3 Homework Assignment – Cellulitis

NB. adhere to the *coding standard* as explained in the document on the website.

NB2. please use the *template file* and only write between the `\\TODO` **and** `\\END TODO` **markers**

A *cellular automaton* consists of a grid of cells, each containing a symbol from a finite set of symbols. The content of the whole grid, called a generation, changes in steps, according to some rules.

The automata we consider in this assignment satisfy the following requirements.

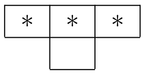
1. The rules use only the current generation to compute the next one.
2. The rules are the same for each cell.
3. The grid is a single row of cells.
4. The new value of a cell is based on the value of the direct neighbours (1 for a cell at the end of the row and 2 for the others) and of the cell itself; this set of cells is referred to as the *neighbourhood* of the cell.
5. a cell can have only two values. it is either *occupied* or *empty*

The assignment is to implement two automata. A third type, the universal automaton has to be implemented if you want to have a score (> 8.0). The description concerning the universal automaton can be found at the end of this text.

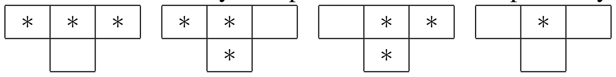
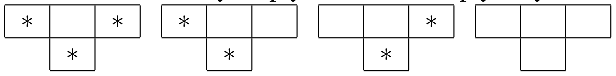
The number of cells in the row is specified in the input, as well as the initial configuration and which automaton has to be executed.

Your program should display the consecutive generations of the automaton, each generation is displayed as one line of output. An occupied cell is displayed as a star $\star$ and an empty cell as a space. The number of generations to be displayed is read from input.

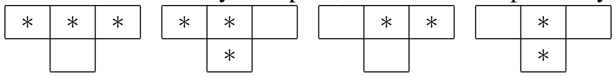
Your program has to implement two automata defined by the following two rule sets. The picture below shows what will happen to the middle cell given the configuration of the neighbours. So this picture tells that a cell that is occupied becomes empty when both neighbors are occupied.

current gen:
results in:


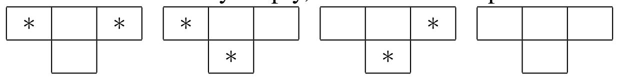
A

1. if the cell is currently occupied, it remains occupied only if exactly one neighbour is occupied;

2. it the cell is currently empty, it remains empty only if both neighbours are empty.


B

1. if the cell is currently occupied, it remains occupied only if the right neighbour is empty;


2. it the cell is currently empty, it becomes occupied if exactly one neighbour is occupied.



3.1 Implementation requirements

The row of cells should be implemented by an array of booleans called currentGeneration, where `true` represents an occupied cell and `false` represents an empty cell.

- `boolean[] currentGeneration;` a boolean array containing the current generation.

Define and use at least the following methods.

- `void readGeneral()` reads the general input (up to the initial configuration).
- `void readInitial()` reads the initial configuration (i.e. the first generation).
- `void readRules()` reads the rules (only implement in case of an universal automaton).
- `boolean newCellValueByA(int k)` returns the value of cell number `k` for the next time step, according to automaton A.
- `boolean newCellValueByB(int k)` returns the value of cell number `k` for the next time step, according to automaton B.
- `boolean newCellValueByRules(int k)` returns the value of cell number `k` for the next time step, according to the rules read from input (only implement in case of an universal automaton).
- `void draw()` draws the current state of the automaton, i.e., the values of `boolean[] currentGeneration` cells, on one line of output.
- `void computeNextGeneration()` computes the next generation and **updates** the `boolean[] currentGeneration`.

You are allowed, end encouraged, to introduce additional methods

3.2 Additional Implementation and development suggestions

It may be convenient to have on both ends of the array an extra element that is always `false`. This way, every row cell has two neighbours, which simplifies the calculations. Note that in the formulation of the rules above it is already assumed that the outer cells of the row have two neighbours, of which one is empty.

It may also be convenient to have another boolean array (e.g. `nextGeneration`) in which you compute the next generation. When you are done with the computation you can assign the new generation to the current generation: `currentGeneration = nextGeneration` and create a new boolean array for the new next generation.

Start with a simple automaton, e.g., one where an occupied cell remains occupied and an empty cell becomes occupied if one or both of the neighbour cells is occupied.

The initial configuration is a sequence of integers followed by the word `*init_end*` (see below). The number of integers is not specified. You may use here the expression `scanner.hasNextInt()`, that is `true` if the next word on input is an integer and `false` if the next word is not an integer or there is no next word (in the case of keyboard input, the latter will never hold).

3.3 Input

1. the letter A, B, or U, signifying the automaton that should be executed: A and B are defined above and U stands for the universal automaton that is defined below.
2. a positive integer L representing the length of the row of cells (not counting the extra border cells you may add in your implementation);
3. a positive number G representing the number of generations that should be displayed, including the initial generation;
4. the word `*init_start*` followed by one or more positive integers, followed by the word `*init_end*`; the integers designate the position of the cells in the row that are initially occupied; the first cell in the row has position 1. All cells not specified are initially empty. When an integer is higher than the number of cells L, it should be ignored.
5. in the case of the universal automaton: 8 numbers (0 or 1), representing the rule sequence (see below).

3.4 Output

A sequence of G lines (G defined above) representing the successive generations. The first line is the initial generation, as specified in the input.

Each line consists of L characters, each character represents one cell. An empty cell is displayed as a space, an occupied cell as the character `*`.

Newlines are not significant in the input. In other words, don't use `scanner.nextLine()`.

Example 1

Input

```
A 11 10
init_start 6 init_end
```

Output

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

Example 2

Input

```
B 61 20
init_start 20 40 init_end
```

Output

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

Universal automaton (for grades > 8.0)

Besides executing two specific automata, your program should be able to read a set of rules from input and then execute the corresponding automaton. This way, your program can execute any automaton satisfying the requirements above. Therefore, we say that it implements the *universal automaton*.

The rules that define an automaton are described in a compact format which is given below.

Remember that the next state of a cell depends only on the current state of the three cells in the neighbourhood cells: its left neighbour, the cell itself and its right neighbour. There are 8 such state combinations, called *patterns*. We give them in the following table. 1 means occupied, 0 means empty. For cells at the ends of the row, the missing neighbour is considered to have the value 0.

pattern 0: 000 (all three cells are empty)
pattern 1: 001 (right neighbour is occupied, other two cells are empty)
pattern 2: 010 (etc.)
pattern 3: 011
pattern 4: 100
pattern 5: 101
pattern 6: 110
pattern 7: 111

For each pattern, we give the value of the cell in the next generation. So we have 8 rules that completely define the behaviour of the automaton. What is needed on input are only these 8 values (0 or 1, where again 0 represents empty and 1 represents occupied), given in the order of the patterns of the table above. We call these 8 values the *rule sequence*.

So what the automaton should do for each cell is the following. If the neighbourhood pattern of a cell in the current generation is found at number n, give value number n of the rule sequence to the cell in the next generation. Numbering of the rule sequence starts at 0.

Example of a rule sequence

Suppose the rule sequence is 0 1 0 1 1 1 1 0. The automaton will then behave as follows. An empty cell with empty neighbours stays empty, since the neighbourhood 000 is the first entry in the table and 0 is the first value in the rule sequence. An empty cell with an empty left neighbour and an occupied right neighbour will change to occupied, since 001 is the second entry in the table and 1 is the second number in the rule sequence. Etc.

More concisely:

1. if a cell's both neighbours are empty, the cell will become (or stay) empty;
2. if a cell is occupied and so are both neighbours, the cell will become empty;
3. in all other cases, the cell will become (or stay) occupied.

Hints

To see which pattern applies for a cell, a sequence of 8 if-statements could be used. However, observe that the ordering of the patterns is as if the patterns are binary numbers, ordered from small to large. You can make use of this observation and get a much shorter piece of Java than with 8 ifs.

Example of the universal automaton

Input

```
  U 11 5
  init_start 5 init_end
  0 1 0 1 1 1 1 0
```

Output

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