# Recursion – Sudoku

We are going to write a solver for sudoku puzzles, a puzzle you likely are familiar with. A sudoku puzzle contains a grid of 3 by 3 boxes, each box of which contains 3 by 3 squares. The puzzle thus comprises 9 by 9 squares in total. Initially some squares are filled with a number, and the challenge of the puzzle is to fill all squares with the numbers 1 to 9 such that: each row, column and box contain each number 1 to 9 exactly once.

## Terminology

A *grid* is a matrix of 9 by 9 squares, containing numbers between 1 and 9 or being empty. A *full grid* is a *grid* without empty squares.

A set of squares is *conflict free* if every number between 1 and 9 occurs at most once in the set. A grid is *conflict free* if every row, column, and box is conflict free.

A grid $h$ is an *extension* of grid $g$ if each non-empty square in $g$ contains the same number in $h$.

A *solution* of a grid $g$ is a full grid that is conflict free and an extension of $g$.

A *puzzle* is a grid that is conflict free and has, usually, exactly one solution.

## Approach

We build our solution step by step. It is not required to stick to these steps, but it may help.

1. Write a class Sudoku, containing an instance variable grid of type `int[][]`. This represents the puzzle and all intermediate steps. Fill the grid with an initial puzzle. Use the value 0 to represent an empty square. An easy way to fill the grid in Java is as follows:

   ```
   grid = new int[][] {
       { 0, 6, 0,  0, 0, 1,  0, 9, 4 },
       { 3, 0, 0,  0, 0, 7,  1, 0, 0 },
       { 0, 0, 0,  0, 9, 0,  0, 0, 0 },
       { 7, 0, 6,  5, 0, 0,  2, 0, 9 },
       { 0, 3, 0,  0, 2, 0,  0, 6, 0 },
       { 9, 0, 2,  0, 0, 6,  3, 0, 1 },
       { 0, 0, 0,  0, 5, 0,  0, 0, 0 },
       { 0, 0, 7,  3, 0, 0,  0, 0, 2 },
       { 4, 1, 0,  7, 0, 0,  0, 8, 0 },
   };
   ```

   Here, we used a puzzle of medium difficulty (for humans) found on the web. Write a method that prints the grid, e.g., as follows.

   ```
   +-----------------+
   |  6  |    1|  9 4|
   |3    |    7|1    |
   |     |  9  |     |
   -------------------
   |7   6|5    |2   9|
   |  3  |  2  |  6  |
   |9   2|    6|3   1|
   -------------------
   ```

```
|     |  5  |     |
|    7|3    |    2|
|4 1  |7    |  8  |
+-----------------+
```

Test it.

2. In the process of solving we will look for an empty square, try a number and see if it fits, or in other words, see if it has no conflict, where conflict means: the same number occurs somewhere else in the same row, column, or box. Write a method `boolean givesConflict(int r, int c, int n)` that determines whether filling in the number $n$ in the square with position $r$ (for row) en $c$ (for column) will give a *conflict*. It is probably a good idea to separate further into three methods rowConflict, colConflict, and boxConflict. Test these methods.

3. Add a method `int[] findEmptySquare()` that tries to find the next empty square in reading order[1]. If it succeeds, it returns the coordinates of this square as {row, column} and otherwise the array {-1, -1} or `null`, whatever you prefer.

   To improve performance, keep track of the last found empty square (in instance variables `rempty` and `cempty`)

   Test this method.

4. Now write a recursive method `void solve()` that counts the number of solutions, and stores the last solution that was found.

   Our strategy is "brute force": we look for an empty square, fill it with a number and see if it fits (i.e., there are no conflicts with other numbers. If it doesn't fit, we try the next number, etc. When we have a number that fits, we fill it in and continue by tackling the smaller problem of finding the solutions of the grid with the extra filled in number.

   We keep only one grid in an instance variable, shared by all recursive calls. When all tries for a certain cell fail, you have to track back and reconsider a previous choice. Recursion will take care of this reconsidering, but you have to re-establish the grid to the state before the call, in other words, clean up your mess behind you.

   An alternative is to add the grid as a parameter to `solve` and make every recursive call on a *new* grid, copied from the previous one and provided with the extra filled in number. This is relatively expensive, however, since copying takes time and all these grids have to be stored. In this approach some more methods need a grid parameter (such as `givesConflict`).

   When you test this method, it may be helpful to add a statement to the method that prints the current grid. To avoid an output flood, start with a grid that is almost solved. Later you can remove the print statement or, even better, have it print the grid only now and then, e.g., after a row is finished, or every 50 calls of solving. To help you with testing, a correct Sudoku (not the solution to the puzzle given above) is printed on the next page.

5. Finally, if there is only one solution, print it using the method described above. If there is more than one possible solution, simply print the number of solutions.

[1]First left to right, then top to bottom.

```
+-----------------+
|1 2 3|4 5 6|7 8 9|
|4 5 6|7 8 9|1 2 3|
|7 8 9|1 2 3|4 5 6|
-------------------
|2 1 4|3 6 5|8 9 7|
|3 6 5|8 9 7|2 1 4|
|8 9 7|2 1 4|3 6 5|
-------------------
|5 3 1|6 4 2|9 7 8|
|6 4 2|9 7 8|5 3 1|
|9 7 8|5 3 1|6 4 2|
+-----------------+
```