Jessica Chen jchen204@u.rochester.edu
Shadiya Akhter sakhter@u.rochester.edu
CSC242 | Adam Purtee
Project 2: Sudoku (Java)

1. Explain design choices (data structures/objects) for representing sudoku board and corresponding constraints; identify computational complexity (time and space); identify key methods.
Our Sudoku solver represents the board as a 9x9 integer array, where each cell holds a number (1-9) or 0 for an empty cell. This choice provides efficient random access, O(1) lookup/modification, and keeps memory usage low. Constraints are implicitly enforced through the getPossibleValues(int row, int col) method, which checks row, column, and 3x3 subgrid constraints using a boolean array (boolean [] used) to track forbidden values. The AC-3 algorithm uses a queue to enforce arc consistency by iterating over empty cells, but it does not actively remove inconsistent values, which limits its effectiveness. The backtracking solver (solveSudoku) uses a heuristic (findUnassigned) to select the next cell with the fewest possible values, reducing search space The time complexity of backtracking in the worst case is O(9^n), but the heuristic improves practical performance. The space complexity is O(1), as the board size is fixed, though recursive calls in the worst case can add O(n) stack depth. Key methods include ac3() for constraint propagation, solveSudoku() for backtracking, getPossibleValues() for checking valid numbers, and findUnassigned() for heuristic-based cell selection.

2. How effective is an approach on pure backtracking (without AC-3)? Can we solve all easy cases & hard cases?
A pure backtracking approach (without AC-3) can solve all solvable Sudoku puzzles, including easy and hard cases, but its efficiency varies significantly depending on the puzzle's difficulty. For easy cases, where most of the cells are prefilled and constraint propagation is minimal, backtracking performs well and can find a solution quickly. However, for harder cases, especially those with few given numbers, pure backtracking can become extremely slow because of excessive guessing and recursion depth. Without forward checking or constraint propagation, the solver may explore many invalid branches before finding the correct solution.

3. How effective is our approach based on the full solution (including AC-3)? Can we solve more problems? How would we characterize change in runtime?
Our approach is much more effective combining AC-3 with backtracking than using pure backtracking alone. AC-3 helps reduce the search space by enforcing constraints before backtracking begins, eliminating impossible values early and preventing unnecessary recursive calls. This makes our algorithm particularly effective for easy and medium puzzles. For harder puzzles, AC-3 cannot guarantee a solution on its own, it reduces the number of choices per cell, leading to fewer dead ends. Compared to pure backtracking, the runtime improves because

backtracking now operates on a smaller, pruned search space, effectively lowering the depth and breadth of the recursive tree. In terms of complexity, AC-3 runs in $O(81*9^2)= O(1)$, since the board size is fixed, while backtracking worst-case remains $O(9^n)$, but with a smaller n after AC-3 filtering. This leads to a substantial speedup! While our approach does not solve all extreme cases, it makes a wider range of problems solvable more efficiently.

4. How did we work as a team?
We first went over the project guidelines and planned a general coding structure (pseudocode) for what the program should do based on the Sudoku game. One of us focused on implementing the AC-3 algorithm, ensuring that we could reduce the search space before applying backtracking. Meanwhile, the other worked on optimizing the backtracking solver, incorporating heuristics like selecting the cell with the fewest possible values to improve efficiency. We were able to do the entire process of the project from implementation to testing together since our schedules align. The work was evenly distributed between the two of us.