Final Project Report: Scrabble-Like Game

Class: CPSC-39

Student: Mohammad Nawid Wafa

Date: May 10, 2025

What My Game Does and Why It’s Useful or Entertaining

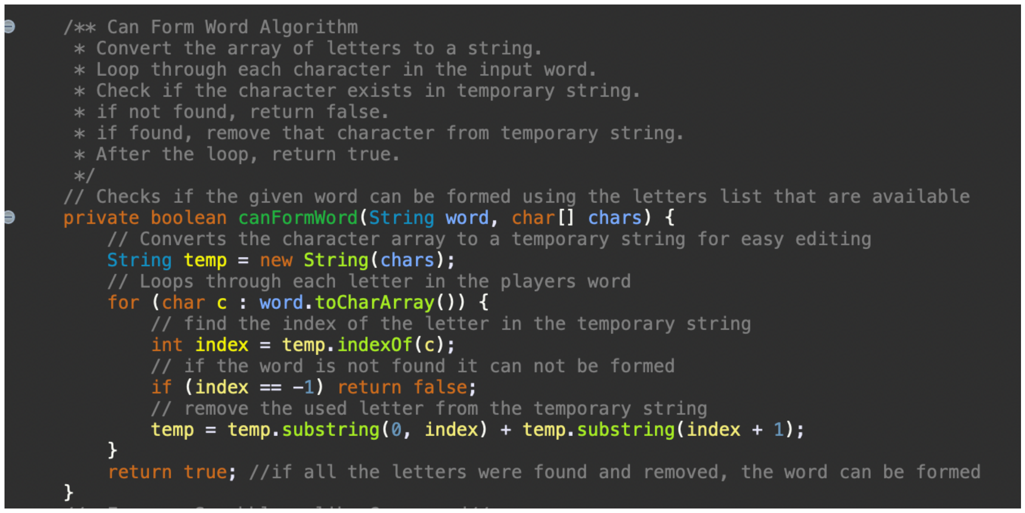
My final project is a two-player Scrabble-like word game built in Java. Players are given a random set of 5 letters that always includes at least one vowel, and they must form valid words using only those letters. The game checks whether a word is real (based on Collins Scrabble word list), whether it was already used, and whether it can be formed from the current letter set. Players receive points based on word length, and the game lasts for 6 rounds.

This game is entertaining because it challenges players’ vocabulary, spelling, and strategic thinking. It’s also educational, as it encourages word recognition and reinforces spelling rules. The game is easy to play in a terminal window and includes features like undo and automatic scoring, which make it smooth and user-friendly.

Algorithms

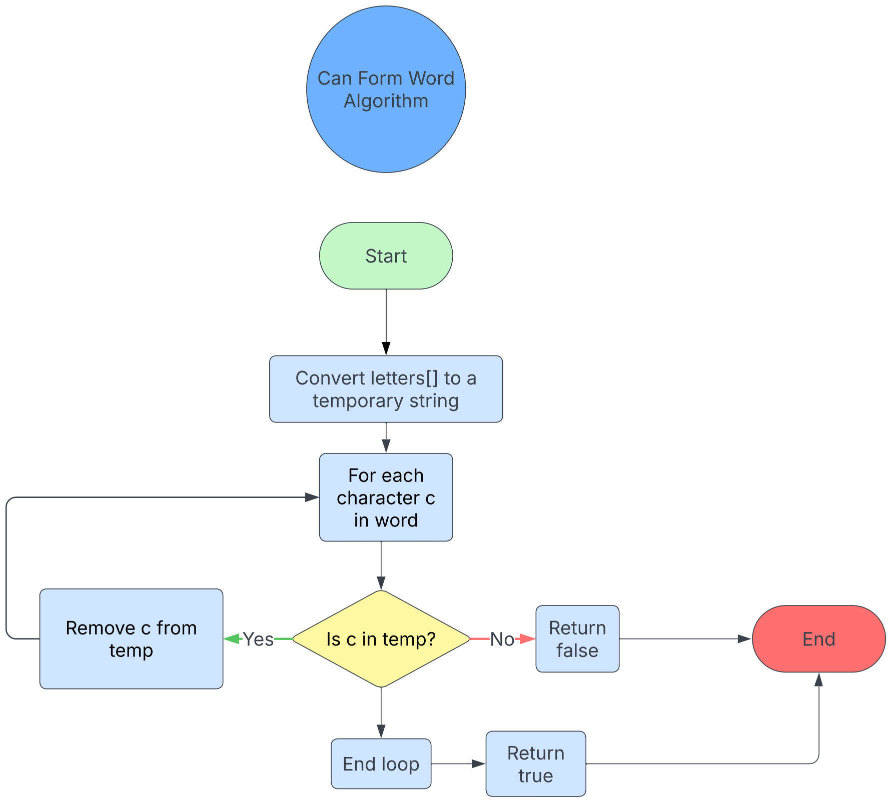
I included three original algorithms in my project. Each one is shown as a flowchart and clearly commented in my source code. These algorithms were developed through a mix of class lessons and independent problem-solving. I used ChatGPT for explanations, flowchart design, and brainstorming logic, but I personally wrote, modified, and tested all algorithms in my own code.

1. Can Form Word Algorithm



```java
    /** Can Form Word Algorithm
     * Convert the array of letters to a string.
     * Loop through each character in the input word.
     * Check if the character exists in temporary string.
     * if not found, return false.
     * if found, remove that character from temporary string.
     * After the loop, return true.
     */
    // Checks if the given word can be formed using the letters list that are available
    private boolean canFormWord(String word, char[] chars) {
        // Converts the character array to a temporary string for easy editing
        String temp = new String(chars);
        // Loops through each letter in the players word
        for (char c : word.toCharArray()) {
            // find the index of the letter in the temporary string
            int index = temp.indexOf(c);
            // if the word is not found it can not be formed
            if (index == -1) return false;
            // remove the used letter from the temporary string
            temp = temp.substring(0, index) + temp.substring(index + 1);
        }
        return true; //if all the letters were found and removed, the word can be formed
    }
```

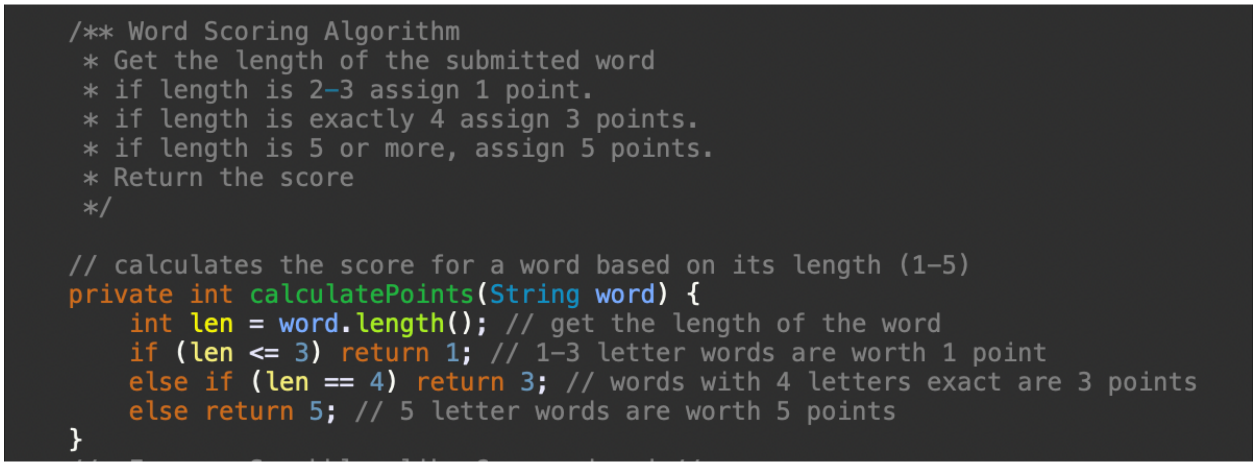
Flowchart



This algorithm checks whether a player's word can be made using the available letters. It loops through each character in the input word, checks if it exists in the letter pool, and removes it once used. If any letter isn't found, the word is considered invalid.

Time Complexity: O(n * m), where n is the length of the word and m is the length of the letter array.

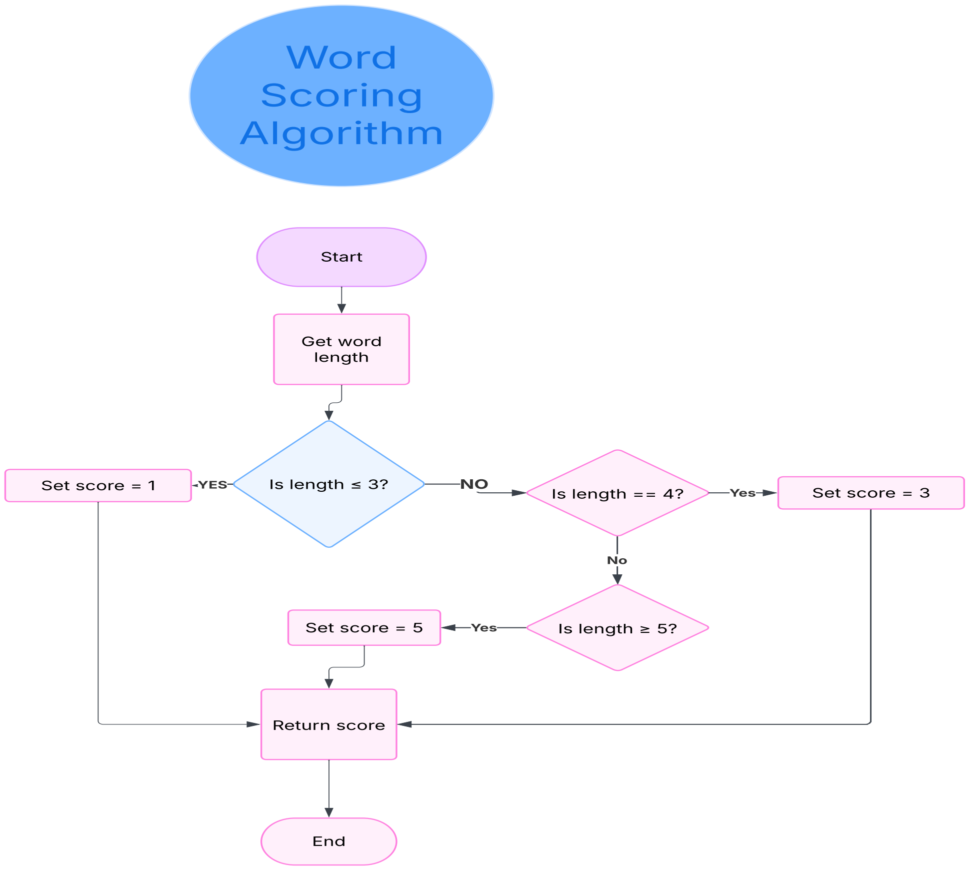
2. Word Scoring Algorithm



```
/** Word Scoring Algorithm
 * Get the length of the submitted word
 * if length is 2-3 assign 1 point.
 * if length is exactly 4 assign 3 points.
 * if length is 5 or more, assign 5 points.
 * Return the score
 */

// calculates the score for a word based on its length (1-5)
private int calculatePoints(String word) {
    int len = word.length(); // get the length of the word
    if (len <= 3) return 1; // 1-3 letter words are worth 1 point
    else if (len == 4) return 3; // words with 4 letters exact are 3 points
    else return 5; // 5 letter words are worth 5 points
}
```

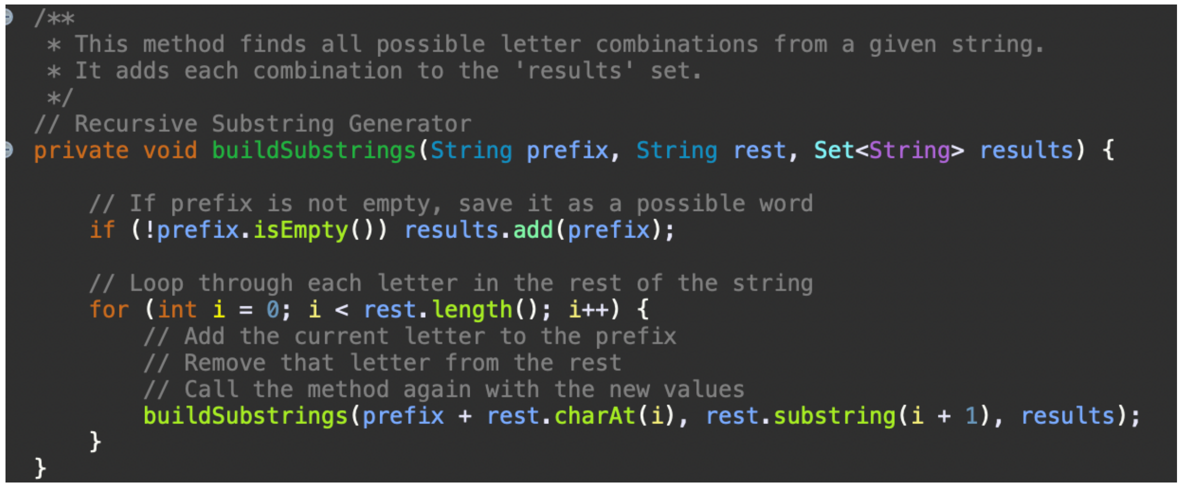
Flowchart



This algorithm calculates how many points a word is worth based on its length. Words with 2–3 letters are worth 1 point, 4-letter words are worth 3 points, and 5+ letter words are worth 5 points. It uses simple if-else logic.

Time Complexity: O(1) — constant time check.

3. Recursive Substring Generator

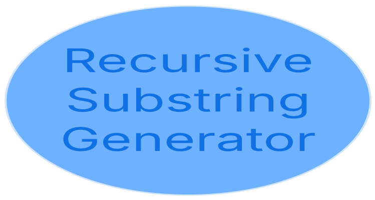


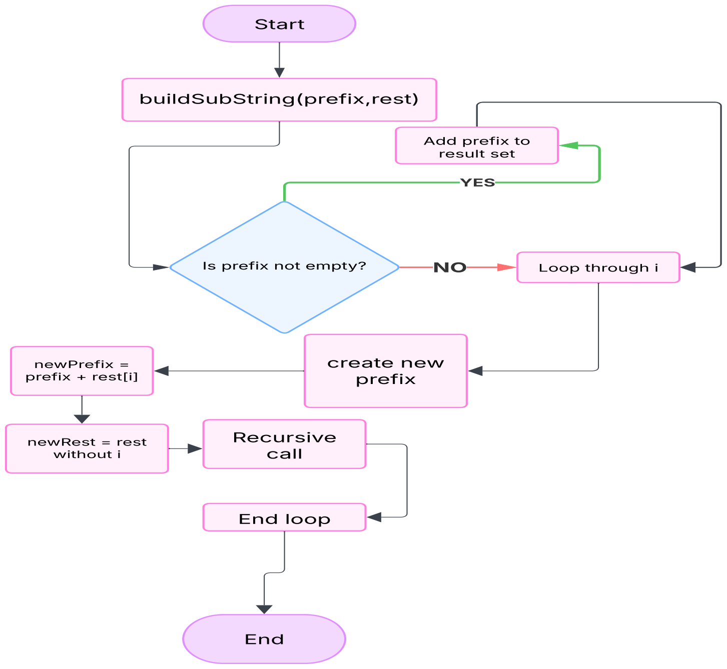
```
/**
 * This method finds all possible letter combinations from a given string.
 * It adds each combination to the 'results' set.
 */
// Recursive Substring Generator
private void buildSubstrings(String prefix, String rest, Set<String> results) {

    // If prefix is not empty, save it as a possible word
    if (!prefix.isEmpty()) results.add(prefix);

    // Loop through each letter in the rest of the string
    for (int i = 0; i < rest.length(); i++) {
        // Add the current letter to the prefix
        // Remove that letter from the rest
        // Call the method again with the new values
        buildSubstrings(prefix + rest.charAt(i), rest.substring(i + 1), results);
    }
}
```

Flowchart





This algorithm uses recursion to generate all possible substrings (letter combinations) from the given letter set. These combinations are stored in a Set, and the program checks which are valid Scrabble words to show players what words they could have made.

- Time Complexity: O(2^n), where n is the number of letters. This is due to the nature of recursion and branching possibilities.

Opportunity I Took Advantage Of

During development, I realized that I could make the game more helpful by showing the player what words they could have made with their letter set. This opportunity led me to implement the recursive substring generator and connect it to my dictionary. It makes the game more educational and engaging by giving feedback.

Error I Faced and How I Fixed It

One challenge I faced was passing the character array to the recursive function correctly. Initially, I used new String(letters), which caused formatting issues and unexpected results. I fixed this by replacing it with String.valueOf(letters), which cleanly converts a char[] into a usable string. This simple fix made the word suggestions work properly.

What I Would Improve in the Future

In a future version of the game, I would like to:

- Add a timer for each player's turn.
- Build a graphical user interface using JavaFX or Swing.
- Add a word difficulty level and bonus points for rare words.
- Allow multiplayer games over a network.

These improvements would make the game more interactive and fun, while also giving players new challenges.

Data Structures Used and Why

Data Structure	Use Case
ArrayList <Word>	Stores and sorts the full dictionary for fast lookup using binary search
HashSet<String>	Tracks words that are already used and prevents repeats
Stack<String>	Enables the undo feature by storing previous words
Queue<String>	Manages alternating player turns fairly
LinkedList<Character>	Stores vowels to ensure every letter set has at least one vowel
String/Char[]	Used for input, word formation, and validation