1. Fibonacci numbers and complexity

Fibonacci numbers are defined recursively as follows:
\[ F(0) = 1, \quad F(1) = 1 \]
\[ F(n) = F(n-1) + F(n-2) \text{ for } n > 1 \]

Write the following methods to compute F(n):
   a) A \( O(2^n) \) method based on the recursive definition
   b) A \( O(n) \) method that uses a loop
   c) A \( O(1) \) method that uses the closed form solution (feel free to look online for the formula)

Run experiments with various values of \( n \) and determine if their analytical running times agree with what you see in practice. Use graphs or plots to illustrate this.

2. Sudoku

Sudoku is a logic-based, combinatorial number-placement puzzle. The objective is to fill an \( n \times n \) grid with numbers so that each column, each row, and each of the \( \sqrt{n} \times \sqrt{n} \) sub-grids that compose the main grid contain all the digits from 1 to \( n \).

This is an example of a traditional \( 9 \times 9 \) grid in Sudoku (notice how each row, each column, and each \( 3 \times 3 \) sub-grid contain all the numbers from 1 to 9).

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

Solving a Sudoku game requires you to find the missing numbers of a partially completed grid. In this lab, we will only consider Sudoku grids where each row and column in the grid is missing exactly one value. This is an example of this particular set of grids:
There are multiple algorithms that can be employed to solve this type of Sudoku grids.

**Naïve Algorithm**

1. Assume that S is a 2-D array where a Sudoku grid is stored.
2. for r =0 to S.length-1
3. for num = 1 to S.length
4. isNumPresent = false
5. for i = 0 to S.length-1
6. if (S[r][i] == num)
7. isNumPresent = true;
8. break;
9. if (!isNumPresent)
10. print("The missing number in row: " + r + " is " + num)

**Not-so-Naïve Algorithm**

1. Assume that S is a 2-D array where a Sudoku grid is stored.
2. for r =0 to S.length-1
3. for num = 1 to S.length
4. isNumPresent[num] = false;
5. for i = 0 to S.length – 1
6. if (S[r][i] contains a known value)
7. isNumPresent[S[r][i]] = true;
8. for i = 1 to S.length
9. if (!isNumPresent[i])
10. print("The missing number in row: " + r + " is " + i)

**Not-Naïve Algorithm**

1. Assume that S is a 2-D array where a Sudoku grid is stored.
2. sum = S.length * (S.length + 1) / 2;
3. for r =0 to S.length - 1
4. missingNum = sum;
5. for i = 0 to S.length – 1
6.     if (S[r][i] contains a known value)
7.         missingNum - = S[r][i];
8.     print("The missing number in row: " + r + " is " + missingNum)

Your main task for this lab is to write a program that does the following:

1. Generate an n×n solved Sudoku game and remove one random element from every row. A simple way of generating a solved Sudoku puzzle is as follows:

   for r =0 to n-1
    startNum = sqrt(n) * (r % sqrt(n)) + (r/sqrt(n));
    for c =0 to n-1
     S[r][c] = ((startNum + c) % n) + 1;

2. Prompt the user to choose an algorithm (Naïve, Not-so-Naïve, Not Naïve)
3. Solve the Sudoku using the selected algorithm
4. Display the time it took the algorithm to complete

Implement the algorithms described above and perform experiments on 9X9, 25X25, 2,500X2,500, and 10,000X10,000 Sudoku grids. Remember that we are only considering Sudoku games where each row and column in the grid is missing exactly one value. Prepare a report describing your work as explained in the syllabus. Determine the running times for each algorithm, compare them, and show your results using tables and/or plots.