Analysis of Algorithms

Project Documentation: Chessboard Game

AKRASH BASHIR	**47571**
MUHAMMAD ARHUM	**44889**
MUHAMMAD AKBAR	**44901**

Project Overview

The Chessboard Game is an interactive React-based chess game built as a university course project. It provides a fully functional chessboard where players can move pieces according to chess rules, with valid moves highlighted for the selected piece. The game enforces turn-taking between white and black players and validates moves dynamically.

Features

- Dynamic Chessboard Rendering: Displays an 8x8 chessboard with Unicode symbols representing chess pieces.
- Piece Movement: Implements rules for valid moves for pawns, rooks, knights, bishops, queens, and kings.
- Turn Management: Alternates turns between white and black players.
- Move Validation: Highlights valid moves for the selected piece.
- Responsive UI: Built with Page.CSS for a visually appealing and responsive design.

Technical Details:

- Framework: React.js

- CSS: Page.CSS and custom styling.
- State Management: React's useState hook for managing board state, selected pieces, valid moves, and turn toggling.

Code Analysis:

1. **CalculateValidMoves Function**

This function calculates all valid moves for a selected piece based on its type and position.

```
const calculateValidMoves = (piece, row, col) => {
 const moves = [];
 const isWhite = piece === piece.toUpperCase();

 const isValid = (r, c) =>
  r >= 0 &&
  r < 8 &&
  c >= 0 &&
  c < 8 &&
  (board[r][c] === "." ||
   (isWhite
    ? board[r][c] === board[r][c].toLowerCase()
    : board[r][c] === board[r][c].toUpperCase()));
```

```
  switch (piece.toLowerCase()) {
    case "p": {
      const direction = isWhite ? -1 : 1;
      const startRow = isWhite ? 6 : 1;

      // Single step forward
      if (isValid(row + direction, col) && board[row + direction][col] ===
".") {
        moves.push([row + direction, col]);
        if (
          row === startRow &&
          isValid(row + 2 * direction, col) &&
          board[row + 2 * direction][col] === "."
        ) {
          moves.push([row + 2 * direction, col]);
        }
      }

      // Capture diagonally
      if (
        col > 0 &&
        isValid(row + direction, col - 1) &&
```

```
    board[row + direction][col - 1] !== "."
  ) {
    moves.push([row + direction, col - 1]);
  }
  if (
    col < 7 &&
    isValid(row + direction, col + 1) &&
    board[row + direction][col + 1] !== "."
  ) {
    moves.push([row + direction, col + 1]);
  }
  break;
}

case "r": {
  for (let i = row - 1; i >= 0; i--) {
    if (isValid(i, col)) moves.push([i, col]);
    if (board[i][col] !== ".") break;
  }
  for (let i = row + 1; i < 8; i++) {
    if (isValid(i, col)) moves.push([i, col]);
    if (board[i][col] !== ".") break;
```

```
  }
  for (let i = col - 1; i >= 0; i--) {
    if (isValid(row, i)) moves.push([row, i]);
    if (board[row][i] !== ".") break;
  }
  for (let i = col + 1; i < 8; i++) {
    if (isValid(row, i)) moves.push([row, i]);
    if (board[row][i] !== ".") break;
  }
  break;
}

case "n": {
  const knightMoves = [
    [row + 2, col + 1],
    [row + 2, col - 1],
    [row - 2, col + 1],
    [row - 2, col - 1],
    [row + 1, col + 2],
    [row + 1, col - 2],
    [row - 1, col + 2],
    [row - 1, col - 2],
```

```
  ];
  knightMoves.forEach(([r, c]) => isValid(r, c) && moves.push([r, c]));
  break;
}

case "b": {
  for (let i = 1; i < 8; i++) {
    if (!isValid(row + i, col + i)) break;
    moves.push([row + i, col + i]);
    if (board[row + i][col + i] !== ".") break;
  }
  for (let i = 1; i < 8; i++) {
    if (!isValid(row - i, col + i)) break;
    moves.push([row - i, col + i]);
    if (board[row - i][col + i] !== ".") break;
  }
  for (let i = 1; i < 8; i++) {
    if (!isValid(row + i, col - i)) break;
    moves.push([row + i, col - i]);
    if (board[row + i][col - i] !== ".") break;
  }
  for (let i = 1; i < 8; i++) {
```

```
    if (!isValid(row - i, col - i)) break;
    moves.push([row - i, col - i]);
    if (board[row - i][col - i] !== ".") break;
  }
  break;
}

case "q": {
  moves.push(
    ...calculateValidMoves("r", row, col),
    ...calculateValidMoves("b", row, col)
  );
  break;
}

case "k": {
  const kingMoves = [
    [row + 1, col],
    [row - 1, col],
    [row, col + 1],
    [row, col - 1],
    [row + 1, col + 1],
```

```
      [row + 1, col - 1],
      [row - 1, col + 1],
      [row - 1, col - 1],
    ];
    kingMoves.forEach(([r, c]) => isValid(r, c) && moves.push([r, c]));
    break;
  }

  default:
    break;
 }
 return moves;
};
```

- **Input:** piece (the type of chess piece), row (current row index), col (current column index)
- **Output:** Array of valid moves for the piece.

Time Complexity:

- **Pawns (p):**

- **Worst-case: O(1)** for basic validation and up to O(2) for diagonal captures and double forward moves.

- **Complexity: O(1).**
- **Rooks (r):**

Moves are calculated along rows and columns.

- **Worst-case**: Iterate over all cells in a row or column.
- **Complexity:** O(7) (linear to the board size in the worst direction).
- **Knights (n):**

Up to 8 potential moves checked.

- **Complexity: O(1).**
- **Bishops (b):**
- Moves are calculated diagonally.
- **Worst-case:** Covers both diagonals.
- **Complexity: O(7).**
- **Queens (q):**
- Combines rook and bishop moves.
- **Complexity:** O(14).
- **Kings (k):**
- Maximum of 8 possible moves checked.
- **Complexity: O(1).**

Space Complexity:

- Stores a list of moves (up to 27 for queens).
- **Complexity:** O(m), where m is the maximum number of valid moves.

2. handleSquareClick Function

Handles clicks on a chessboard square, managing piece selection, move validation, and state updates.

```
const handleSquareClick = (row, col) => {
  const piece = board[row][col];
```

```
 if (selectedSquare) {
  const [prevRow, prevCol] = selectedSquare;
  const selectedPiece = board[prevRow][prevCol];

  const isSameColor =
   piece !== "." &&
   ((selectedPiece === selectedPiece.toUpperCase() &&
    piece === piece.toUpperCase()) ||
    (selectedPiece === selectedPiece.toLowerCase() &&
     piece === piece.toLowerCase()));

  if (!isSameColor && validMoves.some(([r, c]) => r === row && c ===
col)) {
   const newBoard = board.map((r, i) =>
    r.map((p, j) => {
     if (i === row && j === col) return selectedPiece;
     if (i === prevRow && j === prevCol) return ".";
     return p;
    })
   );
   setBoard(newBoard);
```

```
      setSelectedSquare(null);
      setValidMoves([]);
      setTurn(turn === "white" ? "black" : "white");
    } else {
      setSelectedSquare(null);
      setValidMoves([]);
    }
  } else if (
    piece !== "." &&
    ((turn === "white" && piece === piece.toUpperCase()) ||
      (turn === "black" && piece === piece.toLowerCase()))
  ) {
    setSelectedSquare([row, col]);
    setValidMoves(calculateValidMoves(piece, row, col));
  }
};
```

- **Input:** row (row index), col (column index).

Steps:

1. Validates selection (if it is the player's turn).
2. Updates board state if a valid move is clicked.
3. Resets selection and toggles turns.

Time Complexity:

- Calls calculateValidMoves for selected pieces:
- **Worst-case complexity:** O(14) (queen's complexity).
- Updates the board state using nested mapping:
- **Complexity:** O(64) (fixed size of 8x8 board).
- **Overall:** O(64 + 14) ≈ O(64).

Space Complexity:

- Space for valid moves and temporary state updates.
- Complexity: O(m) (maximum valid moves).

3. Rendering Chessboard

The chessboard is rendered dynamically using nested .map calls.

```
<div className="board">
  {board.map((row, rowIndex) =>
    row.map((piece, colIndex) => (
      <div
        key={`${rowIndex}-${colIndex}`}
        onClick={() => handleSquareClick(rowIndex, colIndex)}
        className={`square ${
          selectedSquare &&
          selectedSquare[0] === rowIndex &&
          selectedSquare[1] === colIndex
            ? "selected"
            : ""
```

```
      } ${
        validMoves.some(([r, c]) => r === rowIndex && c === colIndex)
          ? "valid-move"
          : (rowIndex + colIndex) % 2 === 0
          ? "bg-slate-300"
          : "bg-[#358cd8]"
      }`}
    >
      {pieceSymbols[piece]}
    </div>
  ))
 ))}
</div>
```

Time Complexity:

- **Outer loop**: Iterates over 8 rows.
- **Inner loop:** Iterates over 8 columns.
- **Complexity:** O(64).

Space Complexity:

- Space for the JSX elements stored in memory.
- Complexity: O(64).
- Overall Complexity

Time Complexity

- The most computationally expensive operations occur during:

- Valid move calculation (O(14)).
- Board rendering (O(64)).
- State updates (O(64)).
- **Total:** O(64) (board size dominates).

Space Complexity

- Stores a constant 8x8 board state, valid moves, and temporary selections.
- **Total:** O(64) (board state) + O(m) (valid moves).
- Effective complexity: O(64) (as m ≤ 27).

Conclusion

The Chessboard Game is a computationally efficient implementation with consistent time and space complexities. The game's modular structure ensures smooth performance, even with multiple state updates during gameplay.

Thank You!