ATILIM UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING
CMPE 114 – COMPUTER PROGRAMMING II
LAB PROJECT

Instructors: Damla TOPALLI, Ekrem Çağlar YILMAZ
Assistants: Cansen ÇAĞLAYAN, Batuhan COŞAR, Mehtap TUFAN, Ozan Can ACAR
Due Date: **12 June 2020 Friday 23:55**

Objective: You are going to implement a simple cellular automaton which covers only one dimension. Cellular automata are universal computers and they are used to study artificial life [1]. They can be used to simulate a wide range of biological models including swarms, fires, viruses, bacteria etc. You are going to be using arrays to hold your current cell state information.

Restrictions:

- Projects after the deadline will not be accepted, do not mail your projects after the deadline.
- You are going to use C as your programming language.
- You must implement the algorithm based on the limitations described below. Other types of implementations will not be accepted.
- Write clear and understandable code. Use as many functions as you can. Name every variable in an appropriate way. Your functions and variables should be self-explanatory. **Add comments to your code whenever possible.** Readability of your code may cost you a penalty or can grant a bonus to you.
- **This is an individual work; you are expected to work alone. Similarity check will be applied to your codes. Do not cheat you'll get no points if you do.**

Question: You are going to implement a topic called 1D cellular automaton which is the simplest form in the cellular automata domain. In this automaton the whole system has an initial state. Next states are generated from this initial state by applying a rule. Each of the succeeding states are generated from their corresponding previous states. This generation can take place n times and this number will be called **generation** from now on. You are going to represent each state vector with arrays. Only the current and the next state information is needed to be held in a vector so, you only need two 1 dimensional arrays. One for the **current state**, one for the **next state**. Each value in the array corresponds to a cell value. A cell in 1D automata can only have two internal states 0 and 1 (Dead or Alive you may also say). Each cell will change their internal states in the next state according to a certain rule. This rule will be explained later in detail. Each cell has their corresponding neighbor cells, left neighbor and right neighbor. This neighborhood will determine the cell's next internal state at each iteration. You can think the neighborhood like that: Take an index X from your current state array. Your neighborhood will consist from three cells: Left,

middle, right or current_state[X – 1], current_state[X] and current_state[X+1]. Combination of this neighborhood will give you a value. Remember all cells can only have 1 or 0. Let us say the combination is 101. That means the left neighbor is alive, our current index is dead and the right neighbor is also alive. You will take this value and you will get a state for the current index for the next state by looking at your rule. Let's say the rule says 101 is equal to 1. That means in the next state, your current cell (which corresponds to the index of X) will be assigned as 1. You'll continue to perform this process until you reach the generation count. Let's see every parameter & setup in a visual example:

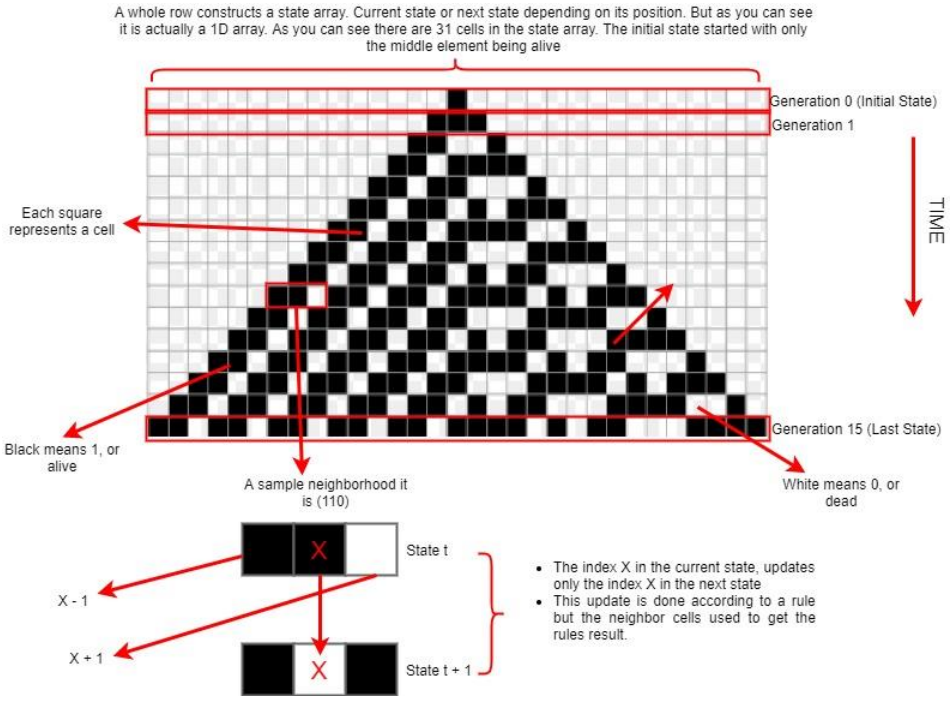


Fig 1: 1D Automaton Definitions

As you can see from the figure every detail of the previously given definitions is explained in detail. Be careful, each index updates the same index in the next state. When a generation is calculated and printed on the console you can assign your next state array to your current state and the remaining part is just to loop this process until you reach the defined generation count. Pseudo code of the desired algorithm can be given as:

```
generation_count = 0
array_size = 50 (or cell_size, each state can hold 50 cells in this case)
middle_point = 24 //finding the middle point because we need to start the initial cell from there
current_state[middle_point] = 1 //only the middle cell is set to 1 (alive) initially
While generation_count < 50 do:
    ForEach cell_x index in current_state[array_size] do:
        left = current_state[cell_x – 1]
        middle = current_state[cell_x]
        right = current_state[cell_x + 1]
        next_state[cell_x] = get_rule(left, middle, right)
    end ForEach
    print(next_state)
    current_state = next_state //Hint: It is easier when you swap these arrays this way
    generation_count ++
end while
```

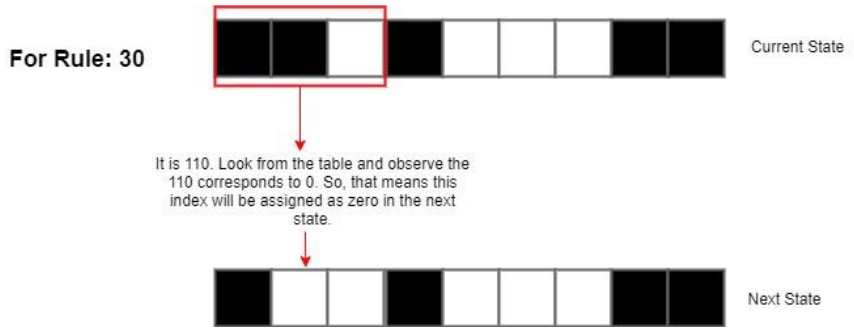
Since a new state for a cell is calculated from a neighborhood that contains 3 cells, you don't need to calculate the states for your first and last cells. So, that means your loop will start from 1 and end when its 48 (for the provided pseudo code example, remember 49 is the last element of the array). Basically, for each cell in the current state array, you'll calculate the next state and you'll continue to do this process until you reach the generation amount. By printing the generated states you'll see that a pattern is generated through this automaton.

Rule Generation: The most important part of the process is the rule generation step. First, the rule size. The rule size is coming from the fact that the state calculation for a single cell is done by using 3 cells from the previous state. So, a 3-cell neighborhood can only take $2^3 = 8$ states (Each cell can only be 1 or 0 hence the 2). For a combination that is matching any of the neighborhood configuration next state value is calculated. There are total of 255 rules in the literature, but you are going to use the rule 30 for your project. The look up table for the rule 30 is given below.

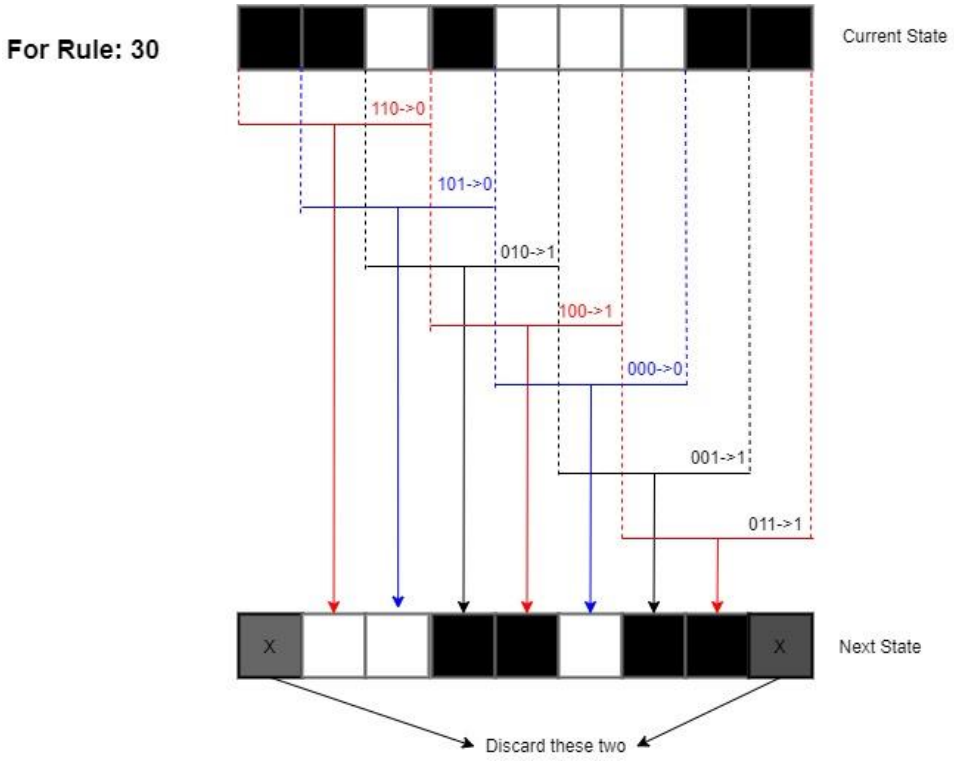
RULE 30 LOOK UP TABLE								
Neighbors	000	001	010	011	100	101	110	111
Next state	0	1	1	1	1	0	0	0

Table 1 Lookup table for rule30

So, based on the neighborhood configuration of your current index's configuration you'll get a next state information for your next state cells. Let's say your current index pointing to a neighborhood which has 101 as its left, middle and right cells (described in details above) you'll assign 0 to your cell which is pointed by the same index and resides in the next state array.

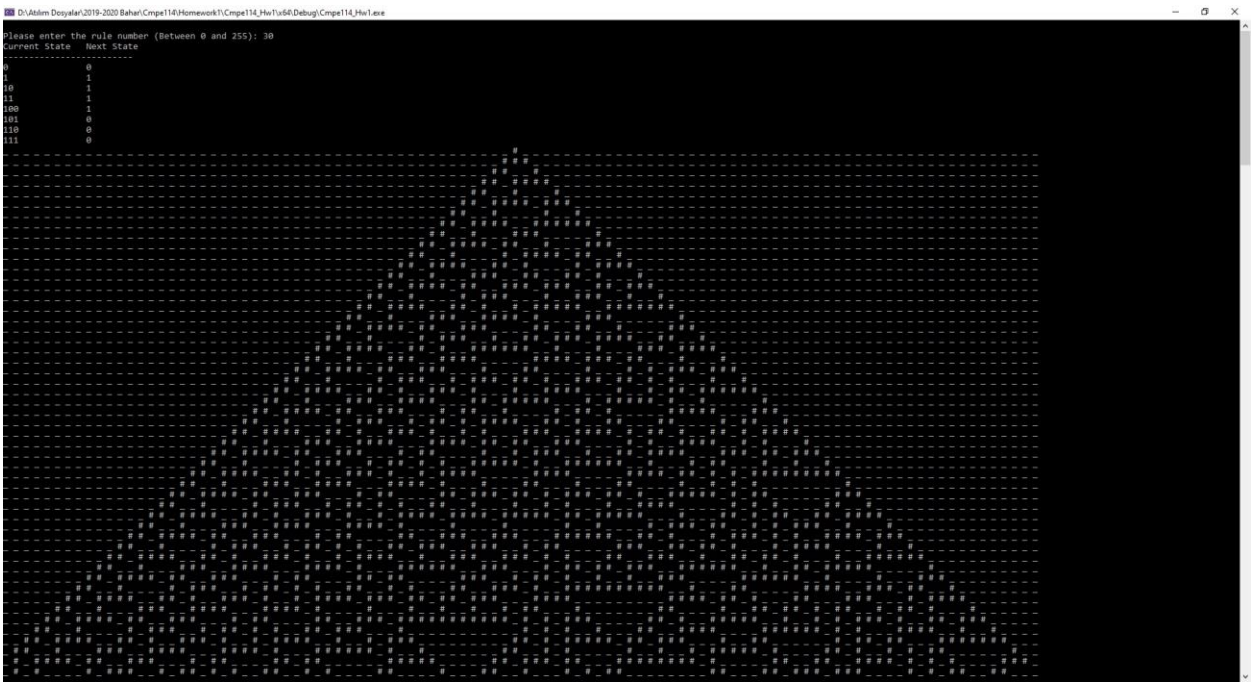


As you can see the cell's current states are read as (110) and its corresponding value from the table is taken. This value assigned to the same index in the next state array. After that, this array is printed on the console. Be cautious that we only updated one cell, let us continue with the other cells.



As you can see from the given figure, we are moving 3 neighbors by 3 neighbors at each time. Taking their values (like 001 and 101 etc.) and finding the corresponding state bits. After that in the next state array we are updating those values accordingly. After those steps you need to print

your next state array on to the console so we can see the pattern. The expected output can be seen below.



You can utilize any mechanics you like to get the next state information. I recommend you to use a 2D array as your lookup table. In that way you can get the states by searching through that array. I am providing a method that achieves the search from a 2D lookup table array, you may use it if you would like to.

```
void search_the_state(int look_up[][2], int *new_state, int state_indice)
{
	//state_indice is the parameter that is being searched in the look up table
	//Ex: 101 (you can use an integer in here no need to work with binaries) is being
	// searched on the lookup table and it corresponds to 5th index of the 2nd
	// column (which has a value of 1). The loop will search the
	//current neighborhood	from the
	//look up matrix and will return the
	//next state value from the pointer parameter new_state
	int lookup_iterator;
	for (lookup_iterator = 0; lookup_iterator < MAX_STATE_COUNT; lookup_iterator++)
	{
		if (state_indice == look_up[lookup_iterator][0])
		{
			//That means the state that consists of the current neighborhood
			//is found on the lookup table get the next state information for
		that cell
			*new_state = look_up[lookup_iterator][1];
			break;
		}
	}
}
```

Of course, you need to fill this array before any search operation. And you will fill the array with the provided values (Look at the rule30 lookup table). You can fill this in your main function as a static instantiation or you can take these values from the user. You can check all the 255 possible combinations from the [2] if you like. Try to write a generic code. Take the array size as 100 (for both current & next state arrays) and generation count as 50. So, your middle point should be 49 in this case. **In this project, you are expected to utilize the following functions and the main function. Be careful that those functions are expected in your code.** So, think your code around them. Mechanisms may change but try to use these signatures as much as possible. I'm providing those functions' signatures below:

```
void print_state(int state_to_print[ ], int state_length);
// Takes an array (state array) and prints it in a format that'll match the provided console output

void initialize_array(int array_to_initialize[], int state_length);
//Takes an array and initilialize it with the expected initial state.
// Remember the middle point in the current state.

void swap_arrays(int current_state[], int next_state[]);
//Takes two arrays as parameters and swap all of the elements in those arrays.

void calculate_next_state(int current_state[], int next_state[], int lookup_table[][2]);
//Takes the current state array and calculates the next state array based on the information
//provided in the lookup table 2D array. You may change this lookup table parameter in any way
//you like. But I have provided a method called search_the_state which will make your search
//easier. So, you may want to use it.
```

You can e-mail ozan.acar@atilim.edu.tr for your questions. **But please search it online before asking. Don't miss the deadline and don't cheat, similarity check will be applied.**

Good Luck.

References:

[1]: Leandro Nunes de Castro, Fundamentals of Natural Computing. Basic Concepts, Algorithms and Applications.

[2]: https://mathworld.wolfram.com/ElementaryCellularAutomaton.html