COS 126 Programming Exam 2 Fall 2019

Instructions. You will have 50 minutes to create and submit one or two programs and a `README` file. Download the project zip file, which includes all the files you will need, from the **`Exams`** page. Do not start (even by reading the next page) until you are instructed to do so.

Resources. You may use your book, your notes, your code from programming assignments and precepts,, the code on the COS 126 course website, the booksite, and you may read Piazza. No form of communication is permitted (e.g., talking, texting, etc.) during the exam, except with course staff.

Submissions. Submit your work using the `Submit!` link on the **`Exams`** page.

Grading. Your program will be graded on correctness, clarity (including comments), design, and efficiency. You will lose a substantial number of points if your program does not compile or if it crashes on typical inputs.

Discussing this exam. Due to travel for extracurriculars and sports, some of your peers will take this exam next week. Do not discuss exam contents with anyone who has not taken the exam.

Honor code pledge. You *must* "electronically sign" the honor code in your `README` file by retyping the pledge and then your name.

Programming Exam 2: Cellular Automata

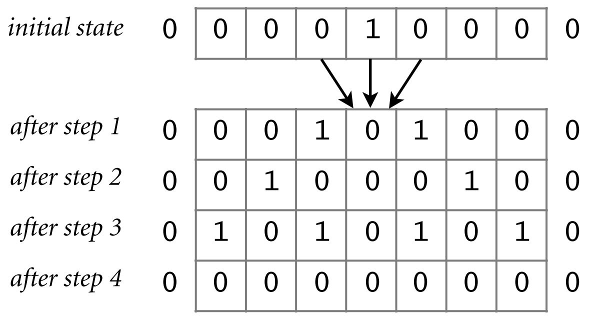
Description. You may have heard of Conway's *game of life*, which is a two-dimensional cellular automaton. Your task in this exam is to simulate *one-dimensional cellular automata*, a similar and simple model of computation that has (amazingly) been proven Turing-equivalent (these machines are as powerful as Turing machines).

A *one-dimensional cellular automaton* is an array of cells that are either *on* or *off* whose states all change each time the machine steps. The changes are governed by a set of rules that specify a new value for each cell, depending on the its current value and those of its its immediately adjacent neighbors. So that every cell has a left neighbor and a right neighbor, we define a dummy cell to the left of the leftmost cell and a dummy cell to the right of the rightmost cell, both of which are always *off*. For simplicity we use `1` to represent *on* and `0` to represent *off*. At left is an example of a 7-state machine (including the dummy `0` bits to its left and to its right), initialized to all `0` except for a `1` in the middle.

0	0	0	0	1	0	0	0	0

Now, to specify how a machine works, we start by considering the three bits given by the states of each cell and its neighbors (in the order left, middle, right). Since each of the three cells are either `0` or `1`, there are eight possible cases to consider. We can convert each of the possible three-bit values to an index between 0 and 7 and then use that index to access an eight-character "rules string" that specifies the new value for the state of the middle cell for each possibility. For example, the table at right shows the interpretation of a `String` variable `rules` with the value `"01001000"` as a rules string for a 1D cellular automaton. Each row in the table tells us how to determine a cell's new state: we convert the three bits representing its old state and the old states of its neighbors to a binary number `i`. The new state of that cell is `rules.charAt(i)`.

`i`	*left middle right* (`i` *in binary*)	*new middle* (`rules.charAt(i)`)
0	000	0
1	001	1
2	010	0
3	011	0
4	100	1
5	101	0
6	110	0
7	111	0



initial state	0	0	0	0	1	0	0	0	0
after step 1	0	0	0	1	0	1	0	0	0
after step 2	0	0	1	0	0	0	1	0	0
after step 3	0	1	0	1	0	1	0	1	0
after step 4	0	0	0	0	0	0	0	0	0

In a step, all cells change independently and simultaneously. The diagram at left shows the result of four steps for our example. After the first step, the center cell changes to `0` because it is in the middle of a `010` pattern (as indicated by the arrows). Also, the cell to its left changes to `1` because it is in the middle of a `001` pattern and cell to its right changes to 1 because it is in the middle of a 100 pattern. Take the time now to verify the changes in steps 2, 3, and 4.

Be sure that you understand how these machines work before reading further.

Part 1 (7 points). Download the project zip file, which includes all the files you will need, from the **Exams** page. Complete the implementation of `CA.java` by implementing a test client, constructor, `toString()`, and `step()` methods.

```
% more CA.java
public class CA
{
   private int n;
   private int[] cells;
   private String rules;

   public CA(int n, String rules)
   // YOUR CONSTRUCTOR HERE

   public void step()
   // YOUR CODE HERE

   public String toString()
   // YOUR CODE HERE

   public static void main(String[] args)
   // YOUR CODE HERE
}
```

Test client. Implement a test client that takes a size, number of steps, and rules string from the command line, builds a CA of the given size, and then runs it for the given number of steps, following the given rules. Print out the initial state of the CA and all state changes.

The constructor. Build an automaton with the given number of cells, all initialized to *off* except for the center one (you may assume that `n` is odd). Note that you need to accommodate the dummy cells at the beginning and at the end—-the easiest was to do so is to use an array of length `n+2`.

The `toString()` method. Return a string with blanks corresponding to *off* cells and asterisks corresponding to *on* cells (exclude the dummy cells). While debugging, you might find it more convenient to use `0`s and `1`s.

The `step()` method. To simulate one step of the automaton, use a temporary array, as follows:

- Create the temporary array.
- Set every entry of that array to the new value dictated by the rules.
- Copy from that array to set the new state of the machine.

The key to the simulation is to, for each cell *i* in the automaton, compute an `int` value `index` that indicates which rule should be applied to find its new value. For example, if `cells[i-1]` is `1`, `cells[i]` is `0`, and `cells[i+1]` is `0`, then your program should compute the value `4` for `index` and use `rules.charAt(i)` for the new value of `cells[i]`.

Examples.

```
% java-introCS CA 31 15 01001000
               *
              * *
             *   *
            * * * *
           *       *
          * *     * *
         *   *   *   *
        * * * * * * * *
       *               *
      * *             * *
     *   *           *   *
    * * * *         * * * *
   *       *       *       *
  * *     * *     * *     * *
 *   *   *   *   *   *   *   *
* * * * * * * * * * * * * * * *
```

```
% java-introCS CA 31 12 01111000
               *
              ***
             **  *
            ** ****
           **  *   *
          ** **** ***
         **  *    *  *
        ** ****  ******
       **  *   ***     *
      ** **** **  *   ***
     **  *    * **** **  *
    ** ****  ** *    * ****
   **  *   ***  **  ** *   *
```

Submit `CA.java` and your `README` using the `Submit!` link on the **Exams** page.

Do not attempt Part 2 until you have submitted your solution to Part 1.

You cannot get credit for Part 2 unless your solution to Part 1 is correct.

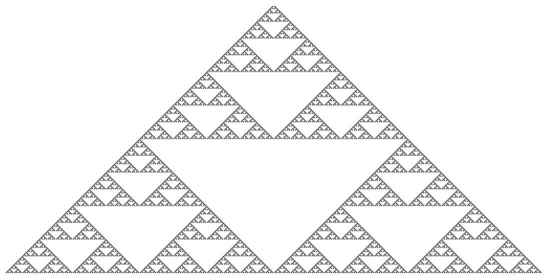
There is no partial credit for Part 2.

Part 2 (0.5 points). Implement a `CA` client `PictureCA` that visualizes the operation of larger automata. Your program must proceed as follows:

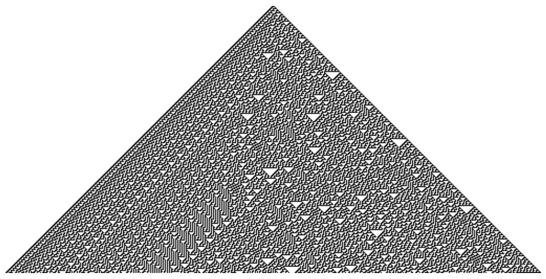
- Create a `CA` with 511 cells.
- Build a 511-by-255 `Picture` with pixel (*i*, *j*) set to `Color.WHITE` if cell *i* is *off* after step *j*, `Color.BLACK` otherwise.
- Show the picture.

Don't forget to import `java.awt.Color` at the beginning of your program.

Your program should produce the following images:



```
% java-introCS PictureCA 01001000
```



```
% java-introCS PictureCA 01111000
```

Submit `PictureCA.java` using the `Submit!` link on the **`Exams`** page.

Now that you're done with the exam, you can enjoy playing with other rules strings. Later, you can look around on the web for other remarkable facts about cellular automata (including the fact that the one on the right, with general initialization, can compute anything a Turing machine can compute)!