Functional Programming: Reversi Game

> - (20 points) Console (or other user) interface.
> - (20 points) Implement game rules, including valid moves and flipping discs.
> - (10 points) Support for two human players.
> - (10 points) Allow different board sizes for varied gameplay.
> - (10 points) Add undo and redo functionality.
> - (15 points) Error reporting for invalid moves.
> - (15 points) Unit and property-based tests.

What the task was

The task was to implement a command-line version of the game Reversi (also known as Othello) in Haskell. The goal was to create a functional game that allows two human players to interact with a board, making valid moves, flipping discs, and alternating turns. The game should also support features such as different board sizes, error reporting for invalid moves, and undo/redo functionality.

However, the undo/redo functionality was not completed, although the project already has a ready architecture for easy implementation.

The architecture of your solution

The solution is structured across four main modules:

- `Main.hs`: The application's entry point handles the game flow and calls other modules to manage the game logic.
- `ConsoleUI.hs`: This module renders the board to the console and parses user inputs. It handles user interaction, such as getting the move from the player and processing the input.
- `GameLogic.hs`: This contains the core rules and logic for the game, including move validation, flipping discs, and switching players.
- `Board.hs`: This module represents the game board and defines the data types for the board and the game state.

  Game State and Board Representation:

  - `GameState`: The `GameState` data structure holds the current board, the current player, and a history of previous boards to implement undo/redo functionality.
  - `Board`: The board is a 2D grid of `Maybe Disc` values, where `Disc` is either `Black` or `White`, representing the two players. Each cell can either be Nothing (empty) or occupied by one of the players.

Why certain architecture decisions were done

- Modular Design: The architecture is designed to be modular, with a clear separation between the user interface (`ConsoleUI`), game logic (`GameLogic`), and board representation (`Board`). This allows for easier testing, debugging, and future expansions (such as adding a graphical user interface or network play).
- History for Undo/Redo: The history field in the `GameState` allows us to implement undo and redo functionality by storing the previous board states. This structure was chosen because it efficiently supports this functionality without overcomplicating the state management.
- Functional Approach: Given that Haskell is a functional programming language, the solution makes use of immutability (e.g., the board and game state are immutable and each move results in a new game state). This aligns well with Haskell's strengths in handling immutable data and pure functions.

- Board Size as a Parameter: The choice to allow dynamic board sizes was made to provide flexibility for different game experiences. It enables the user to play on different grid sizes, enhancing replayability and accommodating various player preferences.

Why certain libraries were chosen

- `Text.Read` (for parsing input): The `readMaybe` function from `Text.Read` is used to safely parse integers from the user input. This helps to avoid exceptions due to invalid input and ensures the game can continue without crashes.
- `Hedgehog` (for property testing): The `Hedgehog` library is used for property-based testing to ensure that the game rules hold under a variety of conditions. Property-based testing is particularly useful for verifying that the game's logic, such as move validity and turn alternation, works correctly across a range of board sizes and states.
- `Tasty` (for unit testing): The `Tasty` testing framework is used for unit tests, which ensures that specific pieces of logic (such as board initialization, move validation, and flipping discs) work correctly. The integration with `Hedgehog` allows for a comprehensive testing suite that covers both edge cases and typical scenarios.
- `Stack` (build system): The Stack build system was used for managing dependencies and building the project. It was chosen because it is simple to use and was already being used in our homework assignments.

Investigation of the performance

While performance wasn't a primary focus for this implementation, it is important to consider the efficiency of the game logic, especially with larger board sizes. The following observations were made:

- Board Representation: The board is represented as a 2D list `[[Maybe Disc]]`. This allows for easy indexing and updating of positions, and I personally think it is quite efficient for large board sizes.
- Move Validation: The `isValidMove` function checks all eight possible directions from a given position. For each direction, the function recursively checks the board to see if any discs can be flipped. In the worst case, this could lead to excessive computation (e.g., for moves near the edges or corners of the board). To optimize this, memoization or caching techniques could be considered to avoid redundant calculations, especially when checking multiple possible moves in the same region. However, implementing such optimizations would be highly complex, time-consuming, and require a significant amount of additional code. Due to its advanced nature, I opted not to pursue this approach.
- Undo/Redo Functionality: Storing the history of board states for undo and redo operations is generally efficient for small to medium-sized boards. However, for very large games, the history could grow considerably, which might lead to memory usage concerns. One possible optimization could be **limiting the depth of the history** or implementing a more memory-efficient data structure.