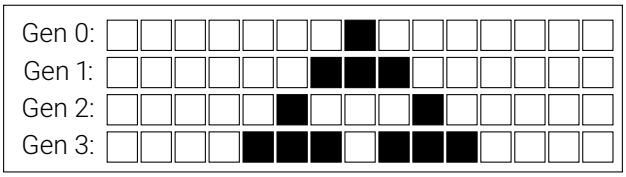
Cellular Automata

AP Computer Science A

Name: _____

Contents

Background



Evolution of a Sample Cellular Automaton

The above image is of the first few generations of an *elementary cellular automaton*, a concept discussed at length in Stephen Wolfram's massive work: *A New Kind of Science*. In this text, Wolfram continues the work first started in the 1940's, when Stanislaw Ulam and John von Neumann suggested the use of cellular automata for a number of different, important areas of research having to do with replication of various different natural and technological systems (von Neumann, in particular, was interested in self-replicating robots).

A cellular automaton attempts to represent these systems as a grid of cells, each of which is in a specific current state. These cells will evolve based on the states of cells in their *neighborhood*, an important concept that will be discussed at length shortly. Once these cells have evolved, the entire set of cells enters a new state, called a *generation* of the cellular automaton.

Although Wolfram's assertation that the study of these "cellular automata" are relevant for all branches of science can be controversial, their academic interest within the computer sciences is fairly undisputed. In this lab, we will be looking at the concept of *elementary cellular automata*, a set of cellular automatons based on a single-dimensional array of cells that evolve according to a number of preset rules.

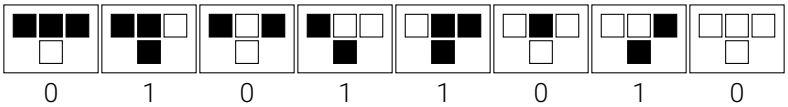
Elementary Cellular Automata

An elementary cellular automaton is a single-dimensional array of cells, each of which can be in one of two states (0 and 1 or "off" and "on", however you'd like to think of it). A neighborhood for each cell is comprised of three cells: the cell itself, and the cells on either side of it (left and right). Additionally, the starting state (often referred to as "generation zero") has all cells in the same state, except for the very center cell. A visualisation of the starting state for every elementary cellular automaton is below.

Generation Zero of Every Elementary Cellular Automaton

Since each neighbourhood contains three cells, there are 8 possible permutations of neighbourhoods. Note that because the cells on either edge of the grid would only have two cells in their neighbourhood, a choice needs to be made as to what pattern to match. In the examples found within Wolfram's work, the missing neighbour is treated as a 0 or 'off'. Unless specified otherwise, you should continue this convention for your work on cellular automata. All 8 permutations (in the order Stephen Wolfram used) are seen below.

By assigning a result to each of these as either a 0 or a 1, we can create a set of rules for each elementary cellular automaton. The following is the set of rules that produced the sample generations above. The block below each neighbourhood represents the state of the cell in the next generation.

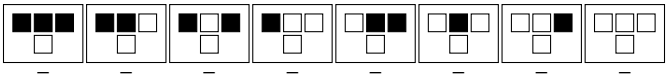


Because the binary string, '01011010', is equivalent to the denary value 90, the above is referred to as 'Rule #90'. In this way, each ruleset for an elementary cellular automaton can be described by its denary equivalent, with a maximum value of $2^8 - 1 = 255$.

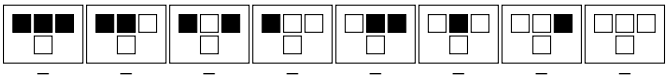
Applications

Question #1: For each of the following rule numbers, write in the appropriate binary digits. Shade each resulting cell, as necessary.

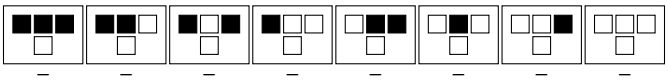
Rule #62



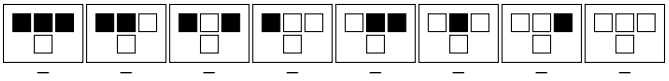
Rule #108



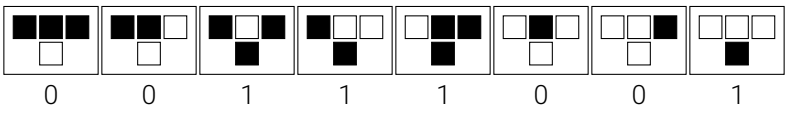
Rule #120

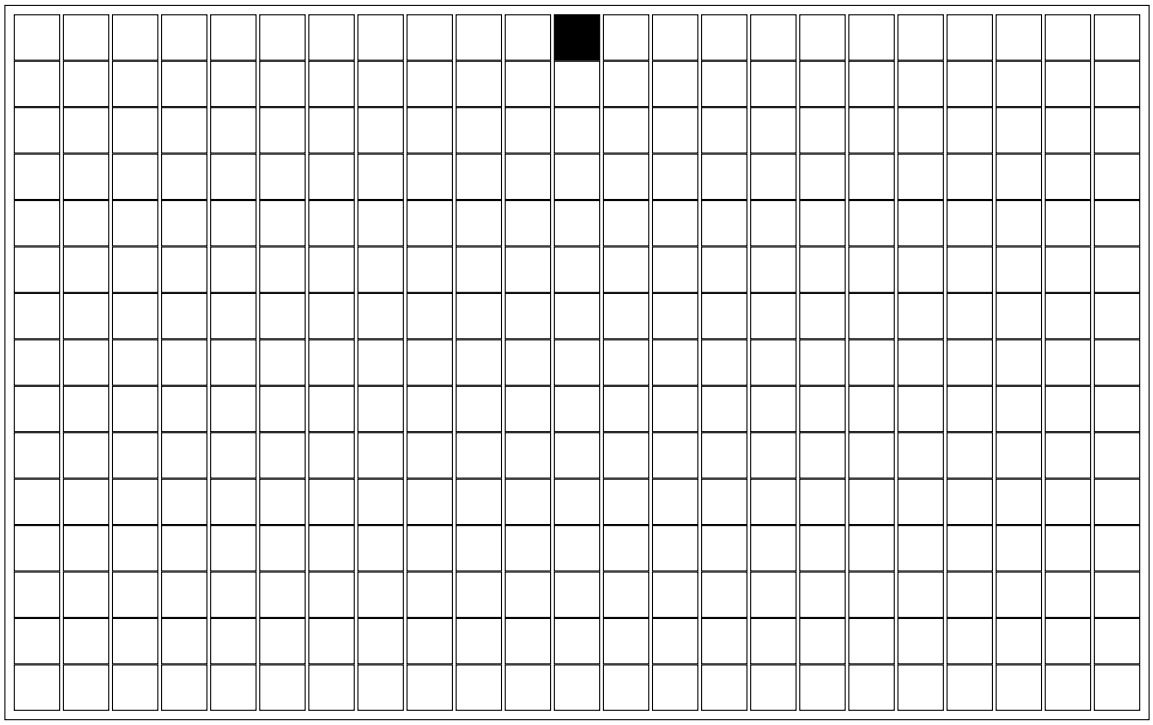


Rule #173



Question #2: Shade in the appropriate boxes for each generation of the following ruleset.





Question #3: What rule number is represented by the pattern in Question #2?

Activity #1

Introduction

In this activity, you will be implementing Wolfram's Classic Cellular Automation in Java. By creating the `CellularAutomaton` class, you will be able to instantiate multiple `CellularAutomaton` objects in order to explore the various different cellular automata that could arise from different sets of rules. See the *Template Class & Test Cases* section for the skeletal outline of the `CellularAutomaton` class that you should use.

Exercises

1. Complete the `CellularAutomaton` class with the following specifications:
   (a) A constructor whose sole parameter is the size (number of array elements) of the automation. Default `rules` values should be set to Rule #90 described in the background. The `cells` array should be initialized so that the middle cell is "on".
   (b) Two helper methods:
      i. `getNeighbourValue` which returns the decimalisation of the binary configuration for a given cell and its neighbours.
      Note: Remember to treat the "edge cases" differently. See the background for more information.
      ii. `getNextGeneration` which evolves the automation according to its rules.
   (c) One public method, `evolve`, which evolves the automation and prints the new generation to the console.
2. Add the following to the `CellularAutomaton` class:
   (a) A helper method, `getRulePattern`, which takes a decimal `int` value and returns a set of rules corresponding to that value.
   (b) An overloaded constructor that allows the specification of a rule number to populate `rules` with.

Questions

Question #4: Briefly explain why the creation of the helper methods, `getNeighbourValue` and `getNextGeneration` may be desirable over placing all of the programming logic in the `evolve` method.

Question #5: Why should `getNextGeneration` return an array of cells representing the next generation rather than simply repopulating the `CellularAutomaton`'s `cells` array?

Activity #2

Introduction

In Stephen Wolfram's elementary cellular automata, the cells at the edge of a group (array index `0` and `length - 1`) assume the missing neighbour is 'off' or 0. In this activity, you will create a number of subclasses of your `CellularAutomaton` class to handle different behaviors for these special edge cases.

Exercises

1. Create the `WrappingCellularAutomaton` subclass of `CellularAutomaton` in which edges "wrap-around". That is, the left-neighbour of cell `0` is actually at index `length - 1` and the right neighbour of cell `length - 1` is actually at index `0`.
   Note: You will only receive full credit for this exercise if you take full advantage of inheritance through subclassing.

2. Create the `SpecialCaseCellularAutomaton` subclass of `CellularAutomaton` in which edges are given a special set of rules to handle only having two neighbours. You should implement a constructor that will accept two rule numbers, one for the main set of rules and one for the special edge cases.
   Note: Because there are only two neighbours for these special edge cases, you will only need to store a total of 4 different cases, with a maximum rule number of 16.

Questions

Question #6: How are the patterns that emerge from the `WrappingCellularAutomaton` different from those of the base `CellularAutomaton` for the same rule number? How are they similar?

Question #7: What base pattern did you choose for the four possibilities of edge cases in `SpecialCaseCellularAutomaton`? What led you to choose this pattern?

Final Analysis

Question #8: Find two sets of cellular automata rules that result in the same pattern. Explain why this occurs.

Question #9: Experiment with changing the pattern for Generation 0 in your `CellularAutomaton` class. How does the starting pattern affect the overall pattern of the cellular automata.

Question #10: Which part of implementing `CellularAutomaton` did you find most challenging? How did you overcome these challenges?

Question #11: Which part of implementing `WrappingCellularAutomaton` or `SpecialCaseCellularAutomaton` did you find most challenging? How did you overcome these challenges?

Question #12: What new programming techniques or knowledge did you learn as a result of this lab?

Template Class & Test Cases

```
/**
 * Cellular Automata Lab (Template Classes and Test Cases)
 * These are the template classes and test cases for the Cellular
 * Automata Lab. Written for the Woodstock School in Mussoorie,
 * Uttarakhand, India.
 *
 * @author Jeffrey Santos
 * @version 1.0
 */

public class CellularAutomata {
  public static void main(String[] args) {
    //  Tests for CellularAutomaton handling of n, the number of cells.
    //   Output: 11 cells, with the center cell filled.
    System.out.println(new CellularAutomaton(11));
    //   Output: 21 cells, with the center cell filled.
    System.out.println(new CellularAutomaton(21));
    //   Output: 51 cells, with the center cell filled.
    System.out.println(new CellularAutomaton(51));

    //  Tests for the evolve() method.
    //   Output: The following lines should print the sample evolution from
    //           the Background section of the lab.
    CellularAutomaton ca1 = new CellularAutomaton(15);
    System.out.println(ca1);
    ca1.evolve();
    ca1.evolve();
    ca1.evolve();

    //   Output: The following lines should print identical output as those
    //           above.
    CellularAutomaton ca2 = new CellularAutomaton(15, 90);
    System.out.println(ca2);
    ca2.evolve();
    ca2.evolve();
    ca2.evolve();

    //   Output: The following lines should print the solution to Activity #1,
    //           Question #2.
    CellularAutomaton ca3 = new CellularAutomaton(23, 57);
    System.out.println(ca3);
    for (int i = 1; i < 15; i++)
      ca3.evolve();

    //   Output: The following lines will produce similar starting output as
    //           the sample given in the Background; however, the edge cases
    //           will begin to shift the pattern once they begin being filled.
    WrappingCellularAutomaton wca1 = new WrappingCellularAutomaton(15, 90);
    System.out.println(wca1);
    for (int i = 1; i < 15; i++)
      wca1.evolve();

    //   Output: The following lines will produce similar output as the
    //           sample given in the background; however, the edge cases will
    //           be handled according to the special rule set.
    SpecialCaseCellularAutomaton scca1 = new SpecialCaseCellularAutomaton(15, 90, 11);
    System.out.println(scca1);
    for (int i = 1; i < 15; i++)
      scca1.evolve();
  }
}

class CellularAutomaton {
```

```
private int[] cells;        //  array to hold the current cell states
private int[] rules;        //  array to hold the current rule values

/**
 * Initializes a CellularAutomaton of array size n. The rules should be
 * populated according to Rule #90 from the background section of the lab.
 * Additionally, the center cell should be turned 'on' (initialized to 1).
 *
 * @param n The size of the array of cells.
 *  Precondition: n > 0
 */
public CellularAutomaton(int n) {
  // To be implemented in Activity #1, Exercise 1A
}

/**
 * Initializes a CellularAutomaton of array size n with a rule set in
 * accordance to the given rule number. Additionally, the center cell should
 * be turned 'on' (initialized to 1).
 *
 * @param n The size of the array of cells.
 *  Precondition: n > 0
 * @param ruleN The denary rule number the cellular automaton should follow.
 *  Precondition: 0 <= ruleN <= 255
 */
public CellularAutomaton(int n, int ruleN) {
  // To be implemented in Activity #1, Exercise 2B
}

/**
 * Evolves the current set of cells according to the stored rule set and
 * prints the result.
 */
public void evolve() {
  //  To be implemented in Activity #1, Exercise 1C
}

/**
 * Helper method to return the denary value representing the given cell's
 * neighbourhood.
 *
 * @param index  The index of the cell to get the neighbourhood value of.
 *  Precondition: index is valid for the current group of cells.
 * @return  A denary value representing the given cell's neighbourhood.
 */
private int getNeighbourValue(int index) {
  //  To be implemented in Activity #1, Exercise 1B(i)
}

/**
 * Helper method to return an array representing the next generation of cells.
 *
 * @return  An array representing the next generation of cells.
 */
private int[] getNextGeneration() {
  //  To be implemented in Activity #1, Exercise 1B(ii)
}

/**
 * Helper method to return an array representing the rule set from a given
 * denary rule number.
 *
 * @param ruleN A denary value representing a rule number.
 *  Precondition: 0 <= ruleN <= 255
 * @return  An array representing the rule set from the given denary rule
 *          number.
```

```
    */
    private int[] getRulePattern(int ruleN) {
      // To be implemented in Activity #1, Exercise 2A
    }

   /**
    * Override of the toString method for printing the cellular automaton.
    *
    * @return  A string giving a visual representation of the current group of
    *          cells in the cellular automaton.
    */
   public String toString() {
     String output = "";
     for (int i = 0; i < cells.length; i++)
       if (cells[i] == 0)
         output += (char)0x25A1;
       else
         output += (char)0x25A3;
     return output;
   }
}

class WrappingCellularAutomaton extends CellularAutomaton {
   public WrappingCellularAutomaton(int n) {
     super(n);
   }

   public WrappingCellularAutomaton(int n, int ruleN) {
     super(n, ruleN);
   }

   //  Additional method(s) to be implemented in Activity #2, Exercise 1
}

class SpecialCaseCellularAutomaton extends CellularAutomaton {
   private int[] specialRules;

   public SpecialCaseCellularAutomaton(int n, int ruleN, int specialN) {
     super(n, ruleN);
     //  Processing of specialN to be implemented in Activity #2, Exercise 2
   }

   //  Additional method(s) to be implemented in Activity #2, Exercise 2
}
```