



Students:

Section 14.1 is a part of 1 assignment:

Programming Project 01 - Totalistic Cellular Automaton (Coding in C: A Total Cell-ebration!)

Includes:  zyLab

Due: 09/12/2024, 11:59 PM CDT



This assignment's due date has passed. Activity will still be recorded, but will not count towards this assignment (unless the due date is changed). See [this article](#) for more info.

Instructor created



14.1 P01 - Coding in C: A Total Cell-ebration!

Project 01 - Totalistic Cellular Automaton

Introduction to Cellular Automata

In this project, a simple version of a totalistic cellular automaton is implemented. [Cellular automaton](#) are a collection of **cells** on a specified grid, often called a **world**, where each cell stores a value representing its active status. Some worlds only allow binary statuses, where each cell is either **active** or **inactive** (i.e. alive or dead, on or off, 1 or 0, etc.), and other worlds allow multiple status *colors*; for example, in a 3-color world, the values 0, 1, and 2 may represent the cell statuses of **inactive**, **at-risk**, and **thriving**, respectively. 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 evolution for Conway's Game of Life is shown below

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



Totalistic Cellular Automata

This project implements the simplest version of a [one-dimensional totalistic cellular automaton](#). The evolution of a one-dimensional cellular automaton can be illustrated by displaying the world's generations one after another, line-by-line. In the following illustration, the initial world, shown on the top row, only has one cell that is **at-risk (-)**, while the rest of the cells are **inactive (whitespace)**. In the second generation, on the second row, there are now three cells **at-risk (-)**. Then, in the third generation, on the third row, two cells are now **thriving (+)** and three cells are **at-risk (-)**. The evolution continues, showing overall growth of the world, as specific cell status vary from **inactivity** to **at-risk** to **thriving**.

```

      -
    ---
  -+-+--
 --    --
-++- -++-
 --  -+-  --
-++---- -+++-
 --  -+++-- --
-++- -- - -- -++-
--  -++++-++++- --

```

All Possible Local Cell States and Local Sums

Our **world** is a one-dimensional collection of cells that can take on one of three statuses, with values of 0, 1, or 2. The **rules** for evolving a cell's status value depend only on the sum of the status values of the cell itself and the nearest neighbors to the left and right. Thus, there are

$3^3 = 27$ possible local cell states, since the left neighbor can take on 3 unique status values, the cell itself can take on 3 unique status values, and the right neighbor can take on 3 unique status values. However, in this totalistic cellular automaton, all that matters is the local sum of the three neighbors, which can only take on the values 0 through 6.

In order to visualize a world, it will be helpful to represent the three possible status values as the following meaningful characters:

status value of 2 is represented as '+'

status value of 1 is represented as '-'

status value of 0 is represented as ' ' (i.e. a whitespace)

Here is a sample world of size 17 that uses this representation, where the local sums are calculated and shown beneath the world representation:

Figure 14.1.2: Sample World (Wrapped) with Local Sums

| | | | | | | | | | | | | | | | | | |
|--------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| value: | 1 | 2 | 1 | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 1 | 0 | 2 | 1 | 1 | 1 |
| Gen 1 world: | - | + | - | | | | - | - | + | + | + | - | | + | - | - | - |
| localSum: | 4 | 4 | 3 | 1 | 0 | 1 | 2 | 4 | 5 | 6 | 5 | 3 | 3 | 3 | 4 | 3 | 3 |

Note that this is a *wrapped* world, such that the first cell and last cell are neighbors. Specifically, the first cell's left neighbor is actually the last cell shown. Similarly, the last cell's right neighbor is the first cell shown. The wrapping results in a local sum of 4 for the first cell, since its left (wrapped) neighbor stores value 1, its own cell value is 1, and its right neighbor stores value 2.

Here are all possible local cell states, grouped by their local sums:

| | | | | | | | |
|------------|-----|-----|-----|-----|-----|---|---|
| Local Sum: | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| States: | +++ | ++- | ++ | +- | -- | - | |
| | | +-+ | + + | + - | - - | - | |
| | | -++ | ++ | -+ | -- | - | |
| | | | +-- | +- | + | | |

| -+- | | - + | | + |

| --+ | | -+ | | + |

| --- |

Evolution Rules

The **rules** for evolving a world in our totalistic cellular automaton from one generation to the next are determined from a user-specified integer. Since there are 7 possible local sum values (i.e. 0-6), each of which can be mapped to one of the three possible status values (i.e. 0, 1, or 2) for the next generation, there are $3^7 = 2187$ unique sets of rules. In fact, the integer values in the range 0-2186 can be uniquely converted to 7 status values, using a *decimal* (i.e. base-10) to *ternary* (i.e. base-3) conversion.

It may be helpful to review *binary* (i.e. base-2) numbers and the binary-to-decimal conversion process. 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,

$$10010100 = 1*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = 128 + 16 + 4 = 148$$

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

The ternary-to-decimal transformation is similar to a binary-to-decimal conversion, but instead of converting numbers from bits that have two possible values, we must convert numbers from ternary digits that can have three possible values. For example,

$$1001210 = 1*3^6 + 0*3^5 + 0*3^4 + 1*3^3 + 2*3^2 + 1*3^1 + 0*3^0 = 729 + 27 + 18 + 3 = 777$$

Since there are only 7 possible local sums for any given cell, rule #s in the range 0-2186 can be uniquely transformed to 7 ternary digits, where each ternary digit can be 0, 1, or 2, which directly represents the next generation's status value for each of the 7 possible local sums. Using rule# 777 from the sample ternary-to-decimal transformation calculation above, we see that **777** is represented in base-3 as **1001210**. If we think of these 7 ternary digits as an array of status values (stored in reverse order), then the array index is the cellular local sum for applying the rule. Thus, for rule 777:

- if a cell's local sum is 0, 4, or 5, then the next generation's status value will be 0;

- if a cell's local sum is 1, 3, or 6, then the next generation's status value will be 1; and
- if a cell's local sum is 2, then the next generation's status value will be 2.

| | | | | | | | |
|-------------|---|---|---|---|---|---|---|
| Local Sum: | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| New Status: | - | | | - | + | - | |

Let's now evolve the sample world from above by applying rule 777 for one generation:

Figure 14.1.3: Sample World Evolution to Generation 2

| | | | | | | | | | | | | | | | | | |
|--------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| value: | 0 | 0 | 1 | 1 | 0 | 1 | 2 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| Gen 2 world: | | | - | - | | - | + | | | - | | - | - | - | | - | - |
| localSum: | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 2 | 1 | 1 | 2 | 2 | 3 | 2 | 2 | 2 | 2 |

Lastly, let's complete the example with evolving the sample world a few generations by applying rule 777:

Figure 14.1.4: Evolution of Sample World using Rule 777 for 5 Generations

| | | | | | | | | | | | | | | | | | |
|--------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Gen 1 world: | - | + | - | | | | - | - | + | + | + | - | | + | - | - | - |
| Gen 2 world: | | | - | - | | - | + | | | - | | - | - | - | | - | - |
| Gen 3 world: | - | - | + | + | + | - | - | + | - | - | + | + | - | + | + | + | + |
| Gen 4 world: | | | | - | | | | | | | | | | | - | - | |
| Gen 5 world: | | | - | - | - | | | | | | | | | - | + | + | - |

Interestingly, the world appears to be flourishing into Generation 3, followed by a rapid downturn with almost all cells becoming inactive in Generation 4, before a slow regrowth into Generation 5. This is a direct result of rule 777 having a 0 status value associated with local sums of 4 and 5. That is, when the world is composed of mostly status values of 1 and 2, the local sums for most cells will be 4 or 5, which will all then evolve to a 0 cell in the next generation. The only thing saving this world is the few cells with local sums of 6, since that evolves to a status value of 1.

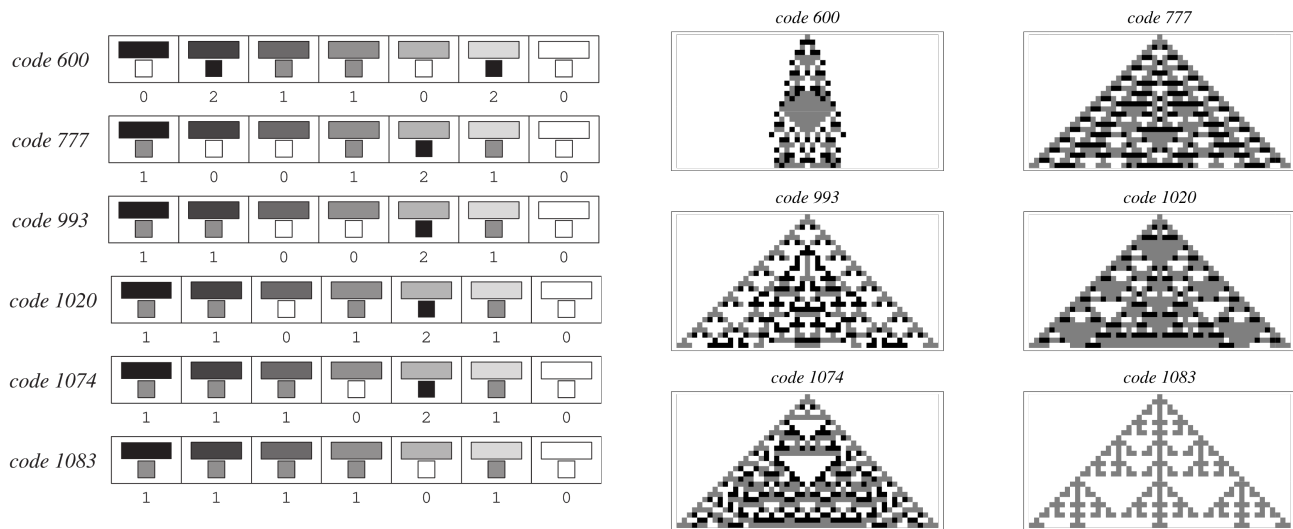
Note that the above example and resulting discussion only pertains to rule 777. The user has 2187 possible rules to apply, many of which will result in unique and interesting behaviors of their own. In summary, this is only ONE sample world (world size and initial status values could vary) applying only ONE rule (out of 2187 possible rules). We could also extend the evolution rules to include more neighboring cells or use a local state analysis in place of the local sum. The possibilities are endless. Thus, we need to put some boundaries for the scope of this project:

Scope/Limitations of our Totalistic Cellular Automaton

- The world ALWAYS has size 65, given as a global constant variable `WORLD_SIZE`.
- The initial world (i.e. generation 0) ALWAYS starts with a single non-zero cell at the midpoint of the world, i.e. index 32, which leaves 32 zero-valued cells on each side of the non-zero cell in the middle.
- Evolution rules ALWAYS use the local sum for the cell, which is determined by the adding the status values for the cell immediately to the left, the cell itself, and the cell immediately to the right; thus, each cell must have a local sum between 0-6.
- The world is wrapped front-to-back and back-to-front; so `world[0]`'s left neighbor is the last element in `world[]`, and the last element's right neighbor is `world[0]`.
- The rule number must be an integer between 0 and 2186, so that it can be converted to a size-7 value array of ternary (base-3) digits, where each digit is associated with an evolution rule for a unique local sum of the left, center, and right neighborhood.

With these limitations, a wide range of interesting behavior results across the possible rules. Here is a sample of world evolutions for 6 rules that exhibit different and interesting behavior (note: in this figure, cell status value is represented by shades of grey; specifically, 2=black, 1=mid-grey, and 0=white; in the evolution rules breakdown, the local sum is colored from white=0 to black=6, with many shades of grey in between representing local sums between 1 and 5; see the [Wolfram MathWorld article](#) for more details):

Figure 14.1.5: Sample World Evolutions using Greyscale Visualization



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{
    int localSum; // total sum of local [left, me, right] cells (possible values 0-6)
    int status;   // this cell's current status (0, 1, or 2)
    int count;    // running accumulated count of this cell's active status for all generations
} 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 **status** subitem for each cell based on its **localSum** subitem for the rule #
2. update the **count** subitem with the new **status** for each cell
3. update the **localSum** subitem for each cell using the left, me, and right cell's **status** subitems
4. print the world

Make sure to fully update ALL cell's **status** subitems before beginning to update the

Sample Output and Solution Executable

Shown below are two sample program outputs. User-input is shown in red within the program output. The sample outputs should be used to develop your program, specifically to match the formatting EXACTLY, including ALL whitespace. The number of whitespaces can be determined by highlighting the sample output. Additionally, a working version of the program is provided as **demo.exe** in the starter file tree. You can run it to play around with generating world evolutions, view the expected behavior of the program, and/or check formatting of outputs, which may be easier for checking whitespace. To run **demo.exe**, you first must use the UNIX command **chmod** to change the permissions of the file to allow all users to execute it, as follows:

```
→ chmod a+x demo.exe
→ ./demo.exe
```

Sample Output (inputs: rule = 777, #generations = 32, initial middle value = 1)

```
Welcome to the Totalistic Cellular Automaton!
```

```
Enter the rule # (0-2186): 777
```

```
The value array for rule #777 is 1001210
```

```
The evolution of all possible states are as follows:
```

```
Local Sum:    6          5          4          3          2          1          0
```

```
States:      |+++|      |++-|      |++ |      |+- |      |-- |      |-  |      |  |
              |+-+|      |+ +|      |+ -|      |- -|      |-  |
              |-++|      | ++|      |-+ |      |--|      |- |
              |+--|      | +-|      |+  |
              |-+-|      |- +|      | + |
```


$$\left| \begin{array}{ccc} - & - & + \end{array} \right| \quad \left| \begin{array}{ccc} - & & + \end{array} \right| \quad \left| \begin{array}{ccc} & & + \end{array} \right|$$
$$\left| \begin{array}{ccc} & & \\ & & \\ & & \end{array} \right|$$

New Status: $[-]$ $[+]$ $[-]$ $[+]$ $[-]$ $[+]$ $[-]$ $[+]$

Enter the number of generations (1-49): 32

Enter the value (1 or 2) for the initial active cell: **1**

Initializing world & evolving...

1

— — —

3

$$- + - + -$$

7

— — — — —

4

$-++-$ $-++-$

12

— — — + — — —

8

$-++---$ $---++-$

16

-- -+++ --

12

$$-++- \quad -- \quad - \quad -- \quad -++-$$

17

-- -++++-++++- --

23

$-++---$ $--$ $--$ $---++-$

20

-- -++++- -++++- --

24

$-++-$ $--$ $--$ $-+-$ $--$ $--$ $-++-$

24

-- -+++++-- -+++++-- --

36

$-++--$ $-----$ $-+-$ $-----$ $-----++-$

30

```

--      -+++---+- -+----++- --
34
-+- -- - - +- - - - +-
24
-- -++++----- -- - - -++++- --
36
-+---- -- ---+++++++--- -- ---+-
44
-- -++++-+- ----- +-++++- --
39
-+- -- -- ---+-----+--- -- -- +-
35
-- -+++++- +- --- +- -+++++- --
43
-+---- --- --- -- +-+ -- --- --- +-
43
-- -+++--+++--++++- -++++--+++--+++- --
60
-+- -- - - --- -- -- -- - -- -- +-
30
-- -++++- --- +-++++++++- --- -++++- --
58
-+---- -- +-+--+--- ----- ---+-+--+ -- ---+-
51
-- -++++-+- +- +------+ +- +- -++++- --
49
-+- -- -- -- +- --- +- -- -- -- +-
39
-- -+++++- +- -+++- ---+-+--- -+++- +- -+++++- --
71
-+---- --- +- --- -+- -+- - --- +- --- +-
54
-- -+++--+- ---+--+-+--- -- -- +-+-+--- -+-+++- --
66

```

```

11111111111122122222333233322221221111111111
123446880124553586790390676810272018676093097685355421088644321

```

Sample Output (inputs: rule = 993, #generations = 49, initial value = 2)

Welcome to the Totalistic Cellular Automaton!

Enter the rule # (0-2186): 993

The value array for rule #993 is 1100210

The evolution of all possible states are as follows:

Local Sum: 6 5 4 3 2 1 0

States: |+++| |++-| |++ | |+- | |-- | |- | | |

 |+-+| |+ +| |+ -| |- -| | - |

 |-++| | ++| |-+ | | --| | -|

 |+--| | +-| |+ |

 |-+-| |- +| | + |

 |--+| | -+| | +|

 |---|

New Status: |-| |-| | | | | |+| |-| | |

Enter the number of generations (1-49): 49

Enter the value (1 or 2) for the initial active cell: 2

Initializing world & evolving...

```

                +
2
                +++
6
                + - +
5
                ++ - ++
9
                +   -   +
5
                +++ --- +++
15
```

[illegible]

38

++ -- + -- + -+ - +-- + --+ - +- + -- + -- ++

42

+ ++- ++ ++ ++ - - + + + - - ++ ++ ++ -++ +

48

+++ + - +--- --- ++ + ++ --- ---+ - + +++

44

+ - + -- + +++ + + + +++ + -- + - +

36

++ --++ +-+ +++ + - ++ +++ ++ - + +++ -++ +--- ++

62

+ + - - + - + --+ ++ - ++ +-- + - + - - - + +

41

+ ++ -- --+-- ++ --++ + ++ - ++ + ++-- ++ --+-- -- ++ +

61

++ ++++ +-+ + - +++ + --- + +++ - + +-+ ++++ ++

59

- + + -- + + - + ++ --+ - +++ + + +++ - +-- ++ + - + + -- + + -

62

+ ++++ ++ + + - + + - - + + - - + + - + + ++ ++++ +

54

-- + + - ++ +++ -----++ +------ +++ ++ - + + --

54

-++- ++ + - ++ - + - - + - ++ - + ++ -++-

42

- -- - + + -- + --+++ --- --- +++-- + -- + + - -- -

46

--++++- ++ ++ ++ ++ -+ -- + -+ +++ +- + -- +- ++ ++ ++ ++ -++++--

80

---- ++ + - + ++ ----

21

-+ +- + ++ --- ++ + -+ +-

27

- ++ - +++ + + -+ +- + + +++ - ++ -

38

+--+ +--- + - ++++ ++++ - + ---+ +-+

42

-- ++ +- ++ --+ -- + + -- +-+ ++ -+ ++ --

42

+ + + - + + ++ ++ ++ ++ + + - + + +

38

```

+++ ++ +- ---      +++ +++ +      + +      + +++ +++      ---++ ++ +++
66
--      -  +-      + -      -  ++      ++ ++      ++ -      - +      -+ -      --
36
-----
11112212222121112222223333423344344332433332222221112122221221111
56875140300272893675675298199717371799189257657639827200304157865

```

Coding Tasks/Milestones

The process to fully implement the totalistic 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 - setValArray()

Write the **setValArray()** function, which converts the input integer **rule** to its ternary (i.e. base-3) representation. Limit the valid values of **rule** to the range 0-2186. Thus, an 7-digit ternary (i.e. 7 base-3 digits) representation is sufficient. Store the 7 ternary digits in the array **valArray[7]**. Note: because arrays are typically presented with index 0 on the left and increasing indices to the right, it may appear that the digits are stored in reverse order. For example, if **rule = 777**, then its ternary representation is **1001210**, which will get stored in **valArray** as **{0121001}**. In addition to updating **valArray**, the function should **return true** if the input value for **rule** is valid (i.e. 0-2186) and **return false** if the input value for **rule** is invalid.

Task 2 - main() - generate the status value array for the user-specified rule

In **main()**, read in a valid **rule #** and generate the rule's 7-ternary-digit status value 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 status value array so that it displays the digits in the correct ternary number order, with the digit for 3^0 at the far right and increasing powers of 3 going to the left. See the sample outputs above for correct formatting of prompts and code output. It is a good idea 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 status value 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. Note that the display for the lines with "*Local Sum:*" and "*States:*" are the same for every run of the program. Thus, these lines can be hardcoded into print statements. However, the "*New Status:*" line depends on the rule # entered by the user and must be programmed to print the correct characters for that specific rule #. For this task (and all tasks), the autograder checks for correct whitespaces. It is a good idea to write additional functions to decompose your code and call them from **main()** for this task.

Task 4 - setSums()

Write the **setSums()** function, which should update the **localSum** subitem for each cell in the world, using the current status values 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 5 - main() - scan-in user-specified inputs and initialize the world

In **main()**, allow the user to enter the number of TOTAL generations for evolving the world, allowing repeated attempts for invalid values. The total number of generations must be between 1 and 49. Then, read in the initial status value for the middle cell, again allowing repeated attempts for invalid values. The initial status value for the middle cell must be either 1 or 2. Lastly, initialize the world with all cells' status values set to zero except ONLY the middle cell, which should be set to the non-zero status value that was just entered by the user. Make sure to also initialize the **count** subitems and set the **localSum** subitem for the initial world. Note: the user enters the TOTAL number of generations to generate, which includes 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 1, then only the initial generation is printed and no evolution steps are performed. 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 6 - evolveWorld()

Write the **evolveWorld()** function, which should evolve each cell's status value to the next generation using its **localSum** subitem. The array **ruleValArray[7]** contains the 7-ternary-digits for the rule #, stored in reverse order when compared to how ternary numbers are typically written. Remember: the rule status value array directly stores the new status values (i.e. for the next generation) associated with the 7 possible local sums. This function should also update the **count** subitems for all cells in the world as the new generation status values are determined; i.e. increase the **count** subitems with the new generation status values. Lastly, this function should return the total sum of all status values in the world after the evolution step is complete. Note: the local cell **count** subitems AND the returned total sum will all be very useful in the following tasks, where the evolution of the world is displayed in **main()**.

Task 7 - main() - evolve and display the world through the user-specified number of generations

In **main()**, evolve the world the user-specified number of total generations. After each evolution step, print the world using the cell representation shown in the sample output, where each cell prints as '+', '-', or ' ' (whitespace) for status value of 2, 1, or 0, respectively. After the world display, but still on the same line, also print out the total sum of all status values for all cells in that generation. After all generations have been printed and the world evolution is complete, print a solid line of underscores that is the same number of characters as the world itself. Once again, it is vital that you follow the format, including and especially whitespaces, of the sample output EXACTLY.

Task 8 - main() - display the total count for each cell vertically underneath the world evolution

In **main()**, complete the world evolution display by printing the total status value count for each cell directly underneath it. That is, during the world evolution, each cell keeps a running total of its status values over all generations, stored in the **count** subitem. Since cell status values are capped at 2 and we only allow 49 generations, the highest a **count** can be is 98. Thus, **count** is at most a two-digit number. However, a two-digit numbers cannot fit underneath a single cell, since each cell is only one character wide. So, we must display the **count** vertically underneath each cell over two lines, by printing the first digit (i.e. the *tens* place) on the first line, and the second digit (i.e. the *units* place) on the second line.

For example, here is the final generation and the total count display for the first sample output from above:


```
--      -+++--+-  ---+--+-+  --      --      -+-+--+-  -+-+--+-  --  
66
```

```
111111111111221222223332333222212211111111111  
123446880124553586790390676810272018676093097685355421088644321
```

Notice that if a cell has a zero total count, as is the case for the first cell in this example, then both digits are printed as blank whitespaces. For cases where the total count is a single-digit number, such as the second through ninth cells in this example, the first digit is left as a blank whitespace, and the single-digit total count is displayed on the second line only.

Here are some observations from the total count display:

- Throughout the world evolution, the first and last cells ALWAYS stored a status value of zero since their total count is zero.
- The accumulated status values for the eighth and ninth cells are both 8.
- The accumulated status value for the middle cell is 27, while its neighbors to the right and left accumulated a greater total count of 32, which is the maximum total count for any cell in the world.
- Generally, the total counts are low near the edges of the world and increase toward the middle of the world.

You should be able glean this information and more from the sample total count display.

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 any additional integer arrays of size `WORLD_SIZE`; the struct array is sufficient to fully complete evolution 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 status value array associated with the rule # to evolve the **status** subitems to the next generation based on the **localSum** each cell in the previous generation. Violations of this requirement will receive a manually graded deduction.

- Coding style is manually graded as potential deductions from the autograded functionality score, up to 25 total points. Style deductions are associated with not following the course standards and best practices laid out in the syllabus, including a program header, meaningful identifier names, effective comments, code layout, functional decomposition, and appropriate data/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, allowing an AI chatbot to write code for you, 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 & Grading

Develop your program in the IDE below. Do NOT use the "Run" button; instead, compile your code (e.g. "**gcc main.c**") and run the resulting executable (e.g. "**./a.out**") in the terminal 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 functionality grading.

Formally, you must submit your final program (**main.c**) to Gradescope, where your code will be manually inspected for style (details in the syllabus) and project requirements (details listed above). The official code submission is your the version you submit to Gradescope (look for *Project01*).

As detailed in the syllabus, students are allocated a certain number of late days throughout the term, where no late penalty is applied. Students can use their late days whenever they choose, simply by submitting their work to Gradescope after the deadline. However, no more than two late days are allowed for each project.

The autograder has a test case suite to check functionality with the following breakdown:

- **[10 points]** Task 1 - `setValArray()`
- **[15 points]** Task 2 - `main()` - generate the status value array for the user-specified rule #
- **[15 points]** Task 3 - `main()` - evolution steps for ALL possible states

- **[10 points]** Task 4 - setSums()
- **[9 points]** Task 5 - main() - user-input and initialize the world
- **[10 points]** Task 6 - evolveWorld()
- **[10 points]** Task 7 - main() - evolve the world through user-specified number of generations
- **[10 points]** Task 8 - main() - display the total counts
- **[11 points]** Full Program Output - for a variety of input rules, number of generations, and initial values

Copyright Statement.

This assignment description is protected by [U.S. copyright law](#). Reproduction and distribution of this work, including posting or sharing through any medium, such as to websites like [chegg.com](#) or AI chatbots like chatGPT, is explicitly prohibited by law and also violates [UIC's Student Disciplinary Policy](#) (A2-c. Unauthorized Collaboration; and A2-e3. Participation in Academically Dishonest Activities: Material Distribution).

Material posted on [any third party](#) sites in violation of this copyright and the website terms will be removed. Your user information will be released to the author.

LAB
ACTIVITY

14.1.1: Totalistic Cellular Automaton

100 / 100

Run

History

Tutorial

main.c

```

1  /*-----
2  Program 1: Totalistic Cellular Automaton
3  Course: CS 211, Fall 2024, UIC
4  Author: Shreya Ganguly
5  -----*/
6
7  #include <stdio.h>
8  #include <stdbool.h>
9
10 const int WORLD_SIZE = 65;
11 int ternary[7]; // Declare global array that holds 7 status values
12
13 typedef struct cell_struct{
14     int localSum; // total sum of local cells, [left] + [me] + [right]:
15     int status;   // this cell's current status: 0, 1, or 2
16     int count;    // running accumulated count of this cell's active st
17 } cell;
18
19 //return true if input rule is valid (0-2186)
20 //return false if input rule is invalid






```

[Submit for grading](#)Coding trail of your work [What is this?](#)

```
8/30 F- S----- 0,0,0,0 U----- 5----- 10 -
10 ---- 5,5 ---- 10 ----- 25,19,25 - 25,25,25 M 25 - 0 -
25 - 25 - 40 ----- 10,40,0,40 - 40 ----- 10,10,0 - 40
,40,40,40,40,10,40 - 10 ----- 40,40,40 - 45,50 T 0,0,50
,50,50,50,50,50,50,50 - 50,50,50,50,50 - W - - 50 - 50 - - - -
50,50,50 - - 50,54,50 - 56,50,56,59,59,0,59,59,50,59,59,59
R - - 50,50,59,50,59,59,59,59,59,59,59,59,50,59,59,54
,59,54,59 F 20,59,20,44,44,59,59,55,25,0,55,30,45,45,59
,59,59,59,59,59,69,79,79,79,79,79,79,0,79 - - 79,79,79
,79,79,79,79,83,0,100,85,100 U 0,0,79,79 M 83,100,100
min:337
```

Latest submission - 12:02 AM
CDT on 09/23/24Submission
passed all testsTotal score:
100 / 100☒ Only show failing tests (40 tests hidden)[Open submission's code](#)

5 previous submissions

| | | |
|---------------------|-----------|--|
| 12:01 AM on 9/23/24 | 100 / 100 | View  |
| 12:01 AM on 9/23/24 | 83 / 100 | View  |
| 10:58 PM on 9/22/24 | 79 / 100 | View  |
| 10:58 PM on 9/22/24 | 79 / 100 | View  |
| 10:58 PM on 9/22/24 | 0 / 100 | View  |

[Trouble with lab?](#)

[Feedback?](#)

Activity summary for assignment: Programming Project 01 - Totalistic Cellular Automaton (Coding in C: A Total Cell-ebration!)

100 /
100
points

Due: 09/12/2024, 11:59 PM CDT

This assignment's due date has passed. Activity will still be recorded, but will not count towards this assignment (unless the due date is changed). See [this article](#) for more info.

[Completion details](#) 