# CSCI 3901 Assignment 4

Due date: 4pm Halifax time, Monday, March 21, 2022 in git.cs.dal.ca at
https://git.cs.dal.ca/courses/2022-winter/csci-3901/assignment4/xxxx.git
where xxxx is your CSID. This repository is being created for you.

## Goal

Work with exploring state space. My own solution uses recursion. Your solution does not need to use recursion, but it might help you.

## Background

Games are a domain in which the player searches through a set of possible local solutions to find an answer that satisfies all of the puzzle's constraints. This assignment has you create a solver for a game.

## Problem 1

### Background

Block puzzles exist in various forms. In short, you are given some outline of space that you must cover with a given set of puzzle pieces. You may have more than one copy of the same puzzle piece and you are allowed to rotate the puzzle pieces (but not flip them horizontally or vertically, in our problem). The puzzle pieces cannot overlap, and every space of the outline must be covered in the end.

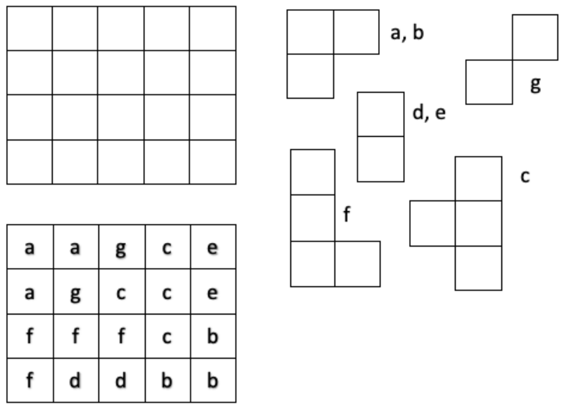

*Figure 1 Puzzle space (upper left) and puzzle pieces with letters to identify them (right). Final solution with the puzzle pieces covering the puzzle space (lower left).*

Sometimes, the given space is just some rectangle (Figure 1). Other times, it could be a shape with some spaces marked as not to be covered (Figure 2).

For simplicity, each of our puzzle pieces will be represented by single character.

## Problem

Write a class called “BlockPuzzle” that accepts a set of puzzle pieces and a puzzle grid (not necessarily square) and provides a solution to the puzzle where all the pieces have been placed in the puzzle space, if a solution is possible.

The class has at least 4 methods:

- addPuzzleRow to define an added row to the rectangular grid of the puzzle space, including spaces marked as not to be covered. The row appears at the bottom of whatever space we have built already.
- addPuzzlePiece to define a number of puzzle pieces of a given shape.
- solve to find a solution to the puzzle, if one exists.
- showPuzzle to return a string representation of the puzzle and it’s state of being solved.

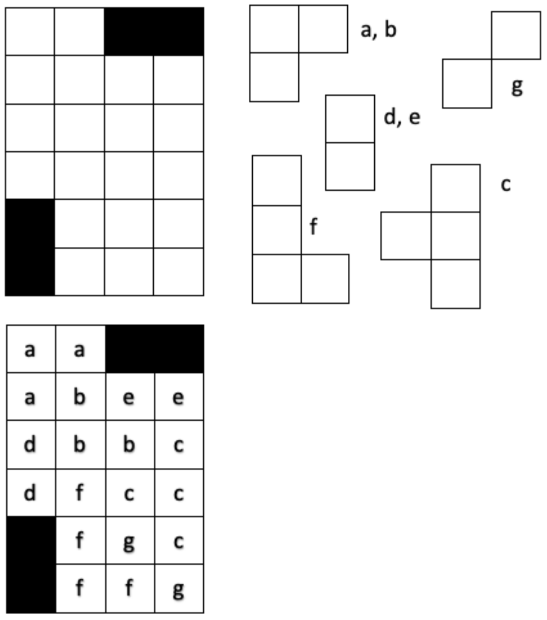

*Figure 2 Sample puzzle space (upper right) with the same puzzle pieces as in Figure 1 (right) and the solution (lower left).*

You get to choose how you will represent the puzzle in your program and how you will proceed to solve the puzzle. Solution strategies are likely to need some trial-and-error with the puzzle as you try out a piece to see if it fits or not with other pieces.

You can also have any of these methods return an exception of your choice (that you document) in error conditions.

### *Method definitions*

#### *void addPuzzleRow( String nextRow )*

Given a puzzle that you have created, add a row at the bottom of the puzzle that you have.

Adding a row should keep the puzzle as a rectangle, so the width of the new row should match the current width of the puzzle (unless it’s the first row, which is defining the width). The next row will be a sequence of characters, where each character represents one space of the puzzle. That character is a space character if the space is to be covered and is a non-space character if the space shouldn’t be covered in the final solution.

#### *void addPuzzlePiece (String piece, String markers)*

Add a puzzle piece to the puzzle.  The “piece” parameter describes the shape of the piece while the “markers” parameter identifies the characters to use to denote each placed piece in the puzzle and so, indirectly, the number of pieces of that shape to be used.  If this method is called with the same piece shape a second time then those are considered as additional pieces of the same shape.

The ”piece” parameter, is a character encoding of a rectangle in which the puzzle piece is placed.  The string has a number of rows for the rectangle, each separated by a carriage return (\n), including at the end of the string.  Each cell of a row is represented by a character; that character is a space if the cell isn’t part of the puzzle piece and is a non-space if the cell is part of the puzzle pieces.

For example, the “c” puzzle piece in Figure 1 would be the string “ *\n**\n *\n”.

Essentially, if you printed the string to the screen in a mon-width font then you would see the picture of the piece on the screen.  This example would produce

```
 *
**
 *
```

The rectangle containing the puzzle piece must be tight on all sides, meaning that there is no blank top or bottom row and no blank left or right column in the rectangle.

The “markers” string is a list of characters that will denote distinct pieces of this shape in the final solution.  The characters are a single character, not separated by one another by anything.  Consequently, the length of the string represents how many pieces of this shape will be in the final solution.

Piece markers must be unique across the puzzle, otherwise we won’t know where they appear in a final solution.

### *boolean solve()*

The solve() method looks for a solution where all the pieces are placed in the puzzle.  The method returns true if a solution was found and false if no solution was found.  If a solution is found then that solution is stored in the object and is ready to be retrieved with the showPuzzle() method.  If no solution is found then the puzzle in the object should be empty, meaning that showPuzzle() would return the initial unsolved puzzle.

### *String showPuzzle()*

The showPuzzle() method returns a string that, if printed, will show the puzzle state. The puzzle state is the rectangular puzzle with one character per cell in a row and with each row separated by a carriage return (\n). The character is either

- # if the cell is not supposed to be covered by a puzzle piece
- A space character if the cell is not covered by a puzzle piece
- A character designating the puzzle piece pattern that is covering that cell

The string ends with a carriage return.

For example, a solved puzzle from Figure 2 would return the following string

“aa##\nabdd\nebbc\nefcc\n#fgc\n#ffg\n”

When printed to the screen in a mono-width font, you then get

```
aa##
abdd
ebbc
efcc
#fgc
#ffg
```

Given this output, neither the # symbol nor the space are valid markers for puzzle pieces.

## *Sample problem*

The problem in Figure 2 would be created with the sequence:

```
addPuzzleRow( “  **” )
addPuzzleRow( “    “ )
addPuzzleRow( “    “ )
addPuzzleRow( “    “ )
addPuzzleRow( “*   “ )
addPuzzleRow( “*   “ )

addPuzzlePiece( “**\n* \n”, “ab”)
addPuzzlePiece( “*\n*\n”, “de”)
addPuzzlePiece( “ *\n**\n *\n”, “c”)
addPuzzlePiece( “* \n* \n**\n”, “f”)
addPuzzlePiece( “* \n *\n”, “g”)
```

## *Assumptions*

You may assume that

- Piece characters of different case (upper or lower) represent different pieces.

## *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.

## *Notes*

- Develop a strategy on how you will solve the puzzle before you finalize and start coding your data structure(s) for the puzzle.
- Work incrementally. The showPuzzle() method is likely to be your friend when debugging. Consider creating extra methods to print parts of the state, like how many puzzle pieces are used. You don't get marks for those extra methods, but they may prove useful when debugging.
- 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.
- If you don't add some degree of cleverness to solving the puzzle then your solve( ) method will run for a very long time on larger grids. There are some marks for adding this efficiency so that we can work with non-trivial grids.

## *Marking scheme*

The nature of this problem is one that you either have a solution or you don't. It's harder to get part-marks for a nearly-solved puzzle since you don't know if the partial solution will lead to an answer or not. So, focus on getting the main puzzle working well before optimizing.

Careful input validation will also be your friend in this assignment since I will have test data that seeks to break your code in different ways.

Above all else, ensure that showPuzzle is working. Notice that that method is the only way that I will be able to get information on the state of your running code.

- Documentation (internal and external), program organization, clarity, modularity, style – 4 marks
- List of test cases for the problem – 3 marks
- Clear explanation of how you are doing your solution (strategy and algorithm), why your strategy works, and what steps you have taken to provide some degree of efficiency (include in your external documentation) – 3 marks
- Solving the puzzle – 12 marks
- The effectiveness of your strategy to be efficient and to keep the running time of your program small (will be compared with timings from my solution on the same input data) – 3 marks