CSIS215 Object oriented programming Project 1 Spring 19-20

Aim: The aim of this project is to strengthen the use of

- User-defined functions.
- Nested selections
- Simple and nested loops.
- 1D and 2D arrays.

Idea: Build a square matrix and test if it is a magic square.

Definition:

Magic square is a **square** grid (where n is the number of cells on each side) filled with distinct positive integers in the range. Such that each cell contains a different integer and the sum of the integers in each row, column and diagonal is equal.

For this project, we will implement two different algorithms that generate magic squares:

1. The first algorithm, implemented in the method **buildMagicSquare1,** requires 3 parameters:
   - a perfect square integer **n,** that represents the number of values in the sequence,
   - an integer **first**, first value of the arithmetic sequence used to generate the magic square,
   - an integer **step** that represents the difference between any two consecutive values in the sequence.

   The dimension of the generated two-dimensional array is (square root of **n**)× (square root of **n**) and its values are the values of the arithmetic sequence that starts with **first** and ends with **first + step * (n – 1)**.

   Example: if **n** is 16, **first** is 6 and **step** is 3, the method generates a 4×4 two-dimensional array that has the values: 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51 (6 + 3 * (16 – 1)).

2. The second one, implemented in the method **buildMagicSquare2**, requires an odd number **dim** as the dimension of the magic square. In this case, the magic square is a **dim×dim** two-dimensional array with values 1 to **dim×dim**.

   Example: if dim = 5, the magic square is 5×5 array with values 1 to 25.

To help you designing your solution you are asked to:

a) Write the boolean method **isPerfectSquare** that tests if its integer parameter **n** is a perfect square.

b) Write a method **createArithmeticSeq** that takes as parameters:

- an integer **n** representing the number of values in the arithmetic sequence,
- an integer **first** representing the first value in the sequence,
- an integer **step**, representing the step between the values of the sequence.

The method then creates and returns, a one-dimensional array of **n** elements ordered in an arithmetic sequence.

For example: if **n** = 16, **first** = 26 and **step** = 5, the arithmetic sequence is: 26 31 36 41 46 51 56 61 66 71 76 81 86 91 96 101.

<u>N.B. This method assumes that the size of its first parameter **n** is a perfect square</u>.

c) Write a method **display1DArray** that displays the values of a one-dimensional array of integers in a tabular format of a 2D shape. This method expects two parameters: an integer **valuesPerLine** (integers per line) and one-dimensional array of integers **arr** to be printed.

For example, for the array: 26 31 36 41 46 51 56 61 66 71 76 81 86 91 96 101 with **valuesPerLine** = 4, the method prints the following (7 places for each element, right justified)

```
26     31     36     41
46     51     56     61
66     71     76     81
86     91     96    101
```

d) Write a method **matricize** that takes a one-dimensional array of integers, as a parameter, and then puts its elements into a square array, whose rows and columns are each equal to the square root of the size of the one-dimensional array, and then returns it.

<u>This method assumes that the one-dimensional array's dimension is a perfect square</u>.

For example, if we pass, to the method, the one-dimensional array **arr** generated by part **b)**, then the function returns the following array:

26	31	36	41
46	51	56	61
66	71	76	81
86	91	96	101

e) Write a void method **reverseDiagonals** that reverses the main and secondary diagonals of a square array. For example, the result of calling this method, using the square array generated by the method in d), is as follows:

101	31	36	86
46	76	71	61
66	56	51	81
41	91	96	26

This method assumes that the two-dimensional array it is manipulating is a square one.

f) Write a method **sum1DArray** that returns the sum of the elements of a one-dimensional array of integers. This method is to be used to sum the elements of a row in a two-dimensional array.

g) Write a method **sumCol** that returns the sum of the elements of a column in a two-dimensional array.

h) Write a method **sum1stDiagonal** that returns the sum of the elements of the first diagonal of a two-dimensional array.

This method assumes that the two-dimensional array is square.

i) Write a method **sum2ndDiagonal** that returns the sum of the elements of the second diagonal of a two-dimensional array.

This method assumes that the two-dimensional array is square.

j) Write a method **buildMagicSquare1** that takes, as parameters, three integers **n, first and step**.

1. It calls the method **createArithmeticSeq** defined in **c)** to generate the arithmetic sequence in a one-dimensional array.
2. It calls the method **display1Darry**, defined in **c)**, to display the generated array as a 2D shape.
3. It calls the method **matricize** defined in part **d)**, to generate, from the one-dimensional array, the two-dimensional array.
4. It call the method **reverseDiagonals** described in part **e)** to reverses the diagonals. Just to mention that the resulting two-dimensional array is not necessarily a magic square. We will use the method **isMagicSquare,** defined in **l)**, for this purpose.

This method assumes that **first** is a perfect square.

k) Write a method **buildMagicSquare2** that builds and returns a magic square, of dimensions **n×n** (**n** is a parameter for this method), using the following algorithm:

Start by storing the first number, which is 1, at position (n/2, n-1). Let this position be (row,col). The next number (equal to previous number plus 1) is stored at position (row-1, col+1) where we can consider each row and column as circular array i.e. they wrap around.

Three conditions hold:

1. The position of next number is calculated by going one row up and moving one column to the right. At any time, if the next row's index is equal to -1, its index resets to n-1. Similarly, if the next column's index is n, its index resets to 0.
2. If the next cell to be filled (let us assume that it is at indices i and j) has already a value, you should then try next to fill a cell whose indices are [i+1, j-2].
3. If the row of the next cell you're trying to fill is -1 and its column is n, you should try next to fill the one at indices [0, n-2].

This method assumes that **n**, the number of rows and columns is odd.

To better understand the logic behind the above algorithm, try to trace it and generate a magic square of size 5×5 similar to the one shown below:

9	3	22	16	15
2	21	20	14	8
25	19	13	7	1
18	12	6	5	24
11	10	4	23	17

l) Write a boolean method **isMagicSquare** that should use the functions implemented previously for this purpose i.e. it should compute the sum of each row, each column, and the diagonals and do the check accordingly.

m) Write a function **printMatrix** that outputs the elements of a two-dimensional array, one row per line. This method uses the method **display1DAray** defined in **c)**.

n) Write a tester file with a main method that tests the methods you wrote for parts **a)** to **e)** using the following steps:

Prompt the user to choose of one the three options:

i. (Option 1) Generate a magic square using the method defined in **j)**:

   1. In this case, ask the user to enter the size of the one-dimensional array, the value to start with and the step for the arithmetic sequence. The validity of these values (i.e. size is a perfect square and the two others are positive) should be checked, separately, one by one. If any one of them is not, you should ask the user to re-enter a new one.
   2. Generate the corresponding two-dimensional array using the method **buildMagicSquare1** defined in **j)**.

3. Test if the generated two-dimensional array is a magic square, using the method **isMagicSquare** defined in **l)**, and display a message accordingly

ii. (Option 2) Generate a magic square using the second method (i.e. the one defined in **k)**:

 1. In this case, ask the user to enter the number of rows or columns of the square array. After making sure that the input is valid (i.e. >0 and odd), generate the corresponding magic square.
 2. Display the generated two-dimensional array.
 3. Test if the generated two-dimensional array is a magic square and display a message accordingly.

iii. (Option 3) Quit the program.

The above loop should continue until the user chooses to quit the program.