I) Problem Description: Scrabble!

The game of "Scrabble" has been a classic since the mid 20th century. In the United States, Scrabble tournaments have been held even since 1973 (https://www.britannica.com/sports/Scrabble). Each player in a game of Scrabble has a set of random letter tiles. The game consists of forming horizontal or vertical words on a board, crossword-style, and trying to earn the highest number of points. Points are scored by summing up the point value of each letter in a word. Scrabble scores each letter according to the following:

1 Point - A, E, I, L, N, O, R, S, T and U.

2 Points - D and G.

3 Points - B, C, M and P.

4 Points - F, H, V, W and Y.

5 Points - K.

8 Points - J and X.

10 Points - Q and Z.

Your task is to implement an algorithm that helps its user emerge from Scrabble victorious. Given an input string of letters, the function will return a Set of strings of the highest scoring words possible from those letters. For example, inputting "crteazxqt" will return the set {"zax"}, which is a 19 point word. A Lexicon of valid English words and a program-wide constant letterPoints has been provided for you. letterPoints maps an integer Scrabble point value to its associated set of letters.

The function prototype has been provided:

Set<string> findHighestScoringWords(**Lexicon&** lex**, string** letters)**.**

Your function should also fulfill the following:

- The program should make use of recursive backtracking to implement its task
- A valid Scrabble word must be 2 letters or longer in length
- If there are no possible valid words from a string of letters, it should return an empty set
- If there are multiple highest-scoring words, it should return all of them
- It should be case insensitive- that is, an input string of letters can be in lowercase, uppercase, or mixed-case and it should still function normally

Feel free to implement any helper functions as needed.

II) Solutions and Test Cases

Solution 1:

```
/* A helper function to add the number of points
 * earned by a single word, according to Scrabble
 * rules
 */
int findPoints(string word) {
    int count = 0;
    for (char ch : word) {
        if (letterPoints[1].contains(ch)) {
            count += 1;
        } else if (letterPoints[2].contains(ch)) {
            count += 2;
        } else if (letterPoints[3].contains(ch)) {
            count += 3;
        } else if (letterPoints[4].contains(ch)) {
            count += 4;
        } else if (letterPoints[5].contains(ch)) {
            count += 5;
        } else if (letterPoints[8].contains(ch)) {
            count += 8;
        } else if (letterPoints[10].contains(ch)) {
            count += 10;
        }
    }
    return count;
}
/* A helper function that builds a stack of highest-scoring words
 * by comparing point values and replacing existing
 * words in the stack such that the stack will only contain the
 * maximum-scoring words at a given time. Assumes there is at least one
 * word already present in the stack. Note: also possible to use
 * alternative data structures, like a queue or set.
 */
void buildStack(Stack<string>& words, string potentialWord) {
    int curPoints = findPoints(potentialWord);
```

```
    int previousPoints = findPoints(words.peek());
    // compare point values
    if (curPoints > previousPoints) {
        /* if found greater point value,
         * replace the word so the stack only contains
         * the maximum scoring words
         */
        words.clear();
        words.push(potentialWord);
    // if point values equal, add the word
    } else if (curPoints == previousPoints) {
        words.push(potentialWord);
    }
}

/* A helper function for findHighestScoringWords
 * that uses recursive backtracking to find the
 * highest-scoring words.
 */
void findHighestScoringWordsHelper(Lexicon& lex, Stack<string>& words, string
remaining, string& soFar) {
    // base case: no decisions remain or prefix not in lex
    if (remaining.empty()) {
        return;
    } else if (!lex.containsPrefix(soFar)) {
        return;

    } else {
    // build the set of words; compare point values to get highest scoring
words
        if (lex.contains(soFar) && (soFar.length() >= 2)) {
            if (words.isEmpty()) {
                words.push(soFar); // if empty, add the word
            } else {
                buildStack(words, soFar);
            }
        }
        // choose
```

```
        for (int i = 0; i < remaining.length(); i++) {
            char letter = remaining[i];
            string rest = remaining.substr(0, i) + remaining.substr(i + 1);
            soFar.push_back(letter);

            //explore
            findHighestScoringWordsHelper(lex, words, rest, soFar);

            //unchoose
            soFar.erase(soFar.length() - 1, 1);

        }
    }
}

Set<string> findHighestScoringWords(Lexicon& lex, string letters) {
    Stack<string> words;
    Set<string> wordSet;
    string soFar;
    findHighestScoringWordsHelper(lex, words, toLowerCase(letters), soFar);
    // build corresponding set from the stack
    while (!words.isEmpty()) {
        wordSet.add(words.pop());
    }
    return wordSet;
}
```

This solution uses a helper function to build a stack of the highest-scoring words, which is called in the recursive backtracking function itself. The stack is built by comparing the point value of its existing word/words to the point value of a potential word, and repeatedly clearing and filling the stack so that at any given point it only contains the highest-scoring words found so far. The information from the stack is then stored into the set. Here, I've used a stack, but the set can be built multiple ways; through a queue, set, etc. If n is the number of characters in the input string of letters, the Big O is O(n * n!). We can visualize the decision tree to see this. If we start with n nodes, at each node there are n-1 decisions to be made. At each n-1 node, there are n-2 decisions, etc. Thus, the recursive backtracking itself is n!, leading to a total Big-O of O(n * n!).

Solution 2:

```
/* A helper function to add the number of points
 * earned by a single word, according to Scrabble
 * rules
 */
int findPoints(string word) {
    int count = 0;
    for (char ch : word) {
        if (letterPoints[1].contains(ch)) {
            count += 1;
        } else if (letterPoints[2].contains(ch)) {
            count += 2;
        } else if (letterPoints[3].contains(ch)) {
            count += 3;
        } else if (letterPoints[4].contains(ch)) {
            count += 4;
        } else if (letterPoints[5].contains(ch)) {
            count += 5;
        } else if (letterPoints[8].contains(ch)) {
            count += 8;
        } else if (letterPoints[10].contains(ch)) {
            count += 10;
        }
    }
    return count;
}


/* A helper function for findHighestScoringWords
 * that uses recursive backtracking to find valid English
 * words from a string of letters. Note that instead of using
 * a helper function to build a stack of the highest-scoring words,
 * in this solution the function just stores all valid English
 * words in a stack to be analyzed later.
 */
void findHighestScoringWordsHelper(Lexicon& lex, Stack<string>& words, string
remaining, string& soFar) {
```

```
    // base case: no decisions remain or prefix not in lex
    if (remaining.empty()) {
        return;
    } else if (!lex.containsPrefix(soFar)) {
        return;

    } else {
        if (lex.contains(soFar) && (soFar.length() >= 2)) {
            words.push(soFar); // found valid word
        }
        // choose
        for (int i = 0; i < remaining.length(); i++) {
            char letter = remaining[i];
            string rest = remaining.substr(0, i) + remaining.substr(i + 1);
            soFar.push_back(letter);

            //explore
            findHighestScoringWordsHelper(lex, words, rest, soFar);

            //unchoose
            soFar.erase(soFar.length() - 1, 1);

        }
    }
}
/* findHighestScoringWords makes use of findHighestScoringWordsHelper
 * to find the stack of valid English words from a string of letters.
 * After finding the stack, this function then sorts through the words to
 * find the maximum-scoring words.
 */
Set<string> findHighestScoringWords(Lexicon& lex, string letters) {
    Stack<string> words;
    Set<string> wordSet;
    string soFar;
    findHighestScoringWordsHelper(lex, words, toLowerCase(letters), soFar);

    // build highest scoring words from the stack
    string maximum;
```

```
    if (words.isEmpty()) {
        return wordSet;
    } else {
        maximum = words.pop(); // set maximum equal to frontmost word
    }
    while (!words.isEmpty()) {
        string next = words.pop(); // compare to next word
        int curMaxPoints = findPoints(maximum);
        int nextPoints = findPoints(next);
        if (nextPoints > curMaxPoints) { // if found higher-scoring word,
update maximum
            wordSet.clear(); // in case we previously added an equal-scoring
word, clear the set to reflect updated maximum
            maximum = next;
        } else if (nextPoints == curMaxPoints) {
        // if equal-scoring word, add it to the set to store it and update
        maximum
            wordSet.add(maximum);
            maximum = next;
        }
    }
    wordSet.add(maximum);
    return wordSet;
}
```

This alternative solution builds the set in findHighestScoringWords itself rather than making use of a helper function in the recursive backtracking. The set is built by going through the stack and continually updating the "maximum" variable to store the highest-scoring word. If there are multiple words found that have the maximum point value, the set will temporarily store them; but note that once a higher-scoring word is found, the set will be cleared to reflect that a new maximum has been found.

Given an input string of letters containing n characters, the Big-O is the same as solution 1, O(n * n!). But one difference is that rather than continuously clearing and filling a stack, this solution goes through the stack and updates the "maximum" variable, only clearing the set when

necessary. Here, the building of the set itself may be more efficient in some cases where there are not many words with the same point value.

For my test cases, I tried to think about edge cases; for example, an empty input string or no possible valid words from the input string. After that, I experimented with different letter-cases because I mentioned that the function should be case-insensitive. I also tried to include tests that had multiple highest-scoring words, along with tests that had just one highest-scoring word.

III. Problem Motivation

Conceptual Motivation

This problem is designed to help students familiarize themselves with the difficult concept of recursive backtracking. A problem like finding a highest-scoring word is particularly well suited for recursive backtracking. Solving this problem requires an understanding of:

- How to implement a solution that uses recursive backtracking, particularly the choose-explore-unchoose strategy
- The ability to decompose complex problems into smaller functions
- Thinking about what information needs to be kept track of (for instance, the letters remaining/the letters used so far)

Personal Motivation

Personally, recursive backtracking has been the most difficult concept that I've encountered in this class. I especially struggled with the recursive backtracking assignment and wanted to get more practice with a problem like Boggle, so I thought of doing something related to a word game. But I also just really love word games and used to play them a lot when I was younger, so I thought it would be cool to actually code something that has practical use in a game. It would be funny to whip out this code the next time I'm playing Scrabble or Bananagrams with my friends.

IV) Common Misconceptions

Misconceptions or bugs could arise in a couple of areas:

- Writing the recursive step

- Writing the correct base cases
- Building the set of highest-scoring words

Writing the choose-explore-unchoose step in actual code may be confusing or difficult. Some students might forget the unchoose step after the recursive call, where the letter added to soFar should be removed before continuing to search for words. Furthermore, it may be difficult for some students to realize what they need to keep track of through each recursive call; that is, they might fail to realize that they need to write a recursive helper function because the provided function prototype does not provide enough parameters to keep track of the built word and the remaining letters.

Writing the correct base cases is also important because they determine when the recursion should return and go back up the call stack. Students may forget to check if the string of remaining letters is empty, or might not realize that if a prefix is not in the Lexicon then they should stop searching.

Building the set of highest-scoring words can be complicated because students need to keep track of both a word itself and its point value. One mistake might be forgetting to clear the words stored so far if a higher-scoring word is found; since the set only stores the maximum-scoring words, if any higher-scoring word is found it means that all previously stored words are immediately irrelevant. Another mistake would be forgetting to keep track of words that have equal point values.

V. Ethics Reflection

My Project

This project does not raise many ethical issues because it deals mainly with finding possible words from an input of letters. However, one thing to note is that algorithms like this should never be used in a competitive setting; in Scrabble tournaments, or perhaps playing competitively online with other people, using a program like this would be a violation of the fairness that should be expected in a competitive setting. I've chosen to explore an ethical issue that is slightly unrelated to the scope of my project, but that I find to be pressing and important.

Article + Ethics Question

https://papers.ssrn.com/sol3/papers.cfm?abstract_id=998565

This article by Daniel J. Solove debunks the "I've got nothing to hide" argument that people often use to express why they often feel indifferent with potential threats to their privacy. The themes presented in this paper touch on some issues that were brought up in the ethics component of Assignment 5 with priority queues and data collection of unhoused persons. As the assignment handout notes, "One of the criticisms of the coordinated entry system is that those that want to access homeless services have no other choice but to go through this system. This raises concerns of privacy and autonomy. Storage of the personal information enables further surveillance and criminalization of the unhoused." Relatedly, Solove's article articulates how the storage and use of personal information can be harmful in ways that are often overlooked by the general public.

Q: (1) Have you used the "I've got nothing to hide" argument before, and has Solove's article changed your stance on it? Why or why not?
(2) Say that you are working with a company or agency that collects personal data; how would you proceed while keeping in mind concerns about privacy?

A: (1) I've relied on the "I've got nothing to hide" argument before, but this article has shown me that the concept of privacy does not have one single definition and is much broader than just having something to hide. By broadening my concept of privacy, I've realized that the harms than can come when powerful organizations invade personal privacy can be much more implicit and dangerous than we might initially think.

I especially liked Solove's analogy of Orwellian vs Kafkaesque; we often view privacy issues under the lens of George Orwell's *1984,* in which we are fearful of an inhibition of our freedoms due to government surveillance. Solove urges us to instead view privacy through a lens using Kafka's *The Trial,* in which a bureaucracy uses people's information to make important decisions about them, yet the people have no say in how their information is used. The issue here is not necessarily one of an explicitly dystopian nature, but rather the issue is about the

relationships between the people and institutions; the invasion of privacy involving the use of data creates a power imbalance.

(2) I think that two things are especially important when an institution seeks to collect personal data: transparency and consent. The people providing their data should not only be able to consent to the data collection but should also be able to have a say in how their information is being processed and used. This involves a responsibility on the developers' part to be explicitly transparent about how the data will be stored and what exactly the data will be used for, and to also provide a way for people to revoke their consent in case they eventually decide that they do not feel comfortable with the institution storing their data. It is important that developers try their best to not normalize a power imbalance between the institutions in power and the people.