Task 1: The Grid Class

The isValid method

My isValid method is split into two parts, the first is a for loop that checks that the rows and columns are valid and the second is a nested for loop that checks each sub grid is valid. In each of these loops it calls a method to do the checks, if one of these methods returns false then isValid returns false, otherwise it will return true.

For the rows and column checks I made a private method isColumnAndRowValid which takes an integer and returns a Boolean. The integer parameter represents either the x or y coordinate of the column or row and the returned Boolean is true if the columns and rows are valid, false if not. This method uses an outer for loop which loops twice, on the first loop it checks the column, second loop it checks the row. When a loop starts it creates an array of 9 Booleans all set to false called foundNumbers. This array is used to show which numbers between 1 and 9 have already been found in the column or row. It then uses a for loop that checks every number in the column or row. If it is not zero and its corresponding value in foundNumbers is false, the value is updated to true. If the value is already true, then that means the number has already been found on that column or row, so the method returns false. If no problems are found it will return true.

For the sub grid checks I made a private method called isSubGridValid which takes two integers and returns a Boolean. The two integer parameters represent the x and y coordinates of the sub grid and the returned Boolean shows if the sub grid is valid or not. Like the previous method it uses an array of 9 Booleans called foundNumbers which are initially set to false. It also uses an equation on the x and y so that they can only be 0, 3 or 6, this is because the function will only work if the given coordinates represent the top left square of a sub grid. It then uses a nested for loop to go through every number in the sub grid. The checks performed on each number have the same functionality as in the isColumnAndRowValid method, meaning that if a number has already been found it will return false, otherwise it will return true.

How I found task 1

I found that implementing the constructor, get and set methods was quite easy. I struggled a bit more with the isValid method. For this I first decided that the best way to make it would be to add three private methods to share the functionality and make it easier to debug and understand, the first method handled columns, the second handled rows and the third handled sub grids, while isValid called them all. Whilst this solution worked, the method to check columns and the method to check rows were practically identical, so I combined them into one function.

Task 2: The Solver Class

How I tested the code

I tested my code for the Solver class by using the toString method I had made for the Grid class to print out the grid at different stages as it was solved. This combined with the examples provided allowed me to see how the solve method would work through different sudoku grids. After I thought that my solution was working correctly, I ran the tests provided to make sure that it was all working.

Big-O complexity

The worst case big-O complexity that I calculated was O(9^n) because for every empty cell there can be 9 potential different options, each of which could then lead to another 9 potential options.

More efficient solution

Off the top of my head, one way to improve the efficiency of the solution would be to make the isValid method in Grid only check one coordinate. This would improve efficiency because rather than checking that the entire grid is valid every time a new number is tried, it would only check the row, column and sub grid the number was placed in, since these are the only things affected by the new value. Another way to improve efficiency could be to have solve take two integers x and y representing the coordinates of either the cell just checked or the cell to be checked next. This would reduce the time spent finding the cell to be checked when solve is called, since it would not have to go over any values before the last cell with a number added.

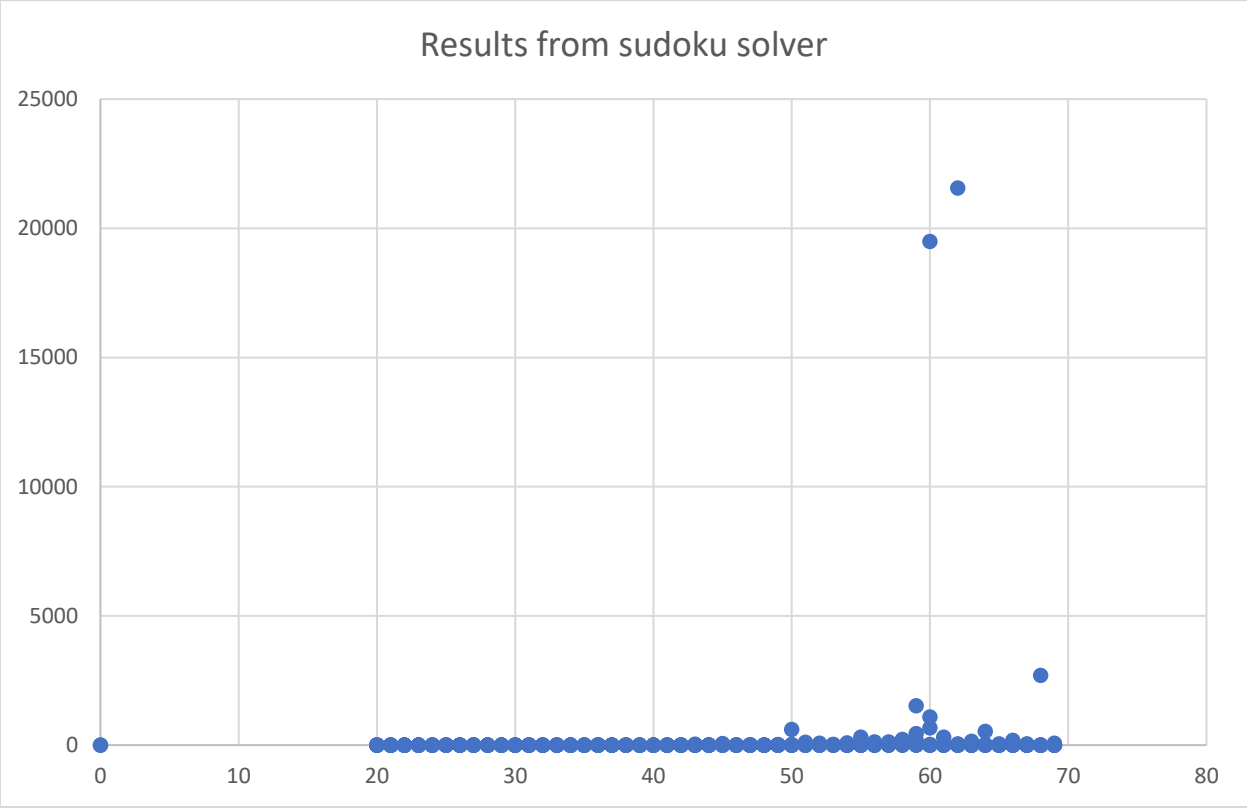
By looking at the algorithms provided online at https://www.geeksforgeeks.org/sudoku-backtracking-7/ I saw that similar improvements to the ones I suggested had been implemented.

How I found task 2

I did not struggle that much with task 2, I think this is because we were provided pseudocode for the solve method, which meant I did not have to worry too much about the logic.

Task 3: Solving Puzzles

Results



Above is a scatter graph with gap count along the x axis and time in milliseconds along the y axis. Unfortunately, I was unable to plot log of time due to many of the puzzles being solved so fast that my program returned 0ms so the log could not be calculated. Whilst the results do not support my big-O in task 2, this could be because that is a worst case scenario, and most puzzles will be solved in a far more tractable time.

How I found task 3

My biggest struggle with task 3 was figuring out why the gap count seemed to have little effect on the time taken for a puzzle to be solved, however I realised that this was because the runtime depended far more on how much backtracking was required, rather than how many gaps had to be filled in.

Task 4: Generating Puzzles

Pseudocode

```
function generatePuzzles takes int gapCount
        if 0 <= gapCount <= 80
                for that loops 81 - gapCount times
                        while new value not added
                                x = random int from 0-8
                                y = random int from 0-8
                                if grid[x][y] contains 0
                                        value = random int from 1-9
                                        set grid[x][y] to value
                                        if grid is valid and can be solved
                                                exit while loop
                                        else
                                                reset grid[x][y] to 0
```

Explanation

My function would take one parameter, an integer called gapCount which would represent how many gaps you would want in the puzzle. It would start by checking that this was a valid number (0 – 80) before starting a for loop that would loop 81 - gapCount times. In each of these loops there would be a while loop that repeats until a new value has been added. To add a new value, it would generate a random coordinate, check that the cell at those coordinates is empty then set the value of that cell to a random number from 1-9. The grid would then be checked using isValid and solve (since a valid grid may still be unsolvable) and if both methods return true it would exit the while loop, otherwise reset the cell to 0.

How I found task 4

Initially I tried to create proper code for my puzzle generator, however I got stuck when checking that the grid was solvable due to the solve method filling in the grid. To overcome this problem, you would have to make a copy of the grid (not a pointer) every time a new value was added, and due to time constraints I opted for pseudocode instead.

Self-evaluation

For me, the most difficult part was creating the Grid class, this is because the isValid method was quite difficult to make and unlike with solve we were not provided with any pseudocode. I also spent a long time figuring out how best to combine my methods for checking columns and rows, whilst the solution I got works, I believe it could be more optimised. The part I did most well on was the Solver class, this is because the solve method is efficient and well commented and I'm happy with how it turned out. Overall, I think I should get around 80% due to my program passing all the tests and completing most of the example puzzles in a very short amount of time, as well as me completing all of the tasks in the assignment with the exception of the proper code for task 4.