

Some of the Algorithms implemented in chess

Core Algorithm: Chess Game Loop

The heart of this chess engine is a robust, turn-based game loop that handles player input, move validation, and game state management. It ensures rules like check, checkmate, pinning, castling, and en passant are enforced while providing a smooth user experience.

The loop prompts players for moves, validates them against chess rules (including special cases like pins and check resolution), updates the board, and alternates turns until checkmate or resignation.

High-Level Overview

The algorithm runs in a repeating cycle per turn, with early exits for game-ending conditions. It prioritizes safety: Invalid moves are reverted, and the board state is never left in an inconsistent state.

Key Principles:

- **Input Sanitization:** All user inputs (piece selection and target square) are validated early.
- **Rule Enforcement:** Uses a dedicated **ValidateMoves** class for complex checks.
- **State Tracking:** Maintains flags for castling rights, king positions, and previous moves (for en passant).
- **Error Handling:** Illegal moves are rolled back with user-friendly feedback.

Step-by-Step Breakdown

Here's the process in plain English, broken into phases for easy following:

1. **Game Initialization:**

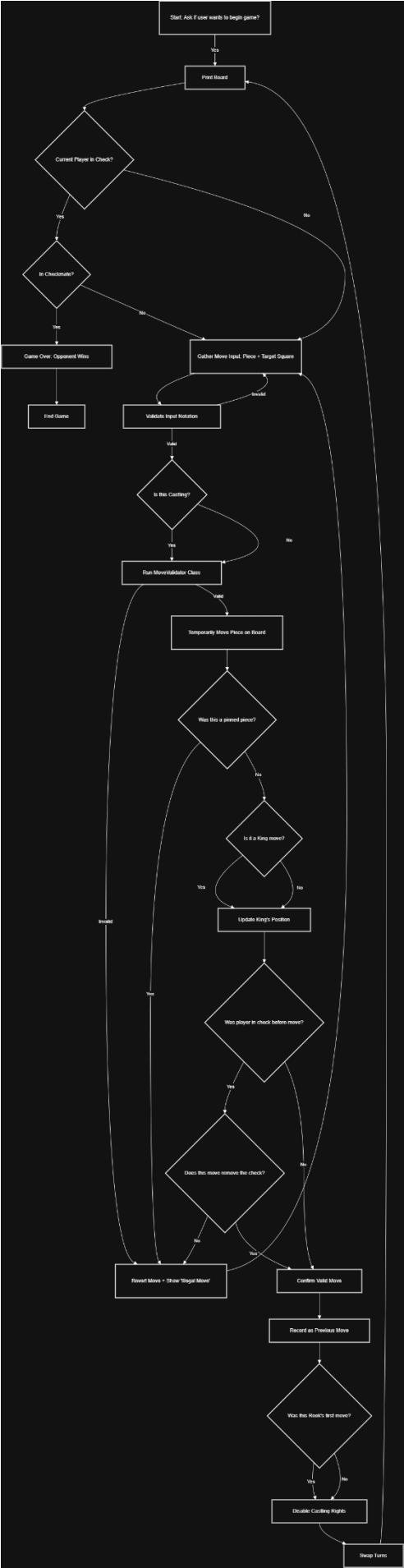
- a. Prompt the user: "Do you want to start the game ? (yes/no) :"
 - b. If yes, initialize a fresh board (8x8 grid with standard starting pieces) and set the current player to White.
- 2. **Display Board:**
 - a. Print the current board state (Using Board -> .print()).
- 3. **Check Game-Ending Conditions:**
 - a. Is the current player in **check**? (Opponent's pieces threaten the king.)
 - i. If yes:
 - 1. Is it **checkmate**? (No legal moves to escape check.)
 - a. If yes: End the game. Declare the opponent the winner and print "Checkmate! [Player] wins."
 - ii. If no checkmate, proceed (player must resolve the check).
- 4. **Gather Player Input:**
 - a. Prompt: "Select the piece to move."
 - i. Validate: Ensure it's the current player's piece, exists, and is his own piece .
 - b. Prompt: "Where to move it? (e.g., e4)."
 - i. Validate: Target square is empty or occupied by opponent; notation is valid.
- 5. **Pre-Move Checks:**
 - a. Is this a **castling** attempt? (King moves two squares toward a rook.)
 - i. Verify prerequisites: Neither king nor rook has moved; no pieces in between; king not in check/passing through check.
- 6. **Full Move Validation:**
 - a. Pass the move to the ValidateMoves class.
 - i. Checks: Piece-specific rules (e.g., knight jumps, bishop diagonals), en passant (if applicable from previous move).
- 7. **Execute the Move:**
 - a. Temporarily apply the move to the board.

- b. If it's a **pinned piece** (movement would expose king):
 - i. Revert to the board and output: "Illegal move: Piece is pinned and cannot move."
 - c. If it's a **king move**:
 - i. Update the king's position in the state tracker.
- 8. **Post-Move Safety Check:**
 - a. If the player was in check before the move:
 - i. Simulate opponent attacks: Does the new position still leave the king in check?
 - ii. If yes: Revert and output: "Illegal move: Does not resolve check."
 - b. Otherwise: Confirm that the move is valid.
- 9. **Update Game State:**
 - a. If valid:
 - i. Finalize the move (capture opponent pieces if needed).
 - ii. Record this as the "previous move" (for en passant in the next turn).
 - iii. If a rook moved for the first time: Disable castling rights for that side.
 - b. Swap turns (White ↔ Black).
- 10. **Loop Back:**
 - a. Repeat from Step 2 until checkmate, stalemate, or user quits.

Edge Cases Handled

- **Invalid Inputs:** Retry prompts with hints (e.g., "Invalid square—try algebraic notation like a1-h8.").

Diagram :



Core Algorithm: Checkmate Detection

The `isCheckmate` function determines if the king of a given color is in **checkmate**—meaning it's in check **and** has no legal moves to escape (king can't flee, can't capture the attacker(s), and no interpositions/blocking for sliding attacks). This is a critical validator called at the start of each turn.

TL;DR: Scans for checking pieces. If none → not checkmate. If multiple → checkmate if king trapped. For single checker: Non-sliders (pawn/knight) → checkmate if king can't escape/capture. Sliders (rook/bishop/queen) → checkmate if no blocks possible, king can't capture, and king trapped. Returns true only on full entrapment.

High-Level Overview

- **Preconditions:** Validates the input king position belongs to the specified color.
- **Key Helpers:**
 - `getCheckingPieces`: Scans entire board for enemy pieces attacking the king.
 - `validKingMoves`: Computes king's safe adjacent moves (avoids enemy attacks, own pieces).
 - `piecesToComeBetween`: For sliders, simulates if **any** own piece can legally block the attack path without exposing king.
- **Efficiency:** $O(\text{board size}) = O(64)$ scans; simulations in `piecesToComeBetween` are $O(n \text{ pieces} * \text{attacks})$.
- **Assumptions:** Relies on precomputed `AttackingInfo` (all attack maps). Single-checker focus for sliders (multi-check rare/hard).

Edge Cases Handled:

- No check → false.
- Double+ check → true if king trapped (can't escape multi-threats).
- King can capture attacker → false (via `validKingMoves` or own attacks).

Step-by-Step Breakdown

1. **Validate King:**
 - a. Ensure position has the player's king. Else, log error → false.
2. **Gather Data:**
 - a. `checkingPositions = getCheckingPieces(...)`: Enemy pieces hitting king.
 - b. `kingValidPositions = validKingMoves(...)`: Safe king escapes/captures.
 - c. `currentAttackings`: Own-side attack map (to check if can capture checker).
3. **Early Exit: No Check:**
 - a. If `checkingPositions.empty()` → false (not even in check).
4. **Multi-Check (≥ 2 attackers):**
 - a. King can't escape multi-threats → true if no `kingValidPositions`.
5. **Single Checker Handling:**
 - a. Get `checkFrom = checkingPositions[0]`.
 - b. **Non-Sliders (Pawn/Knight):**
 - i. Pawn: Immediate true if king is trapped.
 - ii. Knight: true if king trapped (`kingValidPositions.empty()`) **and** can't capture (`!currentAttackings.contains(checkFrom)`).
 - c. **Sliders (Rook/Bishop/Queen):**
 - i. Compute attack path (rook-path, bishop-path, or queen-decided).
 - ii. `piecesToComeBetween(...)`: Any own piece block path **safely** (simulate move → recheck king safe)?
 - iii. true if: No blockers **and** can't capture checker **and** king trapped.
6. **Return:** false unless all escape routes sealed.

- Diagram

