordering of all instructions in all threads, such that instructions from an individual thread occur in order, but instructions in different threads could occur in any order with respect to each other.

For example, consider a program with the following threads:

```
-- Thread 0        -- Thread 1
ld r0, x           ld r0, x
inc r0             inc r0
st x, r0           st x, r0
```

Here is *one* possible interleaving of the instructions in the two threads:

```
-- Thread 0        -- Thread 1
ld r0, x
inc r0
st x, r0
                   ld r0, x
                   inc r0
                   st x, r0
```

The outcome of this interleaving is `x=2`. However, consider another potential interleaving:

```
-- Thread 0        -- Thread 1
ld r0, x
                   ld r0, x
inc r0
st x, r0
                   inc r0
                   st x, r0
```

The outcome of this interleaving is `x=1`. (Recall that each thread has its own set of registers, so Thread 0's `r0` register is distinct from Thread 1's `r0` register.)

Your task is to analyze programs to determine possible execution outcomes by simulating all possible interleavings of instructions.

Program input: The overall program consists of a sequence of 0 or more *simulations*, terminated by an input line `quit`. Each simulation consists of a line with the text `simulation`, followed by one or more *threads*, followed by a line with the text `done`. Each thread consists of one or more instructions (one per line), followed by a line with the text `end`. Example input:

```
simulation
thread
ld r0, x
inc r0
st x, r0
end
thread
ld r0, x
inc r0
st x, r0
end
done
simulation
thread
ld r0, x
ld r1, y
inc r0
inc r1
st x, r0
inc r0
st x, r0
st y, r1
end
thread
ld r0, y
```

```
inc r0
st y, r0
inc r0
st y, r0
end
done
quit
```

Program output: For each simulation specified in the input, a single line of output should be printed starting with the text `Outcomes:`, followed by a space, followed by one or more *outcomes*, separated by a single space. An outcome is a comma-separated sequence of variable values of the form *var*=*value*, where *var* is the name of a memory location, and *value* is the final integer value stored in that memory location (after all threads have completed). The outcomes are ordered lexicographically. Example output (corresponding to the example input shown above):

```
Outcomes: x=1 x=2
Outcomes: x=2,y=1 x=2,y=2 x=2,y=3
```

This page is intentionally left blank.

Problem 7 — Grass Fire

The two-dimensional world of Greeb consists of lush, two-dimensional grassland. Each physical location within Greeb corresponds to a point on a grid, with neighboring locations directly above, right, below, and left. The natives of Greeb enjoy many outdoor activities in its verdant environs, such as two-dimensional croquet.

However, during periods of drought on Greeb, there is a danger of two-dimensional grass fires! Grass fires are caused by two-dimensional flames. A two-dimensional flame has the following characteristics:

- It has a location (x and y coordinates)
- It has a direction (up, right, down, or left)
- If the location of a flame has unburned grass, then the grass at that location will be burned
- If the neighboring location one unit of distance forward (relative to the flame's direction) is already burned, then the flame is extinguished
- If the flame is not extinguished, then
    1. Relative to the direction in which it is moving, the flame will spawn new flames to the locations immediately to its left and right (in each case, only if the adjacent location is unburned); and
    2. It will move one unit of distance forward (based on its direction)

Two-dimensional flames are caused by two-dimensional lightning strikes. A lightning strike, when it hits a location, will create four flames at that location, each moving in a different direction (up, right, down, and left.)

In order to be fully prepared for grass fire season, the Greeb Institute of Science has commissioned you to write a program to simulate grass fires. The input to the program specifies a grid size (number of columns and rows), which is assumed initially to consist entirely of unburned grass. The input also specifies the coordinates of some number of lightning strikes within the grid. The simulation simulates the behavior of all of the flames created by the lightning strikes, keeping track of where each flame is extinguished. The output of the program shows the lightning strike locations and (once the simulation has reached a steady state) where flames were extinguished.

The simulation operates as a sequence of discrete time steps, and should have the following properties:

- At the beginning of each time step, the location of all active flames is marked as being burned
- If the location immediately in front of any flame is burned, the flame is extinguished (and no longer active)
- New flames are spawned to the left and right of active flames if those locations are

unburned and if they are within the bounds of the simulation grid

- Active flames (but not newly-spawned flames) are moved one unit of distance forward, but if the new location is not within the bounds of the simulation grid, the flame is removed from the simulation (this does not count as being extinguished)
- At the end of a time step, newly spawned flames become active

The simulation ends when there are no more active flames.

Note that it is possible that due to spawning, there could be multiple equivalent flames (meaning multiple flames with the same location and direction). Equivalent flames should be considered to be a single flame.

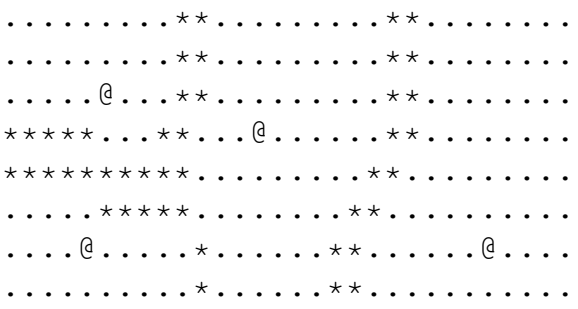
Program input: Each line is an integer width and height of the simulation grid, followed by one or more pairs of integer x/y coordinate pairs specifying locations of (simultaneous) lightning strikes. A 0 (zero) coordinate value indicates the leftmost column or the topmost row. As a special case, a line consisting of just the text `quit` is treated as the end of input, and when encountered, the program should immediately terminate without any further output. Example input:

```
20 7 5 3 15 3
20 7 6 3 15 3
30 8 5 2 4 6 25 6 13 3
quit
```

Program output: For each input, the program should print the final grid, using '`@`' to represent lightning strike locations, '`.`' to represent burned grass, and '`*`' to represent locations where a flame was extinguished ('`@`' and '`*`' take priority over '`.`'). One blank line should be printed between each output grid. Example output (corresponding to the input shown above):

```
..........*.........
..........*.........
..........*.........
.....@....*....@....
..........*.........
..........*.........
..........*.........

..........**........
..........**........
..........**........
......@...**...@....
..........**........
..........**........
..........**........
```



This page is intentionally left blank.