Assignment 1: Cellular Automata

Data Structures, CSCI-UA 102, Section 7
Instructor: Max Sklar

Implement a simulator for Steven Wolfram's Elementary Cellular Automata.

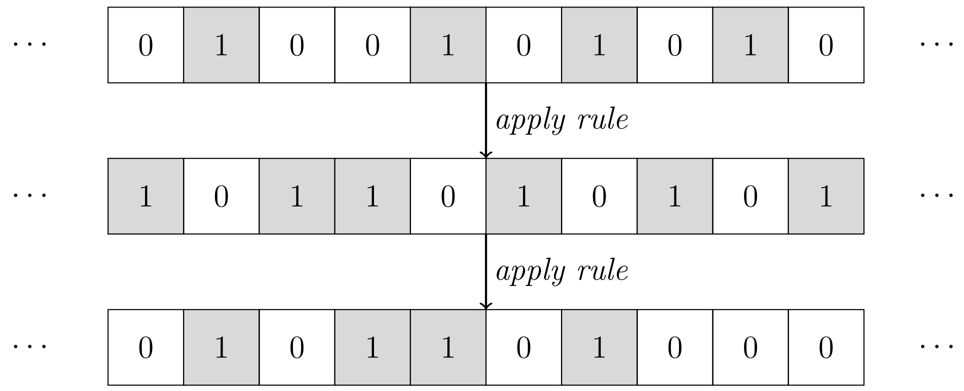
Goals

- Learn how to work with arrays, and in this case a circular array.
- Figure out how to get a more complex array of arguments to work properly.
- Learn about elementary cellular automata, and see visually how they work.
- Work with a randomized algorithm, along with randomization seeds.

Background

Cellular automata is a model of computation that takes place on a grid. In these models, the grid evolves in steps. At each step, the value of an entry in the grid is computed as a constant time function of its neighbors.

Elementary cellular automata (ECA) are the simplest examples of this where interesting results arise. An ECA is based on an array of bits extending infinitely in both directions. At each step, the grid is in a certain state where some of the cells are on and some are off.



Rows of states evolving over time with applied rules

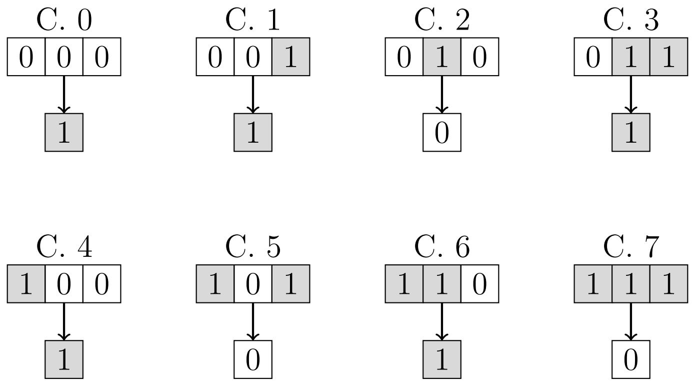
Every time the grid is updated, the value of each cell is a function of its current state, and the state of its left and right neighbors. When computing the value of a cell, there are

8 different configurations of the cell and its neighbors. If a bit representing the outcome is specified for each of these 8 configurations, a complete *rule* is generated that can be used to compute the next state of the ECA.

	L	C	R
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

The 8 possible configurations of a cell and its neighbors.

Because there are 8 configurations, 8 bits specify a rule. An 8-digit binary number is (in decimal) a number from 0 to 255, so the rules can be named simply as rule 0, rule 21, etc. The rule number is calculated by arranging the 8 bit sequentially, with the rule for 000 as the least significant bit (worth 1) and 111 as the most significant bit (worth 128).



The specification for rule 01011011_2 *or 91 in decimal.*

ECAs were studied by Steven Wolfram in the 1980s and 1990s. They are documented in his seminal book 2002, *A New Kind of Science.* Wolfram found that while many ECA rules produce boring results, a few produce intriguing patterns and properties.

- A *computationally irreducible* automata is one where the only way to predict a future state of the system is to apply the rules and run the simulation. Rule 0 is NOT computationally irreducible because it switches all of the cells off, and therefore we can predict what will happen far into the future.

- A *Turing complete* automata is one that can be used to simulate a fully functioning computer. Amazingly, at least one ECA (rule 110) has been proven to be Turing complete!

Currently, Wolfram is using these ideas to look for a computational basis for physics in the Wolfram Physics Project.

Instructions

Your job is to simulate and visualize the elementary cellular automata, as well as including an interesting randomization element.

Your main function should be in a class called `ElementaryCellularAutomata`. It takes a variety of arguments as detailed below, but once it starts running, it will print out the initial state of the ECA, and then on a new line the new state of the ECA each time it evolves.

Storing the State of the Cellular Automata

You will store the current state of your cells in an array of booleans. Unfortunately, your process does not have infinite memory. Instead, you are going to create a finite array of booleans, and the size will be determined by one of the command line arguments. This array is a *circular array* meaning that the last cell connects with the first one. There is nothing special that you need to do when declaring the array, only when you get the neighbors of the rightmost or leftmost cell.

Displaying Cells

You will be provided with the AnsiColor.java class, which defines the 8 possible 3-bit colors that your terminal can display. In order to display a red cell, you could use the following code:

```
AnsiColor c = new AnsiColor("red")
c.printBlock()
```

The method "printBlock" will print out 2 spaces (roughly a square), with the color red in your terminal. In elementary celluar automata, there are 2 possible colors: one representing the off cells and one representing the on cells. These will default to black and white, but the colors used will be another command line argument.

Every time you print the current state of your cells, loop through the array, and call printBlock() on the AnsiColor for that cell. Then print out a newline before moving on to the next state. This loop should go on for a set number of iterations, also given by a command line argument.

Randomization

In traditional deterministic Elementary Cellular Automata (ECA), each rule consists of 8 bits. These bits specify whether a cell should turn "on" or "off" based on the 8 possible configurations of the cell and its neighbors.

In your implementation, you will extend this idea by using probabilistic rules. Instead of 8 bits, each rule will now be represented by 8 doubles (decimal values from 0 to 1). These doubles represent the probability that a cell will turn "on" based on the configuration of its neighbors. For example, if the rule for a specific configuration is 0.3, this means there is a 30% chance the cell will turn "on" in the next state.

Using the Random Number Generator

Random number generators produce bits which are difficult to predict or calculate unless the underlying function is evaluated directly. By providing a randomization seed, we can be sure that our randomizer will produce the same sequence values every time it is initialized. Every time your code runs with the same seed, it should produce the same results.

The random number generator should be initialized one time only:

```
this.random = new Random(seed);
```

Your code should use a single random number generator, and it should only be used to generate doubles:

```
random.nextDouble();
```

There are 2 places where random numbers may be used.

1. Initializing the state of the automaton to determining the starting configuration of "on" and "off" cells.
2. Applying probabilistic rules to determine the state of each cell at each step of the simulation.

Initializing the State

The use of randomness to initialize the state is given by the command line parameter "init".

- When init is 0 or below all of the cells start as off.
- When init is 1 or above all of the cells start as on.
- If init is some fraction between 0 and 1, for example at 0.05, the random number generator is used to initialize a percentage of "on" cells. For example, with init = 0.05, approximately 5% of the cells will be randomly turned "on."

You might have a line of code inside a loop that looks like this:

```
cells[i] = random.nextDouble() < init;
```

Applying Probabilistic Rules

Each rule consists of 8 probabilities, one for each of the 8 possible configurations of a cell and its neighbors. Once the correct configuration is identified and probability retrieved, the next state of the cell is determined as follows:

- If the rule value is 0 or below, the cell will always turn "off."
- If the rule value is 1 or above, the cell will always turn "on."
- If the rule value is between 0 and 1, the random number generator will be used to probabilistically decide whether the cell should turn "on" or "off."

Example:

```
boolean newState = random.nextDouble() < ruleProbability;
```

This compares the random number with the probability associated with the current rule, and the cell turns "on" if the random number is less than the rule's probability.

Do not use the random generator if it isn't needed (in the 0 or 1 cases). If you do, it will break automated testing by producing different values than expected.

Command Line Arguments

The main function in `ElementaryCellularAutomata` accepts arguments in the form of flags, followed by the corresponding values for those flags. Each flag represents a specific parameter, and the values define how the cellular automaton should behave.

Below is a list of flags and their descriptions:

- **-rules**: This flag determines the rule for the cellular automaton. There are two ways to specify the rule:
  - Provide a single integer (Wolfram number), which represents one of the standard rule sets from Stephen Wolfram's elementary cellular automata.
  - Provide 8 doubles separated by spaces, one for each possible configuration of a cell and its neighbors, specifying the transition behavior of the automaton. This list should start with the rule for 000 and end with the rule for 111.
- **-off-color**: This flag defines the 3-bit color used for cells in the "off" state. The default color is `black`, but you can specify a different color by providing an ANSI color name (e.g., `red`, `blue`, etc.).
- **-on-color**: Similar to `-off-color`, this flag sets the color for cells in the "on" state. The default color is `white`.
- **-random-seed**: The seed used for randomization in the automaton. Providing this flag ensures that the randomness is reproducible. It takes a single long value.
- **-size**: This flag sets the size of the cellular automata, i.e., how many cells are present. The default value is 100. You can specify any non-negative integer.
- **-init**: The initialization density of the automaton. It is a floating-point value between 0 and 1 that defines the probability that any given cell will start in the "on" state. For example, a value of 0.1 means 10% of the cells are initialized to the "on" state.
- **-iter**: This flag defines the number of iterations the automaton should evolve for. The default is 1000, but you can specify a different number.

Each flag must be followed by the appropriate number of inputs. The following example shows how you might specify the parameters:

```
java ElementaryCellularAutomata -rules 110 -off-color black -on-color cyan
  -random-seed 42 -size 50 -init 0.2 -iter 500
```

This will set Rule 110 as the rule set, display the off-state cells in red, the on-state cells in green, use a random seed of 42, set the number of cells 50, initialize 20% of the cells to the on state, and evolve the automaton for 500 iterations.

In the next example, we specify the rule for each of the 8 configurations individually, and use probabilistic rules on two of them.

```
java ElementaryCellularAutomata -rules 0 0.5 1 1 1 1 0.5 0 -off-color black
  -on-color white -random-seed 1 -size 90 -init 0.05 -iter 500
```

Hints

- This project has two parts: interpreting the parameters, and running the ECA. You can build one without completing the other. Focusing on one at a time is a good strategy.
- Save the probabilistic rules for last while getting the rest of your code working.
- Think carefully about each of the following operations
  1. Given a cell and its neighbors, what is the next state of this cell?
  2. Given the entire array of cells, how do I update it to produce the next state?
  3. What operations do I need to do every time I update the state?
- In order to get the bit i (starting from the lowest place value) in an integer, the following code can be used:

  ```
  int bit = (n >> i) & 1;
  ```

  Note that it shifts the bits i spaces to the right, and then the &1 will zero out every bit except for the rightmost bit.
- When you are done, try well known interesting rules (like 110) or look at different ones online and see if your simulator produces a similar pattern.
- Not all terminals will treat colors equally or display them exactly as expected (e.g., 'white' might appear grayish). As long as the intended logic is implemented correctly, this is not a concern.
- Be sure to use some of the principles of Object Oriented Programming and break up the project into several classes with defined roles. The solution from your instructor, for example, has 4 total classes: AnsiColor, EcaParams to interpret and hold the values of the parameters, RuleSet to keep track of the rules and apply them to a cell, and ElementaryCellularAutomata to keep the full state and execute a step.

Code Style

Your code style will be checked with the following directives in mind:

- Your code should be readable and understandable.
- You may select any reasonable style (i.e., for indentation or bracket placement), but it should be consistent.
- Please add comments in any part of the code where it is not obvious to the reader what the code is doing and why.
- Please use descriptive and efficient names for your classes, functions, and variables. Local counters that can be still be called something like i for index.

Submission

The entire submission should be a zip file containing all relevant files and folders and should be uploaded to Brightspace by the due date. **The zip file should be of the form [netid].zip where you insert your netid**. The zip file includes:

- Your code, which should include one or more Java files and may or may not be organized into packages. This includes code provided for the assignment.
- Your `readme.txt` file.

README

Included in the `readme.txt` file should be the following:

RUNBOOK: Information on how to compile and run your code in the command line.

TIME SPENT: An estimate of the time spent working on this project.

NOTES: Include the difficulties and roadblocks encountered while working on this project, including notable bugs that needed to be overcome. Are there any quirks in your solution that we should be aware of? Was the outcome what you expected?

RESOURCES AND ACKNOWLEDGEMENTS: Disclose the sources you consulted for help on this project. Possible human sources include: NYU Staff (the instructor, TA, tutor), other students, and anyone who helped you with the project. Sources that you consulted, including code that you found online, in the book, and output from LLMs. Describe the nature of your LLM usage.

There will never be a grading penalty for a complete and accurate accounting of resources and acknowledgments.