San Francisco State University
CSC 220: Data Structures
Dawson Zhou (zhoud@sfsu.edu)

Assignment 2: Recursion

Overview

In this assignment, you'll use what you've learned about recursion to solve three problems: two of which use "standard" recursion, and one of which uses recursion exploration.

Submission

Assignment 2 is due on **October 13, 2021** at **11:59 pm PT**.

Late submissions will be accepted with a penalty:

- October 14, 2021 12:00 am PT to October 14, 2021 11:59 pm PT will receive 75%
- October 15, 2021 12:00 am PT to October 15, 2021 11:59 pm PT will receive 50%

Submit your solution through iLearn. You will need to submit these files:

- **LetterFinder.java**
- **WordCounter.java**
- **WordSplitPrinter.java**

Grading

100 pts total:

- 20 pts: LetterFinder
- 20 pts: WordCounter
- 40 pts: WordSplitPrinter
- 20 pts: Miscellaneous style issues

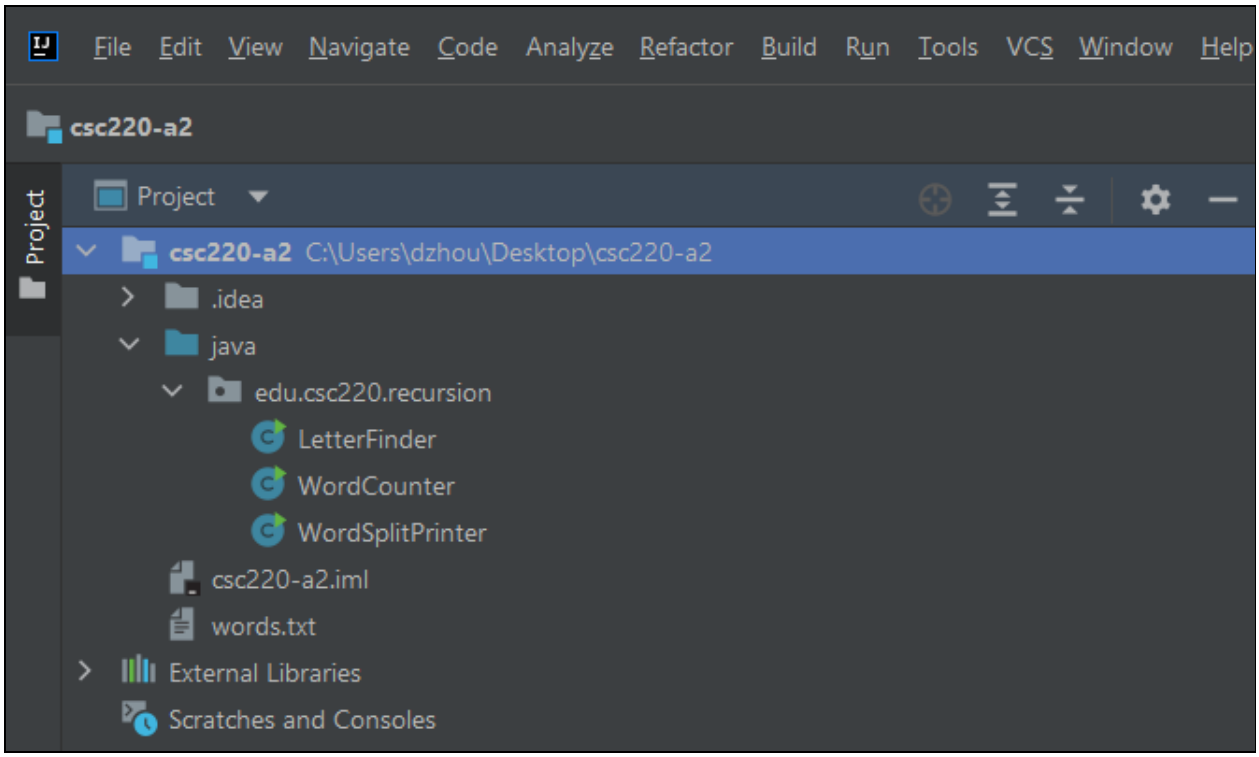
Assignment Setup

Download the folder named “Recursion Starter Code” from iLearn. It will be downloaded as a .zip file which can be extracted to whatever location you want on your computer. Make sure that the folder structure appears as follows:

```
> csc220-a2
      > java
      > words.txt
```

Set up an IntelliJ project choosing “csc220-a2” as the root. For step-by-step instructions on setting up an IntelliJ project, see the “IntelliJ Setup Guide” handout on iLearn (at the top of the iLearn course page).

Once the project is set up, your project navigation panel should look something like this:



Make sure that the “words.txt” file is included as part of the project. It contains a list of commonly used English words and is used in the `WordSplitPrinter` problem. Please reach out ASAP if you have any problems setting the project up.

Detailed Requirements

LetterFinder

`LetterFinder` is a class that defines a method: **`findEarliestLetter`**. It initially appears as follows:

```
public char findEarliestLetter(String word) {
    // TODO: Implement this.
    return 'z';
}
```

You will be replacing the **`TODO`** and the placeholder line (`return 'z';`) with your own implementation. **`findEarliestLetter`** should return the character in the input String word that is considered "earliest," meaning that it appears earlier alphabetically. For example, in the String "cat", the earliest letter is 'a', as 'a' comes before 'c' and 't' in the alphabet. You can assume `input` will never be the empty String, and that `input` will only contain lower-case letters.

Given two characters, you can easily determine which appears earlier alphabetically by using a direct comparison:

```
char foo = // ...
char bar = // ...
if (foo < bar) {
    // This means foo appears earlier alphabetically than bar.
}
```

In your implementation, you'll need to replace `foo` and `bar` with the appropriate characters. To read the character at an index in a String, you can use the **`charAt`** method:

```
char ch = input.charAt(index);
```

In order to receive credit:

- You **must implement** **`findEarliestLetter`** **using recursion**.
- Your implementation **cannot** use any loops.
- Your implementation **cannot** use any instance variables.

- Your implementation **cannot** modify the parameter list or return type of `findEarliestLetter`.

Hint: We want to consider how the answer to a simpler version of the problem can help you solve the original problem. Using a recursive call, you can find what the earliest letter is in `input.substring(1)` -- i.e. everything in input *other than* the first character. How can you combine this information with what you know about the first character to find the earliest letter overall?

WordCounter

`WordCounter` is a class that defines a method: **`countWordsWithLength`**. It initially appears as follows:

```
public int countWordsWithLength(ArrayList<String> words, int targetLength)
{
    // TODO: Implement this.
    return 0;
}
```

You will be replacing the **`TODO`**s and the placeholder lines (`return 0;`) with your own implementation. **`countWordsWithLength`** should return the number of Strings in the `words` list whose length is equal to `targetLength`. See the code in the main method for examples.

In order to receive credit:

- You **must implement `countWordsWithLength` using recursion**.
- Your implementation **cannot** use any loops.
- Your implementation **cannot** use any instance variables.
- Your implementation **cannot** modify the parameter list or return type of **`countWordsWithLength`**.

Note that your implementation **can** modify the `words` parameter as needed.

Hint: Again, you'll want to consider how the solution to a simpler version of this problem will help you solve the original problem. Using a recursive call, you can count the number of Strings that have length `targetLength` in the `words` list with one element of the list removed. How can you combine that information with what you know about the removed word to find the answer to the original problem?

WordSplitPrinter

In lecture, we talked about how we can use recursion exploration to print out all permutations of a given string. For example, we could use recursion on the input “goat” to print out:

```
goat    ogat    agot    tgoa
gota    ogta    agto    tgao
gaot    oagt    aogt    toga
gato    oatg    aotg    toag
gtoa    otga    atgo    tago
gtao    otag    atog    taog
```

If we’d like to find out all the possible anagrams we can form from these, we could check which are complete words (like “toga”), but that wouldn’t be enough. There are also cases like “atgo” where the full string isn’t a word, but we can break it into the separate words “at” and “go”.

To find all anagrams, we need to find the ways we can take one of these permutations and split it into separate English words. That’s where your method, **`findWordSplits`**, comes in:

```
public void findWordSplits(String input, TreeSet<String> allWords) {
    // TODO: Implement this.
}
```

The method takes two parameters: `input` and `allWords`.

- `input` will be a string like “atgo” or “goat” that you’re trying to split into separate words. You **do not** need to find permutations of `input`, since that was part of the demo we did in class. `input` will be a string whose character ordering is already fixed.
- `allWords` is a set containing the words listed in the “words.txt” file -- nearly 10,000 English words. `allWords` is already populated for you! You’ll need to use it when you’re splitting up the input string, since each chunk needs to be an actual word. Remember to pass it to each recursive call as a parameter.

Your task is to implement **`findWordsSplits`** to find every way you can split `input` into valid English words, where every character in `input` is used exactly once. For example, if `input` is “goat”, we would expect these two splits to be printed:

```
go at
goat
```

Other arrangements, such as “g oat” or “goa t”, would not be printed because “g”, “goa”, and “t” are not considered words. The **`main`** method uses “isawabus” as input. Once **`findWordsSplits`** is correctly implemented, you should see the following output:

```
i saw a bus
i saw ab us
is aw a bus
is aw ab us
```

I’ve provided an already implemented method in the starter code called **`printWordSplit`**:

```
public void printWordSplit(ArrayList<String> words) {
    // Already implemented for you.
}
```

You should call **`printWordSplit`** to print out a single arrangement. For the “i saw a bus” example above, you would call **`printWordSplit`** passing in an `ArrayList<String>` containing `["i", "saw", "a", "bus"]` to print it out.

`findWordSplits` is a recursion exploration problem, just like our permutations demo in lecture. Think about how to apply the recursion exploration formula:

1. How do you represent a path of options you’re exploring?
    - The sequence of options picked so far would consist of entire words. For example, you might have chosen “i”, followed by “saw”, followed by “ab”, with “us” still left over. An **`ArrayList<String>`** would work here.
2. At each step, what are your options?
    - At each step, we want to consider what the next word we want to take is. That word can be as short as one character long, or as long as the entire input.
    - Consider an example like “isawabus”. Here are the options for the first word we can take:

        ```
        i
        is
        isa
        isaw
        isawa
        isawab
        isawabu
        isawabus
        ```

You'll need to check for which of these options are valid English words using the `allWords` set. If `allWords` contains a given String, it's considered a valid word.
    - When you choose one of these options, you'll need to consider how to recursively continue your exploration. If your first word was "is", for example, the leftover letters to choose from would be "awabus". These will help you determine what the parameters of the recursive call should look like.
    - You'll need to use the **`String`**'s **`substring`** method to pick out the next word you want to try, and also to determine what the remaining letters are:
        - To get a substring that contains the first `i` characters, use `str.substring(0, i)` where `str` is the name of your **`String`** variable.
        - To get the rest of that string (i.e. everything *other than* the first `i` characters), use `str.substring(i)`.
4. What do you do when you've finished exploring a path?
    - Just call **`printWordSplit`**!

As always, I strongly recommend diagramming out your approach, using a visualization of the various branches, paths, and options like we did in Lecture 11 when we discussed recursion exploration.