INTRODUCTION TO COMPUTER SCIENCE I
PROJECT 7
Sudoku!

1 The game of Sudoku

Sudoku is a popular game giving crossword puzzles a run for their money in newspapers.[1] It's a game well suited for computers because it is a matter of finding legal placements for numbers on the board; it is an example of a *constraint satisfaction problem*.

The game: Sudoku is typically played on a 9×9 grid. Initially, a few of the cells of that grid contain a numbers, all integers between 1 and 9. The goal for the player is to fill in the remaining cells of the grid, also with integers between 1 and 9, given the following constraints:

- Each value (1 through 9) must appear exactly once in each row.
- Each value (1 through 9) must appear exactly once in each column.
- Each value (1 through 9) must appear exactly once in each 3×3 *sub-grid*, where the 9×9 board is divided into 9 such sub-grids.

A *valid* Sudoku game begins with initial numbers that, when combined with the constraints above, admit to **exactly one** complete solution. That is, the initial numbers cannot make it impossible to fill in the board legal, nor can they allow multiple solutions.

2 Your assignment

2.1 Copying files from my directory

You must copy some files from my directory to get started on this project. To do so, follow these steps:

1. Login to `remus/romulus` and open a shell.
2. Create a `project-7` directory and change into it.
3. Issue the following command from within your `project-7` directory:[2]

```
cp -r ~sfkaplan/public/COSC-111/project-7/* .
```

If you look at your directory (use the `ls -l` command), you will see that there is one Java class file (`Support.class`) and three Sudoku *board files* (e.g., `medium.board`).

[1] When students stop knowing what *newspapers* are, I will know that I'm old and should retire.

[2] Remember that this copying step should be performed **only once**. If you do it again later, you will clobber your own work with a fresh copy of my skeleton code. Don't do that.

2.2 The overall goal

You need to write a program, named `Sudoku` (written, of course, in a file named `Sudoku.java`), that can be run like this:

```
(remus)~/project-7> java Sudoku medium.board
Initial board:
0 0 0 0 0 0 0 0 7
0 0 0 0 7 0 3 0 8
0 0 0 5 0 4 0 6 0
0 0 0 0 0 0 0 8 0
7 1 0 0 9 0 0 0 5
8 0 0 0 1 5 9 0 0
3 0 0 0 0 0 0 0 0
0 0 8 9 4 0 6 3 0
0 2 7 6 0 3 0 0 0
Final board:
6 8 1 2 3 9 5 4 7
2 4 5 1 7 6 3 9 8
9 7 3 5 8 4 2 6 1
5 9 4 7 6 2 1 8 3
7 1 6 3 9 8 4 2 5
8 3 2 4 1 5 9 7 6
3 6 9 8 2 1 7 5 4
1 5 8 9 4 7 6 3 2
4 2 7 6 5 3 8 1 9
Correct solution!
```

That is, the file `medium.board` contains a Sudoku puzzle (with `0` values where there would normally be blank entries). Go ahead, open it in *Emacs* and look. Your `Sudoku` program must read that board into a two-dimensional array of `int`. It must then solve the puzzle. That's all there is to it! Well, there are a few more details that you should know about...

2.3 Importing needed classes

Since your program must use a number of special objects for opening and reading files (and catching exceptions that may be thrown by the methods in those objects), then you must tell the compiler about these object types—*classes*, using Java terminology. We have needed to do this kind of *importing* in past projects, but those statements were provided in the partial code that you obtained from me. Here, you are writing the `Sudoku` program from scratch, so you must declare these imports.

An `import` statement must come **before you declare the beginning of the class/program**, right at the very top of your `Sudoku.java` file. There are a number of importations needed in your program. One is for the `Scanner` class, and the others—for `File`, `FileNotFoundException`, and the like—are all part of a collection of classes used for *input/output* functions, or *I/O*. Rather than list each one-by-one, we can use a *wildcard*—a symbol that stands in for a number of possible

class names. Specifically, the use of the asterisk (`*`) is our way of telling the compiler to include a group of classes; in this case, the entire set of Java I/O classes. So, your `Sudoku.java` file should begin like this:

```
import java.util.Scanner;
import java.io.*;

public class Sudoku {
  ...
```

2.4 Using the `Support` methods

There are two methods in the `Support` class that you must use in your program. To use each, you call them from your `Sudoku` program by using the prefix `Support.` (including the dot!). That is, to call, for example, `printResults()` (documented below), then you would write the call as `Support.printResults(isSolved, board)`.

- `public static void printAndVerifyInitialBoard (int[][] board, String initialStatePath)`

  When your program has read the initial state of the board from its file into a two-dimensional array of `int`, it must call this method, passing **both** the 2D array **and** the file name from which it was created. This method will print the initial board; it will also check the 2D array against the file, verifying that the array is correct.

- `public static void printResults (boolean isSolved, int[][] board)`

  After your program has (tried) to solve the puzzle, it must call this method. The first parameter, `isSolved`, should indicate whether your solver believed that it solved the puzzle. The second parameter, `board`, should be the 2D array containing the solved puzzle. This method will print the 2D array that is passed. Moreover, if `isSolved` is `true`, then this method will verify that the puzzle is indeed solved. If `isSolved` is `false`, this method will determine whether a solution should have been found.

2.5 Suggestions for a progression

Do not try to write all of this code at one. Break it into pieces, program and test each piece, and then move onto the next one. This incremental approach will reduce your programming and your debugging time. Here is a suggestions for how to go about this problem:

1. **Read the board files:** Write a method capable of reading a board file and returning its contents as a 2D array of `int`. Test this method by calling it, and then passing its result to `Support.printAndVerifyInitialBoard()` to see if it worked correctly.

2. **Test legal placements:** Write a method that can determine whether the placement of a particular value in a particular position on the board is legal. That is, does placing a value v

in row r and column c violate the rules because v already exists in r, in c, or in the sub-grid to which (r, c) belongs? Test this method by reading in a board file and then placing values that are known to be legal and illegal in particular positions, calling your tester method on each to see if it evaluates the situation correctly.

3. **Write the solver:** With an initial board loaded and a method that can determine whether the placement of a value into a blank location on the board is valid, you can now write the actual solver method. Use `Support.printResults()` to determine whether your solver is doing its job correctly.

2.6 Testing your code

There are three Sudoku board files:

- `easy.board`—The easiest of the puzzles to solve. Start with this one.
- `medium.board`—Fewer initial values are provided than in `easy.board`, making the search a bit harder.
- `evil.board`—A minimal number of values are provided. This solution requires backtracking to work correctly. (That is, certain values will be placed, but their incorrectness will not be evident until much later in the search.)

So long as you call the `Support` methods at the right times, you should be able simply to use these three provided puzzles to test and debug your code. Feel free, also, to find other puzzles and make your own Sudoku board files.

2.7 Extra challenges

The following are possible extensions to this assignment that you may pursue for additional credit. To be clear, the effort for one or both of these challenges substantially exceeds the credit that each would earn you. Although you may earn some small additional credit for (nearly) completing any part of these, you should take them on because you find the challenge compelling. You should also know that there will be little guidance on how to solve these problems. To devise a solution, plan on spending some significant time and effort in puzzling it out.

Creating a puzzle generator: The core of this assignment is creating a program that solves an existing Sudoku puzzle. But how do those puzzles get created? We want an algorithm, and then code that implements that algorithm, to create a new Sudoku puzzle. Your task is simply to create such an algorithm and then to program it.

If you manage to develop such a puzzle generator, you can then take on the additional challenge of making it *optimal*: generate puzzles that contain the *smallest number of initially provided values possible*.

Variables sizes and dimensions: Just as we have used the 4×4 "kids" Sudoku puzzle during lectures in addition to the 9×9 standard puzzle, the game is easily defined for any $k \times k$ configuration for all integer $k > 0$.[3] So, as a first step, **generalize your code to read and solve $k \times k$ Sudoku puzzles** (again, for integer $k > 0$).

Having made this change, now consider that Sudoku need not be only a 2-dimensional game. Consider, for starters, a 3-dimensional Sudoku. Here, there are not just horizontal rows and vertical columns, but also depth-based *tunnels*.[4] Whereas the standard puzzle is a $3^2 \times 3^2$ arrangement, we could consider, say, a $3^3 \times 3^3 \times 3^3 = 27 \times 27 \times 27$ puzzle, one would need to place the digits 1 through 27 such that each digit appears exactly once in each:

1. *row*
2. *column*
3. *tunnel* (third dimensional line from front to back)
4. *subcube* (each cube created by dividing the 3-dimensional cube into 81 separate $3 \times 3 \times 3$ cubes.

3 How to submit your work

Use the `cs111-submit` command:

```
$ cs111-submit project-7 Sudoku.java
```

If you complete any portion of the extra credit projects, submit them something like this:

```
$ cs111-submit project-7-extra SudokuMaker.java SudokuMultiSize.java
```

This assignment is due on Wednesday, May 7, at 5:00 pm

[3] There are also configurations of the puzzle for non-squares (e.g., 3×4), but we can skip those arrangements here.
[4] Perhaps not the best choice of name for this third dimension, but it will do.