Game of Life

In this project you will implement a mathematical game known as “Game of Life” devised by John Conway in 1970. This zero-player game is set on a two dimensional array of cells (potentially infinite in dimensions). Each **cell** is in a **state**. Grid changes over a number of discrete time steps.

Change of cell state determined by its current state, states of its neighbors, and the set of rules.
Reference:

1. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
2. https://playgameoflife.com/
3. https://www.youtube.com/watch?v=ouipbDkwHWA&feature=youtu.be&disable_polymer=true

For better understanding play the game at least once.

In this game Cells are arranged in a two-dimensional Grid

1. Two possible states for each cell
   a) Live
   b) Dead
2. States can change
   a) Living cell can die (death).
   b) Dead cell can become alive (birth).
3. Simple set of rules specifying
   a) Death (overcrowding OR underpopulation).
   b) Birth (reproduction).

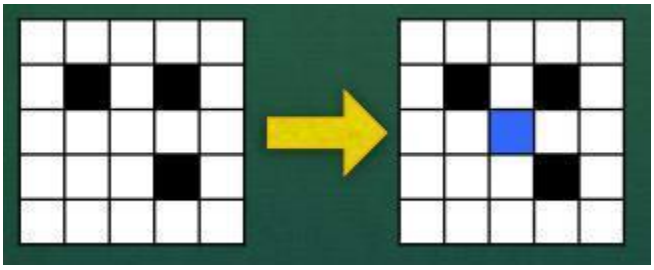
The game starts with an arbitrary pattern of cells set to live status.

The rules of the game are as follows:

If the cell is dead:

Rule: 1

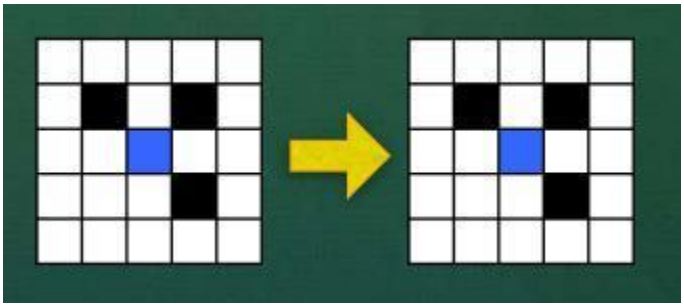
Birth: if exactly three of its neighbors are alive, the cell will become alive at the next step.



If the cell is already alive:

Rule: 2

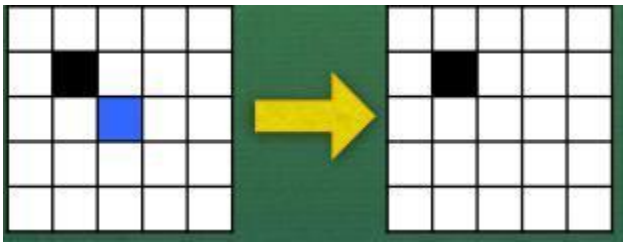
Survival: if the cell has two or three live neighbors, the cell remains alive.



Otherwise, the cell will die:

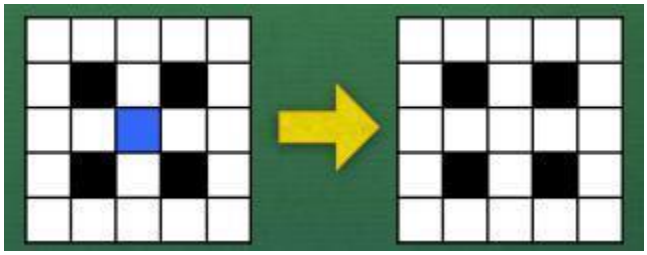
Rule: 3

Death by loneliness: if the cell has only zero or one alive neighbor, the cell will become dead at the next step.



Rule:4

Death by overcrowding: if the cell is alive and has more than three alive neighbors, the cell also dies.



These carefully chosen rules allow for very interesting evolution of cells starting with even simple patterns.

The life is staged on a **two-dimensional grid**, with each cell at unique **(x, y)** coordinates. The top left corner has coordinates **(0,0)** and **x** increases as one moves right and **y** increases as one moves down (typical choice for graphics devices). Simplest choice for representing the cells would be to use a two-dimensional integer array, where each cell contains a simple integer with value i-e

1. Cell alive: Value=1
2. Dead cell: Value=0

Neighborhood

For each cell, there are eight neighbors. If the cell has coordinates (x,y), then (x-1,y-1), (x, y-1), (x+1,y-1), (x-1,y), (x+1,y), (x-1,y+1), (x, y+1), (x+1,y+1) are the neighbors.

Neighborhood of (2,3) cell are:

(1,2) → dead cell

(1,3) → dead cell

(1,4) → dead cell

(2,2) → live cell

(2,4) → live cell

(3,2) → dead cell

(3,3) → dead cell

(3,4) → dead cell

0	0	0	0	0	0
0	0	0	0	0	0
0	0	1	1	1	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Secondary Array

To perform updates efficiently, an additional array (**secondary array (sec-array)**) can be used. The **sec -array** keeps track of only the **live cells**. An entry in the **sec -array** points to the corresponding entry in the grid. The figure 1 shows the relationship between the arrays.

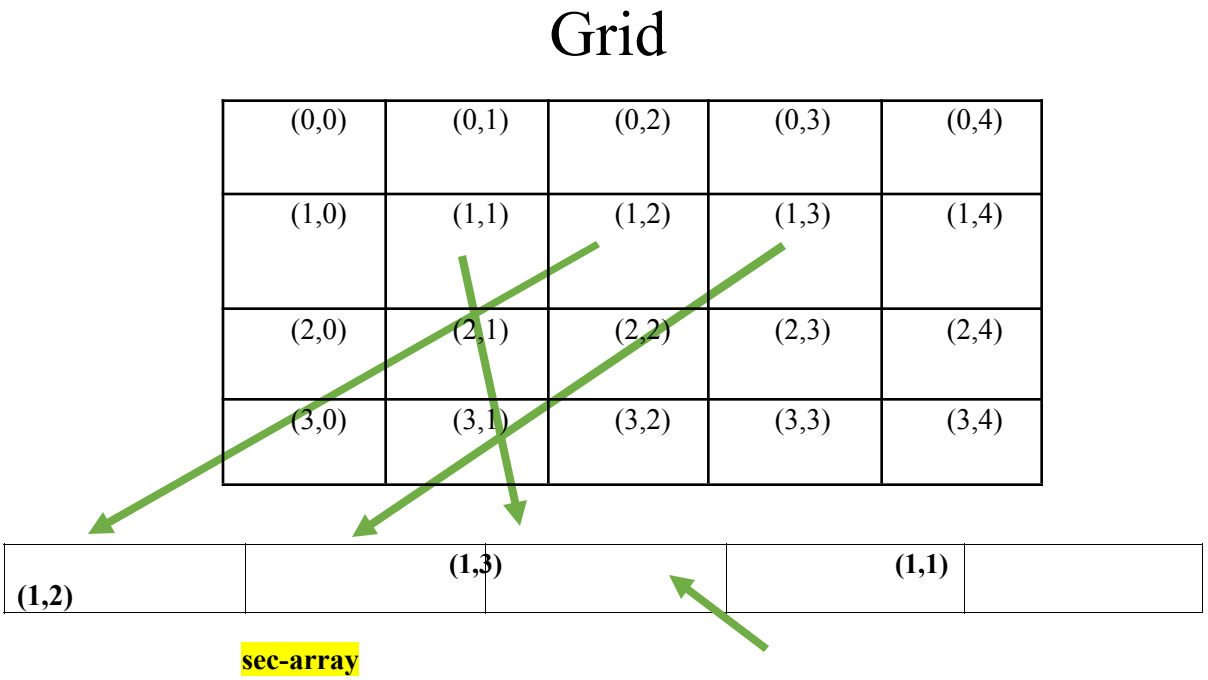


Figure 01 Last

Insertion:

A new element is always inserted at the end of sec- array and last element is maintained. Let's understand using example. Suppose sec – array size is 5 and currently there are 2 elements in array [(1,2), (1,3)]. Now cell of index (1,1) become alive so It will be inserted at 3rd index of sec – array. Last will be updated to 3.

Deletion:

Just deleting an element will not work. You have to swap all right sided elements to left (you can do this because the order of entries is not important). Suppose, cell (1,3) become dead so, now we have to remove this cell from sec – array. Deleting cell (1,3) from state shown in Figure 1 leads to the Figure 2.

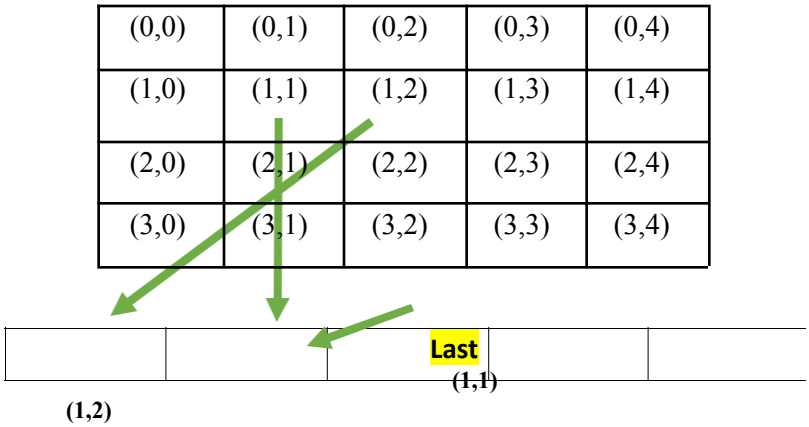


Figure 02

Details of the game algorithm:

Various phases involved in the update are as follows. These phases are primarily for descriptive purposes and you may choose to combine them for efficiency or convenience.

Adding neighbors:

In this phase, the number of neighbors for each cell will be counted. For each cell, there are eight neighbors (as mentioned in **Neighborhood** section). It could be easier to add the neighbors to a different array call it neighborhood array (nei - array), i.e. one array for live cells (sec - array) and one array for their neighbors (nei – array (dead cells)). Since the game rules for how alive cell propagate are different than rule for dead cells, this division of cells into different arrays can make things simpler.

Counting live neighbors:

Count the number of live cells adjacent to each cell (**sec – array** AND **nei - array**). This live neighbor count is what is needed to decide which cell will propagate to the next generation.

Cell propagation:

Once the neighbor counts are known, the status of cells is determined (i.e. is it going to die or alive). Those cells which turnout to be alive will be inserted and those cells which will die will be

deleted from sec - array in next generation and status (0 or false) will be updated to 1 or true in grid (remember, do not insert duplicate enters). **Do not use temporary array for insertion or deletion of sec - array**.

Birth: if exactly three of cells neighbors are alive, the cell will become alive at the next generation.

Display: The cells are drawn onto a console. This phase will be kept simple so that programming can focus on the main aspects of the game.

Additional details and advice

Your program should read input from a file. The input file contains how many generations of life you need to simulate as well as the coordinates of the initial pattern. We will provide some examples for you to work with, and you should try to test with more patterns of your own. The input file format is as follows. Only the numbers will be in the file. The comments following "←" below are only explanatory.

10 ← number of generations that you will need to simulate

6 ← number of cells in the object about to be read

19 9 ← (x=19, y=9) coordinates for the various cells of the object

10 10

11 10

12 10

19 10

19 11

Make your algorithm as simple as possible so that you have less of a chance to introduce bugs.

We suggest the following outline:

1. Implement read/write procedures. Read in your creature from a file and then print it out the grid. Please do not hardcode anything even do not hardcode name of input file.

2. Check that you are able to correctly add the cell neighbors. **Only the ones not already in the array (sec - array) will need to be added.** While adding the neighbors, it is easy to compute the counts too. Print out the sec-array to ensure this works correctly (this is only for debugging purpose).

3. Now apply the rules correctly and perform updates.

4. As far as printing/displaying is concerned, print Grid after each generation.

5. After Completing the simulation, write the live cell coordinates (sec - array) on to the file.

6. **Consider neighbors outside Grid as dead cell.**

7. Do not use temporary array (Grid, sec – array, nei – array) for **insertion** or **deletion**.

8. Grid, sec – array & nei – array should not be global arrays.

DELIVERABLE:

1. Implement complete project for Grid size 20x20, sec – array size 100 and nei – array size 300.

2. main function will only start the game int main()

```
{
        start();
        return 0;
}
```

3. Divide your code in functions (**your project will not be accepted without functions**): Functions are used because of following reasons

    a. To improve the readability of code.

    b. Improves the reusability of the code.

    c. Debugging of the code would be easier if you use functions, as errors are easy to be traced.

    d. Reduces the size of the code, duplicate set of statements are replaced by function calls.

    Your function should include things mentioned above.

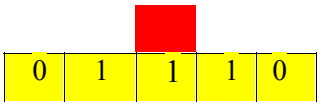
4. **Bonus:** Dynamic growth of

    a. Grid (if cells become alive outside the size of current grid then expand the grid's rows and column by (Row+2)x(Col+2). **And adjust Grid according to newly alive cells**. Suppose current size of grid is 4x5 then expanded size will be 6x7).

    Example:

    Current Grid

    (at Generation n)



0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

According to Rule 1: Red color cells (dead cells) will become alive in next generation. One of the cell is outside Grid so, you have to expand the Grid to size 6x7. Adjust the cells accordingly

Grid will transform like this:

Expended Grid
(at Generation n+1)

0	0	0	1	0	0	0
0	0	0	1	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Note that sec – array & nei – array will also be update.

b. Sec – array (to create space, double the size of array)

c. Nei – array (to create space, double the size of array)

Initially sizes 20x20 for gird, 20 for sec – array & 80 for nei – array.