Lucy Barest

Prof. Aaron

CS231A

3/17/23

**Project 5:** Sudoku Solver

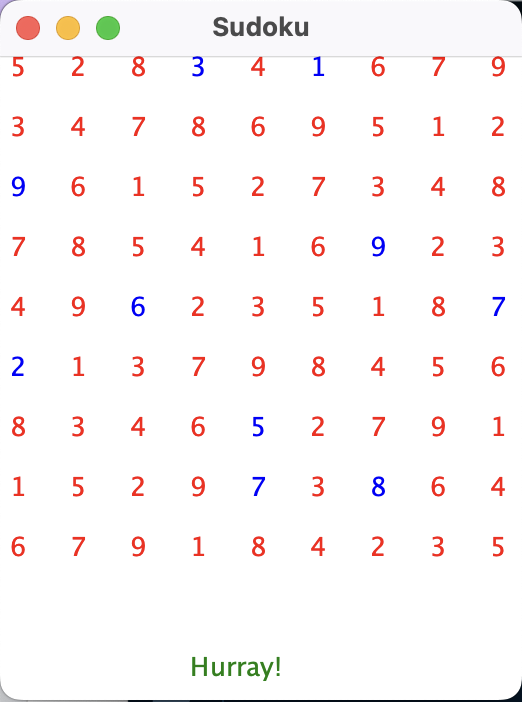
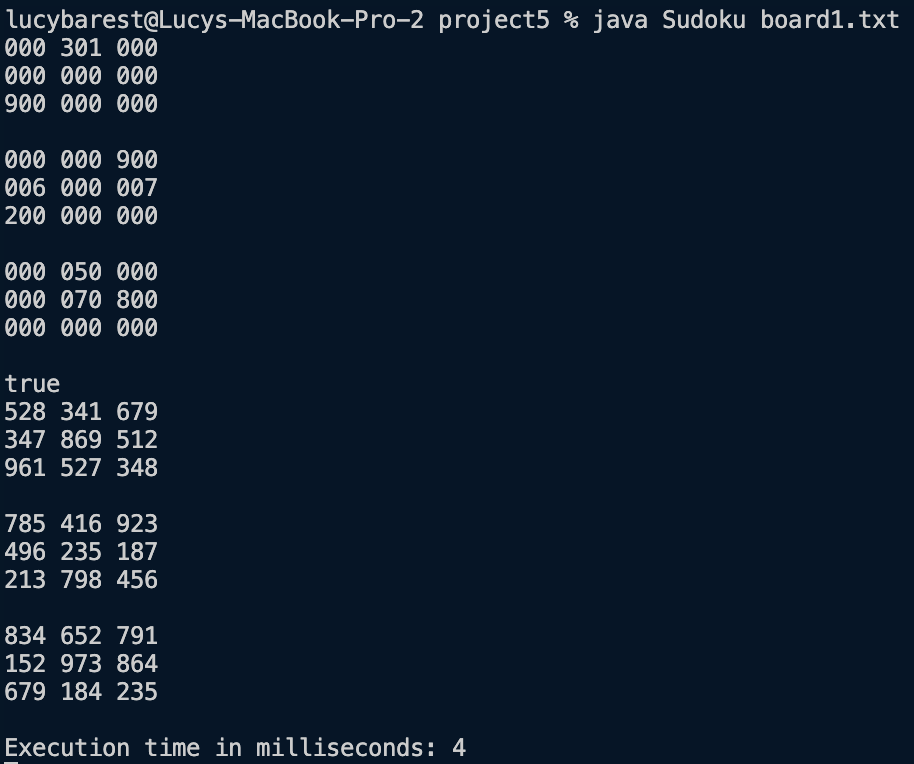
**Abstract:**

Using depth-first search, this project aims to solve a sudoku puzzle - a game that requires a 9x9 board of numbers to be filled in such a way that no number appears twice in any row, column, or 3x3 square. Backtracking is employed when a solution is not possible based on the current board. To facilitate backtracking, a stack data structure is used, with entries being removed from the front of the stack. The result is the creation of a sudoku solver and a method to display the board as it is being solved.

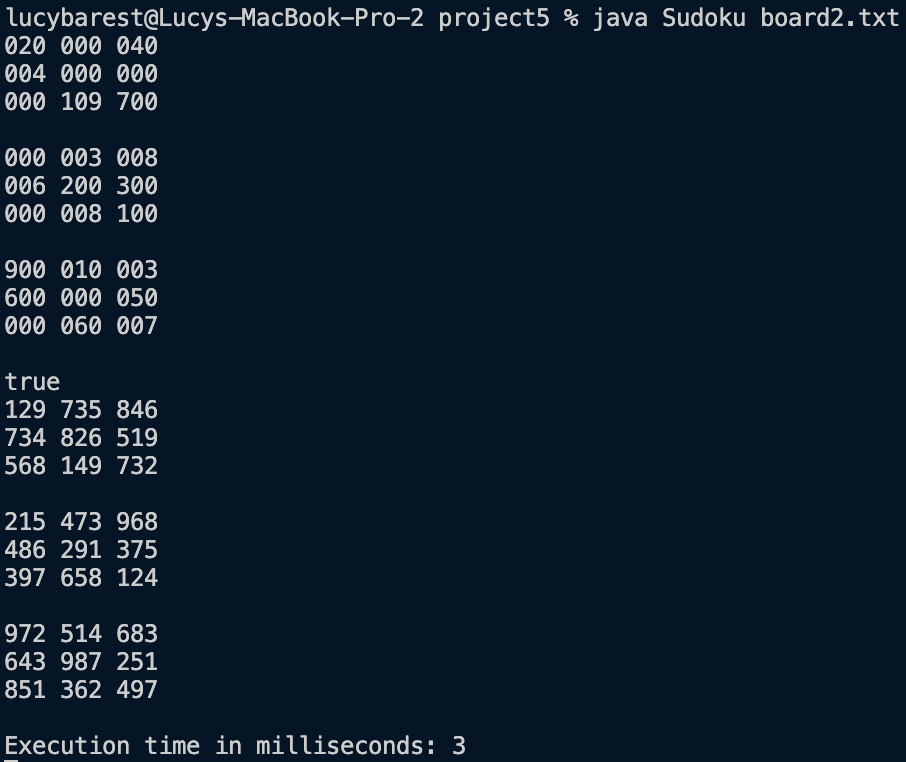
**Results:**

In both board#.txt files, the terminal displays the original text file and then returns a boolean (‘true’ to represent that the board has been solved) with the solved board underneath. Additionally, the non terminal displays show the locked values in blue and the solved values in red with a celebratory ‘Hurray!’ in green text underneath the board.

board1.txt terminal display and solved sudoku board display



board2.txt terminal display and solved sudoku board display



**Exploration:**

*(I am sorry about the size of the images but the resolution should be good enough so that if you zoom in you can see the initial board in the terminal, followed by the boolean representing the solved status of the board, then followed by the final board result underneath the boolean)*

1. All 5 boards with 10 locked cells were solved, 3 out of 5 boards with 20 locked cells were solved, but no trial of the boards with 30, 40, or 50 locked cells were solved. From this we can infer that there is a limit to the amount of locked cells that can be on a board for it to still be solvable.
2. Average execution time:

10 locked cells = (3+4+4+3+3)/5 = 3.4 milliseconds

20 locked cells = (5+3+1+7+3)/5 = 3.8 milliseconds

30 locked cells = (1+1+0+1+2)/5 = 1.0 milliseconds

40 locked cells = (0+1+0+0+0)/5 = 0.2 milliseconds

50 locked cells = (0+1+0+1+1)/5 = 0.6 milliseconds

From this (limited) data I can conclude that on average, when there are more locked cells, it takes slightly longer to solve the board because more locked cells result in more backtracking (because less values work for each cell to begin with).

|  | Board 1 | Board 2 | Board 3 | Board 4 | Board 5 |
| --- | --- | --- | --- | --- | --- |
| 10 Locked |  |  |  |  |  |
| 20 Locked |  |  |  |  |  |
| 30 Locked |  |  |  |  |  |
| 40 Locked |  |  |  |  |  |
| 50 Locked |  |  |  |  |  |

**Extensions:**

For my extension I created slay(good) testing files for the Board, Cell, CellStack, and Sudoku methods. Each file was named with respect to the method it was testing and was tested using different boards (of different locked values in some cases). The output from each test is displayed in the table below.

| BoardTests.java | CellTests.java | CellStackTests.java | SudokuTests.java |
| --- | --- | --- | --- |
|  |  |  |  |

**Reflection:**

In this project, a stack was utilized as the optimal data structure for backtracking because it allows the most recent move to be reversed first, rather than starting from the beginning, which saves time and avoids repetition. Additionally, a stack is ideal for the depth-first search approach because it only requires accessing the first element and adding to the front, making it straightforward to use.

**References:**

Lora LaRochelle - helped with exploration and general confusion with methods

Henry Landay - helped with general confusion with methods

Shelby Roman - helped with general confusion with methods

Anya Jiang - helped develop thorough (and actually useful) test files

Prof. Bender - helped clean up LinkedLists.java for project 4, which carried over to this project