**Algorithms Used:**

**1) Generator**

The ClassicPuzzleGenerator and IcePuzzleGenerator classes are responsible for generating Sudoku puzzles.

* **Classic Puzzle Generation:**
  + **Backtracking Algorithm (for generateCompleteSudoku and solveSudoku):** This is the core algorithm for creating a valid, complete Sudoku board.
    - It starts with an empty board and tries to place numbers from 1 to 9 into each empty cell.
    - For each cell, it shuffles the numbers 1-9 and attempts to place them.
    - If a number is valid (doesn't violate Sudoku rules in its row, column, or 3x3 subgrid), it places the number and recursively calls itself for the next cell.
    - If the recursive call returns true (meaning a solution was found for the rest of the board), it keeps the number.
    - If the recursive call returns false (meaning no solution could be found with the current number), it backtracks, clears the cell, and tries the next number.
    - This continues until the board is filled or all possibilities have been exhausted for a given cell.
  + **Backtracking with Solution Counting (for hasUniqueSolution and countSolutions):** To ensure the generated puzzle has a unique solution, the generator uses a modified backtracking algorithm.
    - It attempts to solve the puzzle, and instead of stopping at the first solution, it continues to explore all possible paths to count how many valid solutions exist.
    - If countSolutions returns anything other than 1, the number previously removed is restored, and another cell is attempted for removal.
* **Ice Puzzle Generation:**
  + The IcePuzzleGenerator leverages the ClassicPuzzleGenerator to first create a standard Sudoku puzzle and its solution.
  + **Heuristic/Rule-based Cell Freezing:** It then identifies "frozen" cells based on a specific rule: a cell can be frozen only if it is empty and has an adjacent empty, fillable cell in the 4 cardinal directions within its subgrid. It randomly selects a certain percentage of these candidate cells to be frozen. This isn't a complex algorithm like backtracking, but rather a set of rules for selecting cells.

**2) Validator**

The StandardSudokuValidator class implements the SudokuValidator interface.

* **Constraint Checking (for isValid):** This method checks if placing a given num at a specific row and col is valid according to standard Sudoku rules.
  + It iterates through the entire row to ensure num is not already present.
  + It iterates through the entire col to ensure num is not already present.
  + It iterates through the relevant 3x3 subgrid to ensure num is not already present.
  + This is a direct application of the Sudoku rules as constraints.
* **Full Board Validation (for isValidSudoku):** This method iterates through every cell on the board. For each non-empty cell, it temporarily removes the cell's value and then uses the isValid method to check if that value could be legally placed there considering all other *existing* numbers. This effectively validates if the current state of the board adheres to Sudoku rules.

**3) Mode**

The ClassicMode and IceMode classes define the game logic and rendering specific to each mode.

* **Game Logic and UI Rendering:** These classes don't employ complex algorithms in the same way the generator and validator do. Their "algorithms" are more about defining the rules of interaction and how the UI behaves.
  + **renderCell:** This method determines the visual appearance and editability of each cell based on whether it's a fixed puzzle cell, an empty editable cell, or (in Ice Mode) a frozen cell.
  + **isValidMove:** Both modes use the SudokuValidator's isValid method to check if a user's entered number is locally valid (row, column, subgrid).
  + **isPuzzleComplete:** Checks if all cells are filled and if the completed board is a valid Sudoku using validator.isValidSudoku.
  + **getHintCell:** Randomly selects an empty, non-fixed cell to provide a hint. This involves iterating through cells and using a random number generator.
  + **checkSolution:** Compares the user's current board state with the generated solution to count errors.
  + **updateFrozenCellState (Ice Mode specific):** This method applies the specific rule for "unfreezing" cells in Ice Mode. It checks if a frozen cell has an adjacent (4-direction) user-filled cell that is *correct* according to the solution. If so, it unfreezes the cell. This is a specific rule implementation.

**Data Structures and Algorithms You Need to Know for Presentation in DSA Class:**

Based on this code, here's what you should focus on for a DSA class presentation:

**Data Structures:**

1. **2D Array (or Matrix):**
   * **Usage:** The Board class internally uses a int[][] grid to represent the Sudoku board. This is fundamental to storing the numbers at their respective row and column positions.
   * **Why it's important:** Efficient for direct access to cells using [row][col] indexing, crucial for checking rules.
   * **In your presentation:** Explain how a 2D array maps to the Sudoku grid, and how element access works (O(1)).
2. **Boolean 2D Arrays:**
   * **Usage:** fixedCells[][] in Puzzle to mark cells that are part of the initial puzzle and cannot be changed by the user. frozenCells[][] in Puzzle (specific to Ice Mode) to mark cells that are currently uneditable due to the game's rules.
   * **Why it's important:** Efficiently tracks cell states (fixed, frozen) with minimal memory overhead.
3. **List (e.g., ArrayList):**
   * **Usage:**
     + List<Integer> numbers in ClassicPuzzleGenerator to shuffle numbers for backtracking.
     + List<Point> positions to store cell coordinates for random removal during puzzle creation.
     + List<Point> emptyCells for hint generation.
     + List<Point> candidateFrozenCells in IcePuzzleGenerator.
   * **Why it's important:** Dynamic arrays that allow adding/removing elements, useful for shuffling and collecting coordinates.
   * **In your presentation:** Briefly discuss ArrayList's dynamic nature and average O(1) append, O(n) removal/insertion in the middle.
4. **Point Class (Simple Object):**
   * **Usage:** Used to encapsulate row and column coordinates (x and y).
   * **Why it's important:** Makes it easier to pass around cell locations as a single object.

**Algorithms:**

1. **Backtracking:**
   * **Concept:** A general algorithmic technique for solving problems recursively by trying to build a solution incrementally, one step at a time, and "undoing" (backtracking) if a solution cannot be completed with the current partial solution.
   * **Application in Sudoku:**
     + **Puzzle Generation:** The most prominent use is in ClassicPuzzleGenerator.solveSudoku to fill a complete valid Sudoku board and countSolutions to check for unique solutions. You try a number, if it's valid, you move to the next cell. If not, or if the path leads to a dead end, you "backtrack" and try a different number for the current cell.
   * **Diagram:** A recursive tree diagram showing choices and backtracks would be excellent.
   * **Time Complexity:** Generally exponential for Sudoku (O(9N2) in the worst case, but heavily pruned by constraints).
   * **Key points for presentation:**
     + Recursive calls.
     + Base cases (board filled).
     + Making a choice (placing a number).
     + Checking constraints (using the validator).
     + Undoing the choice (setting cell back to 0).
2. **Constraint Satisfaction:**
   * **Concept:** Finding values for variables such that they satisfy a given set of constraints.
   * **Application in Sudoku:** The isValid method within StandardSudokuValidator directly implements constraint satisfaction. It checks if placing a number satisfies the row, column, and subgrid constraints.
   * **Key points for presentation:** Explain the three core rules of Sudoku and how isValid checks them by iterating through relevant cells.
3. **Randomization/Shuffling:**
   * **Concept:** Using randomness to achieve a desired outcome.
   * **Application in Sudoku:**
     + Collections.shuffle(numbers, random): Used when filling cells during puzzle generation to try numbers in a random order, leading to different valid Sudoku boards.
     + Collections.shuffle(positions, random): Used to randomly select cells to remove when creating the puzzle from a solved board.
     + random.nextInt(): Used for selecting hint cells.
   * **Key points for presentation:** Discuss how shuffling (e.g., Fisher-Yates shuffle used by Collections.shuffle) helps in generating diverse puzzles and provides random hints.