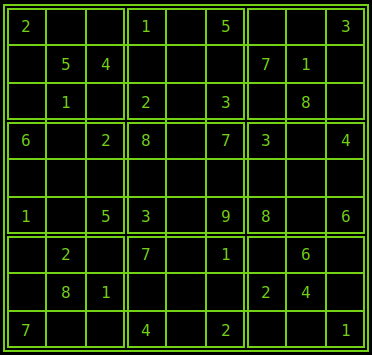
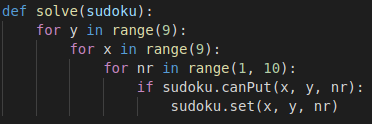
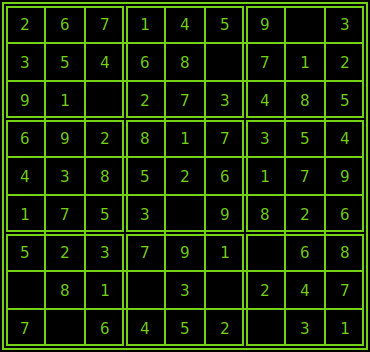
# Sudoku Solver

Sudoku is a puzzle with a 9x9 grid. Each grid is separated into 3x3 squares, columns and rows. The goal of the game is to fill the grid with numbers from 1 to 9 within certain restrictions: numbers should not be repeated in the 3x3 squares, the columns or the rows. When displayed, an unsolved sudoku, will look like this:

 To solve such a puzzle, a human would carefully look at all the already filled cells and fill the ones where there aren’t many options or preferably only one option. However to program such a behaviour would be tedious and unnecessary. A simpler solution would be to go through all the empty cells one by one and check whether we can put any of the numbers from 1 to 9. If so we put the number and continue onto the next cell.

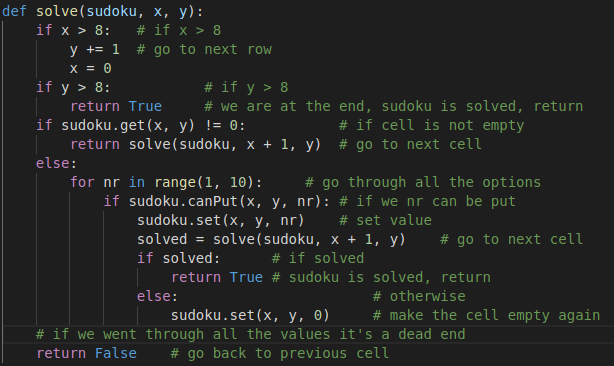
This can be achieved pretty easily by looping through all the cells and, for each of them, loop the numbers 1-9. Then we check whether the number can be put into that specific cell of the sudoku. If so we insert it otherwise we go on to the next cell.

The algorithm would look something like this:

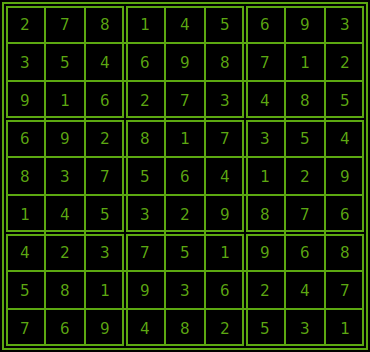


However with this approach we quickly run into a problem where some of our cells are being left empty. This is a result of a cell going through all the values and not finding any that fit that specific cell. Does that mean that the sudoku has no solution? No. It just means that we have hit a dead end and that one or more of the values we put in the previous cells needs to be changed. With the current approach of using for loops to go through all the cells it would be hard to make a system to go back to previous cells (though not impossible). It is much easier to make our function recursive to go through each of the cells and when we hit a dead end we just return to the previous call of the function, I.e. the previous cell.

Our new function looks like this:



When executing this new function we get the desired output:

 The result showed up almost instantly so let’s analyse the code to see what’s the runtime of our code. Since the function is recursive it’s harder then a regular function. Even the worst case scenario is hard to determine since we can be backtracking at any point.

Let’s start by going through the worst case scenario for one cell: the cell is empty and we loop through the values 1-9 and can only put the last value. That would give us a big O notation of O(9). If we multiply this by the number of cells (81) we get a big O notation of O(729). Now, this is not accurate because some cells are going to be filled and also we didn’t account for the backtracking when we hit a dead end. Also only one cell in 9 is going to be a certain value so that means we can get a big O notation of O(9 \* 9 + 9 \* 8 + 9 \* 7 + … + 9 \* 1) = O(405).

According to a study made by Cornell University (<https://arxiv.org/abs/1201.0749>) the sudoku has a minimum of 17 filled cells at the start. Since we are looking at the worst case we can use the values of nine 1’s and eight 2’s being filled which makes 17 cells filled. The O(405 – 9 \* 1 - 8 \* 2) = O(380).

This might seem like it is a very fast algorithm however the wildcard here is backtracking. So let’s try to figure out how many times the algorithm will backtrack in the worst case scenario. The sudoku has 9 \* 9 – 17 = 64 empty cells so the algorithm may backtrack on 63 of the cells. However we do not know how many times it will backtrack on each cell neither do we have a way to find it so let’s mark it as a variable b. The result big O notation is O(380 + b \* 9) because every time the algorithm backtracks it will loop through all 9 of the values and return.

As a conclusion this algorithm can be really fast or really slow depending on how many times it backtracks. There are many ways to improve it; for example we could start the loop from the start or from the end depending on which cells are empty. However that is for another project.

# **Notes**

- To run the project execute the python script main.py. The program will present you with a prompt and ask you to enter a command. All the commands will be listed on the right of the grid. If you need any help with the commands type `help <command>` to show help about any command.

- Otherwise if you just want the solving algorithm without the prompt there is a script solve.py which will load the test.sdk file present on the project and solve it.

- The algorithm for solving the sudoku in main.py and solve.py are a bit different because of some implementation details but the logic is still exactly the same.

- If you find this project on github it’s because I put it there and not because I plagiarized :) . My username is peTheProgrammer. Repository: <https://github.com/peTheProgrammer/Sudoku>