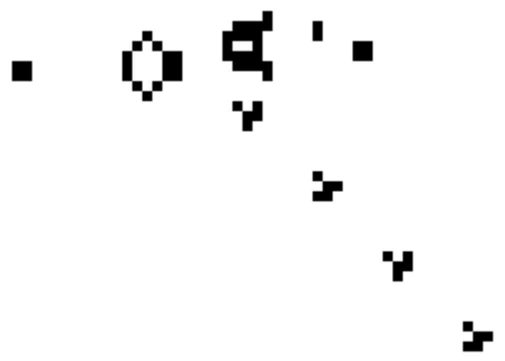
Instructor created

14.1 Project 01 - Elementary Cellular Automaton

Introduction to Cellular Automata

In this project, the simplest version of a one-dimensional cellular automaton is implemented. Cellular automata are a collection of **cells** on a specified grid, often called a **world**, where each cell is either **active** or **inactive** (i.e. alive or dead, on or off, 1 or 0, etc.). Rules are specified for how the status of a cell can change through the evolution of the world, where the rules typically involve an analysis of the active statuses of the nearest neighbors for each cell. For certain world types and rule configuration, cellular automata can act as an idealized simulation for life, characterized by periods of determinism sprinkled with moments of chaos. Conway's two-dimensional Game of Life is perhaps the most well-known cellular automaton. An example world for Conway's Game of Life is shown below

Figure 14.1.1: Conway's Game of Life - Gosper's Glider Maker



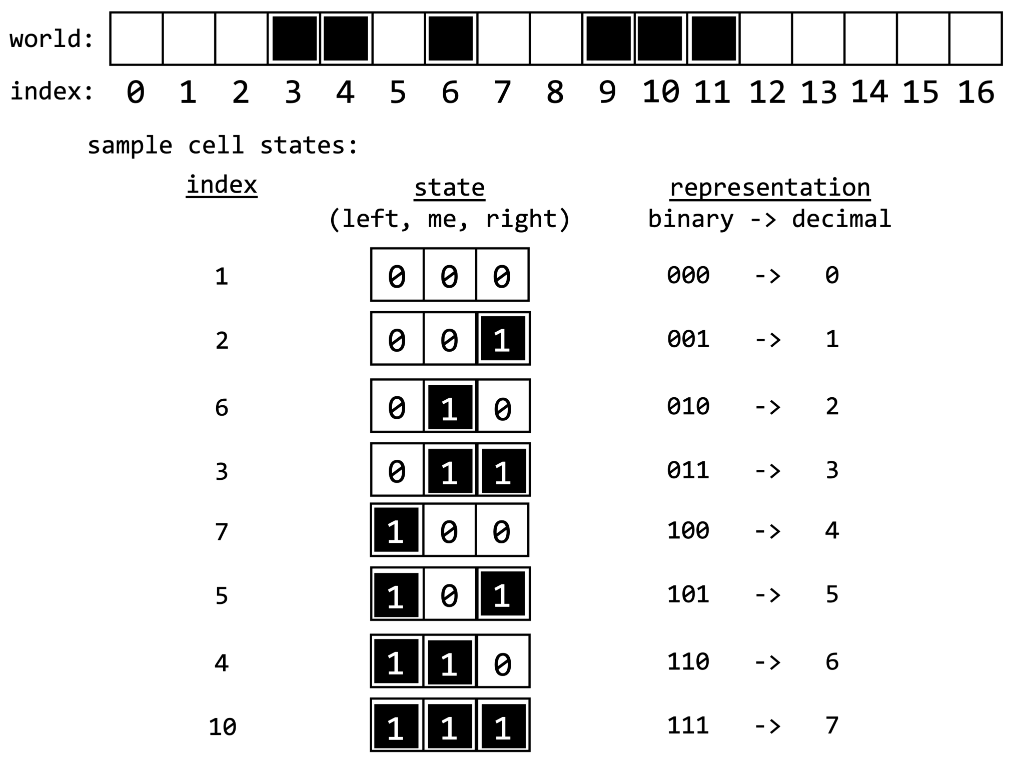
Elementary Cellular Automata

This project implements the simplest version of a one-dimensional cellular automaton.

A Sample World & All Possible States

The **world** is one-dimensional, and the **rules** for evolving a cell's active status depend only on the active statuses of the cell itself and the nearest neighbors to the left and right. A 17-cell sample world is shown below, with eight sample **cell states** that comprise ALL possible cell states, since each cell is either active (1) or not (0), and the state is made up of the 3 nearest cells' active statuses; 2 possible statuses for each cell and 3 total cells, so the total number of possible states is 2^3 = 8. Inactive cells are blank/white, while active cells are filled/black. Note that for this example, the **world is an array** of size 17, where **each element of the array is a cell** composed of a **single Boolean to store its active status** (active=1 or inactive=0) AND a local **Boolean array of size 3 to represent the cell's current state.**

Figure 14.1.2: Sample World with ALL Possible States



index	state (left, me, right)	representation binary -> decimal
1	0 0 0	000 -> 0
2	0 0 1	001 -> 1
6	0 1 0	010 -> 2
3	0 1 1	011 -> 3
7	1 0 0	100 -> 4
5	1 0 1	101 -> 5
4	1 1 0	110 -> 6
10	1 1 1	111 -> 7

Evolution Rules

The **rules** for evolving a world in our elementary cellular automaton are determined from a user-specified integer value in the range 0-255, since they can be uniquely represented by 8-bits. Recall that binary (i.e. base 2) numbers can be converted to the more traditional decimal (i.e. base 10) numbers, using increasing powers of 2. For example,

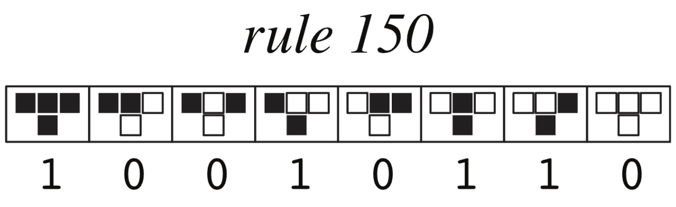
$$10010110 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 128$$

.

For a refresher on binary numbers, review section 2.13 - Binary Numbers.

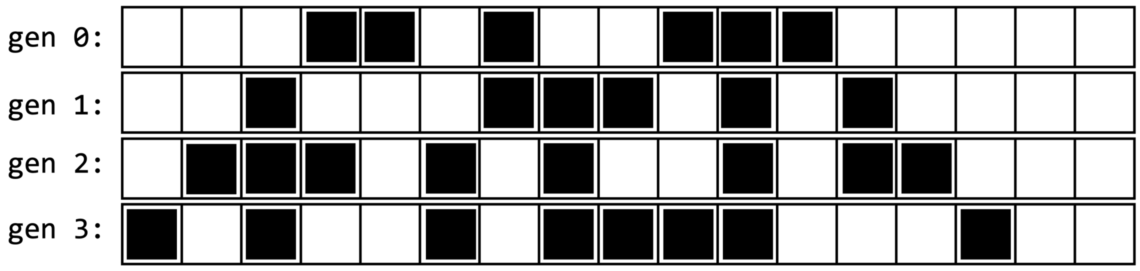
Since there are only 8 possible states for each cell, we can uniquely assign each state to one of the bits for the rule # entered by the user. Furthermore, the bit value in the binary representation of the rule # sets the evolution behavior for the associated state. For example, let **rule = 150**, which was calculated above as a sum of powers of 2 for the following exponents: 7, 4, 2, and 1. Thus rule = 150 is equivalent to 10010110 in binary. Thus, only the cells that have a state equivalent to 7, 4, 2, or 1 will be active in the next generation. Cells with states equivalent to 6, 5, 3, and 0 will be inactive in the next generation. A pictorial representation of the evolution of all possible states for rule 150 is shown below, where the top row gives the state and the bottom row

Figure 14.1.3: Evolution of All Possible States for Rule 150



Note that for rule 150, states with 1 or 3 active cells are active in the next generation, while states of 0 or 2 active cells are inactive in the next generation. Applying rule 150 to the sample world produces the following 3-generation evolution (where each successive generation is shown below the previous generation):

Figure 14.1.4: Evolution of Sample World using Rule 150 for 3 Generations



Of course, this is only ONE sample world (world size and initial active statuses could vary) applying only ONE rule (out of 256 possible rules). We could also extend the the evolution rules to include more neighboring cells. The possibilities are endless. Thus, we need to put some boundaries for the scope of this project:

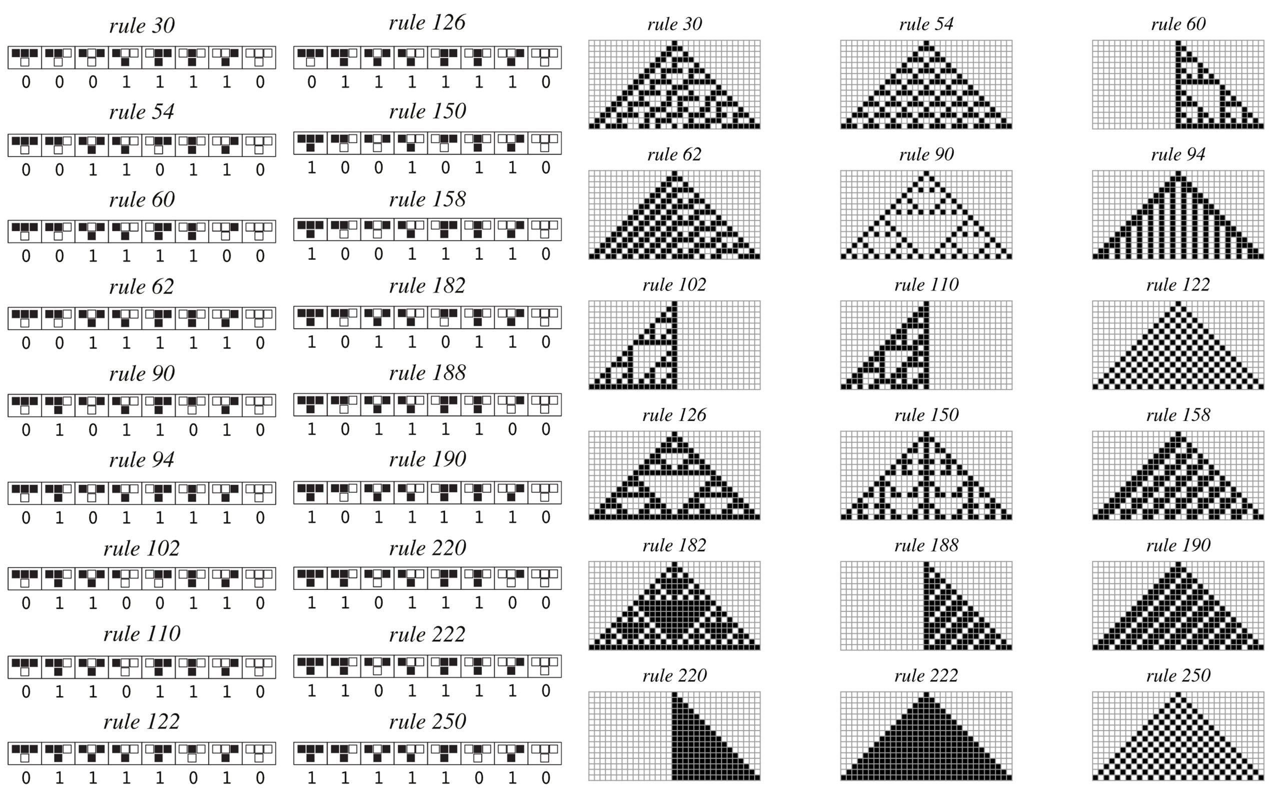
Scope/Limitations of our Elementary Cellular Automaton

- the world will ALWAYS have size 65, given as a global constant variable WORLD_SIZE

- the initial world (i.e. generation 0) ALWAYS starts with a single active cell at the midpoint of the world, i.e. index 32, which leaves 32 inactive cells on each side of the active cell in the middle
- evolution rules ALWAYS use the 3-bit state of the cell, which is determined by the active statuses of the cell immediately to the left, the cell itself, and the cell immediately to the right; thus, each possible state can be represented by a unique 3-bit binary number, which can be converted to a unique decimal number 0-7.
- the world is wrapped front-to-back and back-to-front; so world[0]'s left neighbor is the last element world, and the last element's right neighbor is world[0]
- the rule number must be an integer between 0 and 255, so that it can be converted to a bit array of size 8, each bit associated with an evolution rule for a unique 3-bit state

With these limitations, a wide range of interesting behavior results across the possible rules. Here is a sample of world evolutions for 18 rules that exhibit different and interesting behavior(see the Wolfram MathWorld article for world evolutions for ALL 256 rules):

Figure 14.1.5: Sample World Evolutions



Implementation - an array of cell structs

The starter code for this project includes full scaffolding to achieve all of the required tasks/milestones. You are encouraged to navigate to the bottom of this description to find the

IDE and scan the starter code in main.c, taking note of the struct definition, function definitions, and helpful comments describing each coding task.

We are using a **struct** to represent each **cell** of the **world**, with the following definition:

```
typedef struct cell_struct{
    bool state[3]; //active status for [left, me, right] cells
    bool active; //current active status for this cell
} cell;
```

The **world** is then represented as an **array of cell structs:**

```
cell world[WORLD_SIZE];
```

Implementing the cellular automaton in this way allows a simple procedure to evolve the world to the next generation as follows:

1. update the **active** subitem for each cell based on their **state** array subitem for the rule #
2. update the **state** array subitem for each cell using the left, me, and right cell's **active** subitems
3. print the world

Make sure to fully update ALL cell's **active** subitems before beginning to update the **state** array subitems.

Sample Output (inputs: rule = 90, #generations = 32)

```
Welcome to the Elementary Cellular Automaton!
Enter the rule # (0-255): 90

The bit array for rule #90 is 01011010

The evolution of all possible states are as follows:
|***|   |** |   |* *|   |*  |   | **|   | * |   |  *|   |   |
 | |     |*|     | |     |*|     |*|     | |     |*|     | |

Enter the number of generations: 32

Initializing world & evolving...
                                *
                               * *
                              *   *
                             * * * *
                            *       *
                           * *     * *
                          *   *   *   *
                         * * * * * * * *
                        *               *
                       * *             * *
```

```
                     *   *           *   *
                    * * * *         * * * *
                   *       *       *       *
                  * *     * *     * *     * *
                 *   *   *   *   *   *   *   *
                * * * * * * * * * * * * * * * *
               *                               *
              * *                             * *
             *   *                           *   *
            * * * *                         * * * *
           *       *                       *       *
          * *     * *                     * *     * *
         *   *   *   *                   *   *   *   *
        * * * * * * * *                 * * * * * * * *
       *               *               *               *
      * *             * *             * *             * *
     *   *           *   *           *   *           *   *
    * * * *         * * * *         * * * *         * * * *
   *       *       *       *       *       *       *       *
  * *     * *     * *     * *     * *     * *     * *     * *
 *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Sample Output (inputs: rule = 30, #generations = 32)

```
Welcome to the Elementary Cellular Automaton!
Enter the rule # (0-255): -10
Enter the rule # (0-255): 256
Enter the rule # (0-255): 30

The bit array for rule #30 is 00011110

The evolution of all possible states are as follows:
|***|   |** |   |* *|   |*  |   | **|   | * |   |  *|   |   |
 | |     | |     | |     |*|     |*|     |*|     |*|     | |


Enter the number of generations: 32

Initializing world & evolving...
                                *
                               ***
                              **  *
                             ** ****
```

```
                           **  *   *
                          ** **** ***
                         **  *    *  *
                        ** ****  ******
                       **  *   ***     *
                      ** **** **  *   ***
                     **  *    * **** **  *
                    ** ****  ** *    * ****
                   **  *   ***  **  ** *   *
                  ** **** **  *** ***  ** ***
                 **  *    * ***   *  ***  *  *
                ** ****  ** *  * *****  *******
               **  *   ***  **** *    ***      *
              ** **** **  ***    **  **  *    ***
             **  *    * ***  *  ** *** ****  **  *
            ** ****  ** *  ******  *   *   *** ****
           **  *   ***  ****     **** *** **   *   *
          ** **** **  ***   *   **    *   * * *** ***
         **  *    * ***  * *** ** *  *** ** * *   *  *
        ** ****  ** *  *** *   *  ****   *  * ** ******
       **  *   ***  ****   ** *****   * ***** *  *     *
      ** **** **  ***   * **  *    * ** *     *****   ***
     **  *    * ***  * ** * ****  ** *  **   **    * **  *
    ** ****  ** *  *** *  * *   ***  **** * ** *  ** * ****
   **  *   ***  ****   **** ** **  ***    * *  ****  * *   *
  ** **** **  ***   * **    *  * ***  *  ** ****   *** ** ***
 **  *    * ***  * ** * *  ***** *  ******  *   * **   *  *  *
** ****  ** *  *** *  * ****     ****     **** ** * * *********
```

Coding Tasks/Milestones

The process to fully implement the elementary cellular automaton is broken up into 8 tasks, as detailed in the following 8 sections. The starter code includes **"TODO"** statements for each task. The autograder is organized with test cases associated with each task, in order. Thus, you are strongly encouraged to successfully pass all test cases for each task before moving on to the next task.

Task 1 - setBitArray()

Write the **setBitArray()** function, which converts the input integer **rule** to its binary representation. Limit the valid values of **rule** to the range 0-255. Thus, an 8-bit binary representation is sufficient. Store the 8-bits in the bool array **bitArray[8]**. Note: because arrays are typically presented with index 0 on the left and increasing indices to the right, it may appear that the bits are stored in reverse order. For example, if **rule = 6**, then it's binary representation is **00000110**, which will get stored in **bitArray** as **{01100000}**. In addition to updating **bitArray**, the function should **return true** if the input value for **rule** is valid (i.e. 0-255) and **return false** if the input value for **rule** is invalid.

Task 2 - main() - generate the bit array for the user-specified rule #

In **main(),** read in a valid **rule** # and generate the rule's 8-bit rule bit array, using a call to the function written in Task 1. If an invalid rule # is entered, repeat the prompt and scan another value in. Continue prompting and scanning until a valid value is read in. Print the bit array so that it displays the bits in the correct binary number order, with the bit for 2^0 at the far right and increasing powers of two going to the left. See the sample outputs above for correct formatting of prompts and code output. It is okay to write additional functions to decompose your code and call them from main() for this task.

Task 3 - main() - evolution steps for ALL possible states

In **main(),** use the rule bit array from Task 2 to report the evolution step for all possible cell states. See the sample outputs above for correct formatting of code output. For this task, the autograder even checks for correct whitespaces. It is okay to write additional functions to decompose your code and call them from main() for this task.

Task 4 - stateToIndex()

Write the **stateToIndex()** function, which converts a 3-bit **state** array to the associated **index** of the rule bit array. For example, if **state = {0 1 1}**, then the function should return **3**. As another example, if **state = {1 0 1}**, then the function should return **5.**

Task 5 - setStates()

Write the **setStates()** function, which should update the state array for each cell in the world, using the current active status for the nearby [left, me, right] cells. Remember, the world is wrapped back-to-front and front-to-back. So, you can find world[0]'s left neighbor at the end of the array. Similarly, the last cell in the array has a right neighbor in world[0].

Task 6 - main() - initialize the world

In **main(),** allow the user to enter the number of new generations to evolve the world. Then, initialize the world with ONLY the middle cell active. All other cells begin inactive. Make sure to also set the initial state array for each cell using a call to the function written in Task 5. Note: the user enters the number of generations to generate TOTAL, including the initial state. For example, if the user enters 3, then the final output will include 3 printed generations of the world, i.e. the initial world followed by 2 new generations. If the user enters 0 (or less) for the number of generations, then still print the initial state, but no further evolution generations. See the sample outputs above for correct formatting of code output. It is okay to write additional functions to decompose your code and call them from main() for this task.

Task 7 - evolveWorld()

Write the **evolveWorld()** function, which should evolve each cell's active status to the next generation using its state array. The array ruleBitArray[8] contains the 8-bits for the rule #, stored in reverse order when compared to how binary numbers are typically written.

Task 8 - main() - evolve the world through user-specified number of generations

In **main(),** evolve the world a user-specified number of total generations. After each evolution step, print each active cell as '*' and each inactive cell as ' ' (i.e. whitespace). See the sample outputs above for correct formatting of code output. For this task, the autograder even checks for correct whitespaces. It is okay to write additional functions to decompose your code and call them from main() for this task.

Requirements

- The world must be represented ONLY by the cell struct array as declared in main() in the starter code. No copies of the world can be made. That is, do NOT declare an additional Boolean array of size WORLD_SIZE; the struct array is sufficient to fully complete evolutions steps. Violations of this requirement will receive a manually graded deduction.
- Use the starter code as provided, adding code to complete functions, without making any structural changes: do NOT modify the cell struct definition (no name change, do not add subitems, do not remove subitems, do not modify subitem definitions, etc.), and do NOT change the function headers (no name changes, do not add parameters, do not remove parameters, do not modify parameter types, etc.). Violations of this requirement will receive a manually graded deduction.
- Solve each task and the program at large as intended, i.e. build a cell struct array for the world, and use the bit array associated with the rule # to evolve the active subitems to the next generation based on the state of each cell in the previous generation. Violations of this requirement will receive a manually graded deduction.
- Coding style is manually graded, worth 25% of the total project score. Style points are awarded for following the course standards and best practices laid out in the syllabus, including a header comments, meaningful identifier names, effective comments, code layout, functional decomposition, and appropriate data and control structures.
- Programming projects must be completed and submitted individually. Sharing of code between students in any fashion is not allowed. Use of any support outside of course-approved resources is not allowed, and is considered academic misconduct. Examples of what is allowed: referencing the zyBook, getting support from a TA, general discussions

about concepts on piazza, asking questions during lecture, etc. Examples of what is NOT allowed: asking a friend or family member for help, using a "tutor" or posting/checking "tutoring" websites (e.g. Chegg), copy/pasting portions of the project description to an AI chatbot, etc. Check the syllabus for Academic Integrity policies. Violations of this requirement will receive a manually graded deduction, and may be reported to the Dean of Students office.

Submission

Develop your program in the IDE below. Use the "Run" button to test your code interactively as you develop your program. Use the "Submit for grading" button to test your code against the suite of autograded test cases, which also submits your program for grading. Formally, the official code submission is your latest submission that scores the highest when run through the autograder. Late submissions are allowed for a point reduction. If you submit your highest scoring program after the deadline it will be treated as a late submission, which will incur the point reduction.

LAB ACTIVITY | 14.1.1: Elementary Cellular Automaton 0 / 100