Summary of Solver class

The solver works by using some standard techniques for solving sudoku (see update methods below). If the solver stops making progress using these techniques, it makes a guess for the value of an empty entry of the board, and continues with the standard techniques until it either stops making progress again (in which case another empty entry is "guessed"), the board becomes invalid (in which case it returns to the previous guess and changes it), or the board becomes solved.

Instance variables:

- **n** : The side length of a block of the sudoku (standard sudoku have n = 3).
- **n2** : The side length of the whole sudoku board. Therefore, n2 = n^2 (standard sudoku have n2 = 9).
- **int[][] board** : An n2 × n2 matrix storing the entries of the sudoku board (an empty entry is recorded as -1).
- **Set⟨Integer⟩[] row** : row[i] stores the set of integers currently appearing as entries in row i.
- **Set⟨Integer⟩[] col** : Similar to row, but for columns.
- **Set⟨Integer⟩[][] block** : block[i][j] stores the set of integers currently appearing as entries in block (i, j) (we view the blocks as forming an n × n matrix).
- **Set⟨Integer⟩[][] candidates** : candidates[i][j] stores the set of possible values for board[i][j].
- **Set⟨Integer⟩[][] rowPoss** : rowPoss[i][k] stores the possible locations of k + 1 in row i. More precisely, it stores those j such that k + 1 is a possible value for board[i][j].
- **Set⟨Integer⟩[][] colPoss** : colPoss[j][k] stores those i such that k + 1 is a possible value for board[i][j].
- **Set⟨Integer⟩[][][] blockPossRow** : blockPossRow[i][j][k] stores those r such that k + 1 might appear in the portion of row r contained in block (i, j).
- **Set⟨Integer⟩[][][] blockPossCol** : Similarly for columns
- **int totalPoss** : Stores the sum of the sizes of all the sets in the matrices candidates, rowPoss, colPoss, blockPossRow, blockPossCol.
- **int unknowns** : Stores the number of currently empty entries on the board.

Important methods:

- **solve()** : The primary method, attempts to solve the sudoku. Returns true if sudoku was solved, false if sudoku was found to be invalid. It is based on the methods updateAllWhilePoss() and tryPossibilities(Solver solver).

- **updateAllWhilePoss()** : Repeatedly updates instance variables until either board is completely filled in (so sudoku is solved), or totalPoss stops changing, meaning that further updates will have no effect.

- **tryPossibilities(Solver previousSolver)** : Recursively tries filling in an unknown entry of board and running updateAllWhilePoss() until board is either solved or is found to be invalid.

- **updateBoardViaCandidates()** : If candidates[i][j] has been reduced to a singleton {k}, sets board[i][j] = k.

- **updateBoardViaRowColPoss()** : If rowPoss[i][k] has been reduced to a singleton {j}, sets board[i][j] = k + 1. Similarly for colPoss.

- **updateBoardViaBlockPoss()** : If blockPossRow[i][j][k] = {r} and blockPossCol[i][j][k] = {s}, sets board[r][s] = k + 1.

- **updateCandidatesViaRowColPoss()** : If the location (i, j) fails to be given as possibility for k + 1 by either rowPoss[i][k] or colPoss[j][k], removes k + 1 from candidates[i][j].

- **updateCandidatesViaBlockPoss()** : If location (i, j) fails to be given as possibility for k + 1 by either of the corresponding blockPossRow or blockPossCol sets, removes k + 1 from candidates[i][j].

- **updateRowColPossViaSelf()** : Checks for the following situation: there is a value k, a set A of size m, and a set R of m rows (more precisely indices of rows) such that for each $i \in R$, rowPoss[i][k] $\subseteq A$. In this situation, we may remove A from rowPoss[r][k] for each $r \notin R$. Similarly for columns.

- **updateBlockPossViaSelf()** : Similar to above, checks for the following situation: there are i, k, a set A of size m, and a set C of size m, such that for each $j \in C$, blockPossRow[i][j][k] $\subseteq A$. In this situation, we may remove A from blockPossRow[i][s][k] for each $s \notin C$.