CS 314 - Specification 7 - Anagrams

Programming Assignment 7 Individual Assignment. You must complete this assignment on your own. You may not acquire from any source (e.g. another student or an internet site) a partial or complete solution to a problem or project that has been assigned. You may **not** show another student your solution to an assignment. You **may not** have another person (current student, former student, tutor, friend, anyone) "walk you through" how to solve an assignment. You may get help from the instructional staff. You may discuss general ideas and approaches with other students but you may not develop code together. Review the class policy on collaboration from the syllabus.

- Placed online: Tuesday, February 28
- 20 points, ~2% of final grade.
- Due: no later than 11 pm, Thursday, March 9
- General Assignment Requirements

The purposes of this assignment are:

1. to use various data structures
2. to implement a program that uses multiple classes
3. to implement a recursive backtracking algorithm
4. to improve the efficiency of a brute force recursive backtracking algorithm through creative approaches

Thanks to Stuart Reges for sharing this assignment with me.

In this assignment you will implement several classes that allow a user to find anagrams of words and phrases they type in. **You are not allowed to create a multi threaded application for this assignment.**

Summary: An anagram is formed by taking all of the letters in one word or phrase and scrabbling them to form a new word or phrase. All of the letters from the original word / phrase must be used and no letters can be added. The new word or words must be valid based on some dictionary. For example, an anagram of the phrase "**`Isabelle Scott`**" is "**`obstacle stile`**".

There are many online anagram solvers. One of the better ones can be found at http://wordsmith.org/anagram/index.html.

See the guide below for suggestions on how to complete the assignment. **We expect you to implement the anagram finding algorithm as outlined in the video explanation on Canvas. Do NOT search the web for algorithms to "find anagrams" for this assignment.**

There is also a video on Canvas explaining the recursive backtracking approach you are expected to use to solve this problem. Canvas -> CS314 -> Files -> Assignment Help Videos -> A7_Anagrams.mp4

Files:

Source Code	AnagramMain.java The driver program for the completed anagram solver.	Provided by me. Do not alter except for printing out time results if you would like.
Source Code	LetterInventory.java A class that represents a collection of letters from the English alphabet. **You must add the standard header with your information and the academic honesty statement. Failure to do so will cause you to lose points on the assignment.**	Provided by you.
Source Code	AnagramFinderTester.java. A class with tests for the LetterInventory and your AnagramSolver classes. Add at least 2 tests per public method and the constructor in LetterInventory.java. (Delete the provided tests after you are sure you code passes them.) The tester requires the d3.txt file for the dictionary and this file with the expected results. Here is the expected output of the tester without the anagrams shown and with the anagrams shown. Your times will vary. **Any of the given tests that complete in less than 0.2 seconds should take less than 1 second on the CS department machines. Any of the given tests that take >= 0.2 seconds should take no more than 5X my time when run on the CS department machines.** Sometimes solutions are slow because you have missed a base case and made too many calls to your recursive helper method. Here is some test output with the number of calls made to the recursive helper method. Note, your number of calls DOES NOT have to match those shown. Finally, there is a very clever solution that can significantly improve the time it takes to find some anagrams. Here is the test output for that approach and the number of calls made to the recursive helper method. **It is up to you to explore various approaches to try and speed up your algorithm. There is no one right way to do this. It is up to you to try different things. Neither the TAs, nor the instructor, have a "magic answer" that works for all cases. I want to stress this. Don't come to lab hours expecting us to tell you exactly how to speed up your program. We will give high level suggestions at best, not detailed advice. It is up to you to create ideas and test them out.**	Provided by me.
Source Code	AnagramSolver.java. A class that solves and returns a list of anagrams for a given phrase and maximum number of words.	Provided by you
Source Code	Stopwatch.java. A class for calculating elapsed time when running other code. You may use this to see how long it takes to find anagrams and compare results on Piazza.	Provided by me.
Dictionary Files	d1.txt A small dictionary file with 56 words. d2.txt A medium size dictionary file with 3927 words. d3.txt A large dictionary file with 19911 words. **Note, when we test your program, we will use other dictionaries that contain one and two letter words. Do not assume all words in the dictionary will have a length of 3 or greater. There can and will be words of length 1 or 2 in the dictionaries we use to test your solution.**	Provided by me.
Sample Output	Sample run of anagram solver using d1.txt. Other than the timing results, your program must match this output given the same input. I have trimmed many of the anagrams for clarity, but when you run AnagramMain it prints out all the result. The output also shows the timing data for the sample solution and the tests in AnagramFinderTester. Here is a run of the anagram solver using d3.txt with all the output.	Provided by me.
Submission	Turn in a7.zip with these three files: AnagramFindrerTester.java, AnagramSolver.java, LetterInventory.java.	Provided by you.

Submission: Complete the header in AnagramFinderTester.java and AnagramSolver.java. Copy it to LetterInventory.java a Replace <NAME> with your name. Note, you are stating, on your honor, that you did the assignment on your own.

Create a zip file name a7.zip with your AnagramFinderTester.java, LetterInventory.java, and AnagramSolver.java files. The zip file must **not** contain any directory structure, just the required file.

See this page for instructions on how to create a zip via Eclipse.

Turn in a7.zip via your Canvas account to programming assignment 7.

- Ensure you files are named AnagramFinderTester.java, LetterInventory.java, and AnagramSolver.java. Failure to do so will result in points off.
- Ensure AnagramFinderTester.java, LetterInventory.java, and AnagramSolver.java are part of the default package. Do not add a `package` statement to either file.
- Ensure your zip has no internal directory structure. When the file is unzipped no directories (folders) are created. (Note, the built in unzip feature of some versions of Windows and Apple OS X "help" by adding a directory when you unzip with the same name as the file. The unzip we use on the CS Linux system does not do this. Neither do unzip programs such as 7zip.)
- Ensure you submit the assignment under Programming Assignment 7 in Canvas.

Checklist: Did you remember to:

1. review and follow the general assignment requirements?
2. follow the specific requirements and restrictions for this assignment?
3. work on the assignment by yourself?
4. fill in the header in AnagramFinderTester and copy it to LetterInventory.java and AnagramSolver.java?
5. ensure your LetterInventory and Anagram Solver classes pass the tests in AnagramFindrerTestert? Add at least 2 tests per public method and constructor (yes this assignment requires tests of the constructor) for LetteInventory?
6. complete the AnagramSolver and implement a recursive backtracking algorithm with the required efficiency improvements?
7. ensure your program matches the sample output file when run?
8. Turn in a file name a7.zip file with your Java source code files LetterInventory.java, AnagramSolver.java, and AnagramFinderTester.java to assignment 7 on Canvas by 11 pm, Thursday, March 9

Specification:

The LetterInventory class: Implement a class to model a letter inventory. A letter inventory object stores the number of times each English letter, 'a' through 'z' occur in a word or phrase. So for example the letter inventory of the word **"tall"** is

1 a, 2 l's, and 1 t

The letter inventory for **"Isabelle M. Scott!!"** is

1 a, 1 b, 1 c, 2 e's, 1 i, 2 l's, 1 m, 1 o, 2 s's, and 2 t's

Notice the letter inventory ignores characters that are not English letters. Also notice that the letter inventory is case insensitive. **"Isabelle M. Scott!!"** has 2 s's.

Implement the LetterInventory class. **You must use an array of ints to store the number of times each English letter occurs**. The inventory is case insensitive so the length of the array will equal the length of the alphabet being used, in this case 26. Use a class constant for the value 26. The class keeps track of the total number of letters in the inventory so this value can be returned quickly without adding up all the individual counters.

Provide the following constructors and methods:

- a **constructor** that accepts a String. The precondition requires the String to not be null. A LetterInventory is constructed based on the String ignoring case. This method shall be no worse than O(N + M) where N is the number of characters in the String and M is the number of letters in the alphabet our LetterInventory is using. Recall you can test if a given character is a letter between 'a' and 'z' with the following boolean expression: 'a' <= ch && ch <= 'z'.
- a method named **get** that accepts a char and returns the frequency of that letter in this LetterInventory. The precondition requires that the char be an English letter. It may be an upper or lower case letter. The answer returned must be the same given the upper or lower case version of a letter. This method shall be O(1).
- a method named **size** that returns the total number of letters in this LetterInventory. This method shall be O(1).
- a method named **isEmpty** that returns true if the size of this LetterInventory is 0, false otherwise. This method shall be O(1) with respect to the size of the alphabet.
- a method named **toString** that returns a String representation of this LetterInventory. All letters in the inventory are listed in alphabetical order. For example if a LetterInventory is created from the String "tall" toString returns the String `"allt"`. If a LetterInventory is created from the String **"Isabelle M. Scott!!"** the method returns the String `"abceeillmosstt"`.
- a method named **add** with one parameter, another LetterInventory object. The method returns a new LetterInventory created by adding the frequencies from the calling LetterInventory object to the frequencies of the letters in the explicit parameter. The precondition requires that the LetterInventory sent as an explicit parameter not be null. The postcondition requires that neither the calling object or the explicit parameter are altered as a result of this method call. This method shall be O(M) where M is the number of letters in the alphabet our LetterInventory is using.
- a method named **subtract** with one parameter, another LetterInventory object. The methods returns a new LetterInventory object created by subtracting the letter frequencies of the explicit parameter from the calling object's (**`this`**) letter frequencies.

  So l**`et1.subtract(let2)`** is analogous to **let1 - let2**

  If any of the resulting letter counts are less than 0 the method shall return null. the precondition requires the LetterInventory sent as an explicit parameter not be null. The post-condition requires that neither the calling object or the explicit parameter are altered as a result of this method call. This method shall be O(M) where M is the number of letters in the alphabet our LetterInventory is using.
- **override** [override, override, override! Do not overload equals. **The parameter shall be of be Object.**] the **equals** method from Object. Two LetterInventorys are equal if the frequency for each letter in the alphabet is the same.

You may add other private methods and constructors if you wish. Note, `LetterInventory` objects are immutable. There are no methods that alter a given `LetterInventory` object after it is created.

When you implement the anagram solver you will not use all of these methods, but you are required to complete the class. Your completed `LetterInventory` class must pass all of the tests in `AnagramFindrerTester`. You must add at least 2 tests per constructor and public method in `LetterInventory` to the `AnagramFindrerTester` class.

The Anagram Solver class: Now that you have a LetterInventory class to handle the low level details, implement a class to find all possible anagrams for a given phrase. (Realize that the legal anagrams for a phrase will vary based on what dictionary is used, the maximum number of words allowed in the anagram, and whether words can be repeated in the anagram or not.) For example the anagrams of "Isabelle Scott" using the dictionary in file d3.txt and a limit of no more than 2 words per anagram are:

```
best, oscillate
bestial, closet
bleat, solstice
blest, societal
closet, stabile
obstacle, stile
solstice, table
```

Note, there are two special cases your do not have to worry about.

- First, a given word or phrase **is** considered an anagram of itself if the word is in the dictionary. So "dog" is an anagram of "dog"
- Second, you may use words more than once in the anagram. So for example one of the anagrams of "dog dog" is "god god".

Provide the following for the `AnagramSolver` class:

- a public constructor that has one parameter, a Set<String>. (A set of strings.) The set contains the dictionary this AnagramSolver is to use when forming anagrams. This constructor creates and stores letter inventories for each element of the set. Store the original words in the dictionary and their corresponding letter inventories inside the AnagramSolver class. You want to be able to access the letter inventory of a given word quickly, therefore store each word and its associated LetterInventory in a Map.

  - your class must create the `LetterInventorys` for a given word in the dictionary exactly one time. (Doing it more than once is an inefficiency you must avoid in this assignment.)

  Although none of the provided dictionaries contain words of length 1 or 2, your program must be able to handle dictionaries that do. This isn't a special case, but in the past students have tried to optimize their code assuming there won't be any words of length 1 and then failed tests on dictionaries that contained words with a length of 1 or 2. I recommend adding words to the dictionary files of length 1 and 2 to test your program. You can share results of these tests on Piazza.

- a public method named **getAnagrams** that has two parameters: a String, and an int. The String is the target word or phrase and your method will form anagrams out of that word or phrase. The String sent as a parameter does not have to be in the AnagramSolver's dictionary. It may contain characters which must be ignored. (Recall this is handled by the LetterInventory class already.) The int parameter indicates the maximum number of words allowed in the anagrams of the given String. If the int is 0 there is no maximum number of words for the anagrams being formed. The precondition requires the String not be null and the String contain at least one English letter. The int parameter must be greater than or equal to 0.

  The method returns a `List<List<String>>`. In other words a List of Lists of Strings. `List` is a Java interface The `ArrayList` class implements the `List` interface so you could return an `ArrayList<ArrayList<String>>`. Don't alter the return type from `List<List<String>>`. Returning an `ArrayList<List<String>>` will work because `ArrayList` is a kind of `List`. It implements the `List` interface. Note, you will create a variable of type `ArrayList<List<String>>` and add new variables of type `ArrayList<String>` to it.

You can and should add private helper methods as necessary to break the problem up into manageable parts. **By way of comparison my AnagramSolver class, including a nested class for a Comparator, is roughly 135 lines including comments, blank lines and lines with a single brace.**

Finding the anagrams of a given String is a classic recursive backtracking problem similar to the N queens problem, the phone mnemonic problem, and the fair teams problem. **I think this problem is most similar to the phoneMnemonics problem from assignment 6. There are of course differences in the details, but the basic structure is quite similar.**

You shall complete some preprocessing each time your anagram finding method is called. In most cases, many of the words in the dictionary cannot be part of an anagram for the given word or phrase. For example the word "queen" will not be in any anagram of "Isabelle Scott" because there is no 'q' in "Isabelle Scott". Likewise the word "casa' cannot be part of an anagram of "Isabelle Scott: because there are two 'a's in "casa" but only 1 'a' in "Isabelle Scott". You will, of course, use the LetterInventory class to simplify the task of determining if a given word could possibly be part of an anagram. (The subtract method is VERY useful.)

Creating this smaller list of words from the dictionary (we can access the corresponding LetterInventories from our original map that is an instance variable) shall not alter the AnagramSolver's main dictionary. Once this smaller list of words is created you are ready to carry out the recursive backtracking method to find all anagrams of the given String. **I recommend using the smaller list of words as your "choices" for the recursive step.**

You can also improve your efficiency by **virtually** shrinking the dictionary of possible words as your proceed deeper into the recursion. I don't not recommend trying to actually shrink the dictionary at each step.

Note that the returned `List<List<String>>` must have the following properties:

- The words in a particular anagram are in sorted order
- The list does not contain duplicate anagrams
- The list of anagrams is sorted with anagrams with the fewest words coming first. In other words all anagrams with 1 word come before all of the anagrams with 2 words which come before all the anagrams with 3 words and so forth. Also, the anagrams with the same number of words are sorted based on the Strings within the anagram. Refer to the sample output for examples.

Accomplishing all that sorting can be a chore. You might try to do it as you go or simply wait until you have found all the anagrams and then process it. The second option may be a cleaner solution, because it separates the recursive backtracking from the sorting. You are free to use methods from the Collections class (not the Collection interface, the Collections class. With an **s**. The Collections class contains a number of static methods that perform common task on various kinds of Collection objects.) to make this easier as well as other data structures.

Initially there is no way to sort a list of anagrams. Lists and ArrayLists are not Comparable. One approach would be to create a class that implements the Comparator interface. The Comparator interface allows you to define how two objects should be compared if they are not Comparable. You will need a separate class, but don't put in in a separate file. Instead, you can make your class that implements the Comparator interface a nested class, similar to the iterators you have implemented, except this class will be declared static because it doesn't need to know about the outer class.

```
public class AnagramSolver {
    // code for AnagramSolver

    // example of a nested class
    private static class AnagramComparator implements Comparator<List<String>> {
        public int compare(List<String> a1, List<String> a2) {
            // code for compare
        }
    } // end of AnagramComparator

} // end of AnagramSolver class
```

The compare method for the Comparator interface is similar to the compareTo method for Comparable. If a1 is "less than" a2 return a negative int. If a1 "equals" a2 return 0. If a1 is "greater than" a2 return a positive int.

When you create a class that implements the Comparator interface correctly you can call the sort method in the Collections class that takes in a list and a comparator to sort the List of Lists of Strings.

Back to the CS 314 homepage.