Twenty-Second Annual
Willamette University—TechStart High School Programming Contest

Saturday 8 March 2008

Contest Problems

1. Goldilocks and the binary digits

Take an integer `n` and convert it into base two (i.e., binary form, as 1s and 0s with *no leading 0s*). Now compare the number of 1s to the number of 0s: imagine that the 1s are heavy, like water, and the 0s are light, like bubbles of carbonation. If the binary representation has more 1s than 0s, it is "`Too heavy`"; if it has more 0s than 1s, it is "`Too light`"; and if it has the same number of each, it is "`Just right`".

Your program should read in positive integers (between one and one billion, inclusive) and report whether they are too heavy, too light or just right. For example, on input `292` your program should produce the output "`Too light`" since the binary representation of `292` is `100100100`.

2. Word order

Write a program that will read in a line of text and print all the words in decreasing order of length. ***If two words happen to have the same length, print them out in the same order they appeared in the original text.*** The "words" will be separated by spaces, ***but include any adjacent punctuation*** (so anything that is not a space is part of the word; no tabs or newline characters will be in the input, except a newline for the end of the line). For example, if the input text is:

```
Print words in decreasing order of length (punctuation is part of a word).
```

Your program should give this output (or you may print the "words" one per line instead, if you like):

```
(punctuation decreasing length word). Print words order part in of is of a
```

3. Spinning sequences

Given a sequence of non-negative integers, we can "rotate" it by shifting it to the right, wrapping the rightmost numbers around to come in on the left side. How many places should we shift it by? Let's use the first (leftmost) number in the sequence as the "control number".

Your program should read in a sequence of numbers and tell how many times the sequence can be rotated like this before it yields a repetition, i.e., a sequence which has *appeared before anywhere in the process.* For example, the following sequence rotates 4 times before returning to a previously seen order:

```
        5 2 3 1 7 0 2

=>      3 1 7 0 2 5 2

=>      2 5 2 3 1 7 0

=>      7 0 2 5 2 3 1

=>      7 0 2 5 2 3 1
```

If the first value in a sequence is 0, we will assume that it must still be rotated once, by that zero value, before returning to its original state (and the same for any other "self-generating" sequence).

4. Digital roots

The *digital root* of a (non-negative) integer is the number you get by repeatedly adding up the digits until only a single digit remains. For example, for the number 974685:

9 + 7 + 4 + 6 + 8 + 5 = 39

3 + 9 = 12

1 + 2 = 3

So, the digital root of 974685 is 3 and it took us 3 steps to find it.

Write a program which will read in a positive integer and write out *both* its digital root *and* the number of steps required to get it. Note that zero is a possible input (it's non-negative): it counts as one digit, and one-digit inputs always count as taking zero steps, since no addition is required.

5. Anti-sorting!

Sorting is a common problem and sorting programs are a common assignment in programming classes. So teachers often need data which is specifically *not* in sorted order, to use when testing student programs. Your teacher has asked you to write an "anti-sorting" program which will jumble up its input in a very specific way. The user enters a series of integers (in an input box, or on a line, separated by spaces) in arbitrary order. The program outputs the largest number first, then the smallest, then the second largest, then the second smallest, etc., through all the numbers from the input. As an added feature, *any number that is input more than once should be reported only once* (this prevents teachers from making tricky data sets where the same number occurs twice!). Notice that it's the numeric magnitude of the numbers that counts here.

For example, if the input is

```
13   532   67   -67   0   -8   9   67   67
```

your output should be

```
532   -67   67   -8   13   0   9
```

6. Cipher seizure

The fine people at Homeland Security need your help in deciphering some coded messages they intercepted, sent by two suspected terrorist organizations, the EFF and the ACLU. We know that the messages were coded with a so-called *caesar cipher*, in which the letters of the alphabet are shifted by a fixed amount, with the end of the alphabet "rotating" back through to the front. For example, if the shift amount is 5, then the letters "`ABC`" would become "`FGH`" and the letters "`XYZ`" would become "`CDE`". In order to *decode* a message with a known shift factor, you merely shift or rotate the code letters *backward* by the same amount.

Unfortunately, all the Homeland Security agents have is the coded message and a certain *keyword* which they think the message contains. Your program should look for the keyword among all possible decodings (reverse shiftings) of the coded message, showing the corresponding decoded message whenever it finds a match. For example, if the keyword is "`HAL`", and the coded message contains the word "`IBM`", then a shift factor of 1 is a possibility, so you would print out the whole message shifted backward by 1.

However, note that you might find the keyword in several different versions of the decoded message, corresponding to different shift amounts. For example, if the keyword was "`OH`" and the coded message had both "`MF`" and "`AT`" in it, then the correct shift amount might be either 24 or 12, since "`OH`" shifted by 24 gives "`MF`", but "`OH`" shifted by 12 gives "`AT`". If you find different possible decodings of this kind, you should print all of them out: the agents will then manually inspect the decoded messages to determine which one is likely to be the actual intended one. If the keyword *cannot* be matched in the message, say so.

Note that a zero shift factor is possible: some of these terrorists are not so bright! All inputs will be words in uppercase separated by spaces, with no punctuation.

For example, if the input is:

```
coded message: AT FTQ DMHQZ RXUQE MF PMIZ
keyword:  OH
```

your program should report something like this:

```
if message was shifted by 24, decoding = CV HVS FOJSB TZWSG OH ROKB
if message was shifted by 12, decoding = OH THE RAVEN FLIES AT DAWN
```

7. Word ladder

A *word ladder* is a sequence of words where each word can be gotten from the last one by the use of one of a few simple rules. The possible rules for changing one word into another are:

- add a single letter;
- remove a single letter;
- change a single letter;
- or rearrange the order of the letters (i.e., make an anagram).

The idea of word ladders comes from a game invented by Lewis Carroll, author of *Alice in Wonderland*. In the original game, part of the fun is that each word in the sequence has to be a real English word. We don't want your program to have to use a dictionary, so *we* will worry about whether the words are real or not: your job is to write a program that reads in a sequence of words (in a GUI, or in a command line until a period is read) and decides whether or not they can be arranged into a word ladder. If they can, you should print out a list of the words along with the type of move. For example, if your program sees these words:

```
trap bat strap part tap cat tarp tab parts .
```

it should return something like this:

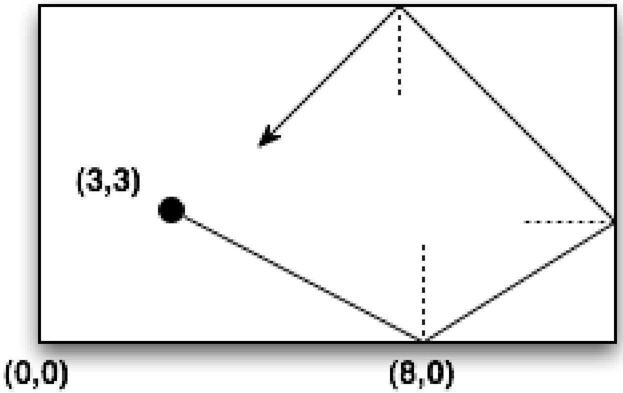
```
Successful word ladder: cat bat tab tap tarp part parts strap trap
```

(Notice that the order of the input words is *not the same* as the output: your program has to *find* the order.) If there is no way to rearrange the words into a word ladder, print out a message saying so.

8. Nights of the rectangular table

Art and his buddy Lance spend a lot of nights down at the *Camelot Bar and Grill* playing pool, billiards and similar games. The Camelot has an unusual practice table with no pockets: this makes for an excellent place to try out those tricky shots where the ball bounces around several times off the bumpers (the "walls" of the table). Art would like you to write a program to help him plan out these tricky shots: he'll supply a starting point for the ball, a "target" point on one of the walls and a *total travel distance* for the ball (based on how hard he hits it with the cue stick). Your program should read these values and tell where the ball will end up, in terms of (x,y) coordinates, after it rolls toward the target and bounces, possibly several times, off the walls. Each time the ball hits a wall, it bounces away at the same angle to the wall, but in the opposite direction. You will need to compute and accumulate the distance the ball travels on each "leg" of its journey to determine where it will stop. Note that the ball might hit a wall "straight on", in which case it bounces straight back. We will not use test data where the ball hits a corner exactly to avoid this special case.

Your program should read in an initial ball position as a point, a target point on one of the walls, and a value representing the total distance the ball will travel. It should report the final resting place of the ball when it stops, as a point. Each point will consist of a pair of decimal (floating point) values. We will assume that the table is 12 units wide (x direction) and 6 units tall (y direction) as shown in the following diagram.



9. Cloud hopper

Even little froggies go to heaven … if they can only find the way! Our little frog spirit awakens after an ill-timed road crossing (*Frogger,* anyone?) to find himself floating on a cloud. He looks up plaintively to see the lush lilly-pads and plentiful flies on the fabled Cloud Nine nearby. Between him and Cloud Nine are a number of other clouds, floating at various altitudes in the sunny skies of the amphibian afterlife. Froggy can jump only a certain distance with each leap, but if he can jump from one cloud to the next, in the right order, perhaps he can find his way to froggy bliss on Cloud Nine.

Your program will read in the three-dimensional coordinates of a bunch of clouds: the first one is our froggy's starting cloud and the last is his goal, Cloud Nine. Our coordinates will always be integers, for simplicity, but you will want to work with non-integer distances. (The distance between two points is the square root of the sum of the squares of the differences between coordinates in each dimension.) Froggy can jump up to *seven* units of distance in one jump, and no further (so the next cloud in each jump must be at least that close). Your program should print out *any* successful path for froggy as a list of cloud coordinates and distances between them (to verify that he can make the jump). *Of course the order will usually be different than the order they happen to be read in.* If there is no possible path between clouds, print out an appropriately sad message to this effect. (For simplicity's sake, we are assuming that our froggy must jump from the center of one cloud to the center of the next: that is, our clouds are just points, not 3-dimensional shapes … and thank goodness for that!)

Here is an example set of cloud coordinate inputs:

```
0 0 0;  6 10 1;  3 7 3;  0 4 2;  3 10 0;  4 0 2;  11 10 2
```

and a corresponding set of outputs (this isn't the only solution, but you only need to find one):

```
from (0, 0, 0)  to (0, 4, 2)   (distance = 4.47)
from (0, 4, 2)  to (3, 7, 3)   (distance = 4.35)
from (3, 7, 3)  to (6, 10, 1)  (distance = 4.69)
from (6, 10, 1) to (11, 10, 2) (distance = 5.09)
```

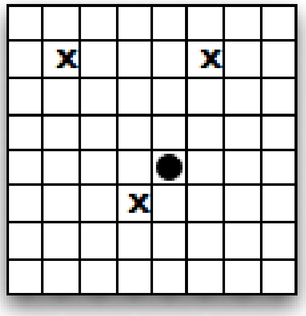
10. Checker challenge

A checkerboard has eight rows and columns of squares. Let's number the squares so that the lower left one is square (1,1) and the upper right is square (8,8). Checkers can move diagonally forward, to the left or to the right, and they are not allowed to move onto a square that is already occupied by another checker. (Our checkers will not do any jumps, just plain moves.) So a checker starting at some point on the board will have several *paths* it can take to reach the other side of the board (i.e., row 8): no path is allowed to go through an occupied square, but there might still be a lot of options available. We can represent each path as a sequence of moves, "`L`" or "`R`", for diagonally forward to the left or right, on each turn.

Your program should read in the "coordinates" of a starting square and some occupied squares of a board, and should print out a list of all "paths" that can be taken by a checker. You should report if there is no path or if the piece is already on the last row. The input will consist of one line containing the coordinates of the starting square, then several lines each with the coordinates (column or *x*, then row or *y*) of the occupied squares, as two integers. If you need it, you may request a line with `0 0` to indicate end of data.

Sample input (in the picture, xs mark occupied squares and our checker is the black dot):

```
5   4
4   3
2   7
6   7
0   0
```



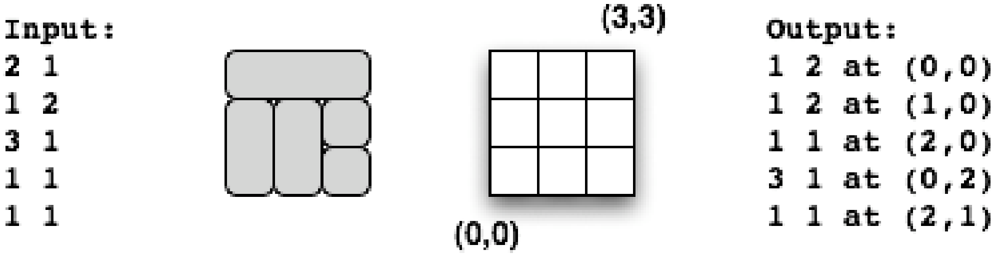
There are seven possible paths to the last row, so your program should show:

```
LLRL
LLRR
LRLL
LRLR
RLLL
RLLR
RRRL
```

11. Tile assembly

Say we have a bunch of tiles of various sizes, but always rectangles, and always sized as integer multiples of some common unit (e.g., an inch). Can the tiles be arranged into a perfect square, with no "holes", no ragged edges and no tiles left over? The tiles may be turned to fit, if needed, but should align to a grid.

Your program should read in a sequence of pairs of positive integers: each pair represents the size of one tile. When all the tiles are read in, you should tell *whether or not they can be arranged to make a square; if they can, you should also tell how.* To report how, give the orientation of each tile as a pair of integers, width first, then height, along with the "coordinates" of the lower left-hand corner where the tile should be placed. For example, here is a possible input, a sample arrangement (not the only one, of course) and an output:



```
Input:
2 1
1 2
3 1
1 1
1 1
```

```
Output:
1 2 at (0,0)
1 2 at (1,0)
1 1 at (2,0)
3 1 at (0,2)
1 1 at (2,1)
```

As an example of an input with no solution, consider this set of sizes: 3×3; 1×4; 1×4; 2×2; and 2×2. Note that the total "area" is 25, and no tile is longer than 5 units, but these tiles cannot be arranged into a 5×5 square. *Our inputs may include tiles whose total "area" is not a square, or ones where some tiles are just too long:* you might want your program to check for these possibilities and report "No solution" before starting to search for one.

12. Block assembly

Consider a similar problem as the *Tile assembly* above, but now with 3-dimensional blocks. Each block will be described by *three* positive integers, giving its size in each of three dimensions. Now the blocks should be arranged to make a solid cube, with no "hollow" parts, ragged edges or blocks left over. Note that blocks can now be turned more ways than just horizontally and vertically. (How many different orientations are possible, while still aligned on the grid?) Your output should be similar to that for the tile assembly, but with three numbers for the orientation of the block and 3 coordinates for position. As before, you may want to check some basic "feasibility" conditions before you begin to search.

[Discussion idea for *after* the contest: can you write *one* program that solves both problems, just by changing a constant "2" to "3" somewhere in the program? I think it might be possible in some languages!]

13. Polynomial parsing

Everybody who does programming uses a little algebra: after all, we need variables and arithmetic to get the job done. Sometimes we end up with some pretty big formulas, or *expressions,* in our code: variables, constants, arithmetic operators, and lots of parentheses. Time to simplify!

Your program should read in a large expression, as described below, and "reduce" it down to a simple *polynomial,* i.e., something like $5x^2+3x-7$. In fact, since we don't want you to have to print out all those exponents, let's just write out what are called the *coefficients* of the polynomial. For our example, that would be: `5 3 -7`. We'll be able to tell which exponents go where by the length of the list: this one must start at x^2 because it has 3 coefficients. For this reason, you should be sure to write out any zero coefficients, *but don't write out any leading zeros* (just like for numbers: we write 103 and 256, but not 00256).

The expressions you read in will consist of constant numbers (like 3 or 0, but never negative), the variable x (possibly several times), and the arithmetic operators +, – and * (that's multiplication): no division! Every use of an operator will have exactly two arguments and will be surrounded by parentheses, like (2+3). But, the arguments may themselves be other expressions; so, in general, you might see something like this:

```
((((x*x)+(3*x))-((8*x)*((6-x)*2)))*(x-1))
```

You could simplify this expression by hand like this (of course, your program might use another method):

```
((((x*x)+(3*x))-((8*x)*((6-x)*2)))*(x-1))
```

$$= (((x^2+3x)-(8x*(12-2x)))*(x-1))$$

$$= (((x^2+3x)-(96x-16x^2))*(x-1))$$

$$= (17x^2-93x) * (x-1)$$

$$= (17x^3-93x^2) - (17x^2-93x)$$

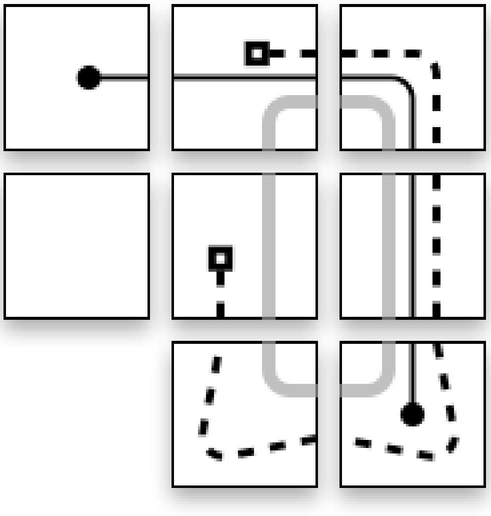
$$= 17x^3 -110x^2 +93x +0$$

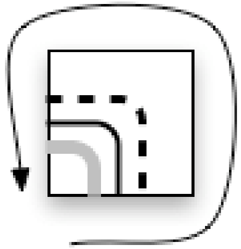
So, if your program read the original expression above, it would print out: `17 -110 93 0`.

14. Rivers, roads and rails

There is a popular game (published by **Ravensburger**) called *Rivers, Roads and Rails:* it consists mainly of square tiles with various patterns of rivers, roads and rail lines painted on them. Typical tiles might have one each of the three lines coming in on one edge and leaving on the opposite edge, or perhaps turning a corner and coming out an adjacent edge. Not every tile has all three line styles (river, road and rail), and some even have patterns where a particular line enters the tile on one edge, but does not leave on another (usually the picture on the tile then shows a lake, parking lot or train station where the line stops mid-tile).

The goal of the game is for players to put down tiles so that their edges match with each other, and so that all of the tiles are used up. There are some other rules for play between several players, but we will simplify the problem to this: given a set of tiles, can you arrange them on a grid so that all the edges match and are "used up", i.e., no "dangling" lines are available on any given edge? For example, here is a picture of a good arrangement of tiles (the grey lines are rivers, the solid lines are roads and the dashed lines are rails):





Note that all edges of every tile that have a river, road or rail are matched up to an adjacent tile. Also notice that we have a blank tile: it must be placed, but can be put anywhere that its (blank) edges match up with blank edges of adjacent tiles.

All the tiles should be connected into one large group: two or more groups, disconnected, is not allowed! The *order of the lines matters,* but we will assume that the actual physical alignment always works out for tiles with matching edges. "Donut" arrangements (a hole in the middle) are OK, as long as edges match.

The data you will read will be a sequence of tile descriptions; each one will consist of four groups of letters to describe the four sides of a tile. We will use the letters "`R`", "`D`" and "`L`" for river, road and rail, respectively (these are the final letters, since the initials are all "`R`"). The sides of a tile will be separated by commas, and each tile will be separated from the next by a period. If a side of a tile is completely blank, we will use a "`B`" just for clarity. So a description of the full set of tiles on the left above might look like this:

```
B,D,B,B.  R,RDL,B,D.  RDL,B,B,LDR.  B,B,B,B.  LR,B,R,B.  RDL,B,LDR,B.
B,LR,RL,B.  B,B,LDR,RL.
```

Your output should tell where each tile is placed *and how it is turned.* Use integer coordinates of tile size to tell the lower left-hand corner of the tile, and then describe the edges of the tile as above: that way, if you have had to rotate the tile, it will be easy to tell what this means for the edges. Tiles can be given in any order, but the descriptions should match (all) the tiles you read in. The above picture might start like this:

```
(1,0): B,LR,RL,B.   (2,0): B,B,LDR,RL.   (2,2): RDL,B,B,LDR.
(0,1): B,B,B,B.     (1,1): LR,B,R,B.     ... etc., for all tiles ...
```

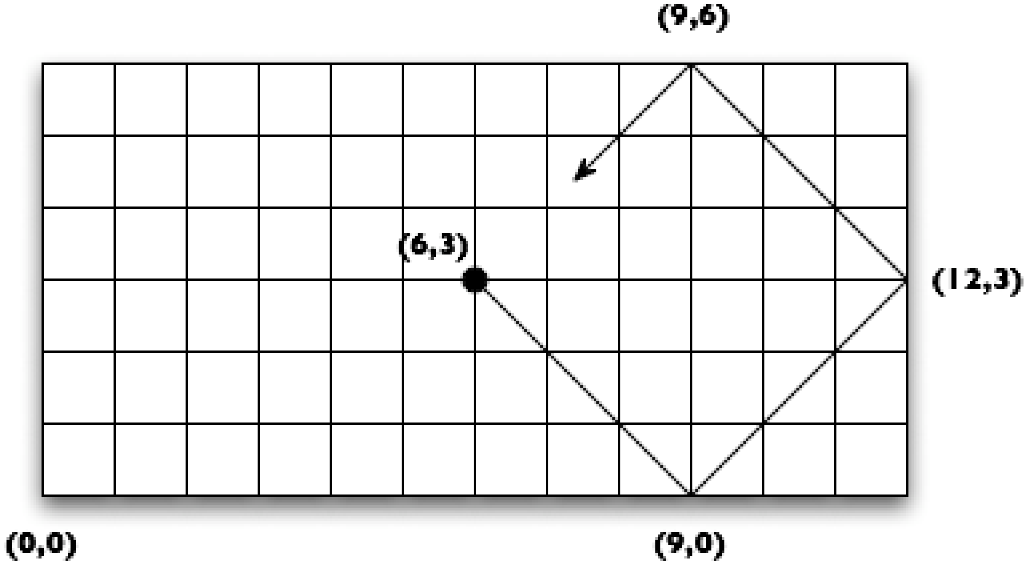
8. Nights of the rectangular table—CLARIFICATION AND EXAMPLE!

This sheet provides a worked out example and some clarification about input and output for problem #8.

Say that we place a ball at position (6,3) on the table (the midpoint of the whole table) and aim at point (9,0) (thus on the bottom wall) with a force sufficient to drive the ball 15 units.

These values conveniently give 45° angles, which won't happen all the time, but they make it easier to verify the bounce points and distances—your program will need to handle more complex situations, but a little algebra should suffice to program it.

So our situation looks like this:



Our input data would be as follows (just the numbers, not the side comments):

```
6.0 3.0          — start position
9.0 0.0          — target position
15.0             — cue stick force, in units traveled
```

In order to facilitate scoring, let's require that your program print out each impact/bounce point (hint: these will always be on a wall: what does this say about their coordinates?), as well as the accumulated distance traveled so far. Then we might see something like this:

```
from: 6.0 3.0  to:  9.0   3.0     (total distance so far: 4.243)
from: 9.0 0.0  to: 12.0   3.0     (total distance so far: 8.486)
from: 12.0 3.0 to:  9.0   6.0     (total distance so far: 12.729)
from: 9.0 6.0  to:  7.394 4.394   (total distance so far: 15.000)
final position: 7.394 4.394
```

This format should give you some ideas about how to proceed in your programming. Note that the first three "legs" of the journey have lengths equal to three times the square root of 2 (why?); and note that the horizontal and vertical distance from the last wall position are proportional to the square root of two and the final leg length (again, why?). If you can generalize these ideas to different angles, you should be able to solve the problem!