# Sudoku Documentation:

This document presents the design of the provided Sudoku solver, detailing the chosen data structures and algorithm, along with an analysis of their time and space complexity.

## High-level approach:

- My Sudoku solver uses a recursive backtracking search:
    - It scans the grid for the first empty cell (0).
    - It tries candidate numbers *1*..*n* in that cell.
    - For each candidate, it checks Sudoku constraints (row, column, subgrid) with isValid.
    - If valid, it places the number and recurses to solve the remainder of the grid.
    - If the downstream recursion fails, it resets the cell to 0 (backtracks) and tries the next candidate.
    - If all cells are filled without conflicts, the puzzle is solved.

## Low-level approach:

1. Simplicity and direct mapping to the Sudoku matrix.
2. Excellent cache locality for row/column scans used in “isValid”.
3. Minimal overhead—just the numbers, no auxiliary containers required.
    a. Generality: Works for any n x n where n is a perfect square (supports 4x4, 9x9, 16x16, etc.).
    The “subgrid” size is computed as “subSudokuSize = (int) Math.sqrt(n)”.

4. Implicit recursion stack:
    a. Backtracking leverages the call stack to remember decision points. No explicit stack structure is needed, keeping the code concise.

5. In-place mutation:
    a. The same int[][] is updated as the search proceeds, avoiding extra copies.
    b. The logical basis: Minimizes memory usage and reduces allocation overhead.

## The algorithm details:

**Backtracking search (solveSudokuWithBacktrackingAlgorithm):**

Depth-first search over partial assignments with pruning via constraint checks.

The reasons behind my choice:

1. Simplicity of implementation and correctness for all valid inputs.
2. Strong pruning from Sudoku's constraints—fast enough for typical puzzles.
3. Extensible: can incorporate heuristics (e.g., variable and value ordering) without changing the core structure.

**Constraint checking (isValid):**

1. Row check: scans the row for duplicates of num.
2. Column check: scans the column for duplicates of num.
3. Subgrid check: scans the k x k subgrid that contains (row, col).
4. Complexity per check is linear in n (explained below).
5. Note on the subgrid condition: if (sudoku[i][j] == num && (i != row && j != col)) return false;

- How does it work:

  The method first checks the entire row and column separately; thus, the subgrid check only needs to detect duplicates that are not in the same row or column, which the (i != row && j != col) predicate achieves. This also avoids flagging the current cell in isSolved where the value is momentarily present.

**Complexity analysis:**

Let $n$ be the grid width/height, where $n = k^2$ and $k = sqrt(n)$ (e.g., $n=9$, $k=3$). Let $E$ be the number of empty cells at the start.

- Per-constraint check cost (isValid):
  - Row scan: $O(n)$ comparisons.
  - Column scan: $O(n)$ comparisons.
  - Subgrid scan: $k * k = n$ comparisons.
  - Total per call: $O(n)$ (specifically up to $3n$).

- Backtracking worst-case time complexity:
  - At each empty cell, up to $n$ candidates can be tried.
  - Each candidate attempt performs an $O(n)$ validity check.
  - In the worst case (pathological or unsolvable boards), the search explores up to $n^E$ assignments.

    - Upper bound: $O(n * n^E) = O(n^{(E+1)})$. Since $E \leq n^2$, a coarse worst-case bound is $O(n^{(n^2+1)})$, often summarized as exponential in the number of empty cells.
    - For a standard 9x9 with many clues, real-world performance is typically fast due to strong pruning; but theoretically, the problem is NP-complete.

- Backtracking average/typical behavior:
    - With standard Sudoku puzzles (well-posed with a single solution and reasonable clue counts), the algorithm tends to run in milliseconds to seconds, depending on ordering and density of clues.

- Space complexity:
    - Grid storage: $O(n^2)$ integers in-place.
    - Recursion stack: depth up to $E$ (one frame per empty cell along a search path), so $O(E)$ frames. Each frame holds a constant amount of data, hence $O(E)$ additional space.
    - Total auxiliary space: $O(n^2 + E)$; since $E \leq n^2$, this is $O(n^2)$.

- "isSolved(int[][])" complexity:
    - Iterates over all $n^2$ cells, calling "isValid" per cell.
    - Time: $O(n^2) * O(n) = O(n^3)$.
    - Space: $O(1)$ beyond the input grid.

**Strengths of this approach:**

1. Simple, readable, and maintainable core.
2. Works for multiple grid sizes (4x4, 9x9, 16x16, ...), if $n$ is a perfect square.
3. Minimal memory footprint due to in-place backtracking.

**Potential optimizations:**

1. Variable ordering heuristic (MRV): Choose the next empty cell with the fewest legal candidates rather than the first encountered. This reduces branching early.
2. Value ordering heuristic: Try more promising numbers first (e.g., those least constraining neighbors).
3. Constraint propagation can be optimized by maintaining candidate sets (or bitmasks) for rows, columns, and subgrids, so that validity checks become effectively O(1). With this representation, each assignment automatically prunes the relevant domains (forward checking), and using three n x n boolean arrays or integer bitmasks for rows, columns, and subgrids enables both checking and undoing moves in constant time.

**When to consider a different algorithm:**

- Honestly,  if you want to solve a very large or adversarially hard Sudokus (e.g., `16x16` with few clues, or bigger) at scale or under strict time constraints, consider bitset-augmented backtracking with MRV or an exact-cover solver (Algorithm X). These approaches maintain the same worst-case exponential bound but greatly reduce explored states in practice.