

ND Programming Contest, Spring 2008

ND ACM Computer Club

Saturday, April 5, 2008

#1: Tri it – it’s fun.

The Problem

The Alien Lord Gleepzorp uttered thus:

Earth was invaded by aliens, and we aliens won. Deal with it. Among the most disruptive of our alien concepts (even worse than appointing Britney Spears as dictator of Earth) is our exclusive use of trinary (base 3) arithmetic. All of your Earthling-built computers were gathered up and ground into powder (we think it’s delicious and we sprinkle it on our toast).

Our computer corporation (Alienware, Inc. – Michael Dell is one of us in a great make-up job) has hired you to implement a fixed-point arithmetic-logic unit for the first tri-valued-logic CPU. As a first step, you are to write a simulator for a few of the math operations. Be aware that we aliens use the “balanced ternary” notation, where each ternary digit (a “trit”) has the value +1, 0 or –1 (these values are denoted 1, 0 and 1 respectively). Write a program that will add or subtract two integer ternary numbers of six digits or less. You may assume that no operand to any test case will exceed this number of trits in length. To spare your poor human brain from worrying about how to format the input to the adder/subtractor, you should assume that the input arguments are formatted as character strings, with “+” representing 1, “-” representing 1, and “0” representing 0.

Example: the balanced ternary number 1101 (formatted as “+-0+”) converts to the following form in your pathetic Earthling decimal system, with fractional notation:

$$1 \times 3^3 + (-1) \times 3^2 + 1 \times 3^0 = 27 - 9 + 1 = 19$$

Thus endeth the speech of Lord Gleepzorp.

The human editor adds: your program’s input will consist of a single integer (regular base-10), followed by a newline; this integer n indicates the number of test cases. Following this line will be n lines, each of which has a balanced ternary number, a space, a character “+” or “-”, another space, and then a second balanced ternary number (that is, the line is in infix notation). Given this test case, your program should write the result to standard output in balanced ternary notation, one result per line. There should be no leading zeroes in your output.

Sample Input

```
2
+-0+ + -+0-
+-0+--+ - 0000--
```

Sample Output

```
0
+-0+00
```

Problem contributed by Prof. Patrick J. Flynn

#2: When Network is Notwork

The Problem

OIT workers came in this morning to discover a disaster: a horde of burrowing hamsters, released the previous night on South Quad by several students in search of excitement, infiltrated the steam tunnel system and severed all fiber lines to dorm ResNet switches. There is only so much spare fiber in the South Bend metropolitan area; workers must therefore order more, but they want to order the minimum amount needed. Your job is, given a description of the network (in the form of a graph, where each edge represents a fiber link and is labeled with the distance), calculate the minimum length of fiber required so that a connection is possible from any node to any other (that is, so that the graph is fully connected).

Hint: This is a Minimum Spanning Tree problem; it helps to be greedy.

Your program will read a number N , indicating the number of test cases. For each test case, read a number M , indicating the number of nodes in the graph; these nodes will be known by numbers 0 to $M - 1$. Then read a number E , indicating the number of graph edges; there will follow E lines, each of the format $n_1 n_2 l$, where n_1 and n_2 are the node IDs and l is the edge length. Produce one line of output: the minimum cost to build a spanning tree.

You may assume no more than 100 nodes, and no more than 100 edges.

Sample Input

```
2          // 2 test cases
3          // test case 1: 3 nodes
3          // test case 1: 3 edges
0 1 10     // node 0 to node 1: length 10
1 2 10     // node 1 to node 2: length 10
2 0 50     // node 2 to node 0: length 50
4
5
0 1 10
1 2 10
2 3 10
3 0 10
0 2 5
```

Sample Output

```
20        // minimum spanning tree: 0 <-> 1, 1 <-> 2
25        // minimum: 0 <-> 1, 0 <-> 2, 2 <-> 3
```

#3: Forward Irish Notation

The Problem

Reverse Polish Notation became popular with HP calculators in the 1970s. The notation, used to enter expressions on pocket calculators, is postfix: all arguments to an operator (such as addition) come first, and then the symbol for that operator. So, for example, $2 + 2$ becomes “2 2 +” and $3 * (1 + 2)$ becomes “3 1 2 +*”. This is easily evaluated using a stack: when a number is read, push it to the stack, and when an operator is read, pop the top two values off the stack, perform the operation, then push the result back to the stack.

Little-known at the time, but recently unearthed among some old files, was a grassroots effort among Notre Dame EE students in the mid-1970s to replace the odd RPN with a more appropriate and localized notation: Forward Irish Notation. Where RPN says “2 2 +”, FIN says “+ 2 2”; similarly, “3 1 2 +*” becomes “* + 1 2 3”.

Your task is to implement a converter that reads RPN expressions and produces the equivalent FIN expressions. Read a line with a single value N , indicating the number of test cases; then, for each test case, read M , a token count, followed by M whitespace-delimited tokens. Produce the resulting expression with tokens delimited by single spaces, one expression per line.

You may assume that each expression consists of no more than 100 tokens. The tokens “+”, “-”, “*” and “/” are operators; the rest (literal numbers or variable symbols) should be interpreted as values.

Sample Input

```
2 // 2 expressions follow

7 // 7 tokens in first expression
1 2 + 3 * 4 / // ( (1 + 2) * 3 ) / 4

9 // 9 tokens in second expression
b b * 4 a c * * - // ( b^2 - 4*a*c )
```

Sample Output

```
/ * + 1 2 3 4
- * b b * 4 * a c
```

#4: My Easter is Easter Than Yours

The Problem

The date of Easter falls on “the first Sunday after the first full moon on or after the day of the vernal equinox,”¹ based on the ecclesiastical rather than astronomical full moon. This is all very well and good, but as a mere mortal without an intuitive understanding of lunar calendars, how are you to know when to celebrate your Easter holiday?

Never fear; many a medieval mathematician has toiled away to make your job solving this problem incredibly easy. This particular computation proved so important in earlier times that it is called simply Computus²; though it originally involved creating a large table procedurally, the Meeus/Jones/Butcher algorithm has been developed as a simple formulaic method. The date of Easter can be calculated as follows, given the year Y , where integer division $/$ leaves no fractional part:

```
a := Y mod 19 ; b := Y / 100 ; c := Y mod 100 ; d := b / 4 ; e := b mod 4
f := (b + 8) / 25
g := (b - f + 1) / 3
h := (19*a + b - d - g + 15) mod 30
i := c / 4 ; k := c mod 4
L := (32 + 2*e + 2*i - h - k) mod 7
m := (a + 11*h + 22*L) / 451

month := (h + L - 7*m + 114) / 31
day := ((h + L - 7*m + 114) mod 31) + 1
```

Your program should read a number N from standard input, then read N years (in 4-digit format); for each year, print the date of Easter as MM-DD on one line.

Sample Input

```
4
2008
2002
1993
2020
```

Sample Output

```
03-23
03-31
04-11
04-12
```

¹http://en.wikipedia.org/wiki/Easter#Date_of_Easter

²<http://en.wikipedia.org/wiki/Computus>

#5: State Your State, Please

The Problem

Professor Absen-Mindt (it's a German name, you are told) is teaching FSM 101 this semester, an introductory course on Finite State Machines, at Logic-Design University (LDU). He is about to give an exam in which he asks his students to execute several finite state machines on certain sets of input and name the final state of each machine; however, he has not taken the time to produce a solution set, and needs one urgently before he gives the test. He has enlisted your help: you are to write a program that will provide the solutions to each of his problems.

Your program will take a description of a Finite State Machine as input, along with an input stream. Fear not; the FSM is represented simply as a directed graph, and for simplicity's sake, every state (node) will have exactly two transitions (out-edges). The input stream is simply a stream of bits, and in each state, the machine reads one bit; depending on the bit, it will take one transition edge or the other. Your program should write the final state to standard output.

Read a number N from standard input; this is the number of test cases. For each test case, read a number S , indicating the number of states, which will be labeled with IDs 0 to $S - 1$. Then read S lines describing the S states, each of the form $S'_0 S'_1$, indicating that for the state described, a 0 in the input stream will cause a transition to state S'_0 and a 1 will cause a transition to S'_1 . Then, after these state descriptions, read one line, which will contain a string consisting of the characters "0" and "1"; start the FSM in state 0, and execute the FSM on this input stream. Print the final state to standard output.

You may assume no more than 100 states.

Sample Input

```
2 // 2 test cases
2 // first test: 2 states
0 1 // state 0: '0' -> state 0, '1' -> state 1
0 1 // state 1: '0' -> state 0, '1' -> state 1
010101 // input for test 1
3 // second test: 3 states
1 2 // state 0: '0' -> state 1, '1' -> state 2
2 0 // state 1: '0' -> state 2, '1' -> state 0
0 1 // state 2: '0' -> state 0, '1' -> state 1
111000001110 // input for test 2
```

Sample Output

```
1
0
```

#6: It's Spring-Time!

The Problem

Spring has come to South Bend; contemplating this one morning, Hooke's Law suddenly entered your mind and you had an irresistible urge to simulate systems of springs.

Hooke's Law states that a spring with spring-constant k , displaced (stretched or compressed) away from its rest length by distance x , will exert a force in the direction of its displacement of magnitude F , such that $F = -kx$.

Assume that a number of springs obeying Hooke's Law are anchored in a two-dimensional plane at given coordinates; the second endpoint of each spring is tied to a single shared 1-kg weight. Assume that each spring has a rest length of 0, so that x in Hooke's law above is simply the vector from the anchor point to the weight's location, and so the force can be obtained by scalar-vector multiplication of that vector difference. Assume also that Newtonian mechanics apply, that is, that $F = ma$.

Given the list of springs, anchor points, and spring constants, together with the initial coordinates of the weight, run a simple iterative simulation with timestep $dt = 0.01$ s. That is, at every iteration, compute the force as the sum of force due to each spring; then compute the acceleration by the law given above; then add this, times dt , to velocity (which starts at 0); then add this, times dt , to position (with initial value given). Run this simulation for 1 s, or 100 iterations, and report the position of the central mass at this time.

Your program should read N , the number of test cases; then, for each test case, M , the number of springs. For each spring, read a line $x\ y\ k$, giving the x- and y-coordinates of the spring's anchor point and the spring constant. Finally, read a line $x_0\ y_0$, giving the initial coordinates of the central mass. Print a line $x_f\ y_f$ to standard output after running the simulation as described above. Round the result to 3 places after the decimal point.

You may assume no more than 100 springs.

Sample Input

```
1          // 1 test case
4          // test case 1: 4 springs
-1 0 15    // spring 1: anchored at (-1, 0), k = 10
0 1 15     // ...
1 0 15     // ...
0 -1 15    // spring 4
0.5 0.5    // initial position
```

Sample Output

```
0.034 0.034
```