CS 455 Programming Assignment 4

Spring 2020 [Bono]
Due: Wednesday, April 15, 11:59pm

Introduction and Background

In this assignment you will get a chance to use some of the Collection classes and methods we have covered recently. This will enable you to write a faster-running program with less effort than you would otherwise. In this assignment you will also get an opportunity to do your own design; the design outline we provided you is less constrained than in past assignments: you will be deciding on the exact interface and representation for your most of your classes. You'll also get some practice with command-line arguments and text file processing.

This assignment concerns the game of Scrabble. You may know the game of Scrabble better as Words with Friends. If you want to try out Words with Friends yourself you can download the free app for your smartphone. However, the programming assignment is not to create the game itself, but to write a console-based program that finds all possible words that can be made from a rack of Scrabble tiles (so it could help someone playing Scrabble). We'll elaborate on the exact requirements of this assignment in the section on the assignment below.

A rack of Scrabble tiles (the little number is the score for playing that tile)

The assignment files

Note: the blurbs below do not describe what each of these classes are, and how they fit together. For more details on that, see the section on the class design.

The starter files we are providing for you on Vocareum are listed here. The files in **bold** below are ones you create and/or modify and submit. The ones not in bold are ones that you will use, but not modify. More details about the java classes below are in the section on class design. The files are:

- **`WordFinder.java`** This class will contain the `main` method, and any other helper methods that you design. (You create this file.)
- **`AnagramDictionary.java`** All anagram sets from a dictionary. We have provided the interface for you. This class is discussed more here.

- **`Rack.java`** Stores the current rack. You can decide on the representation and public methods for this class. We wrote the private static `allSubsets` method for you, discussed later.
- **`ScoreTable.java`** This class has information about how much each scrabble letter is worth. (You create this file.)
- `sowpods.txt` The Scrabble dictionary we will be using. The version given here is all lower case letters. Go here for an explanation of its odd name.
- `testFiles` A subdirectory with some data files and corresponding output for help in testing. The `README.txt` file in that directory explains the files and how to use them.
- **`README`** See section on Submitting your program for what to put in it. Before you start the assignment please read the following statement which you will be "signing" in the README:

  > "I certify that the work submitted for this assignment does not violate USC's student conduct code. In particular, the work is my own, not a collaboration, and does not involve code created by other people, with the exception of the resources explicitly mentioned in the CS 455 Course Syllabus. And I did not share my solution or parts of it with other students in the course."

Note: you may have additional files, see the section on the class design for more about this.

The assignment

You will be implementing a program, called WordFinder, that when given letters that could comprise a Scrabble rack, creates a list of all legal words that can be formed from the letters on that rack. To solve the problem you will also need a scrabble dictionary (we'll provide that for you). Some particulars of the Scrabble dictionary: it only has words of length two or more, and it includes all forms of a word as separate entries, e.g., singular plus plural, verb conjugations.

For example, if your rack had the letters **c m a l** you could rearrange the letters to form the words `calm` or `clam`, but you could also form shorter words from a subset of the letters, e.g., `lam` or `ma`. It's generally difficult to figure out all such sequences of the letters that form real words (unless you are a tournament Scrabble competitor who knows the Scrabble dictionary very well).

For your program, you will display all such words, with the corresponding Scrabble score for each word, in decreasing order by score. Each letter has a score associated with it, the score for a word is the sum of the scores of each letter in that word. For words with the same scrabble score, the words must appear in alphabetical order. Here are the results for a rack consisting of "cmal" (using the sowpods dictionary) in the output format you will be using for your program (user input is shown in italics):

```
Rack? cmal
We can make 11 words from "cmal"
All of the words with their scores (sorted by score):
8: calm
8: clam
7: cam
```

```
7: mac
5: lac
5: lam
5: mal
4: am
4: ma
2: al
2: la
```

We'll provide you the Scrabble score for each letter later in this document.

Here's more about exactly how to run your program and what happens:

Your program will take an optional command-line argument for the dictionary file name. If that argument is left off, it will use the Scrabble dictionary file `sowpods.txt` (see assignment files) from the same directory as you are running your program. If the dictionary file specified (either explicitly or the default one) does not exist, your program will print an informative error message (that includes the file name) and exit.

Once the program starts it will print the message:

```
Type . to quit.
```

Then the program will run in a loop on the console, printing the prompt "Rack? " (as seen in the earlier example) and reading and processing each rack you enter, until you tell it to exit. The use tells the program to exit by typing in "." at the prompt (i.e., a period). We aren't use a command such as "quit" as the sentinel, since that could be a legal rack.

We have provided you a few sample data files, and corresponding correct reference output from running those on the sowpods.txt (the Scrabble dictionary given) in the `testFiles` directory. Your output must match the reference output character by character.

The real game of Scrabble has only upper-case letters on tiles, but for our program we'll accept any sequence of non-whitespace characters as a legal "rack." However, words will only be able to be formed from actual letters if that's what's in the given dictionary. E.g., if the rack given is "abc@" you will report the words such as "cab", but there will be no words containing "@", since @ doesn't appear in any dictionary words.

The program will work on both lower-and-upper case versions of dictionaries, but all processing will be case-sensitive. E.g., if the dictionary given has only upper-case versions of words, it will find words from a rack such as "CMAL", but won't be able to find any words from the rack "cmal".

Some other differences between this program and Scrabble:

- The real game of Scrabble also has two blank wild-card tiles. Your program is not required to handle blanks.
- In Scrabble you almost always have a rack of exactly seven letters. For this program you can enter any number of characters for a rack. If the rack has more than seven characters, you will report words from the dictionary that have more than seven characters too.

- This program just deals with forming words only from what's on the rack, it doesn't consider any tiles that are on the Scrabble board.

This shows how to run your program:

```
java WordFinder [dictionaryFile]
```

Note: in this common format for showing Unix command-line syntax the square brackets (i.e., []) are not part of the command that is typed: it is just syntax indicating that the command line argument shown is optional.

Additional program requirements are described in the following sections and summarized here:

- **Approach.** you are required to use the second approach discussed below, under Approach. The class design we started goes along with that approach.
- **Efficiency.** you will get more credit if you have an efficient solution. We discuss the efficiency of the approach you are required to use in the next two sections.
- **Class design.** you are required to design and implement the classes discussed in the section on class design. We will also be evaluating the quality of the fleshed out version of this design.
- **Error checking.** the only error you have to handle is if the dictionary file given is not found. This was discussed in the section on the assignment. You are not required to check that all the words from the dictionary file are unique.
- **README.** as usual, you are required to submit a `README` file. See the end of this document for what needs to go in it for this assignment.
- **Style / Documentation / Design.** Also as usual, your program will be evaluated on style and documentation. See the section on grading criteria for more details.

Approach

There are two distinct ways to approach this problem. One is to read in the dictionary, and then for each rack given, compare each word in the dictionary to that rack to figure out whether that word can be formed from some or all of the letters in that rack, creating a list of the legal words as you go. This is faster to process the dictionary, but slower to process each rack.

The second approach, which is the one you will be using for the assignment, involves preprocessing the dictionary so that you organize the words by the set of letters each one contains (this set is actually a multiset, because letters can appear more than once in a word; the rack itself is also a multiset). Then for each rack you'll generate all the subsets of that multiset of letters, and for each subset add all the words from the dictionary that have exactly the same elements as that subset. This is slower to process the dictionary, but once we do this processing, it's faster to process each rack than the first approach. This approach is explained in more detail in the following two sections.

It's a little complicated to describe in big-O terms the time for each approach, but what makes the first approach slower for processing one rack is traversing the whole dictionary (which will typically be large) *for each* rack. For the second approach, the slow part of processing a rack is creating all the subsets. The worst case for creating the subsets is if there are no repeated letters in the rack (i.e., largest number of subsets created). Even though generating the subsets for such a rack would take O($n * 2^n$) for a rack of n unique characters (because there are 2^n subsets when there are no repeat characters, and n n steps to form each subset), n will typically be small: for a 7-tile rack: 2^7 is only 128, times 7 is 896). In a solution we wrote using this approach, processing the sowpods dictionary took under half a second, and processing a 7-character rack with no repeating characters, and consisting of the most commonly occurring letters in English took under 15 milliseconds (this is test file `testFiles/aestnlr.in`). (Commonly occurring letters will result in a larger resulting word list.) These runs were done on Vocareum.

Some of the time spent for processing a rack in the second approach is to get the list of anagrams for each subset; we'll discuss that further in the next section.

The rest of the time spent processing a rack is to sort the resulting word list (which we would have to do for either approach).

For full credit on this assignmnet you'll need to use this second approach for the assignment; we'll go into further details about it in the following sections.

The `AnagramDictionary` class

For the approach we're using, we said that you would organize the dictionary words by the (multi)set of letters a word contains. If two words contain the same exact letters in a different order, they are called anagrams of each other. If a rack (or subset of that rack) has all the same letters (and multiplicity of those letters) as a particular word in the dictionary, that word, plus all of its anagrams from the dictionary should all be added to the list of words reported by our WordFinder program.

You are required to create an `AnagramDictionary` class to handle this. It will have a `getAnagramsOf` method that finds all anagrams of a particular string efficiently. For, example, suppose we have a variable, `dictionary`, of type `AnagramDictionary`, that contains data from the sowpods dictionary. If we did the call

```
dictionary.getAnagramsOf("rlee")
```

it would return an ArrayList of words with the contents: ["leer", "lere", "reel"] (not necessarily in that order). Note, "rlee" is not a real word, but the method does not require you to pass it a real word. But the anagrams returned are real English words.

How to do this efficiently? One insight is that if we put two words into some kind of canonical form, then we could figure out if they are anagrams of each other by just comparing the canonical versions of them for equality. This canonical form will be a sorted version of the characters in the word. In the earlier example given the rack contained "cmal". The sorted version of this rack is "aclm". The first two words listed in the output are "calm" and "clam", anagrams of "aclm", or put another way, these first two words are the only dictionary words we can make using all the letters on the rack, and all the other words listed are anagrams of *subsets* of "cmal".

For full credit your `AnagramDictionary` is required to find all the anagrams of one String in time linear in the size of the output set (not including the time to sort the letters in the String given).

Finding all the subsets of the rack

Finding all subsets of a multiset is a somewhat difficult recursion problem on its own, so to make this assignment easier, we wrote the code for you to do that (static method `allSubsets` in `Rack.java`). The method is static because, like some other recursive methods we have written, it takes all of its data as explicit parameters; also, this means `allSubsets` works regardless of what representation you choose for your Rack objects (it will not be accessing any Rack instance variables). The solution is similar in structure to the method to compute all permutations of a string given in Section 13.4 of the textbook. You will likely have to write a wrapper method that calls `allSubsets` with the correct starting parameters.

The `allSubsets` method uses a particular representation for the rack which we'll explain with an example here. Earlier we mentioned that a rack is a multiset of letters (set because we don't care about the order of the letters, and multiset because letters can appear more than once). Suppose our rack is:

a b a d b b

Gathering together the like letters, we could rewrite this as "aabbbd". We could also say that 'a' appears with multiplicity 2, 'b' appears with multiplicity 3, and that 'd' appears with multiplicity 1. `allSubsets` expects the rack information to be in two parallel arrays: one has the unique letters, and the other has the multiplicity of that letter at the same array index. The array of unique letters is actually a String, so we can do String operations on it. For the example given, we could create this rack representation as follows:

```
// create variables for the rack "aabbbd"
String unique = "abd";
int[] mult = {2, 3, 1};

// example to show relation between values in unique and mult:
for (int i = 0; i < unique.length(); i++) {
   System.out.prinln(unique.charAt(i) + " appears " + multi[i] + " times in the rack");
}
```

Like other examples of recursion over an array that we've seen, `allSubsets` will take a third argument, `k`, which is the starting position of the part of the array that this recursive call will process. So for this code, it's the starting postion from which to find the subsets. So, for example, if we called

```
allSubsets(unique, mult, 1);  // starts at position 1 in unique and mult
```

it would find all the subsets of the rack "bbbd" (i.e., it wouldn't consider the subsets that included any 'a's in it).

Class design

Unlike the previous programs in this course, this time you are going to design your own classes, with some guidance. Consequently, part of your style score will be based on the quality of your design.

When doing an object-oriented design, you first come up with a candidate set of classes, choosing a name for each, and identifying the responsibilities of each in the context of the larger program overall. We have done that step for you here. We are requiring you to have at least the following four classes in your solution, with the responsibilities described. You are allowed to add more classes to your design as you see fit. The four, with their overall responsibilities described, are:

`WordFinder`
: This contains the `main` method. This class will have a main that's responsible for processing the command-line argument, and handling any error processing. It will probably also have the main command loop. Most of the other functionality will be delegated to other object(s) created in `main` and their methods.

`Rack`
: This corresponds to the idea of the rack in the problem description. Thus, wherever your program is using a rack, it should be using an object of type Rack. As previously discussed, we have already provided the code for a private static `Rack` method `allSubsets`.

`AnagramDictionary`
: This will contain the dictionary data organized by anagrams. It is required to have at least the two public methods whose headers are given in the starter file. You are allowed to add other methods to this interface. This class was discussed in more detail in the section about it.

`ScoreTable`
: This class has information about Scrabble scores for scrabble letters and words. In scrabble not every letter has the same value. Letters that occur more often in the English language are worth less (e.g., 'e' and 's' are each worth 1 point), and letters that occur less often are worth more (e.g., 'q' and 'z' are worth 10 points each). You may use hard-coded values in its data. Here are all the letter values:

- (1 point)-A, E, I, O, U, L, N, S, T, R
- (2 points)-D, G
- (3 points)-B, C, M, P
- (4 points)-F, H, V, W, Y
- (5 points)-K
- (8 points)- J, X
- (10 points)-Q, Z

This class should work for both upper and lower case versions of the letters, e.g., 'a' and 'A' will have the same score. Hint: You can index an array with a char that is a lower case letter by treating it as an int and subtracting 'a' from it (because the internal numeric codes for letters are all sequential). E.g., If your letter is 'd', ('d' - 'a') = 3 and if it's 'e', ('e' - 'a') = 4.

Although you haven't done much class design yourself, you have seen many examples of well-designed classes in the textbook, lecture, labs, and assignments in this class. We recommend you review the following sections of the textbook that give hints on deciding what classes and methods would make sense for a program design, before you start on your own design: 8.1, 8.2, 12.1, and 12.2 (the last two of these were not in the original readings).

One thing to keep in mind is you want the code that operates on some data to be in the same class that contains that data. One sign that your design doesn't have that feature is if your classes tend to have a lot of `get` and `set` methods and not much else. That would indicate that all the code operating on this data is outside of the class itself.

Hopefully we've made clear the importance of making all instance variables private. But even if you make your data private there are other ways to expose the implementation of your objects. For example, if you have a class that contains an ArrayList, and also provide an accessor method for this ArrayList, it gives clients the ability to change the contents of that arraylist from outside of the object methods, possibly invalidating the object. (We discussed these types of issues and how to cope with them in the lecture discussion of side effects in week 7.)

You are welcome to add additional classes as part of your design. These ones would be designed and implemented by you, of course. If you have more classes, just make sure the additional .java files are in your Vocareum home directory when you submit the assignment. If a class is just used by one other class, you could put it in the same file as that class, or a separate file. If it is used by multiple classes, it should be in its own file. Make sure you discuss these additional classes in your design write-up in your README (including telling us where to find them).

Development hints

As usual, we recommend creating test drivers for any non-trivial class you implement to make it easier to debug your code. That should be pretty easy here, because the classes are somewhat independent from each other. (WordFinder is an exception since it already is a main program.)

You'll want to test your complete program (and your AnagramDictionary) on a small dictionary file *before* subjecting it to sowpods.txt. We provided a sample small dictionary and input and corresponding output for some racks in the `testFiles` directory (more about that in the next paragraph). If you find AnagramDictionary-related bugs, you may want to use an even tinier dictionary for when you are single-stepping, etc.

Once you have all your modules working, you can also check if your program produces the right answers for sowpods.txt with the other test input files and corresponding output in the `testFiles` directory. Note: The `testFiles/README.txt` file describes what's what that directory.

Grading criteria

This program will be graded approximately 2/3 on correctness, 1/3 on design, style, and documentation. As usual we will be using the style guidelines published for the class. There was more about design issues in the section on class design in this document. Another issue that will come into play is good use of the parts of the Java library we have learned about. E.g., it's better to use one of the Java `sort` methods than reimplementing it.

README file / Submitting your program

Your `README` file must document known bugs in your program, contain the signed certification shown near the top of this document, and contain any special instructions or information for the grader.

In addition, for this assignment, your README must also document your design. This includes the approach you took to solving the problem (i.e., description of the data structures and algorithms involved). One part of this was discussed in the section on approach. You will also include there information about how your class design relates to this approach, including what data structures and algorithms are encapsulated in which of your classes.

When you are ready to submit the assignment press the big "Submit" button in your PA4 Vocareum work area. Because you may have additional files in your program, it will try to compile all files in your work area, and test the resulting program on the small dictionary data we gave you in `testFiles` (not on sowpods). As usual, you will want to submit for the first time well before the final deadline, so you have time to fix any errors you get on the submit script.

Passing these submit checks is not necessary or sufficient to submit your code (the graders will get a copy of what you submitted either way). (It would be necessary but not sufficient for getting full credit.) However, if your code does not pass all the tests we would expect that you would include some explanation of that in your README. One situation where it might fail would be if you only completed a subset of the assignment (and your README would document what subset you completed.)
