The Tree Of Life

This challenge asks you to write a program which computes cellular automata state in a non-standard world. Instead of each cell living on a grid it lives on a binary tree.

Overview

A Cellular Automaton (CA) is a set of *cells* which evolve together in stages. At each stage, a given cell changes according to a given *rule* which depends upon values of cells in its *neighborhood*.

Commonly, CAs consist of a *grid* of cells so that each cell has a neighborhood of 8 cells.

```
1---2---3
|   |   |
4---X---5
|   |   |
6---7---8
```

The dynamics of a CA on a grid is thus determined by a way of mapping all 9 of the values near `X` to a new value which `X` takes in the next iteration.

```
rule(v(1), ..., v(8), v(X)) = ...
```

An example grid CA is Conway's *Game of Life* which stores boolean values at each cell and has the following update rule:

```
Consider the sum N = v(1) + ... + v(8).
If v(X) then
    If N < 2 : return False
    If N > 3 : return False
    otherwise : return True
else
    If N == 3 : return True
```

A Game is played by picking an initial state of the grid and updating each cell according to the given rule repeatedly. Conway's rule produces startlingly interesting behavior for many initial states—in fact, given an infinite grid there exist starting states which compute an encoding of any computable function. In other words, the Game of Life is *Turing complete*.

Rule encoding

CA update rules can be complex, but oftentimes they are serialized to a simple format.

Most of the time there are only finitely many possible CA rules. Rules over binary-valued cells are usually assigned numbers by extending an ordering on the neighborhood of cells. For example, if we consider a linear automaton then the neighborhood of a cell looks like

```
1---X---2
```

a rule takes the form

```
(v(1), v(X), v(2)) -> {0, 1}
```

and so if we assign an order to the input space we can write any such rule as a binary number

```
111 110 101 100 011 010 001 000
-------------------------------
 0   0   0   1   1   1   1   0   "Rule 30"
 0   1   1   0   1   1   1   0   "Rule 110"
```

Clearly there are 256 such rules and so we can read any 8-bit number as a rule specification for a linear CA over binary values.

Problem statement

Instead of computing a CA on a grid, we'll compute a CA on a binary tree. The formulation of CAs above does not depend upon arranging the cells in a grid—we can pick any topology so long as there's a notion of the *neighborhood* of a cell.

So the task is to build a CA evaluator on binary trees. A conformant program should take a serialized binary tree and a rule (encoded as an integer) as input. Then, given a sequence of counts of iteration steps and associated "paths" into the tree the evaluator must *interactively* print out the binary value stored in the tree at the given path after the specified number of steps.

Importantly, while paths are guaranteed to always be valid, step counts are *not guaranteed* to always be *positive*. We may ask you travel back in time---though never beyond the beginning of time.

1. Parsing trees

The initial state of the binary tree automaton is a non-empty binary tree. A conformant program should parse its initial state from a format like as follows

```
X            --> A leaf containing a single "on" cell

.            --> A leaf containing a single "off" cell

(X . .)       --> A branch holding an "off" with
                     An "on" leaf as the left subtree
                     An "off" leaf as the right subtree

(X . (X . .))  --> A branch with an "off" cell with
                     An "on" leaf as the left subtree
                     The prior example as the right subtree
```

2. Parsing rules

The behavior of the automaton is governed by a choice of update rule operating on the 4-cell (unbiased) neighborhood of each cell. There are thus 2^(2^4) = 2^16 = 65536 possible rules and a conformant program must determine the rule in force by evaluating a given 16 bit code given as an unsigned integer.

In pictures, the (maximal) neighborhood of a cell in a binary tree looks like and is numbered like

```
   (1)
    |
   (3)
  /   \
(2)   (4)
```

and we'll encode each rule as a big-endian 16-bit word beginning with the big encoding the "all on neighborhood"

```
1111 1110 1101 1100 1011 1010 1001 1000 ...
_   _   _   _   _   _   _   _
```

Here are a few examples:

```
Rule 7710: (An analogue of Rule 30)
 (X X X X)  -->  .
 (X X X .)  -->  .
 (X X . X)  -->  .
 (X X . .)  -->  X
 (X . X X)  -->  X
 (X . X .)  -->  X
 (X . . X)  -->  X
 (X . . .)  -->  .
 (. X X X)  -->  .
 (. X X .)  -->  .
 (. X . X)  -->  .
 (. X . .)  -->  X
 (. . X X)  -->  X
 (. . X .)  -->  X
 (. . . X)  -->  X
 (. . . .)  -->  .

Rule 42354:
 (X X X X)  -->  X
 (X X X .)  -->  .
 (X X . X)  -->  X
 (X X . .)  -->  .
 (X . X X)  -->  .
 (X . X .)  -->  X
 (X . . X)  -->  .
 (X . . .)  -->  X
 (. X X X)  -->  .
 (. X X .)  -->  X
 (. X . X)  -->  X
 (. X . .)  -->  X
 (. . X X)  -->  .
 (. . X .)  -->  .
 (. . . X)  -->  X
 (. . . .)  -->  .
```

3. Parsing Paths

Paths are pointers into a tree. The simplest path is the empty path which points to the root value of the tree. From there, they are merely sequences of directions, left (denoted by the single character `<`) and right (denoted by `>`). Some example paths are

```
[<><<<><>>]
[>>><<><]
[]
[><<<<<<<<<>]
```

where the third example was the empty path.

Full input specification

A conformant program works as follows. First, it reads two lines setting up the initial state: a rule integer and a serialized binary tree. The next line contains a natural number, *n (1 <= n <= 1100)*, indicating the number of queries that will be asked.

Then, it begins to execute the *n* queries. It reads a single line indicating the first query and must immediately print out the value stored at the path in the tree after the indicated number of rule iterations followed by a newline. For cells which are "on", emit a `X` and for those which are "off" emit a `.`. This repeats *n-1* more times.

An example transcript follows where lines sent from the server are preceeded by `>` and lines returned from the client are preceeded by `<`.

```
> 42354
> ((. X (. . .)) . (X . (. X X)))
> 6
> 0 []
< .
> 2 [><]
< X
> 0 [><]
< X
> 0 [<>]
< .
> 1 [><]
< X
> 0 [<>]
< X
```

For rule `r`, number of cases `n`, and step increments `s` we have `0 <= r <= 65535`, `1 <= n <= 1100`, and `-1000 <= s <= 1000`.

To see a bit of what's going on behind the scenes, here are snapshots of the first 5 tree states at times [0, 4]

```
0 -- ((. X (. . .)) . (X . (. X X)))
1 -- ((X . (. X .)) X (. X (X . X)))
2 -- ((. X (X . X)) . (X X (. X .)))
3 -- ((X . (. X .)) X (X . (X X X)))
4 -- ((. X (X . X)) . (. X (X . X)))
```

Sample Input

```
42354
((. X (. . .)) . (X . (. X X)))
6
0 []
2 [><]
0 [><]
0 [<>]
1 [><]
0 [<>]
```

Sample Output

```
.
X
X
.
X
X
```

Tested by Javran (Fang) Cheng