H2 Computing Practical Worksheet – T1W2

1 Sudoku is a number-placement puzzle. The objective is to fill a 9×9 grid with digits so that:

- each column contains all of the digits from 1 to 9 (each once)
- each row contains all of the digits from 1 to 9 (each once)
- each of the nine 3×3 subgrids contains all of the digits from 1 to 9 (each once)

The puzzle setter provides a partially completed grid.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

A typical Sudoku puzzle

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

And its solution

For more detailed information, you may refer to:
https://en.wikipedia.org/wiki/Sudoku

1a Design object-oriented classes that will allow you to:

- Store the contents of a Sudoku game grid
- Initialise a blank Sudoku grid
- Populate and/or update the Sudoku grid – i.e., populate/update any number of specific cells in the grid
- Check the validity of a puzzle – i.e., ensure that the grid is legal based on the requirements for a solution (or partial solution)

Your design should reflect the following object-oriented principles:

- Encapsulation (for data hiding and implementation independence)
- Inheritance (for code reuse)
- Polymorphism (for code generalisation)

You may also refer to the following guide for your design:
https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

1b Augment your code from **1a** such that it now also includes a brute-force Sudoku solution generator. More specifically, your solution generator should take as input 1 parameter**: the number of unique Sudoku solutions desired**, and then using a brute-force algorithm, generate the required number of complete (i.e., all grid cells are filled in) and legal Sudoku solutions.

Your solution generator should be deterministic.

This will require you to sequentially populate cells until either an illegal entry is made (in which case, the generator should move on the next iteration, discarding the current solution), or until a complete and legal solution is found, in which case, that solution is stored before moving on to the next iteration.

Along with your generator, **you must also include methods that support persistence**. To do this, you may either implement you own convention for saving the solution data in a text or binary file, or, you may use the Pickle module: https://docs.python.org/3.1/library/pickle.html

In your code, you must also leave a comment concerning the computational complexity of your Sudoku solution generator.

1c Augment your code from **1b** such that you now also include a puzzle generator. Unlike the solution generator from 1b, the puzzle generator will work as follows:

- Requires 1 argument: **the number of blank grid cells (b)**
- Uses the solution generator to generate 20 Sudoku solutions and then randomly choose 1
- Randomly removes approximately b/9 from each row or column or 3×3 subgrid
- Determines and returns (along with the puzzle) the number of possible solutions that are applicable to the current puzzle (this should be another method that is adapted from the solution generator method)

Once again note that your code **must also include methods that support persistence**.

In your code, you must also leave a comment concerning the computational complexity of algorithm you have used to check the number of possible solutions for the generated puzzle.

1d Write a different solution generator using one of the methods described in:
https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

Again, **you must also leave a comment concerning the computational complexity of this Sudoku solution generator.**