Knight's Tour

The Knight's Tour is a classic chess puzzle. The challenge: starting from any square on a chessboard, can a knight visit every square exactly once?

A knight moves in an "L" shape—two squares in one direction and one square perpendicular to that (or vice versa). From a square in the middle of the board, a knight has up to 8 possible moves. Near the edges or corners, that number shrinks.

A *complete* tour visits all 64 squares. If the knight gets stuck with unvisited squares remaining, the tour is incomplete.

Part 1: Random Walk

Create classes to model the board and squares, then implement a random walk algorithm.

Class Square

Represents a position on the board. Should store the row and column of the square.

Planning: The Square Class

List the fields and methods you need and describe them?

Questions to consider:

- Should the fields be public or private? Do you need accessor methods?
- What format should `toString()` return? Something like "(3, 5)" or "Square at row 3, col 5"?

Class Board

Manages the state of the chessboard. Use a 2D array of booleans where `true` means the square is available (not yet visited) and `false` means it has been visited. The Board should:

- Initialize all squares as available
- Track how many open (unvisited) squares remain
- Provide methods to mark a square as visited and to check whether a square is available

Planning: The Board Class

List the fields and methods needed. Your constructor should create a board of any size. Describe the fields and methods and how/when they are used

Questions to consider:

- What's the best way to keep `openSquares` accurate? Recount every time, or update incrementally?
- Should `fillSquare()` accept row/col as separate parameters, a Square object, or both (overloaded)?
- Will you ever need to "unfill" a square (mark it available again)? Might that be useful later?

Finding Valid Moves

Write a method that takes a Board and a Square and returns an ArrayList of all valid moves from that position. A move is valid if it follows the knight's L-shaped pattern, stays within the board boundaries, and lands on an unvisited square.

Questions to consider:

- How will you represent the 8 possible offsets? Parallel arrays? An array of pairs?
- Should this method be part of the Board class, or a separate helper method?

How Will You Test validMoves?

This method is critical—if it's wrong, nothing else will work. Describe how you'll verify it returns the correct moves.

Testing plan: (Hint: try a corner, an edge, and the center. Count expected moves by hand first.)

Random Walk Algorithm

Starting from square (0, 0):

- Mark the starting square as visited
- Find all valid moves from the current position
- Randomly select one of the valid moves
- Repeat until no valid moves remain

Deliverable

Run 100 random walk trials on an 8×8 board and report the average number of squares remaining. Most random walks will not complete the tour.

AVERAGE NUMBER OF REMAINING SQUARES _____

Part 2: Warnsdorff's Heuristic

Random selection is inefficient. Warnsdorff's rule, published in 1823, provides a simple heuristic that dramatically improves success rates.

The Heuristic

Instead of choosing randomly among valid moves, always move to the square that has the *fewest* onward moves. In other words, look ahead one step: for each candidate square, count how many valid moves would be available from that square, then choose the square with the smallest count.

The intuition is that squares with fewer onward moves are harder to reach later, so you should visit them while you still can. Corners and edges have limited accessibility—if you don't visit them early, you may not be able to reach them at all.

Implementation

Write a method that examines each valid move, temporarily considers what moves would be available from that position, and returns the move with the minimum count. Be careful not to permanently modify the board state while evaluating candidates.

Planning: The getBestMove Method

Questions to consider:

- How do you "temporarily" fill a square to count onward moves without messing up the real board?
- What if two squares tie for the fewest onward moves? Does it matter which you pick?

How Will You Test the Heuristic?

Describe how you'll verify the heuristic is choosing correctly and improving results.

Testing plan:

Final Testing

Run your heuristic-based algorithm on an 8×8 board. A correct implementation should complete the tour (0 squares remaining) on most or all attempts. Compare your results to the random walk—the difference should be dramatic.

QUESTION: Which (if any) starting squares on an 8x8 board do NOT complete a full tour?

Challenge (Optional)

Test your implementation on different board sizes and starting positions. Are there configurations where it fails?