

2007
ACM Northeast
North America
Programming Contest

at

SUNY-Oswego

October 6, 2007

Contest Questions

Problem 1: Room Full 'o Mousetraps!

The floor of a small room is covered with a 16x16 grid of tiles. Each tile contains one mousetrap. When all have been set, a tennis ball is released from a specified direction. The ball lands on a mousetrap, and the trap snaps closed with enough force to send both it and the ball back into the air. The ball and trap land on other traps, thus causing a chain reaction.

The rules governing landing and directions are as follows.

1. A single trap or ball cannot bounce more than three times. On the fourth landing, the object must come to rest.
2. Objects that land on an empty space, or a space occupied by a snapped trap, do not bounce.
3. The direction of an object leaving a tile is a function of the initial direction incremented in a circular manner (N-E-S-W-N-E, etc.) by the bounce number. For example, if an object arrives with a direction of north on the second bounce, it departs with a direction of south.
4. An object travels 3 tiles following the first bounce, 2 tiles on the second bounce, and 1 tile on the third bounce. If an object strikes a wall, it lands in the tile directly below the point of contact.
5. An object that lands on a tile and "launches" a trap will land before the displaced object.

Input

The input to your program is the initial direction of the tennis ball and the tile coordinates of the first bounce. Example

North, 2, 2

means that the tennis ball was heading north when it landed on the second tile of the second row (referenced from the upper left corner of the room).

Output

Your program must output a 16 x 16 grid showing which tiles contain traps that are still set (indicated by a '1') and which tiles contain objects that have come to rest (indicated by 1 '0') when all bouncing has stopped. For the above input, output is:

```
0000010010010010
0001011011011010
0110010010010010
0011111011011010
0011011011011010
0010010010010110
0011010010010010
0011010010010010
0010110110110110
0011010010010010
0010010010010010
0010110110110100
0011010010010010
0010010010010010
0010110110110100
0010010010010010
0010110110110100
0010010010010010
```

Problem 2: nPVI from Esac Notation

Background

1. The **normalized pairwise variability index, nPVI**, is a measure of durational variability in speech and music. This problem focusses on its role in analyzing durational variability in melody. The metric is defined as follows, where m is the number of durational events and d_k is the duration of the k th event.

$$nPVI = \frac{100}{m-1} * \sum_{k=1}^{m-1} \left| \frac{d_k - d_{k+1}}{\frac{d_k + d_{k+1}}{2}} \right|$$

2. The **Essen Folksong Database** is very popular among music cognition researchers for a variety of reasons, not the least of which is its size. The songs in the database are represented textually using a somewhat surprising syntax, called **Esac** notation. Only the last three items below, especially those containing the “period” and the “underscore”, are particularly relevant for your current needs.

- 1, 2, 3, 4, 5, 6, 7 – pitches in the central octave
- -1, -2, -3, ... – an octave below
- +1 +2 +3 ... – an octave above
- 0 – a rest
- # – half a tone up
- b – half a tone down
- *blank blank* – end of bar
- *hard return* – end of phrase
- *blank//* – end of melody
- . after note – duration increased 50
- _ after a note – double duration
- A note (a digit) without any sign after has a duration equal to the shortest time unit.

For example:

- (a) The following melody consists of eight notes of equal duration.

5432 1511 //

- (b) The following melody consists of 8 notes. The first four are of one duration. The second four are each twice as long in duration.

4321 4_3_2_1_ //

- (c) The following melody again consists of eight notes, but this time the odd notes are twice as long as the even notes.

```
4.32_1 4.32_1 //
```

- (d) The following melody consists twelve notes, the odd ones being 50% longer than the even ones.

```
5.43.2
```

```
1.-7-7b.1
```

```
1.1#7.1 //
```

- (e) The following little melody consists of nine notes. Suffice it to say that the durations of the notes can be thought of as: “1 1.5 1.5 2 3 3 4 1 1”.

```
12.3.4_ 2_3_ 4_21
```

Problem

Write a program which reads an Esac encoded melody from the standard input stream and displays its nPVI value.

Sample Session

Suppose that the program is a Java program called NPVI.

```
os) java NPVI
```

```
5432 1511 //
```

```
nPVI = 0.0
```

```
os) java NPVI
```

```
4321 4_3_2_1_ //
```

```
nPVI = 9.523809523809524
```

```
os) java NPVI
```

```
4.32_1 4.32_1 //
```

```
nPVI = 66.66666666666666
```

```
os) java NPVI
```

```
5.43.2
```

```
1.-7-7b.1
```

```
1.1#7.1 //
```

```
nPVI = 40.0
```

```
os) java NPVI
```

```
12.3.4_ 2_3_ 4_21 //
```

```
nPVI = 32.14285714285714
```

Problem 3: Gate Level Simulation

A gate-level circuit consists of three types of gates: And, Or, and Inverter.

Each type of gate has a gate delay -- the amount of time it takes the circuit to compute the logic function.

The input pins for both And and Or gates are numbered 1 and 2. Since Inverters have only one input, the pin number is not needed.

Write a gate-level simulator for an arbitrary logic circuit. An example of the definition of a circuit is shown below. A logic circuit is defined by means of an input file, which consists of the following:

- Gate delays are defined first, and these are specified with lines that begin with the keyword "delay". The time is in nsec. It might be the case that not all gates have a gate delay specified - and this is perfectly acceptable if that gate type is not used in the circuit.
- The rest of the file is organized as a sequence of connections defined as:

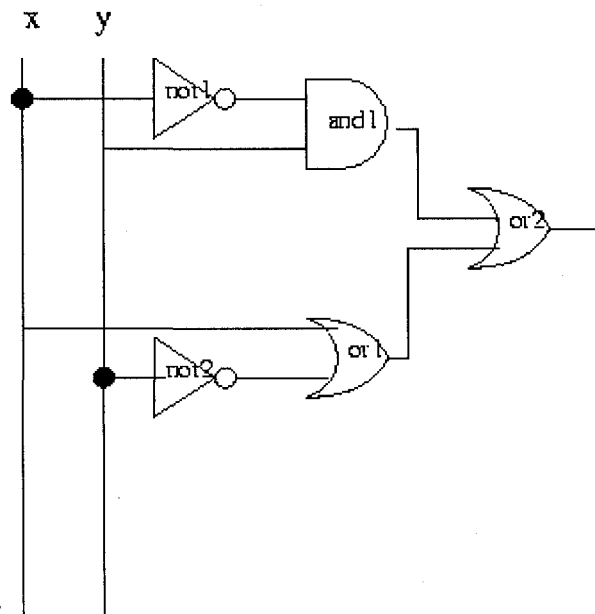
```
and<number> pin_number input
or<number> pin_number input
not<number> input
```

input is either a gate name or a signal name. pin_number is 1 or 2.

The last lines of the input file give the signal name and signal value.

Input to the simulation is a file that looks like this:

```
delay and 0.6
delay or 0.4
delay not 0.1
and1 1 not1
not1 x
or1 2 not2
or2 1 and1
not2 y
and1 2 y
or1 1 x
or2 2 or1
x 0
y 1
```



This input corresponds to the following circuit:

Notice that there is no ordering for gate inputs and output.

Simulation time begins at 0.0 nsec.

The output of the simulation consist of the output from each gate and the time at which the output appears on the gate. Notice that the or2 gate does not have an output until 1.1 nsec because it doesn't have all it's inputs until 0.7 nsec.

OUTPUT:

not1 1 after 0.1 nsec

not2 0 after 0.1 nsec

and1 1 after 0.7 nsec

or1 0 after 0.5 nsec

or2 1 after 1.1 nsec

Circuit output: 1 after 1.1 nsec

Problem 4: Colors in a Pixel

An 8-bit "truecolor" image stores the information about a pixel in a single 8 bit byte. A typical implementation uses 3 bits for red (R), 3 green (G) and 2 bits for blue (B), high- to low-order.

A simple way to filter some noise out of an image is to give each pixel a new set of colors, where each color (RGB) is computed as the average of that color in each of its 8 neighbors.

In the image below, the pixel at (5,4) denoted with the 'x' has neighbors denoted as 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'. The filter algorithm would simply add the values of the red (R) component in the 9 pixels and divide by 9. It would do this for the green (G) and the blue (B) components as well. Then x's color would be assigned these new values. Note that this can not be done "in place" because we can't change x's values until we've used its component values to alter its neighbors' values.

	0	1	2	3	4	5	6	7
0	o	o	o	o	o	o	o	o
1	o	o	o	o	o	o	o	o
2	o	o	o	o	o	o	o	o
3	o	o	o	o	a	b	c	o
4	o	o	o	o	d	x	e	o
5	o	o	o	o	f	g	h	o
6	o	o	o	o	o	o	o	o
7	o	o	o	o	o	o	o	o

For a pixel in the corner of an image there are only 3 neighbors to be averaged into the pixel's value. For other edge pixels there are 5 neighbors. The problem is to write a filter like that described above.

Input will be an unknown number (N) of lines of ASCII text, each of the same length. Each character represents one byte in the image; each line is the width of the image. The new line character at the end of a line is not part of the data and should be discarded before processing.

Output should be the N lines with each character as altered by the algorithm. Output should include a line feed at the end of each line. For example, given the input:

```
ABCDEF
GHIJKL
MNOPQR
```

The output would be:

```
AAEEEE
EEEEII
IIIIIM
```

For an input of identical bytes:

```
AAAAAAAAAAAA
AAAAAAAAAAAA
AAAAAAAAAAAA
```

The output would be identical to the input:

```
AAAAAAAAAAAA
AAAAAAAAAAAA
AAAAAAAAAAAA
```

Problem 5: Just A Love Matching

Clients of your on-line match-making service list their preferences. Each member of group X is to be matched up with a member of group Y with at least one matching preference.

For example, if group X contains Joe, Jim, Fred and group Y contains Jane, Bette, Tina, and we have the following preferences:

```
Joe: horses, long walks, instant messaging
Jim: horses, cars, art
Fred: carpentry, computers

Jane: fishing, carpentry, horses
Bette: shopping, long walks
Tina: horses, art
```

the only match possible is

```
Fred - Jane
Jim - Tina
Joe - Bette
```

The problem to be solved is, given the two lists of user names and preferences, print out a set of matches, if one exists. If no match exists, print out "no match". If more than one exists, print out an arbitrary match. Input will have the form as given in the example. The first N names are group X and the second N names constitute group Y. There is no blank line between the groups.

```
{Name 1}: {comma separated list of preferences}
{Name 2}: {comma separated list of preferences}
:
{Name N}: {comma separated list of preferences}
{Name N+1}: {comma separated list of preferences}
{Name N+2}: {comma separated list of preferences}
:
{Name 2N}: {comma separated list of preferences}
```

Output should be the matching names separated as above, by space dash space, printed in alphabetical order of the names in the first group.

Problem 6: Packing Parity

A simple communications protocol breaks up text messages into "packets" of fixed length for transmission. Each packet consists of a source name, destination name, message data, and an even parity bit. The parity bit is computed by counting the number of '1' bits in the ASCII character codes used in the source and destination names, and the message data. If the count is even, the parity is even and the bit is set to '0'. If the parity is odd, the bit is set to '1'.

Your program takes as input a single line containing the source name, destination name, message, and packet length (in bytes), where each field is separated by "|" (The "|" character never appears as part of a field). The output is a series of lines that adhere to the packet format and packet length boundary. Messages that do not completely fill all packets are padded with training spaces such that all packets are the same size.

Example

Input

```
Jim|Cathy|Here's my message to you|16
```

Output

```
Jim|Cathy|Here's m|0 1  
Jim|Cathy|y messag|0 1  
Jim|Cathy|e to you|0
```