Kevin Chen
kjc004
EECS 348
Sudoku Solver

1) Code
    a) Attached, in python. One small modification was made to the starter code in `sudoku.py`. All other code is in `solver.py`. To use, run `python -i solver.py` from the command line or load the file from IDLE. Have the appropriate sudoku file in the folder, then run `solve(filename, method)`, where `filename` is the name of the file and `method` is either `"forward"` or `"back"` (by default `"back"`).
    b) I conceived of and wrote all of the code.
2) Write-up
    a) Code representations:
        i) Board: an instance of the provided start code `SudokuBoard` class. More specifically, the `CurrentGameboard` is an array of each of the rows of the sudoku board.
        ii) Variables: each entry in the sudoku board is a variable. It can be accessed by the `CurrentGameboard` array by choosing a row and column index.
        iii) Constraints: Sudoku rules / must be a proper sudoku board (column, row, box restraints). I wrote a function called `is_valid_board_p` to check if a given board is valid.
        iv) States: each given instance of the `SudokuBoard` class is a different state, with a different `CurrentGameboard`.
    b) Forward Checking:

An array `allowed_values` is calculated through each recursion listing the allowed values for each empty space in the sudoku board. I select one of those variables, assign it one of those values, then recurse. At the beginning of each recursion, I check if `allowed_values` indicates that any spot doesn't have any allowable values -- if this is true, then I stop this branch and move on to the next possibility.

    c) Performance:

Size	Backtracking	Forward Checking	MRV + MCV	MRV + MCV + LCV
4x4	36 0.007 seconds	5 0.004 seconds	3 0.002 seconds	3 0.008 seconds
9x9	23152 10.28 seconds	847 2.606 seconds	530 1.122 seconds	345 0.815 seconds
16x16	did not finish	> 70 000	5747	> 230 000

		did not finish	30.66 seconds	did not finish
25x25	did not finish	did not finish	did not finish	did not finish

Interesting additional case (originally implemented by accident): MRV + *Least* Constrained Variable (opposite MCV):
4x4: 3 checks, 0.002 seconds; 9x9: 73 checks 0.165 seconds; 16x16: 2530 checks, 15.33 seconds; 25x25: > 4 000 000 checks

d) In general, adding heuristics and forwarding checking very definitely and significantly impacts the performance, reducing it significantly from straight recursion. Unexpectedly, the 16x16 did not finish when all of the heuristics were implemented compared to just MRV + MCV, and an accidental implementation of MRV + least CV outperformed everything. However, it's extremely possible that some combination of heuristics just happened to be very well suited for a specific board or problem. More generally though, MRV seemed to be the most impactful heuristic, scaling very well as the board size increased. For even the 9x9, the heuristics were very crucial in increasing our ability to solve larger boards and forward checking scales much better than simple backtracking.

e) Heuristics:

   i) MRV + MCV: MRV is implemented by changing the process by which we choose the coordinate we edit and recurse down. We do this by checking the number of possible values for all of the coordinates left open on the board. Since we already wrote a function to get all the allowable values of an spot, we used this function on the entire board. We then sort by this number, with the minimum at the front of the list. To add MCV as a tiebreaker, we take advantage of the fact that Python's `sort` method is stable, i.e. the relative order before the sort is maintained. We sort by the "constrainingness" of the spot before we apply the previous MRV sort. This is down by counting the number of zeros that placing a value in this spot would affect, e.g. all zeros in row alignment, column alignment, and in the same box as the current coordinate.

   ii) LCV: After implementing the above, we take the list of allowed values for the coordinate we're recursing down sort these by their "constraintness" -- this will change the order in which we try these values. To do this, we sort each value by a score determined by the number of aligned (row, column, box) empty coordinates that could possibly also take on that value.