Project 2
Sudoku Solver

Due April 12th

Summary

Design a program that solves “Simple” Sudokus.

Background

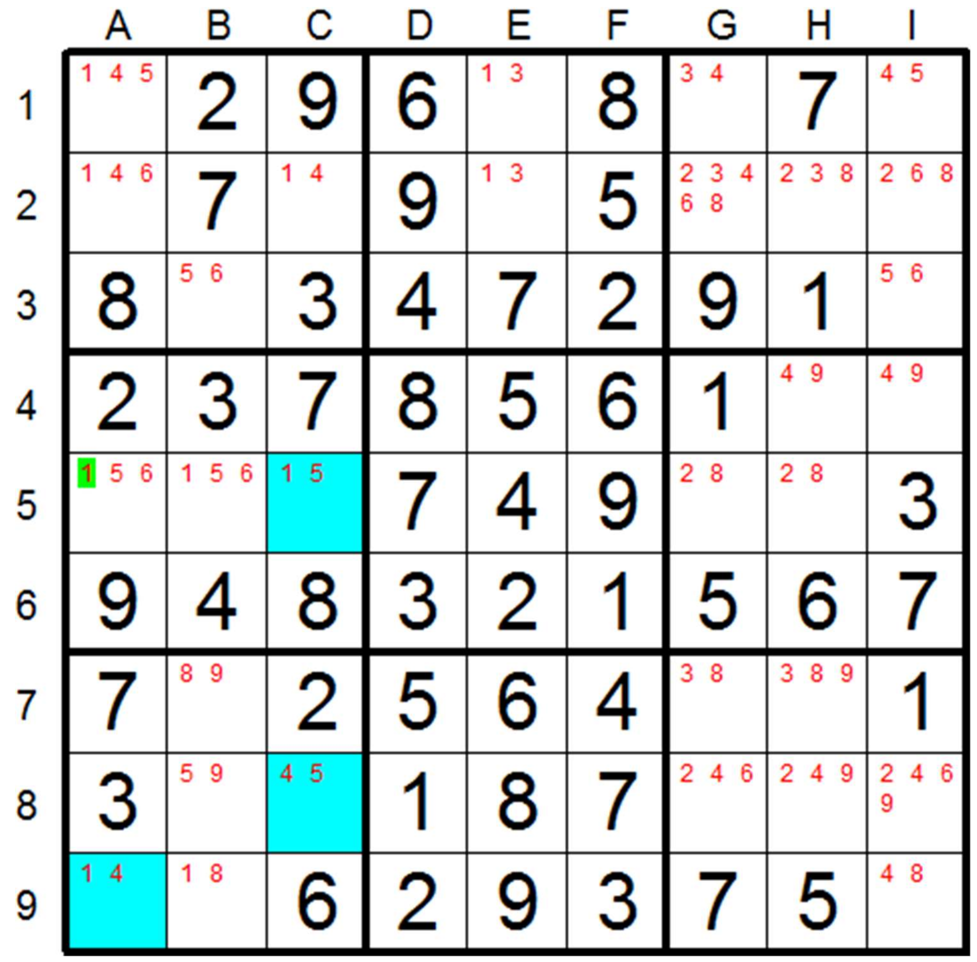
Sudoku

Sudoku is a math game that consists of a 9 by 9 grid, where each cell can have a value of 1 to 9. A correct solution to Sudoku should not have a duplicate/repeated value in any row, column, or a 3 by 3 grid.

General Strategy

Generally, the player will first write down all the candidate values for each cell based on the initial (given) values. Then, the player will assume the value of a cell that would in turn, affect the candidate values of other cells. This will cause a “chain reaction” until either the solution is found, or a dead end is reached. In case of a dead end, the player would know the assumption before was wrong, and take a different assumption. The process will be repeated until the puzzle is solved. It is quite possible that multiple assumptions need to be made to solve some of the more difficult Sudokus.

In the picture example, E1 and E2 can both be 1 or 3. Assuming E1 is 1, then E2 must be 3, and A1 can no longer be 1, and other numbers on the 2nd row cannot be 3. If this continues and eventually reach a dead end, then we know that E1 cannot be 1, which means it must be 3.



<u>The "Simple" Sudokus</u>

In most Sudokus, the player have to assume the value of some positions and back track in case it doesn't work out. Some, however, would never require any assumptions. In those Sudoku problems, you can always find one (or more) position(s) that only a unique value would work (also known as a "gimme"). As soon as the value a filled in, new cell(s) that have trivial solution(s) will emerge. By repeating this, the puzzle will eventually be solved, without ever needing to make any assumptions.

Writing the program

Data Structure

The very first component you should consider in this project is probably the grid. You will need that class to store the current state of the puzzle, maybe keep track of all the candidates, and definately functions to verify if rows, columns, and 3x3s are valid. New functions would become necessary as you continue on the project.

IO

The input and output will be CSV files. CSV files are comma separated values that you can view and edit with programs such as Excel, and easy to be read and edited as plain text. Each row is corresponding to a line in the file, and each column is separated by the comma character.

You will need a class that read/write CSV files and find a way to convert between the file and the Data Structure you have defined yourself.

In the project template, you will find 4 CSV files. The 2 easy puzzles can be solved by only finding trivial values. If your code is working, it should be able to solve both of the easy puzzles. The harder puzzle requires assumptions and can only be solved using a combination of Stack and Queue. *You only need to solve the easy ones to receive full credits. The harder ones are much more challenging and for extra credits.*

Execution Stack (Optional but recommended)

The execution stack itself does not need to be implemented, because it's built in by Java. However, your data structure must support recursive calls. This would mean the puzzle object can be passed to a different function to be modified further until a solution is found. Be very careful about the difference between passing the "object itself", or a "copy of the object".

Execution Queue (Optional for Extra Credits)

If you are feeling adventurous, you can attempt to solve the harder puzzle. To do this, you will need to evaluate the options within all the candidates and pick out the easiest ones first. Sorting is not required and seems overkill for this project. Since practically you only want to approach candidate cells with 2 or 3 possible values, you can simply use a few dedicated containers (such as HashSets or even Arrays) to store them.

Rubrics/Grading Guide

This project (as well as all the other projects) will be graded "max = 10 point", then scaled by multiplying 1.5 or 2.0 depending on your performance on the other projects.

(3 points) Grid/Puzzle Object
- Storing the 9x9 grid
- Having proper check methods
- Having sufficient manipulation methods

(3 points) Main logic
- Successfully constructing the puzzle from file
- Finding the trivial values (gimmes)
- Updating the grid accordingly

(2 points) Frontend file I/O
- Reading the CSV file (puzzle)
- Writing the CSV file (solution)

(1 point) General code structure / readability

(1 point) Documentation

(? points) Extra credits possible
Implementing an execution queue
Solving the hard puzzle

Extra Notes

It could be easier to first work out a “brute-force” solution before attempting to optimize your algorithm. Although the “brute-force” solution will be frowned upon, and unlikely to yield any extra credits, it should be a “safety net” to guarantee you most (if not all) of the base points.