Project 2: Sudoku Validation Solver
Shwetha Raju & Riya Patel
Due: March 11th, 2022
CSC 345 - 01

Implementation of Required Methods:

The goal of this project was to develop and analyze three different design choices for implementing a sudoku puzzle validation check program. These three options are placed into case statements in the program that choose a specific option and evaluate the sudoku board using that designated option.

The sudoku board is read and stored into a global variable called *sudoku* that is used throughout the program. To determine if the rows, columns, and squares in the provided sudoku board are valid, a global variable called *validFlag* is set to 1 which indicates a valid board, when one of the following functions finds an issue with a board, the *validFlag* is set to 0. This was done by utilizing the three following options:

Option 1: Implementing 11 Threads:

- 1 Thread for checking all 9 Rows
- 1 Thread for checking all 9 Columns
- 9 Threads for checking 3x3 Squares

Accomplished by using the following functions:

- **validRows**: This function iterates through the stored global *sudoku* variable and checks each row for duplicate numbers; this is done using a flags array that increments at each occurrence of a number.
- **validColumns**: This function iterates through the stored global *sudoku* variable and checks each column for duplicate numbers; this is done using a flags array that increments at each occurrence of a number.
- **validSquares**: This function takes in the input of every top left value of each of the 3x3 squares present inside a 9x9 sudoku board. This occurs when both the row and the column values are divisible by 3. Next, the function iterates through the values present in the square and checks for duplicate numbers; this is done using a flags array that increments at each occurrence of a number.

Option 2: Implementing 27 Threads:

- 9 Threads - 1 Thread for Each Row
- 9 Threads - 1 Thread for Each Column
- 9 Threads for checking 3x3 Squares

Accomplished by using the following functions:

- **validR**: This function takes in the input only of the **row** index and iterates through that one specific row and checks for duplicate numbers; this is done using a flags array that increments at each occurrence of a number.
- **validCol**: This function takes in the input only of the **column** index and iterates through that one specific column and checks for duplicate numbers; this is done using a flags array that increments at each occurrence of a number.
- **validSquares**: *This function is called twice and used in both Option 1 and in Option 2.*

Option 3: Implementing 11 Processes:

- 1 Process for checking all 9 Rows
- 1 Process for checking all 9 Rows
- 9 Threads for checking 3x3 Squares

Accomplished by using the following functions:

- **validRowsProcess**: Similar logic used in validRows used in Option 1, however, this function uses process-based commands such as the exit command.
- **validColsProcess**: Similar logic used in validColumns used in Option 1, however, this function uses process-based commands such as the exit command.
- **validSquaresProcess**: Similar logic used in validSquares used in Option 1 & 2, however, this function uses process-based commands such as the exit command.

Comparing Option 1, Option 2, & Option 3:

Table 1 : Runtime Testcases for Invalid & Valid Sudoku Boards for all 3 Cases

	INVALID SUDOKU			VALID SUDOKU		
Run	**Case 1**	**Case 2**	**Case 3**	**Case 1**	**Case 2**	**Case 3**
1	0.000841	0.002442	0.000558	0.001169	0.002097	0.000623
2	0.001337	0.002157	0.000571	0.001158	0.001995	0.000652
3	0.000725	0.002403	0.000421	0.001036	0.002007	0.000451
4	0.001102	0.001831	0.000403	0.001058	0.001662	0.000491
5	0.000775	0.001621	0.000622	0.001156	0.002143	0.000591
6	0.001011	0.001736	0.000563	0.001247	0.00177	0.000462
7	0.000827	0.001454	0.00046	0.001081	0.00121	0.000432
8	0.000757	0.001479	0.000572	0.001227	0.001884	0.000491
9	0.000702	0.001668	0.000532	0.001143	0.001429	0.000468
10	0.00136	0.002554	0.000579	0.000811	0.001254	0.000369
11	0.000954	0.001255	0.000468	0.000535	0.000833	0.000408
12	0.001094	0.001723	0.000577	0.000541	0.001466	0.000464
13	0.000525	0.001727	0.000533	0.000595	0.0014	0.000452
14	0.000648	0.001441	0.000501	0.000898	0.001521	0.000548
15	0.001517	0.00136	0.000399	0.000772	0.00147	0.000476
16	0.000502	0.001249	0.000486	0.000782	0.001201	0.000375
17	0.000992	0.001622	0.000465	0.000822	0.001564	0.000421
18	0.001248	0.001255	0.000534	0.000722	0.001313	0.000392
19	0.000612	0.001636	0.000398	0.000855	0.001385	0.000369
20	0.000913	0.001624	0.000465	0.001199	0.001574	0.000466
21	0.000901	0.001019	0.00039	0.000819	0.001321	0.000399
22	0.001148	0.001492	0.000515	0.000782	0.001749	0.000518
23	0.00098	0.001758	0.000474	0.000761	0.001441	0.000418
24	0.000997	0.001955	0.000521	0.001116	0.001374	0.000424
25	0.000838	0.001443	0.000471	0.000867	0.002184	0.000614
26	0.000434	0.000859	0.000363	0.000899	0.001121	0.000427

27	0.000407	0.001419	0.000353	0.000441	0.001213	0.000399
28	0.001154	0.001968	0.000582	0.00081	0.001307	0.000376
29	0.00067	0.001425	0.000398	0.00106	0.001418	0.000395
30	0.000582	0.000867	0.000395	0.001058	0.001076	0.000374
31	0.000876	0.001283	0.00038	0.000606	0.001361	0.000376
32	0.000918	0.002181	0.000682	0.000752	0.00153	0.000381
33	0.000432	0.001268	0.000361	0.0008	0.001802	0.000569
34	0.00079	0.00149	0.000472	0.00089	0.001386	0.000369
35	0.000921	0.000787	0.000404	0.0009	0.001687	0.000627
36	0.000652	0.001492	0.000511	0.000838	0.001574	0.000476
37	0.000881	0.00152	0.000457	0.000512	0.00121	0.000566
38	0.000836	0.001316	0.000402	0.001247	0.001968	0.000419
39	0.000857	0.001411	0.00047	0.000439	0.001424	0.000474
40	0.00075	0.001209	0.000378	0.000804	0.001549	0.000535
41	0.000799	0.001416	0.000386	0.000556	0.00139	0.000382
42	0.000413	0.001184	0.000393	0.000601	0.001477	0.000392
43	0.000915	0.001206	0.000615	0.000706	0.001449	0.000509
44	0.000854	0.001395	0.000393	0.000914	0.001234	0.000413
45	0.000665	0.001655	0.000428	0.00041	0.00157	0.000384
46	0.000577	0.000995	0.000368	0.00091	0.001251	0.000368
47	0.000756	0.001139	0.000343	0.000727	0.001424	0.000407
48	0.000778	0.001531	0.000387	0.004646	0.001743	0.000463
49	0.000492	0.001595	0.00043	0.000625	0.001218	0.000393
50	0.000425	0.001373	0.000492	0.001062	0.001394	0.000412

Table 2 : Runtime Averages for Invalid & Valid Sudoku Boards for all 3 Cases

	Invalid Solution Runtime Average (s)	Valid Solution Runtime Average (s)
Case 1	0.0008228	0.0009273
Case 2	0.00151776	0.00150046
Case 3	0.00046642	0.0004532

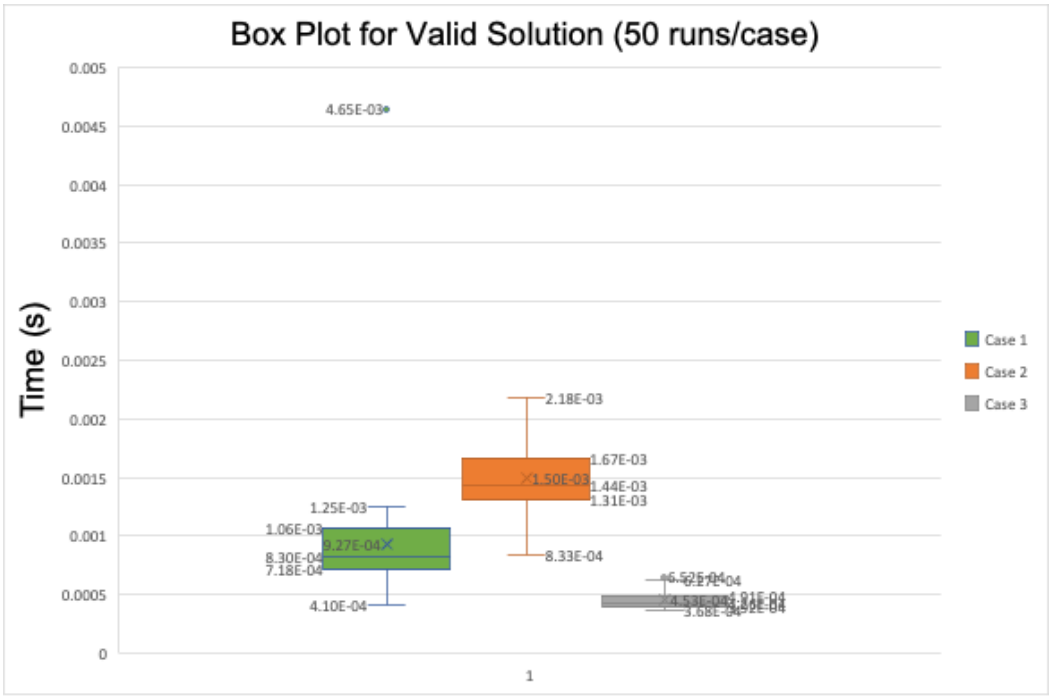

Figure 1: Box Plot Diagram for Valid Solution Test Cases

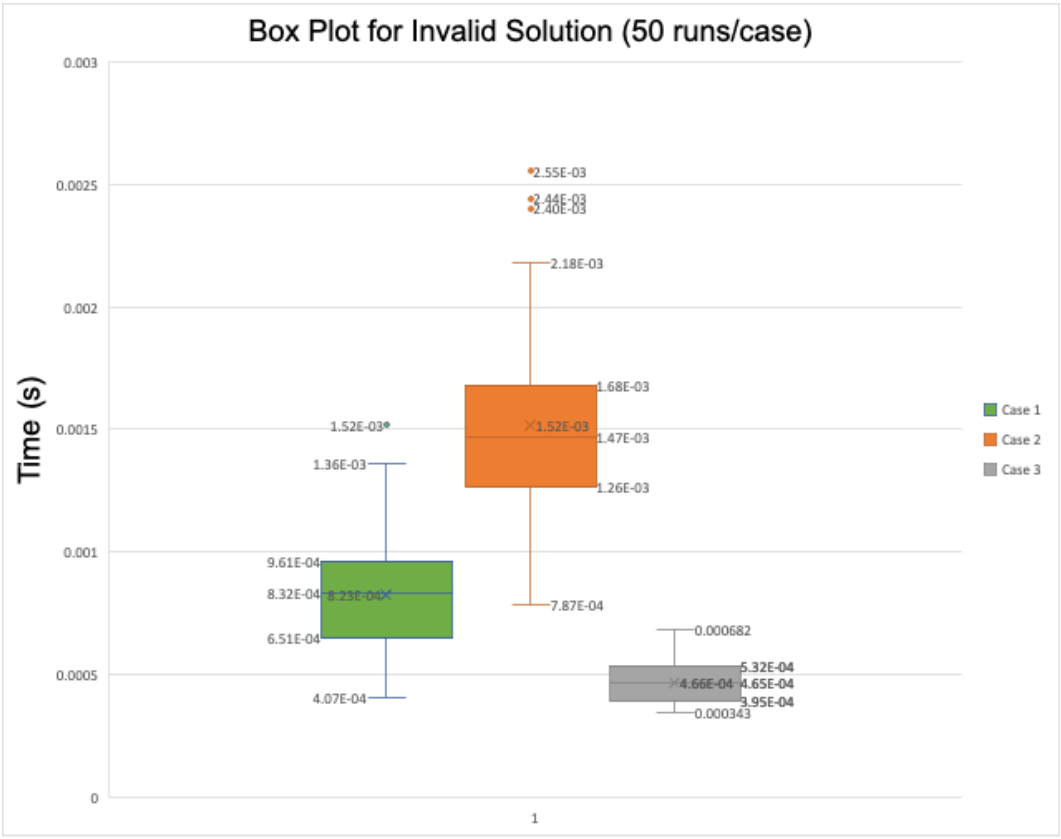

Figure 2: Box Plot Diagram for Invalid Solution Test Cases

The first table shows all of the data obtained from running our test cases for cases 1, 2, and 3. To ensure that the different options were tested thoroughly, all three options were run 50 times each, and on valid and invalid sudoku boards. To streamline our data all three options were compared to one another and all compared to each other in one singular table. This provided a wide range of data that would ultimately give a very well-rounded look into the results.

From this collection of data, the averages for each option and for invalid and valid solutions are displayed in the second table. Taking a look at the invalid solution runtime averages, the average for case 1 is 59.3841% faster than the case 2 averages. Comparatively, averages for case 3 are 55.2861% faster than averages for case 1. Next, to compare the results for valid solution runtime averages; average run times for case 1 were 47.2172% faster than case 2 average runtimes. Following this trend, case 3 average run times were 68.6853% faster than case 1 run times. These results are also visually represented in box graphs as shown in Figures 1 and 2. These figures remain consistent with the results that were shown previously as well.

With the above analysis, it can be observed that the initial hypothesis of "*There is no statistically significant difference between methods."* is proved to be false. From our analysis, case 3 proved to be the most effective (fastest), case 1 being the second fastest, and case 2 being the least efficient (slowest). This is because case 3 only runs 3 processes vs case 1 that runs 11 threads and case 2 that runs 27 threads. Furthermore, there are no parameters being passed into case 3 and case 1 that allow these results to be more efficient as it's not waiting for any input; unlike case 2 which also is waiting for parameter inputs.