CSCI 2110 Assignment 3

Due date: 11:59 pm Halifax time, Friday, March 4, 2025 in Brightspace.

Problem 1

Goal

Do problem solving where you have options to make to get an answer and you are likely needing to try several different choices before finding an answer to any instance of the problem.

Background

In this assignment, you will be solving an edge-matching puzzle, sometimes called tetravex (see https://gamegix.com/tetravex/game for one example).

You are given an n x m rectangular grid (most often an n x n square) and a set of nm puzzle pieces. Each puzzle piece has an identifier for each edge. Figure 1 shows a sample puzzle space (the blue rectangle) and pieces (edge identifiers are numbers / colours on the edge).

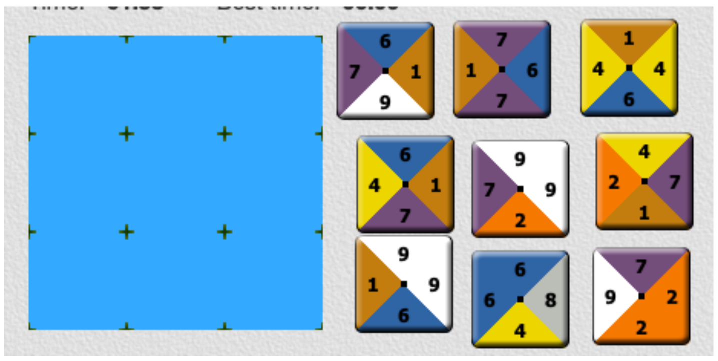
Figure 1 Sample tetravex puzzle from gamegix.com

The task is to put the puzzle pieces into the rectangular grid so that:

- Each piece is used exactly once
- Each edge meets up with its neighbouring pieces so that the edge identifiers match

In a general game, the puzzle pieces may need to be rotated to fit into the puzzle; simpler puzzles give you the puzzle pieces with a correct orientation for the solved puzzle.

Figure 2 shows a solution to the puzzle in Figure 1.

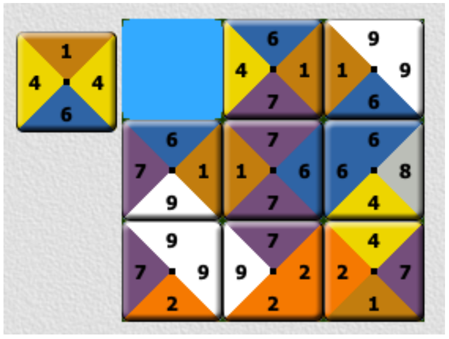
Figure 2 Solution to the tetravex puzzle from Figure 1.

Problem

Write a class called "Tetravex" that accepts a puzzle and ultimately solves the puzzle.

The class has at least 4 methods:

- boolean loadPuzzle(BufferedReader stream) – Read a puzzle in from the given stream of data. Further description on the puzzle input structure appears below. Return true if the puzzle is read. Return false if some other error happened.
- boolean solve() -- Do whatever you need to do to find a solution to the puzzle. The solution is stored within the class, ready to be retrieved. Return true if you solved the puzzle and false if you could not solve the puzzle with the given set of words.
- boolean placePiece(int pieceName, int xPosition, int yPosition) – put the piece named "pieceName" in column xPosition and row yPosition in the puzzle grid. The leftmost column is column 1 and the bottom row is row 1. Return true if the piece can be positioned at the location and has proper edge matchings with any other adjacent pieces already in the puzzle. Return false if you cannot put in the piece and respect the puzzle solution.
- String print( ) – print the current puzzle state to the returned string object.

You get to choose how you will represent the puzzle in your program and how you will proceed to solve the puzzle. You can also have any of these methods return an exception of your choice (that you document) in error conditions.

Inputs

The loadPuzzle( ) method will accept a description of the puzzle as an input stream. The input stream will have 2 parts to it.

- Part 1: Describes the puzzle square itself. On one line of input, you will get the width of the rectangular grid followed by the height of the rectangular grid. Both values are integers and the values are separated by at least one space.
- Part2: Describes the puzzle pieces, one piece per line. The line for a puzzle piece has 5 space-separated values: a "name" for the puzzle piece and then 4 integers representing the numbers along the edges of the puzzle piece, in counterclockwise order starting with the top edge. The "name" for the puzzle pieces is an integer that is unique among all puzzle pieces. We use the name to print the final puzzle. We use the edge integers to match with other puzzle pieces.

Example: The puzzle in Figure 1 would be represented as the following input. The "names" of the puzzle pieces are integers and need not be consecutive in order or be shown in sorted order.

```
3 3
10 6 7 9 1
11 7 1 7 6
17 1 4 6 4
12 6 4 7 1
38 9 7 2 9
23 4 2 1 7
25 9 1 6 9
22 6 6 4 8
```

42 7 9 2 2

Outputs

The print( ) method produces a String that can later be printed. The output contains 2 sections, with each section separated from the other by a line of 20 minus signs (-):

- The first section shows the placed puzzle pieces.
- The second section shows the list of unplaced pieces.

Remember to terminate each line in your string with a carriage return (\n) character.

First section: placed pieces
If the grid has n columns and m rows then this first section will have m rows and n tab-separated columns, which gives all n x m cells of the grid. For each grid cell, print the name of the puzzle piece in that grid cell.

The first section for the solution in Figure 2 looks like:

```
17\t12\t25\n10\t11\t22\n38\t42\t34\n
```

That, when printed, looks like

```
17	12	25
10	11	22
38	42	23
```

If no puzzle piece is placed in the grid cell then the puzzle piece name is "xxx".

Second section: unplaced puzzle pieces
List all the unplaced puzzle pieces, one per line of output, with the piece name and then the 4 edge integers (top edge first, counterclockwise order of edges), all tab-separated. The order of the puzzle pieces is arbitrary, but would be convenient if it matched the order described in the input.

If all puzzle pieces are placed then this second section contains no lines after the dividing line between section 1 and section 2.

Complete output
Printing Figure 1 before being solved would have the following output (when printed to the screen):

```
xxx	xxx	xxx
xxx	xxx	xxx
xxx	xxx	xxx
--------------------
10	6	7	9	1
```

```
11	7	1	7	6
17	1	4	6	4
12	6	4	7	1
38	9	7	2	9
23	4	2	1	7
25	9	1	6	9
22	6	6	4	8
42	7	9	2	2
```

While printing the completed puzzle of Figure 2 has the following output (when printed to the screen):

```
17	12	25
10	11	22
38	42	23
--------------------
```

Assumptions

You may assume that

- The puzzle piece names are positive integers.
- The number of puzzle pieces provided will not be more than the number needed to fill the grid.

Constraints

- You may use any data structures from the Java Collection Framework.
- You may not use an existing library that already solves this puzzle
- If in doubt for testing, I will be running your program on tiberlea.cs.dal.ca. Correct operation of your program shouldn't rely on any packages that aren't available on that system.
- Do not rotate the pieces to put them into the puzzle. It's an interesting twist to this problem, but it adds more complexity to your solution than is warranted by this particular assignment (for some people).

Notes

- Work incrementally.
  - First write the code to read in a puzzle. Test that.
  - Next, write the code to print a solution, whatever solution you have. Test that.
  - Then, write the code to place a piece into the puzzle. Test that.
  - Next, write your code to solve the puzzle in any way. Test that.
  - Last, implement some code to speed up your solution. That step is likely going to have you choose pieces or locations to add to your puzzle strategically. Picking that piece or space may need you to have some data structure(s) to store your pieces. Picking a location differently may need you to augment your data

structure that stores the puzzle board or do some searching for a location on your puzzle board.

Notice that the marking scheme has marks for designing and describing a solution strategy that are separate from marks for implementing a strategy. So, you need to come up with some strategy, but not implementing it doesn't throw away all the assignment.

- Develop a strategy on how you will solve the puzzle before you finalize and start coding your data structure(s) for the puzzle.
- Recall that you should first seek _a_ solution to solving the puzzle. That alone can be tricky in some instances. Even consider a brute-force version that tries all numbers in each cell.
- You may want to create another "print" method to print the actual edge values set into the grid as you add puzzle pieces. This other print method is likely to be useful when you are debugging your program.
- This assignment can be solved with or without recursion. You choose how to solve it.

Marking scheme

- Commenting, program organization, clarity, modularity, style – 4 marks
- Explanation of how you are doing your solution (strategy and algorithm) and what steps you have taken to provide some degree of efficiency. Include this information in an external PDF – 4 marks
- Ability to read in a puzzle and print out the puzzle that you have just read – 4 marks
- Ability to print a puzzle that is partly or fully solved – 2 marks
- Ability to place pieces individually in the puzzle and maintain the solution constraints…basically able to solve the puzzle by hand – 6 marks
- Ability to have your program solve the puzzle on its own in any way – 4 marks
- Correct implementation of some efficiency strategy that provides a meaningful speed-up to your program on big puzzles – 4 marks

Test cases

List of test cases to come.