1. Statement of Approach and Its Difficulty: My approach models sudoku as an exact cover problem (Knuth, 2000), where each valid digit placement is a candidate that satisfies four constraints: cell, row, column, and box uniqueness. I represent each of the 729 candidates with a 4-element array storing the corresponding constraint indices. A depth-first recursive backtracking algorithm based on Algorithm X is used to explore valid combinations. Unlike Knuth's original Dancing Links implementation, my version simplifies constraint management using a 729×4 NumPy array and a dictionary that maps constraints to candidate rows. Constraint state is efficiently updated through in-place modifications and selectively backed up for backtracking, avoiding the overhead of duplicating data structures.

2. Core Algorithm Description: The core of the implementation is Knuth's Algorithm X, a depth-first recursive algorithm for solving exact cover problems. First the sudoku is reduced to an exact cover matrix (Stolaf.edu, 2021) (a 729 x 4 NumPy array) where each row in the matrix represents a candidate digit placement, and each column corresponds to one of four constraints: cell occupancy, row uniqueness, column uniqueness, and box uniqueness(ref). There are 729 possible candidates—one for each digit (1–9) in each of the 81 cells—and 324 constraints in total.

Constraint Type	Formula	Value Range
Cell	Row * 9 +col	0-80
Row-Digit	81 + row * 9 + digit	81-161
Col-Digit	162 + col * 9 +digit	162-242
Box-Digit	243 + (row / 3) * 27 + (col / 3) * 9 + digit	243-323

For example, the candidate row [0, 85, 166, 247] satisfies the constraints for filling cell 0, placing a digit in row 0, column 3, and box 2 respectively. A supplementary structure, constraint_map, maps each of the 324 constraints to the candidate rows that satisfy it.

The algorithm selects the constraint with the fewest candidates to minimise branching, recursively explores each option. When a candidate is selected, all other candidates that share any of its four constraint indices are removed from the matrix, and the constraints that the selected candidate satisfies are marked as covered and eliminated from further consideration. If a chosen path leads to a dead end, the algorithm backtracks by restoring the previous state and trying the next candidate. This process continues until either all constraints are satisfied—producing a complete solution—or no valid options remain, meaning the sudoku is unsolvable. The solved sudoku is then reconstructed from the stored candidate choices or filled with -1 if unsolvable.

3. Optimisations: I used a minimum remaining value heuristic—selecting the constraint with the fewest candidates to reduce branching. Additionally, I updated the constraint map in place and only backed up affected entries for fast backtracking. Third, candidate constraints were precomputed in a NumPy array, enabling fast access and reducing memory overhead. Instead of using Knuth's large 729×324 binary matrix, my implementation only stores the 4 non-zero constraint indices for each candidate and a map of which candidates belong to each constraint. Candidate elimination and restoration are handled using Python's `dict` and `set` structures, which offer average-case O(1) time complexity. Finally, using sets helped quickly detect and remove conflicting candidates. These optimisations reduce the number of choices the algorithm has to explore and speed up each step. While the worst-case time complexity is still exponential—up to O(b^{324}), where b is the average number of options per constraint—the solver performs efficiently in practice. Most puzzles are solved in under a second by avoiding unnecessary recursion and taking advantage of the sparse structure of the constraint graph.

4. Reflections and Suggestions for Further Work: The solver performs reliably on all difficulties of sudoku puzzles, consistently producing correct solutions in under one second. Most puzzles have a runtime of <0.003s and memory usage is fairly low due to compact data structures. To optimise my approach further, using more advanced data structures such as Dancing Links could further reduce the cost of backtracking; combined with an implementation in a low-level language like C, where memory can be managed directly using pointers, resulting in faster execution and reduced overhead.

References:

Knuth, D.E., 2000. Dancing links [Online]. arXiv.org. Available from: https://doi.org/10.48550/arXiv.cs/0011047.

Russell, S.J. and Norvig, P., 2016. Artificial Intelligence: a Modern Approach. 3rd ed. Upper Saddle River: Pearson.

Stolaf.edu, 2021. Available from: https://www.stolaf.edu/people/hansonr/sudoku/exactcovermatrix.htm.