The project under review is a **Boggle Solver**, implemented in Python. The program is designed to find all valid words that can be formed from a given NxN Boggle grid based on a provided dictionary of words. The solver uses recursive depth-first search (DFS) and a Trie data structure to efficiently search for words and prefixes within the grid.

The project is organized into three primary classes:

1. `Node` **Class**
    - Represents a single node within the Trie data structure.
    - Each node stores a dictionary of child nodes (`children`) and a boolean flag (`is_end_of_word`) indicating whether the node marks the end of a valid word.
2. `Trie` **Class**
    - Implements the Trie (prefix tree) data structure to support fast word and prefix lookups.
    - **Key methods:**
        - `insert(word)`: Adds a word into the Trie by creating child nodes for each character.
        - `search(word)`: Checks whether a given word exists in the Trie.
        - `startsWith(prefix)`: Determines whether any words in the Trie start with a given prefix.
    - The Trie structure enhances performance by allowing efficient pruning of recursive searches when prefixes are invalid.
3. `Boggle` **Class**
    - Core class that performs the actual word search on the grid.
    - **Attributes:**
        - `grid`: 2D list of letters or letter combinations (e.g., “Qu”, “St”, “Ie”).
        - `dictionary`: List of valid words to search for.
        - `solutions`: List of valid words found in the grid.
    - **Key methods:**
        - `getSolution()`: Validates input, builds a Trie from the dictionary, and initiates DFS from each grid cell to find valid words.
        - `find_words()`: Recursively explores adjacent cells to form potential words, checking validity using the Trie.
        - `convert_to_lower()`: Normalizes all letters in the grid and dictionary to lowercase for case-insensitive matching.
        - `is_grid_valid()`: Ensures each grid cell contains a valid single letter or approved multi-letter combination using a regular expression.

The program’s **approach** is systematic and optimized for performance:

1. Validate and normalize inputs.
2. Construct a Trie from the dictionary for O(L) word and prefix lookups.

3. Perform recursive DFS from each grid cell, exploring all adjacent tiles (including diagonals).
4. Accumulate valid words of length three or more into a solution set.

Overall, this project demonstrates effective use of **object-oriented programming** principles and algorithmic problem-solving to implement a scalable and efficient Boggle word finder.

My group members were Giliad Dawite and Caleb Orr (no response).

Defects

1. **Data Mutation Risk**
   - The `convert_to_lower()` method modifies the original `grid` and `dictionary` objects directly.
2. **Minor Typographical Error**
   - Typo “alreadsy” instead of “already” (line 60).
3. **Redundant Lowercasing Operation**
   - The grid cells are lowercased again in line 63 even though they were already converted earlier.

Recommendations

1. **Documentation & Clarity**
   - Add a docstring to the `Boggle` class summarizing the algorithm and purpose.
   - Add docstrings to public methods and type hints to improve IDE support and readability.
2. **Code Organization**
   - Consider extracting `Node` and `Trie` into separate modules to simplify future testing and modularity.
   - Rename variables for clarity (`grid_size` instead of `N`, `current_word` instead of ambiguous names).
3. **Regex Precision**
   - Add anchors (`^...$`) to the regex pattern to make matching more precise.
4. **Testing & Validation Enhancements**
   - Include an expected output comment in the `main()` demo to help verify correctness quickly.

Review Time

- Giliad took two days reviewing my code and found three defects.