CS102 – Algorithms and Programming II
Programming Assignment 5
Spring 2024

ATTENTION:

- Compress all of the Java program source files (.java) files into a single zip file.
- The name of the zip file should follow the below convention:
  CS102_Sec1_Asgn5_YourSurname_YourName.zip
- Replace the variables “Sec1”, “YourSurname” and “YourName” with your actual section, surname, and name.
- You may ask questions on Moodle and during your section’s lab.
- Upload the above zip file to Moodle by the deadline (if not, significant points will be taken off). You will get a chance to update and improve your solution by consulting with the TAs and tutors during your section’s lab.

GRADING WARNING:

- Please read the grading criteria provided on Moodle. The work must be done individually. Code sharing is strictly forbidden. We are using sophisticated tools to check the code similarities. The Honor Code specifies what you can and cannot do. Breaking the rules will result in disciplinary action.

Recursive Patterns

This assignment focuses on using recursion to create and display various character patterns. You cannot use any loops (for, while, foreach, etc.) or the Stream API; you have to simulate your loops using recursion. You cannot use any global variables other than the 2D array itself. You are free to design your recursive methods as you want; you may use any number of helper methods, variables, and classes. You should have the following methods:

- `createArray(int numRows, int numColumns)`: Initializes an empty 2D char array using the given sizes; you should keep a `char[][] array` class member and use it in your methods.

- `fillChar(char c, int rowStart, int rowEnd, int columnStart, int columnEnd)`: Fills the desired portion of the 2D character array with the given character. This method will start the recursion; you can call different helper methods within this method to recurse. For example, a separate method can handle filling specific rows, and you may call it repeatedly to fill the desired portion. Start indexes are inclusive, and end indexes are exclusive.

  Suppose we used `createArray(4, 5)` to produce the following empty 2D array:

Then, `fillChar('x', 2, 4, 1, 4)` should fill this array as follows:

	x	x	x	
	x	x	x	

Notice the start and end indices correspond to the index numbers of the cells. The method should operate on the class member `array`, so if we call `fillChar('y', 1, 3, 0, 5)`, the method should work on the `array` to produce the following result:

y	y	y	y	y
y	y	y	y	y
	x	x	x	

You can assume the input parameters of the `fillChar` method are always within the array bounds.

- `pattern1()`: Produces the first pattern on the existing `array`. This pattern fills the `array` with alternating rectangular boundaries filled with chars # and *. Make use of your `fillChar` method in your recursive patterns. For example, if we created the array using `createArray(10, 10)`, calling `pattern1()` fills the array as follows:

```
**********
*########*
*#******#*
*#*####*#*
*#*#**#*#*
*#*#**#*#*
*#*####*#*
*#******#*
*########*
**********
```

For example, `createArray(7, 19)` and `pattern1()` gives the following result:

```
******************
*################*
*#**************#*
*#*############*#*
*#**************#*
*################*
******************
```

Note that the dimensions of the array do not need to match, and based on the size, the boundary at the center may be a line. A different example for `createArray(4, 3)` and `pattern1()`:

```
***
*#*
*#*
***
```

- `pattern2(int fillWidth, int shiftAmount)`: Produces the second pattern on the existing `array`. This pattern fills the rows of the array with * chars of the given width; in each row, the * pattern is shifted by the given shift amount. Any cell that is not a * is filled with # in this pattern. You can start by filling the whole array with #. Then, fill each row with the desired width and shift the starting index for the next row. For example, `createArr(3, 13)` with `pattern2(5, 0)` generates the following pattern:

  ```
  *****########
  *****########
  *****########
  ```

  As the filled rows are not shifted, the * chars appear in the same column; `createArr(3, 13)` with `pattern2(5, 2)` generates the following pattern:

  ```
  *****########
  ##*****######
  ####*****####
  ```

  This time, each consecutive line is shifted by 2. Assume the rows are wrapped so that if we need to print more *, they appear at the beginning of the same row. For example, `createArr(7, 13)` with `pattern2(5, 4)` generates the following pattern:

  ```
  *****########
  ####*****####
  ########*****
  ****########*
  ##*****#####
  #######*****#
  ***########**
  ```

  A different example with `createArr(4, 19)` with `pattern2(9, 4)`:

  ```
  *********##########
  ###*********######
  ########*********##
  **##########*******
  ```

- `pattern3()`: Produces the third pattern on an empty `array`. This pattern fills the first row and first column with 1. Then, it fills the rest of the array such that for each element arr(x,y) = (arr(x-1,y) + arr(x,y-1)) % 10. In other words, each element is

equal to the sum of its up and left neighbors in Modulus 10 (the least significant digit). This ensures that each element can be represented with a char. Since the array we use is made of char variables, you either need to work on a separate integer array or convert the chars to integers using the method `Integer.parseInt(String s)`.

For example, `createArr(5, 5)` with `pattern3()` generates:

```
11111
12345
13605
14005
15550
```

A different example using `createArr(9, 35)` and `pattern3()` generates:

```
11111111111111111111111111111111111
12345678901234567890123456789012345
13605186556815063100136051865568150
14005640506450069000140056405064500
15550600551500065555600051550060555
16166222723888849494000056166622727
17840246358642043760000051284024130
18600262508480047400000056860026700
19555735008200041555555506400028555
```

The third pattern requires you to recurse following a specific path; for example, to calculate arr(5,5), you need to have arr(4,5) and arr(5,4) ready. You may check if an element is set or not by comparing it to the char '\u0000'. When an empty char array is created, each element is set to '\u0000' by default in Java.

- int findMaxRowSum(): Assuming `pattern3()` is called before, this method finds the highest row sum. You may implement an `int findRowSum(int row)` method to calculate the sum of values in a specific row recursively. Again, since the `array` includes char values, you must use the `parseInt` method. This time, you will calculate the integer sum without any Modulus operation. Then, based on these row sums, you need to return the highest sum.

  For example, calling `createArr(5, 5)` with `pattern3()` and `findMaxRowSum()` returns 16, which is achieved by the last row: `1+5+5+5+0=16` Note that you do not need to find the index of the row, returning the maximum sum is sufficient.

- `print()`: Prints the current array on the screen. This method should also use recursion and avoid using loops or the Stream API. You may add additional helper methods to print a row, which can be called recursively to print all the rows.

Preliminary Submission: You will submit an early version of your solution before the final submission. This version should at least include the following methods completed:

- `createArray`, `fillChar`, `pattern1`, `pattern2`, and `print`.

You will have time to complete your solution after you submit your preliminary solution. You can consult the TAs and tutors during the lab. Do not forget to make your final submission at the end.

Even if you finish the assignment in the preliminary submission, you should submit for the final submission on Moodle.

Not completing the preliminary submission on time results in a 50% reduction of this assignment's final grade.