Name: ⟨ Julius Putra Tanu Setiaji ⟩ Matric: ⟨ A0149787E ⟩
Name: ⟨ Tan Zong Xian ⟩ Matric: ⟨ A0167343A ⟩

1 Algorithm used

We model the sudoku puzzle as a constraint satisfaction problem (CSP). The algorithm used is constraint propagation and backtracking with minimum-remaining-value (MRV) heuristic. We will explain the algorithm more in depth below.

1.1 Initialisation

1. Traverse the input puzzle to generate lists of sets containing non-zero numbers in each row, column and 3x3 subgrids.
2. Generate the set of possible values for each cell using the set difference of the domain of each cell ($[1, 9]$) and the existing numbers in the same row, column and 3x3 subgrid as the cell (this was computed in step 1).

   During this step, if any empty cell (value $= 0$) has a possibility set of size 1, push the cell's coordinates and its sole possible value onto a queue for constraint propagation in Phase 1.2.

1.2 Set value and propagate constraint

While the queue is not empty:

1. Pop from queue to get cell coordinates and value v.
2. Set the cell in the answer matrix to v.
3. For each cell in the same row, column or 3x3 subgrid:
   (a) Remove v from its set of possibilities.
   (b) If the cell's possibility set is empty, stop this phase as it means that the answer matrix has an inconsistent assignment. Note that if this phase is called from Phase 1.3, this will result in backtracking.
   (c) If the cell's possibility set size is 1 and its coordinates are not in the queue and its value in the answer matrix is zero (not assigned yet): push the cell's coordinates and its sole possible value onto the queue.

1.3 Backtracking

1. Find the cell with possibility set with the smallest possibility set size and size > 1 (if the size is 1, then the cell has already been assigned a value).
2. Create a copy of the current solver object, push a possible value onto its queue and return to Phase 1.2.
3. If the result is a complete and consistent assignment, return it. Otherwise, backtrack to next possible value.

2 Complexity Analysis

Let n be the dimension of the puzzle i.e. 9.

2.1 Time Complexity

Each step of the initialisation is $O(n^2)$ as they each traverse the puzzle once. In Phase 1.2 constraint propagation, when a cell's value is set, updating for same row, column and 3x3 subgrid take $O(n)$ each. This takes $3n$ steps which reduces to $O(n)$. This has to be done once for each cell in the puzzle, giving a total of $O(n^3)$ in the ideal case without backtracking.

2.2 Space Complexity

Each instance of the solver object has stores the answer matrix and the queue of values to propagate. The queue will not have duplicate elements so in the worst case, the queue's size is $O(n^2)$. The sets of possible values each has a constant size of the whole domain of the worst case, thus their size is $O(1)$. Since each cell has an associated set, in total the size of all the sets is $O(n^2)$/ Hence each instance of the solver object takes $O(n^2)$ space.

Backtracking is depth-first search. For the provided puzzle, our recursion depth was 16. Hence we store at most $16O(n^2)$.

3 Optimisation

When we first implemented the algorithm, we had a lot of instance attributes in the `Solver` class which turned out to be redundant. Removing these instance attributes allowed us to obtain some speedup as there are less attributes to be copied in Phase 1.3 during backtracking search.

To ease implementation, we also made use of Python's built-in `copy.deepcopy` function. While it is convenient, it is also highly inefficient as it is designed very broadly. For example, it keeps a memoisation table to be able to perform deep copying of recursive data structures. We know for a fact that we have no such recursive data structures in our solver, thus using other constructs to copy is so much faster.

In fact, when we use the built-in Python `cProfile` module to profile our solver, most of the time was spent on performing `copy.deepcopy` during Phase 1.3 (more than 75% from our profiling). Thus, we can get the most performance boost by optimising the copy. Initially, we tried using the `cPickle.dumps` and `cPickle.loads` methods in-memory. Immediately we gain an improvement that cuts the run-time by a factor of 4. However, the time spent copying still accounts for 65% of our runtime. So instead, we change the `Sudoku` class initialiser to take in keyword arguments, and allow keyword arguments to override states instead of calculating them from the `puzzle` that is passed in. Copying is achieved by copying each of the state (instance attributes), and passing them to the initializer. This is the final method that we use in our solver, with the copying only taking about 11% of the total running time, which is faster by about a factor of 8 compared to our original solver. Just for good measures, we also memoised the function `same_square` since the function is pure.