I opted to implement a depth-first search with backtracking for my sudoku solver. This approach was relatively easy to implement as depth-first search algorithms are widely discussed and employed in various contexts and therefore many resources exist to aid understanding of their functionality. In addition, heuristics can be added to the base algorithm in an easy, modular fashion to boost performance and efficiency without making large changes and refactors to the main algorithm. Yato & Seta (2003) have proven that the problem of solving sudoku puzzles of arbitrary size is NP-complete and therefore no algorithm currently exists which can solve any sudoku in consistent polynomial time - the existence of such would provide a solution to the Millenium Prize Problem. Some have written sudoku solvers using the A* algorithm which purportedly performs better in terms of time complexity, but it is more resource intense and harder to implement. This considered, the depth-first search with backtracking combined with heuristics presents an approach that is easily understood and implemented, while also achieving fast results for the constraint of a 9x9 grid (though my code has been intentionally designed to work for any size of square grid).

`fillRemainingGrid()` is a recursive method used to perform the aforementioned depth-first search. It works first by iterating through each cell in the sudoku grid checking if it's empty, then attempting to enter a value from the domain into this cell. If the entry is valid the method calls itself to check the next empty cell and repeat this process. If the entry is invalid the current iteration is terminated and removed from the stack of method calls - this is how the backtracking works. This approach does of course have its drawbacks. The time in which a solution can be found for a given starting sudoku varies greatly and therefore the worst- and best-case time complexities differ wildly. If the first number entered into an empty cell happens to be correct and therefore no backtracking is needed, the amount of recursive calls of the main loop drops considerably. In this case the time taken to solve the puzzle scales with the number of empty cells rather than the size of the input. Formally the worst case scenario has the time complexity $O(n^m)$, where n is the size of the domain (9 in our case) and m is the number of empty cells. Memory complexity is also a consideration as each recursive call of the main method results in another element being added to the call stack, meaning a sudoku of greater width and domain size would require an increasing amount of memory to be solved.

To allow my algorithm to perform more efficiently I added a heuristic which determines which values in the domain would be valid entries *before* attempting to add any to the grid. This means the number of recursive calls is greatly reduced: if one value in a row is missing it can only have one valid entry, so determining that value and entering it requires just one call of the main method, whereas the primitive version of the method would potentially need to be called 9 times (the size of the domain). This approach is inspired by the hidden and naked pairs techniques used by human sudoku solvers - by checking each 'house' (row, column and block) for all its legal candidates the domain for future empty cells shrinks and in turn the number of recursive method calls drops considerably.

There is plenty of room for improvement in my work, with the easiest and most obvious change being the introduction of more heuristics. Techniques such as the X- and Y-wing, or unique rectangles, are used by human solvers to further whittle down the valid entries for a given cell. The use of these techniques in my algorithm would further reduce the amount of brute-force value entry and therefore the time- and memory-heavy backtracking also.

Yato, T. and Seta, T., 2003. Complexity and Completeness of Finding Another Solution and Its Application to Puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*. E86-A(5), pp.1052-1060.