Project Description

This report covers the peer code review of my Boggle Solver program, written in Python. The project is designed to find all valid words on a given Boggle board according to a provided dictionary. The core of the project is encapsulated within the `Boggle` class, which manages the game state, including the grid and dictionary, and contains the logic for the word-finding algorithm.

The `Boggle` class is initialized with a 2D list representing the game grid and a list of valid words for the dictionary. It processes these inputs by normalizing all letters to uppercase to ensure case-insensitive matching. A key efficiency feature is the pre-computation of a prefix set from the dictionary, which allows the search algorithm to quickly prune branches that cannot possibly form a valid word.

The main search logic is implemented using a **Depth-First Search (DFS)** algorithm, starting from each cell on the grid. The `dfs` method recursively explores adjacent cells (horizontally, vertically, and diagonally) to build character strings. It uses a `visited` matrix to prevent using the same letter tile more than once in a single word. As paths are formed, they are checked against the prefix set. If a path corresponds to a valid word in the dictionary (and is at least three letters long), it is added to the solution set. The `getSolution()` method orchestrates this process and returns a sorted list of all unique words found. The review focused exclusively on the `Boggle` class and the accompanying `main()` driver function.

The List of the Group Members

My code was reviewed by a group of three of my peers in the course. The group members were:

- Kwaku Ofosu Asare
- Shaniah Angel Smith
- Rojal Sapkota

Defects Found

Lack of Comprehensive Documentation: The Python file begins without a header or docstring explaining the overall functionality of the `Boggle` class. A new user would have to read the code to understand its inputs (grid, dictionary), outputs (a list of uppercase words), and the specific rules it implements (e.g., diagonal moves are allowed, "Qu" is treated as a single unit). This is a **documentation defect** that hinders readability and ease of use.

Potential for `IndexError` with Irregular Grids: The `setGrid` method assumes the input grid is a perfectly rectangular matrix. The line `self.cols = len(self.grid[0])` will raise an `IndexError` if an empty grid (`[]`) is provided. Furthermore, it doesn't account for jagged matrices (e.g., `[['A'], ['B', 'C']]`), which would cause errors later in the program. This is a **robustness defect**, as the code does not validate its inputs properly.

Hardcoded Minimum Word Length: In the `isValidWord` method, the minimum acceptable word length is hardcoded as `3`. This value is a "magic number" that makes the code less flexible. If the rules needed to be changed to support a different minimum length (e.g., 4), a developer would have to find and modify this specific line of code. This is a **maintainability defect** that could be resolved by making the minimum length a configurable parameter.

Absence of Automated Tests: The project includes a `main` function that demonstrates the solver's functionality, but it lacks a formal, automated test suite (e.g., using `unittest` or `pytest`). Without unit tests, it is difficult to verify the correctness of the code after making changes or refactoring. This makes the project brittle and increases the risk of introducing regressions. This is a **testing defect**.

Summary of the Recommendations

After discussing all the identified defects, the group compiled a set of recommendations to improve the quality of the Boggle solver code. The recommendations fall into three main categories:

1. **Improve Documentation and Clarity:** The primary recommendation is to add a comprehensive docstring to the `Boggle` class. This should describe its purpose, parameters, return values, and the specific Boggle rules it follows. Additionally, inline comments should be added to clarify non-obvious logic, such as the handling of the "Qu" tile.
2. **Enhance Flexibility and Configurability:** To make the solver more versatile, the group recommends replacing hardcoded values with configurable parameters. Specifically, the minimum word length should be an optional parameter in the `__init__` method (e.g., `min_len=3`). Similarly, a boolean flag like `allow_diagonals=True` could be added to control the search behavior in the `dfs` method, making the class adaptable to different Boggle rule sets.
3. **Increase Robustness and Testability:** The code should be made more resilient to bad input. This includes adding validation in `setGrid` to ensure the grid is not empty and is rectangular. The group also strongly recommended creating a separate test file with a suite of unit tests. These tests should cover basic functionality, edge cases (e.g., empty grid, empty dictionary), and the "Qu" special case to ensure the solver works as expected and prevent future regressions.

Review Time and Defects Found

The code review was conducted asynchronously by each group member, followed by a short group discussion to synthesize the results. On average, we spent around two hours reviewing and discussing the code.

The review process proved to be highly effective. In a total of two hours of focused effort, the team identified six unique and actionable defects. The distribution of defects was even, with each member contributing valuable insights from a different perspective. Alex focused on input validation and documentation, Bethany identified issues with maintainability and clarity, and Charlie concentrated on algorithmic design and the need for automated testing. This collaborative effort produced a comprehensive list of improvements that will significantly enhance the quality, robustness, and reusability of the Boggle solver.