NAME: **login id:**

Precept:

COS 126 Midterm 2 Programming Exam Fall 2012

This part of your exam is like a mini-programming assignment. You will create two programs, compile them, and run them on your laptop, debugging as needed. This exam is open book, open browser—but only our course website and booksite! Of course, no internal or external communication is permitted (e.g., talking, email, IM, texting, cell phones) during the exam. You may use code from your assignments or code found on the COS126 website. When you are done, submit your program via the course website using the submit link for Precept Exam 2 on the Assignments 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.

Even though you will electronically submit your code, you must turn in this paper so that we have a record that you took the exam and signed the Honor Code. *Print your name, login ID, and precept number on this page* (now), and **write out and sign** the Honor Code pledge before turning in this paper. *Note*: It is a violation of the Honor Code to discuss this midterm exam question with anyone until after everyone in the class has taken the exam. You have 90 minutes to complete the test.

"I pledge my honor that I have not violated the Honor Code during this examination."

Signature

Part 1A	/10
Part 1B	/10
Part 2	/10
TOTAL	/30

Your task in this exam is to simulate *one-dimensional cellular automata*, a simple model of computation that has (amazingly) been proven Turing-equivalent (these machines are as powerful as Turing machines).

Description. You may have heard of Conway's *game of life*, which is a two-dimensional cellular automaton. 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 depend on the states of each cell and its immediately adjacent neighbors. So that every cell has a left neighbor and a right neighbor, we define an imaginary cell to the left of the leftmost cell and an imaginary 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*, so that we can use a 3-bit string to represent the states of each cell and its neighbors (with the state of the cell of interest in the middle). Since each cell and each of its two neighbors are either *on* or *off*, there are eight possible cases to consider, and we can represent rules with an eight-character string, by specifying the new value for the state of the middle cell for each possibility. For example, the following table explains the meaning of the rules string `"01001000"`.

rules string index	0	1	2	3	4	5	6	7
left middle right	000	001	010	011	100	101	110	111
new middle	0	1	0	0	1	0	0	0

Each column in the table (and each character in the rules string) determines a cell's new state: the 3-bit string representing its old state and the old states of its neighbors is used as the binary representation of the column index, and the new state of that cell is the entry of the rules string at that index. In a step, all cells change independently and simultaneously. For example, suppose that we have an 11-state machine. According to these rules, if the state of the machine is

```
0 0 1 1 1 0 1 0 0 1 0 0 0
```

then it will change to

```
0 1 0 0 0 0 0 1 1 0 1 0 0
```

after the next step. The first cell changes to `1` because it is in the middle of a `0 0 1` pattern; the next cell changes to `0` because it is in the middle of a `0 1 1` pattern; and so forth. *Be sure that you understand how this machine works before reading further*. Here are the next few steps:

```
0 0 1 0 0 0 1 0 0 0 0 1 0
0 1 0 1 0 1 0 1 0 0 1 0 0
0 0 0 0 0 0 0 0 1 1 0 1 0
0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0 1 0 0 0 0
```

Part 1A (10 points). First, write a skeleton class `CA` that implements the following API:

	`public class CA`	
	`CA(int N, String rules)`	*create a (2N+1) state automaton based on the given rules*
`void`	`step()`	*simulate one step of the automaton*
`String`	`toString()`	*return a string representation of the current state of each cell*

The constructor should save the argument values in instance variables and construct a one-dimensional array named `cells` of `int` values that will represent the state of the automaton (each cell is `0` for *off* and `1` for *on*). To keep things symmetric, build an automaton with $2N+1$ states, so that `cells` should have $2N+3$ entries, to include the dummy cells at the ends. To get things started, initialize the center cell to *on*. The `toString()` method should return a string of length $2N+1$ with spaces corresponding to *off* cells and `1`s corresponding to *on* cells (and excluding the dummy cells). The `step()` method is Part 1B; for now, implement that as

```
public void step() { }
```

then implement a `main()` test client that takes `N` and `rules` from the command line and runs the machine for *N* steps, printing initial state and the state of the automaton after every step. To be sure your output is correct, you might start with your `toString()` method outputting `0`'s instead of spaces. This should produce the following result:

```
% java CA 5 01001000
00000100000
00000100000
00000100000
00000100000
00000100000
00000100000
```

Nothing happens because `step()` does nothing, but now you can implement `step()` without having to worry about all this other code.

Before moving on, remember to change the `"0"` in `toString()` to a blank, so that we only see the *on* states in the output.

You do not need to submit this code separately, but *be sure to follow the submission instructions at the bottom of the next page*, whether or not you get Part 1B working.

Part 1B (10 points). Next, implement the `step()` method for `CA`. This requires:

- Creating a temporary array.
- Setting every entry of that array to the new value dictated by the rules.
- Using that array to set the new state of the machine.

The key to the simulation is to compute an `int` value `index` that indicates which rule should be applied. 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 check whether the fifth character in the rule string is 0 or 1.

Don't forget to make sure that you use the dummy values at the end of the array to compute new values but that you never change those dummy values.

If your program works correctly, you'll see that it produces the Sierpinksi triangle for the `"01001000"` rules and amazing patterns for others, such as the `"01111000"` rules illustrated at right.

```
% java CA 15 01001000           % java CA 15 01111000
               1                                1
              1 1                              111
             1   1                            11  1
            1 1 1 1                          11 1111
           1       1                        11  1   1
          1 1     1 1                      11 1111 111
         1   1   1   1                    11  1    1  1
        1 1 1 1 1 1 1 1                  11 1111  111111
       1               1                11  1   111     1
      1 1             1 1              11 1111 11  1   111
     1   1           1   1            11  1    1 1111 11  1
    1 1 1 1         1 1 1 1          11 1111  11 1    1 1111
   1       1       1       1        11  1   111  11  11 1   1
  1 1     1 1     1 1     1 1      11 1111 11  111 111  11 111
 1   1   1   1   1   1   1   1    11  1    1 111   1 111   1  1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  11 1111  11 1  1 11111  1111111
```

Submission. Submit the file `CA.java` via Dropbox at

https://dropbox.cs.princeton.edu/COS126_F2012/Exam2

(This is the submit link for Precept Exam 2 Part 1 on the Assignments page.) Be sure to click the *Check All Submitted Files* button to verify your submission.

Grading. Your program will be graded on correctness and clarity (including comments).

Part 2 (10 points). *Be sure that you have successfully completed both Part 1A and Part 1B and have submitted* `CA.java` *before attempting this part of the exam, which counts for fewer points* (and which you won't easily be able to test without a working solution to Part 1).

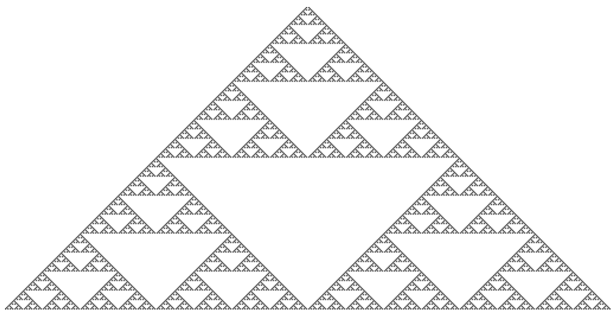
Next, implement a `CA` client `PictureCA` that visualizes the operation of larger automata. Your program must proceed as follows:

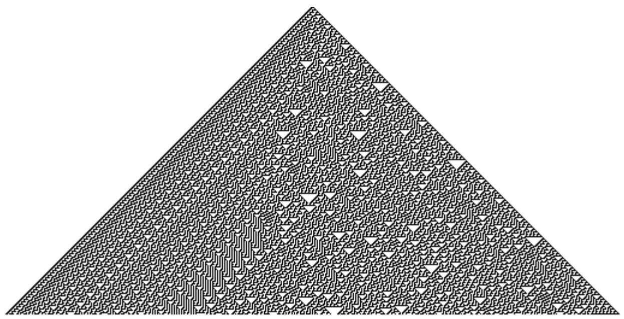
- Create a CA with *N*=255. It will have 511 cells.
- Build a 511-by-256 `Picture` with pixel (*i*, *j*) set to `Color.WHITE` if cell *i* is *off* after step *j*, `Color.BLACK` otherwise.
- Show the picture.

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

You program should produce the following images:

```
% java PictureCA 01001000        % java PictureCA 01111000
```





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

Submission. Submit the single file `PictureCA.java` via Dropbox at

https://dropbox.cs.princeton.edu/COS126_F2012/Exam2

(This is the submit link for Precept Exam 2 Part 2 on the Assignments page.) Again, be sure to click the *Check All Submitted Files* button to verify your submission.

Grading. Your program will be graded on correctness and clarity (including comments).