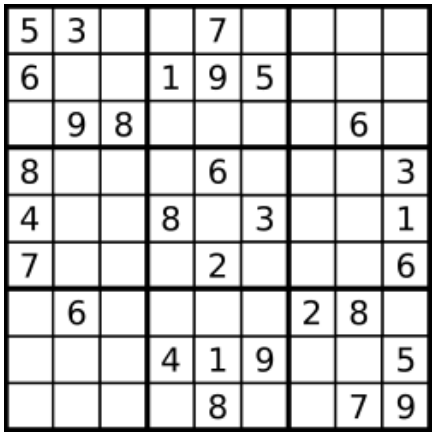
# CS 416 Program 4P – Sudoku

June 24, 2016

**Important Dates and Requirements**

- The program is due Friday, June 24 at 23:59 p.m. Late: -3 (Sat), -7 (Sun), -12 (Mon)
- The last submission day is Monday, June 27 at 23:59 p.m.



**Goals**: Use recursion in game environment to support *backtracking*.

**Scenario**

*Sudoku* is a simple number puzzle that uses a 9x9 array of numbers as shown on the right. A correct solution to the puzzle requires the player to fill in all the empty squares with digits from 1-9 such that every row contains all digits from 1-9, every column has the digits 1-9 and the nine 3x3 subblocks each have the digits 1-9. Your task is to write a program that will solve incomplete Sudoku puzzles that will be read from files.

**High level approach**

You should use a *brute force* approach, which means that you won't try to make your program smart, just persistent! In other words it will simply try every possible digit in each empty square until it finds one that does not violate the *row*, *column*, and *subblock* rules. As soon as it finds one that works, it goes to another empty square and tries to put a digit there and continues in this way until the boxes are all filled ***or*** it finds an empty box that cannot be filled with any digit without violating one or more of the rules. When that happens, your program must *backup* to a previous square and "undo" the last digit it placed there and try another. This process continues until the puzzle is solved or the program claims that there is no solution.

**Algorithm**

In order to make this work, your overall approach has to be transformed into a rigorous algorithm that organizes the *trial-and-error* process: you need to "know" where you are at all times and what digits you have and have not tried in any given square. It's pretty straightforward to just traverse the squares of the puzzle by row (or by column) looking for an empty square. When you find one, you try all the digits from 1 to 9 (this can be inside a *for* or *while* loop). If you find a digit that can go there, you put it into the square, assume that it is the correct digit and *recurse* to look at the "next" blank square. You'll surely reach a point where the choices you've made so far are not correct and all numbers will be incorrect for a given empty square. So, you return failure to the caller of this invocation. The caller will then try the next number in its sequence, if there is one, which will cause another recursion. If the caller just finished its loop (it tried a 9 that failed), it will return failure to *its* caller, and so forth. Eventually, (if the board is a valid board) a correct number can be placed on the last available blank and that invocation sees that there are no more blanks left and returns success, which ripples all the way back up the recursion stack.

**Program**

Your program should read the data file and display the puzzle defined by the file **and** print it to *System.out* in a form that is a readable representation of the 2D array. The GUI provides a **Solve** button; when pressed, your code should respond by filling in a valid solution, updating the display to show that solution, and printing the solution to *System.out*, **or** reporting that there is no solution.

The starter code opens an initial test data set from the command line and has 2 buttons and slider. The *Solve* button should call a method that re-initializes the current puzzle to its starting state and then solves it. The initial numbers (or the cells they are in) should be displayed with a different color than the numbers that your program inserts into blank spaces.

Your program should update the display after every change to the puzzle, both the change of a number to another number or to a space (on backup). This is actually a non-trivial issue since these changes occur during the execution of your solution code during which there is no easy way to interact with the user. The *GUI* starter code includes an *updateDisplay* method that helps. You can invoke this method with the *GUI* object as the first parameter (*this*) and a wait time (*_pauseDelay*), which is set by the slider. It forces a display update (via the *paintImmediately* method call) and then waits for *_pauseDelay* milliseconds. There is also code in that method to **not** do the updates based on user choices as describe below.

**Code issues**

1. Don't write a "smart" program that mimics what a human would do. Be a dumb (but fast) computer that solves systematically.
2. You want to have a clean way of keeping track of where you are and how to get to the *next* cell in the traversal of the array; you want that to be a method of some class. Given a current location of (r,c), the "next" cell is usually just (r,c+1) – unless c+1 gets to be 9 (one past the index of the last column), in which case it is (r+1,0) – unless r becomes

9, in which case you've gotten to the end of you traversal. This should only happen if you got there with a successful placement of all your digits and you looking for another space – that means you have solved the puzzle and can return success.

3. Building the puzzle display isn't very hard. I used *JButton*s for each square and put them into a *JPanel* using a *GridLayout* with a 9x9 grid. Use the Java API docs and online tutorials to get the buttons to look reasonably nice.

**Testing strategy**

It is not easy to test algorithms like this one that require trial and error and then *backup* and try again. It can be helpful to watch the effects of your computation by updating the display every time you change a value. For early testing with simpler puzzles and a long delay time, you can get a sense of whether your code is behaving as you expect. Unfortunately, you cannot do any other interaction while your program is executing, which means, in particular, that you cannot slow down or speed up the rendering of your computation except before it starts. To help you test you should do the following:

1. Be sure that your code that changes values in the puzzle calls the *GUI.updateDisplay* method every time you change a value. You'll then have access to the interactive visualization of the execution results of your program.
2. Use simple test data sets first: *test0-nobackup.txt* has been carefully chosen so that no backup is needed to get a solution. In my solution *test1.txt* requires 7 backups if done by row and 12 if done by column. *test5.txt* requires over 5300 backups.
3. Allow the same puzzle to be solved multiple times. This means that you must be careful to "clean up" your previous solving results before doing it again. You must restore your data to its state before you started putting your numbers in.
4. Use long delays for the simpler data sets, shorter as they get bigger.
5. You can turn the updates off completely with a toggle button or you can pick the "Show a few" toggel, which will show the puzzle display after every 1000 updates.

**Required output**

We will be grading the correctness of your program mainly by running it in batch mode and automatically comparing your output with ours. Therefore, you must print some specific information to *System.out* **in the order listed below with the specified constraints:**

1. The name of the file containing the initial puzzle. It must be left justified with no other characters on the line.
2. The final state of the puzzle board. Each row in your puzzle solution should be on 1 line. There should be 1 space in front of each number. This will require exactly 9 lines. The *GUI.array2DtoString* method prints a 2D array in this format; this is there only as a model for the output format. You do not need to store your puzzle as a 2D array of ints (I didn't), but you can edit this method to access any 2D array or don't use it at all.
3. If you found no solution, do not print the puzzle board. Instead print just 1 line that says "No solution" left justified.
4. The next line should print the total number of recursions your code made to solve the puzzle and the total number of *backup*s you did. There should be exactly 2 numbers on this line in the specified order with a space between them and nothing else. These should be printed even if there was no solution.
5. **No other output should be printed to *System.out*.** Any debugging code or other output you would like to do should be sent to *System.err*.

**Notes**

1. Starter code is in ~cs416/public/4P: *Sudoku.java* and *GUI.java* along with some data sets. You should not change *Sudoku.java*. *GUI.java* implements all you need for interaction, but its buttons and slider need to be connected up with *responder* code.
2. The starter code uses the Java *JFileChooser* class to facilitate your ability to load and test different puzzles. This has been wrapped in a method, *Utilities.getFileName()* that is part of the *Utilities* class that is in *cs416.jar*. **If you get compile errors from this method, make sure you replace your cs416.jar with the version in ~cs416/public/ .**
3. Beware of using "x" and "y" notation vs. "row" and "column". The convention is to order these as *x,y* and *row,col*. However, the *row* coordinate is a *y* and the *col* is an *x*. I find it most clear to use *row, col*. The worst thing to do is use them both in different places!
4. Make sure your code is well-modularized, with no long methods.
5. Pay attention to the style conventions; we'll run *checkstyle* on it.

**Approximate point allocation**

60 Puzzle solution
20 Visualization of the solution; clean display; proper integration of the buttons, multiple runs, etc.
20 Correct, properly formatted textual output including statistical information