Text File Analyzer

1. Task Overview

The task was to create a Haskell application that performs the following actions:

- **File Analysis:** Analyze a user-specified text file to compute various metrics, including:
  - Number of words
  - Number of lines
  - Number of characters
  - The most frequently occurring word
- **N-Gram Analysis:** Perform an n-gram analysis based on user input and display the results.
- **Word Cloud Representation:** Generate a word cloud representation of the text data by repeating words proportional to their frequency.
- **Testing:** Implement automated tests to verify the correctness of the program components and ensure robust performance.

2. Solution Architecture

The solution consists of three main modules: **Main.hs**, **Reader.hs**, and **Spec.hs**, each serving a specific purpose in the application:

2.1 Main.hs

- Handles user interaction (input/output).
- Reads the file, checks its existence, and processes its content.
- Orchestrates various tasks like word counting, line counting, frequent word detection, n-gram analysis, and word cloud representation.

2.2 Reader.hs

- Implements the core logic for text file analysis.
- Contains functions for word counting, line counting, and identifying the most frequent word.
- Handles n-gram generation and counting occurrences of n-grams.
- Includes utility functions for word cleaning and generating word clouds.

2.3 Spec.hs

- Implements tests using the **Tasty** and **Hedgehog** libraries.
- Verifies the correctness of core functionalities like word counting, line counting, character counting, frequent word detection, and word cloud generation.

3. Architectural Decisions

3.1 Modular Design

Why: The solution is divided into distinct modules for maintainability, readability, and ease of testing.
Benefit: By separating user interaction (*Main.hs*) from core logic (*Reader.hs*), the codebase is more flexible and reusable. Tests can focus solely on the Reader module without needing to simulate user input/output.

3.2 Lazy I/O

File reading (`readFile`) is done lazily in `processFileMaybe`.
Why: Lazy I/O minimizes memory usage for large files, as the file is read in chunks only as needed.

3.3 Frequent Word Calculation with Grouping and Sorting

Sorting and grouping were used to count word occurrences efficiently.
Why: This approach is simple and works well for medium-sized datasets.

3.4 N-Gram Helper Function

The `ngramHelper` function uses recursion to generate n-grams from the tokenized input.
Why: Recursive design is idiomatic in Haskell and ensures concise and clear implementation.

3.5 Property-Based Testing with Hedgehog

Why: Hedgehog was chosen for its ability to generate random test cases, ensuring robust validation of edge cases (e.g., empty strings, multiline content).
Benefit: Property-based tests automatically verify the correctness of functions against invariants like non-negative word counts.

0.1 3.6 Testing with unit tests

Why: Since it's hard to test Frequent word and cloud with property testing so unit tests can do the work by tackling different edge cases.

4. Library Choices

4.1 Base Libraries

- **System.Directory:** Used to check file existence with `doesFileExist`.
- **Data.List:** Provides list operations like `group`, `sort`, and `sortBy` for counting word occurrences and generating n-grams.
- **Text.Read:** Facilitates safe parsing of integers with `readMaybe`.

4.2 Tasty and Tasty.HUnit

Why: Tasty is a flexible testing framework that integrates well with Haskell projects. Tasty.HUnit allows for precise unit tests with clear assertions.
Benefit: The framework's modular structure enables organizing related tests into groups.

4.3 Hedgehog

Why: Hedgehog supports property-based testing, making it ideal for testing string-processing functions with randomly generated data.
Benefit: It ensures robustness by generating test cases that may not be manually considered.

4.4 Down (from Data.Ord)

Why: Used for sorting by descending order, simplifying the implementation of frequent word detection and n-gram ranking.

5. Performance Investigation

5.1 Time Complexity

- **Word Counting:** Tokenization with `words` is $O(n)$, where n is the string length. Overall complexity: $O(n)$.
- **Line Counting:** Splitting with `lines` is $O(n)$. Overall complexity: $O(n)$.

- **Character Counting:** Using `length` is $O(n)$. Overall complexity: $O(n)$.
- **Frequent Word Detection:** Sorting is $O(m \log m)$, where m is the number of words. Grouping and counting add $O(m)$. Overall complexity: $O(m \log m)$.
- **N-Gram Analysis:** Generating n-grams is $O(m \times n)$. Sorting and grouping n-grams is $O(k \log k)$, where k is the number of n-grams. Overall complexity: $O(m \times n + k \log k)$.

5.2 Memory Usage

- Lazy I/O minimizes memory usage by reading files incrementally.
- List-based operations (e.g., `words`, `lines`) can lead to higher memory usage for large datasets.