CSC171 — Project 2 — Puzzling Arrays

Due: Wednesday, November 10th by 1159PM EDT.

Objective

For this project, you are asked to write a program which generates puzzles and allows a user to interactively solve them. This will give you an opportunity to practice and demonstrate your understanding of arrays, control flow, encapsulation, and general object oriented design.

Program Requirements

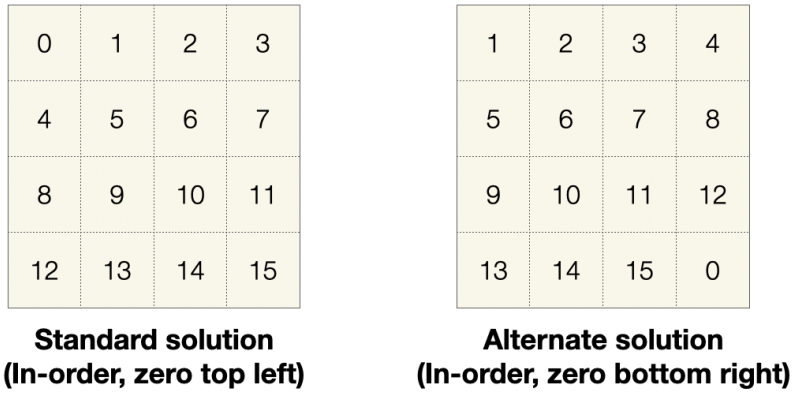
The specific kind of puzzle we will explore is known as a "Sliding Block Puzzle". Sliding block puzzles consist of an NxN grid of movable tiles (aka "blocks"), where each tile is labeled with a separate integer. For this project, you may assume that all puzzles are 4x4, for a total of 16 blocks. One special block is the "blank", which we will represent using zero. The blank can exchange places with its neighbors directly above, below, left, or right. If this were a physical puzzle, you would actually slide the tiles around. Since we are building a software model of a puzzle, *you can think of a move as swapping the zero with and adjacent integer.*

The objective of the puzzle is to find a sequence of moves such that all the non-blank tiles are in-order at the end, with the blank either at the top-left of the board, or at the bottom right. Refer to Figure 1a for examples of the board in each of its "solved" configurations. After that, take a look at Figure 1b and Figure 1c for an example of solving a puzzle through a sequence of moves.

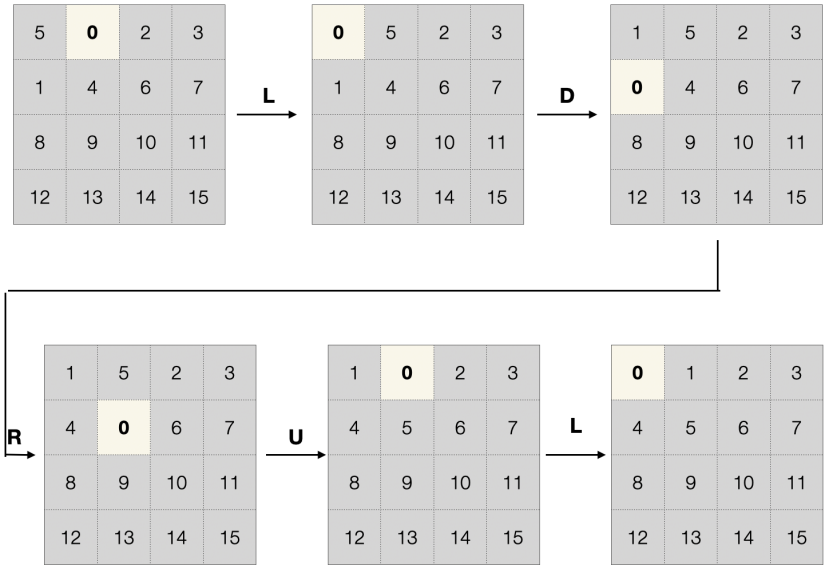
Your submission should include three public classes (and therefore, three Java files). The core functionality of your project should be in a class named *Puzzle*. This class should represent the state of a puzzle (likely as a 2d array) as well as provide methods to simulate a move, load a puzzle from input data, scramble a puzzle, and to check if a puzzle is solved or not. The other two classes should be named *Validate* and *Interact*, which should implement validation mode and interactive mode, respectively.

Originally I asked you to prompt the user for their preferred mode; however, this alterative representation should be easier for us to grade, which hopefully means we can get you feedback more quickly for project two than we have been able to do for the first project. You have total freedom in representing interactive mode – as long as it is clear how to "play" the puzzle (which should be in your readme.) For validation mode, you can print

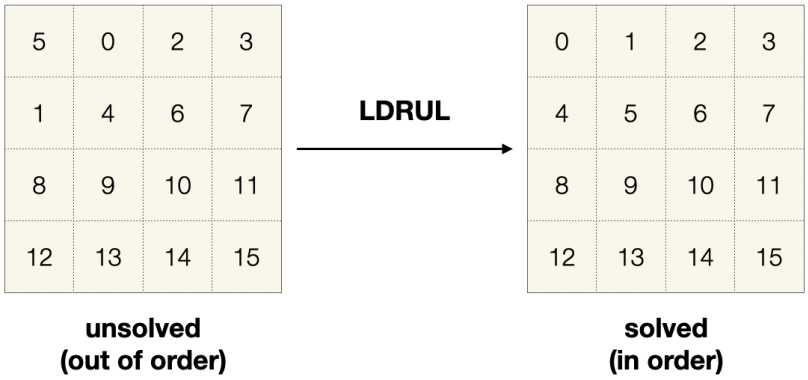
anything you like to the screen (i.e., a welcome message); however, the last line of output MUST contain the word "success" if the users input solves the puzzle, and otherwise it should say "failed". I am hoping to write a script which will test your puzzle on various inputs and simply look for the word "success" to determine how well it performed.



(a) Examples of puzzle goal configurations.



(b) Example of a puzzle solution, in steps.



(c) Example of the same puzzle solution, end to end.

Interactive Mode

For the first part, the user should be allowed to select a difficulty (easy, medium, or hard) and then your program should generate a puzzle of appropriate difficulty and then drop into "puzzle game mode", which will be a loop that prints the current puzzle configuration to the screen, prompts the user for a move, and then applies the move and repeats. If the user selects a move which is not valid, your program should warn the user but otherwise ignore the input. If the user manages to solve the puzzle, your program should print a congratulatory message and indicate the total number of steps that the user provided in order to solve the puzzle. (E.g., something like "Congratulations! You solved a medium difficulty puzzle in 20 moves!"

Some puzzle configurations are not solvable. In fact, any configuration which requires swapping ONLY two pieces cannot be solved while following the rules of the game.

A simple method to guarantee that randomly generated puzzles remain solvable is to begin with a solved puzzle and apply a random sequence of moves. More challenging puzzle configurations can be generated by applying longer sequences of random moves. Note that we are leaving unspecified what exactly is meant by "easy", "medium" and "hard" – try to solve a few yourself and select thresholds that make sense to you. Be sure to describe your choices in the readme.

Validation Mode

For the second part, the user should be prompted to enter a puzzle configuration on the commandline. A puzzle configuration will always consist of 16 integers given across four in the same order — left to right, top to bottom. The user should be expected to enter the puzzle configuration with each row of integers as a separate line. (E.g., each row of the puzzle should correspond to one line of input.) The integers should be separated by whitespace

We will use the validation mode to, well, validate the accuracy of your implementation, by presenting your program with a series of puzzles paired with both valid and invalid solutions. For this reason, it is important that all submissions use the same representations for moves. A move should be represented using the motion of the blank tile, which can either move Up, Down, Left, or Right. A move sequence will be represented as a String of letters - with no spaces - where each letter is either U, D, L, or R. E.g., the string "UURD" corresponds to the move sequence: up, up, right, down.

Collaboration

You may choose to complete this assignment either on your own (i.e., solo) or with a partner. Students working as a team will receive the same grade. Groups of more than two students are not allowed. If you work with a partner, you must ensure that names and emails for both students are listed as comments at the top of each and every Java file. Moreover, your README file must describe the contributions of each team member. If you work with a partner, only ONE person should submit the zip archive. (This last requirement has changed since the first project.)

Academic Honesty

The idea of sliding block puzzles is very old — dating back at least to 1880. There are several puzzle solvers online, and probably even a few Java implementations which are similar in scope to this project. *Searching online for solutions (e.g., via github) is STRICTLY FORBIDDEN and would be considered as a significant instance of academic dishonesty.* Any submission of plagiarized code (even with modifications) will be reported to the board of academic honesty and penalties will be applied.

All that being said, you are welcome to research the general concept of sliding block puzzles on the internet. Although you are not asked to write a program to SOLVE the puzzles, there are a variety of very interesting techniques to make that happen, and if you are so inclined then you are encouraged to take a look. Where we draw the line is looking for CODE solutions. Reading about the 15 puzzle online? Totally OK! Looking for Java (or other language) source code? Definitely NOT allowed!

Submission and Grading

All submissions must include at least one Java file and a README. You are free to create additional Java files if you wish (e.g., in case you prefer to create multiple classes.) The README must be a plain-text file. If you submit pdf, rtf, or docx files, we will deduct points. Your readme file should include your name and email, as well as a description of the overall structure of your code. If you worked as a team, you must clearly state this and describe how you divided up the work. Be sure to specify which class contains your main method. You should not have any package declarations in your code (if so, we will deduct points.)

All submissions will be graded according to the following approximate rubric:

10	Readme
10	Encapsulation
15	Main Control Flow
15	Puzzle Generation
25	Interactive Solving
25	Validation Mode

Revisions

1. Oct 15 - First version.
2. Oct 17 - Added explanation of "swapping" along with new figures and examples (included on blackboard).
3. Oct 25 - Added requirement to split program into three classes (separate validate and interact mode, with utility class puzzle.) Deadline extended by one week (from Nov 3 to Nov 10) to compensate for changing the requirements.