Project 1
A Simple Program with Complex Behavior

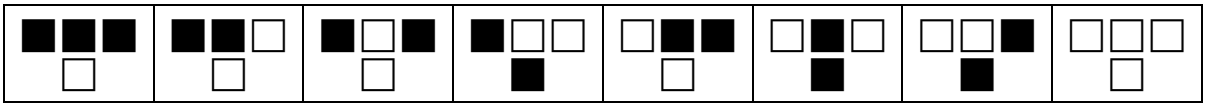
Due: Monday, Feb. 17 at 11:59 PM

> Over and over again we will see the same kind of thing: that even though the underlying rules for a system are simple, and even though the system is started from simple initial conditions, the behavior that the system shows can nevertheless be highly complex. And I will argue that it is this basic phenomenon that is ultimately responsible for most of the complexity that we see in nature.
>
> —Stephen Wolfram, in *A New Kind of Science*

Imagine a row of equal-sized squares. Let some be colored white and the others black. For example, here is one possible row:

Suppose that after some time, the colors can change. The new color of each square is determined by its current color and the colors of its left and right neighbors. If exactly one of these squares is black, the new color is black. Otherwise, the new color is white.

This rule can be expressed graphically with the following table:



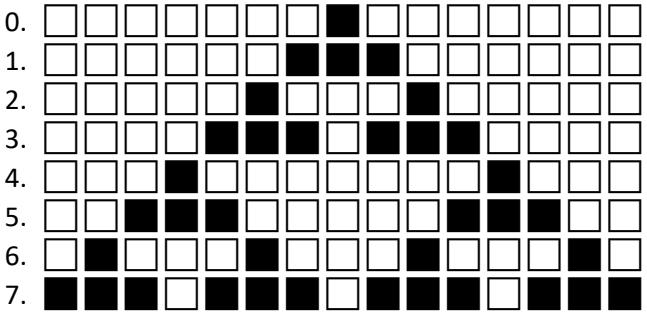
The top row of each column shows one of the eight configurations of a square and its neighbors. The bottom row shows the new color of the center square.

For instance, the rightmost column shows a white square with two white neighbors. The rule says that the new color of the center square will be white. The second-rightmost column shows a white square with a white left neighbor and a black right neighbor. The rule says the new color will be black.

If we apply this rule to the entire row of squares, we get the following new colors:

The center square remained black because both of its neighbors were white. Meanwhile, its neighbors flipped from white to black because they each had a single black neighbor.

What kind of behavior will this system exhibit if we apply the rule multiple times? To visualize how it changes, we can show each new state on a new line. Below is how the system evolves when the rule is applied seven times:



This rule and initial state evidently produce a fractal pattern!

Elementary Cellular Automata: The system described above is an example of an elementary cellular automaton (ECA). Each square is called a "cell," each row is called a "generation," and each application of the rule is called a "step."

Cellular automata such as this are "elementary" in the following sense:

1. The cells are arranged in a one-dimensional grid.
2. Each cell can be in one of two states.
3. The rule that determines the next state of a cell depends on the cell's current state and the states of its two nearest neighbors.

The polymath Stephen Wolfram studied these systems extensively in the 1980s and made some startling discoveries, which he details in his 1200-page tome, *A New Kind of Science*.[1] The simplicity of ECAs suggests that they would evolve in simple ways, but this is not always the case. With the proper choice of rule, an ECA can produce chaotic, irregular patterns that are too complex to predict. The only way to know the state of these automata after *N* steps is to apply their rules *N* times.

For Project 1, we will write a program that does exactly that. Given a rule, an initial state, and a number of steps, our program will calculate the state of the ECA after each step.

Wolfram Code: There are 256 possible rules that an ECA can follow. Wolfram devised a way of numbering these rules that has become known as the "Wolfram Code."

To understand the code, note that white and black squares are arbitrary ways of representing cells. Suppose we use 0 and 1 instead. The rule table from earlier now looks like this:

111	110	101	100	011	010	001	000
0	0	0	1	0	1	1	0

The 0s and 1s on the bottom row uniquely specify the rule. The Wolfram Code interprets these bits as the binary number 00010110. The value of this number in base 10 is

$$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = \boxed{22}$$

Therefore, this is Rule 22.[2]

Given a rule number, the corresponding rule table is found by converting the number to base 2. For instance, Rule 30 refers to the table

111	110	101	100	011	010	001	000
0	0	0	1	1	1	1	0

because the binary number 00011110 is equal to 30 in base 10.

---

[1] Although long, the book is written for a general audience and can be read for free online. Chapter 2 is a lucid introduction to ECAs and includes amazing pictures of automata evolving over thousands of steps. We highly recommend that you at least skim the first section: https://www.wolframscience.com/nks/p23--how-do-simple-programs-behave/.

[2] If this is confusing, recall how place-value notation works in base 10. The decimal number 451 has the value

$$4 \cdot 10^2 + 5 \cdot 10^1 + 1 \cdot 10^0.$$

If 451 is interpreted as a number in another base, say 8, its base-10 value is calculated by replacing the 10s with 8s:

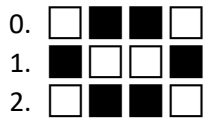
$$4 \cdot 8^2 + 5 \cdot 8^1 + 1 \cdot 8^0 = 4 \cdot 64 + 5 \cdot 8 + 1 \cdot 1 = 297$$

You can use the method toBinaryString in the Integer class to convert integers to base 2 in Java.

Boundary Conditions: You may have noticed that a rule table does not specify the next state of the leftmost and rightmost cells in a row. For instance, what is the left neighbor of the leftmost cell?

The cells in an ECA are often treated as extending infinitely to the left and right. We will use periodic (i.e., circular) boundary conditions instead. The rightmost cell is treated as the left neighbor of the leftmost cell. Similarly, the leftmost cell is treated as the right neighbor of the rightmost cell.

For example, here is Rule 22 applied twice to an ECA with four cells:



The leftmost cell flips from white to black on the first step because its left neighbor (the rightmost cell) is white. It then flips back to white on the second step because both it and its left neighbor are black.

Class Diagram: Our program consists of five classes: Automaton, Rule, Generation, Cell, and a driver class. Below is a partial UML diagram for Automaton:

Automaton
+Automaton(ruleNum: int, initState: boolean[]) +Automaton(filename: String) +getRuleNum(): int +evolve(numSteps: int): void +getTotalSteps(): int +getState(stepNum: int): boolean[] +getStateString(stepNum: int): String +toString(): String +save(filename: String): void +getFalseSymbol(): char +setFalseSymbol(symbol: char): void +getTrueSymbol(): char +setTrueSymbol(symbol: char): void

Note that only the public methods are shown. It is up to you whether this class should contain any data or private methods. The public methods, however, should work as follows:

- Automaton(int ruleNum, boolean[] initState): Construct an ECA from a rule number and an array of boolean values. The values represent the initial states of the cells, with false and true corresponding to 0 and 1 respectively.
- Automaton(String filename): Construct an ECA from the information in a text file. The file will be formatted as follows:

```
<rule-number>
<false-symbol> <true-symbol>
<initial-state>
```

For example, below is the content of a file that defines the same ECA shown at the beginning of this document:

```
22
0 1
000000010000000
```

The first line indicates that the ECA evolves according to Rule 22. The second says that '0' and '1' will represent false and true. The bottom line is the initial state: fifteen cells with only the center cell in the true state.

- getRuleNum(): Return the Wolfram Code for the rule that governs the evolution of the ECA.
- evolve(int numSteps): Evolve the ECA a given number of steps.
- getTotalSteps(): Return the total number of steps that the ECA has evolved. This is equal to the total number of generations minus 1, since the initial state is generation 0.
- getState(int stepNum): Return an array with the states of the cells after the given step. The step number must be less than or equal to the total number of steps.
- getStateString(int stepNum): Return a String that represents the states of the cells after the given step. The String has one character for each cell, and the symbols used to represent false and true are those returned by getFalseSymbol() and getTrueSymbol().
- toString(): Return a String that represents the entire evolution of the ECA. The String consists of the state String for every generation joined together by newline characters.
- save(String filename): Save the output of toString() to a file with the given name. (Overwrite the content of the file if it already exists.)
- getFalseSymbol(): Return the character that represents false. Use '0' as the default if the ECA is not constructed from a text file.
- setFalseSymbol(char symbol): Set the character that represents false.
- getTrueSymbol(): Return the character that represents true. Use '1' as the default if the ECA is not constructed from a text file.
- setTrueSymbol(char symbol): Set the character that represents true.

Unlike our recent lab assignments, you get to design the remaining classes: Rule, Generation, and Cell. The test cases used to grade this assignment will not call the methods in these classes directly. They will only call the public methods of Automaton.

In addition to the classes described above, write a driver class to test the methods of the other classes.

Javadoc: Write a Javadoc comment for each of the classes, fields, and methods in your program. Details can be found in Section 17.7 of our zyBook.