Game of Life Description

Question 1: Program for the Game of Life

Let’s consider a square board, or more formally, a ***grid***. For example, Figure 1 shows a 4 x 4 grid (i.e., a grid with 4 rows and 4 columns). Imagine every square in a grid has life, and it can be either alive or dead. In Figure 1, we have colored the alive squares black, and those that are not black are dead squares. A square is said to be a neighbor of another square if the two squares share an edge or a corner. Note that except those on the boundary, every square has exactly eight neighbors. In Figure 1, next to the 4x4 grids, we have given a table showing, for each square, the number of living neighbors of that square. For example, for the top-left square (i.e., the square at row 1, column 1), it has one living neighbor (the one at row 2 and column 2), thus the top-left entry of the table has the integer 1.

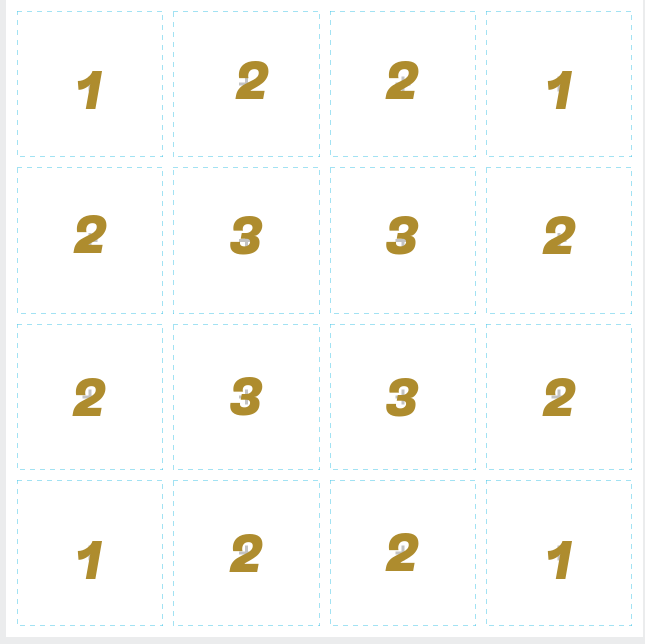


Figure 1. A 4x4 grids with 4 living squares at the center

In the game of life, the board is living generation by generation, and there are some specific rules to determine whether a square is dead or alive from one generation to the next generation: Suppose we are in the ith generation.

1. If a square is alive in this generation, and it has no alive neighbor, or it has only one alive neighbor, then it is dying of loneliness, and it will be dead in the i+1st generation.
2. If a square is alive and has four or more alive neighbors, then it is dying of overcrowding, and it will be dead in the i+1st generation.
3. An alive square with **either two or three** alive neighbors remains alive in the i+1st generation.
4. Consider any dead square in this generation. If it has **exactly three** alive neighbors, then it will become alive in the i+1st generation; otherwise, it will remain dead in the i+1st generation.

This question asks you to write a python program that shows the configuration of the squares after the game has run *n* steps from some input initial configuration. As an example, Figure 2(c) shows the configurations after two steps.



1. Initial configuration. (b) 2nd gen. (c)3rd gen. configuration

Figure 2. Two steps of game

***Important: input and Output***

The input of your program should be as follows:

The first line of input: an integer specifies the size of the grid.

The second line of input: *n,* the number of steps to be played

The next two lines of input: the first line is an integer specifies the row, and the next one the column, of the first alive square.

The positions of the remaining alive squares are given by similar pair of input lines that follow.

After the positions of all the alive squares are given, the input is terminated by the letter ‘t’. For example, the input for the initial configuration of Figure 2 with two steps to be played should be

5

2

2

3

3

3

4

3

t

Then, the output of the program should be as follows:

00000

00100

00100

00100

00000

In general, if the input is for a grid of size k, then your program should output k lines, the ith line for the configuration of the ith row after *n* steps, in which a ‘0’ represents a dead square, and ‘1’ represents an alive square.

**IMPORTANT NOTES:** Since handling grid with arbitrary size requires us to handle lists dynamically, and this is not easy for beginners. (As a matter of fact, we will have more discussion on list later in the course). Thus, you may assume that you only need to handle the case when the size is exactly 5 (i.e., all of our testing cases will only test input with grid size equal to 5). Then, you may store the grid as a list of lists. To be precise, you may first declare the grid as a variable cur\_conf as follows:

cur\_conf=[[0,0,0,0,0],

[0,0,0,0,0],

[0,0,0,0,0],

[0,0,0,0,0],

[0,0,0,0,0]]

Then, to make the top-left square of the grid alive, you can use the assignment: cur\_conf[0][0] = 1.

Another Hint: To determine whether a square will be alive or dead in the next generation, your program needs to check the number of its alive neighbors in the current generation. Since the number of neighbors of a square on the boundary is not 8, but can be 3 or 5, the checking can be quite complicated. One simple trick to simplify the coding is to declare cur\_conf as a 6x6 list, instead of 5x5, and assume all the squares on the boundary are extra squares and they are always dead. Then, the squares of your board are those between row 1 and 5, and column 1 and 5, and they all have 8 neighbors.