You may be familiar with the number placement puzzle Sudoku, which works as follows: a 9 x 9 grid is partially populated with digits from 1 to 9. I’ll refer to this grid as the “clue.” The purpose of the game is to fill in the remaining grid positions with digits from 1 to 9 subject to three constraints:

1. Each digit appears only once in each row
2. Each digit appears only once in each column
3. Each digit appears only once in each “region.”

Regions are a subdivision of the board into 3 x 3 sub-areas. Here is an example puzzle with a solution:

The “regions” are indicated by the different shadings. Constructing puzzles is tricky; it is very easy to construct a puzzle that has no solution. Humans have developed various strategies to solve these puzzles, and working them out by hand can be quite satisfying. However, what we want to do is develop a program that will take the clue and compute a solution for us. From the Eight Queens example given in class, you may recognize this as a situation where backtracking can help us out. In fact, this puzzle is somewhat similar to Eight Queens except that there are multiple symbols for each slot rather than just one, and the constraints are a bit more complicated. We can simply start at the upper left corner, figure out what the set of possible value for that cell are (3 and 8 in this case), pick one for that cell, then move on to the next empty cell and repeat the process. If we hit a cell
where there are no candidate values before we finish the puzzle, we simply back up to a
cell where we have other possibilities to try, pick a new candidate, and continue.

This is a brute-force, exhaustive search method for finding a solution. More elegant
algorithms have been discovered. For one example, try Googling “Donald Knuth”
“dancing links” Sudoku. For our purposes, the brute force approach works just fine on a 9
x 9 grid.

I have written a skeleton program for a solver that you can download from the class
assignments web page. Basically, your job is to fill in the solve(row, column) method.
You may find the print( ) method useful for debugging purposes. It prints a character-
based representation of the current state of the solution on standard output. I have also
included some debugging code that stops the program upon every call to solve( ), prints
the solution so far, and waits for keyboard input. Set the DEBUG constant to true to turn
this code on.

There are also three puzzle files on the class assignment page, puzzle1.txt, puzzle2.txt
and badpuzzle.txt. The first two have solutions, the last one does not. It doesn’t really
matter where you put these, the application will let you browse for the files.

When you are done, email me your SudokuSolver.java file. Any code you add should
contain adequate comments so that I can easily see what you intended. You may turn this
in any time before our next class meeting.