Casey Levy – CS 325 – Portfolio Project

1. **Puzzle**
   - The puzzle I chose to complete was **Sudoku**

2. **Rules**
   - Sudoku is a numbers game where the player must fill in rows, columns, and sections of boxes with numbers 1-9 and are not allowed to repeat any numbers within each respective row, column, or section. The game begins with a 9x9 board where many of the boxes will be pre-filled with a random number between 1-9. Each 9x9 board is broken up into 9 smaller sections, each with a 3x3 size. Each of these 3x3 sections must contain the numbers 1-9 in them as well and no numbers may be repeated. As you fill up your 3x3 sections, you must also ensure no numbers within your rows and columns are repeated. To win the game, you must ensure each row, column, and 3x3 section contain the numbers 1-9 with no repeating numbers. This particular Sudoku program contains a GUI with a "Reset" button to randomly generate a new board, as well as a "Verify" button to verify any moves the player makes to help them create a solution and a "Solve" button to automatically solve the board. The player is also given the ability to delete numbers from boxes, but ONLY from boxes where they've placed a number. Boxes pre-filled by the program cannot be removed.

   - **Note:** Sudoku boards may be much larger than 9x9, but for the scope and purposes of this project, the size is kept at the Sudoku standard board of 9x9.

3. **Solution Algorithm**

   lines 34 – 51 in sudoku_cert.py file

```
def solve_game(arr):
    solve_arr = arr
    empty = check_box_empty(solve_arr)

    if not empty:
        return solve_arr
    else:
        r, c = empty

    for x in range(1, 10):
        if verify_move(solve_arr, x, (r,c)):
            solve_arr[r][c] = x

            if solve_game(solve_arr):
                return solve_arr
            solve_arr[r][c] = 0

    return False
```

- Here, I am running a recursive algorithm to check for correct solutions within the board. This algorithm works together with the other methods for the game to ensure the numbers assigned to each box are valid and correct. If they are, this algorithm recursively runs through the grid and attempts to finish and complete the puzzle. The runtime complexity for this is **O(n^2)** since it must run through ***nxn*** boxes to verify integers placed by the ***verify_move()*** method.

4. **Verification Algorithm**

lines 54 – 72 in the sudoku_cert.py file

```
- def check_solution(game_board):
-     row = [r for r in game_board if not check_row(r)]
-     g_board = list(zip(*game_board))
-
-     for x in g_board:
-         if 0 in x:
-             return False
-
-     col = [c for c in game_board if not check_row(c)]
-     g_boxes = []
-
-     for x in range(1,9,3):
-         for y in range(1,9,3):
-             g_box = list(itertools.chain(r[y:y + 3]for r in g_board[x:x + 3]))
-             g_boxes.append(g_box)
-
-     solve_box = [b for b in g_boxes if not check_row(b)]
-
-     return not row or not col or not solve_game
```

- Here, I am running a double ***for*** loop to verify each box, column, and row. Based on validity, a Boolean value is returned.

5. **Algorithm Correctness**
   - The above verification algorithm is **NP-complete** since it runs in **polynomial time**. This gives us the ability to reduce the problem. If we iterate through each boundary and run a solution check on each row, we can revert the puzzle to its original problem. Reduction shows polynomial time which results in an **NP-complete** problem.

6. **Verification Algorithm Time Complexity**
   - Runtime: **O(n^2)**

Bonus

- **A.** Code to find solution to the puzzle is within this program using the “Solve!” button on the GUI
- **B.** **Solution Algorithm Complexity: $O(n^2)$**
- **C.** GUI shown when program is executed. See README.