CSE143—Computer Programming II
Programming Assignment #6
due: Thursday, 5/20/21, 11 pm

This assignment will give you practice with recursive backtracking. You are to create a class called AnagramSolver that uses a dictionary to find all combinations of words that have the same letters as a given phrase (see the sample log for examples).

Your class must include the following public methods.

Method	Description
AnagramSolver(List<String> list)	This method constructs an anagram solver that will use the given list as its dictionary. You should not change the list in any way. You may assume that the dictionary is a nonempty collection of nonempty sequences of letters, that it contains no duplicates, and that it doesn't change in state as the program executes.
void print(String s, int max)	This is your method that will use recursive backtracking to find combinations of words that have the same letters as the given string. It should print to System.out all combinations of words from the dictionary that are anagrams of s and that include at most max words (or unlimited number of words if max is 0). It should throw an IllegalArgumentException if max is less than 0.

Your print method must produce the anagrams in the same format as in the sample log. The easiest way to do this is to build up your answer in a structure like a list or a stack. Then you can simply "println" the structure and it will have the appropriate format.

You are required to solve this problem by using recursive backtracking. In particular, you are to write a recursive method that builds up an answer one word at a time. On each recursive call, you are to search the dictionary from beginning to end and to explore each word that is a match for the current set of letters. The possible solutions are to be explored in dictionary order. For example, in deciding what word might come first, you are to examine the words in the same order in which they appear in the dictionary.

The solution to the 8 queens problem provides a good example to follow for your own code. An important aspect of the 8 queens solution was the separation of the recursive code (Queens.java) from the code that managed low-level details of the problem (Board.java). You are required to follow a similar strategy here. The low-level details for the anagram problem involve keeping track of various letters and figuring out when one group of letters can be formed from another group of letters. It turns out that the LetterInventory class that we wrote for assignment 1 provides us with the low-level support we need (review handout #2 to remind yourself of the available methods).

For any given word or phrase, what matters to us is how many of each letter there are. Recall that this is exactly what the LetterInventory keeps track of. In addition, the "subtract" method of the LetterInventory is the key to solving this problem. For example, if you have a LetterInventory for the phrase "george bush" and ask whether or not you can subtract the LetterInventory for "bee", the answer is yes. Every letter in the "bee" inventory is also in the "george bush" inventory. That means that you need to explore this possibility. Of course, the word "bee" alone is not enough to account for all of the letters of "george bush", which is why you'd want to work with the new inventory formed by subtracting the letters from "bee" as you continue the exploration.

Part of your grade will be based on the efficiency of your solution. Recursive backtracking is, in general, highly inefficient because it is a brute force technique that checks every possibility, but there are still things you can do to make sure that your solution is as efficient as it can be. Be careful not to compute something twice if you don't need to. And don't continue to explore branches that you know will never be printed. And you are required to implement the following two optimizations:

- There is no reason to convert dictionary words into inventories more than once. You should "preprocess" the dictionary in your constructor to compute all of the inventories in advance (once per word). You'll want fast access to these inventories as you explore the possible combinations. A map will give you fast access. Remember that we used the combination of the SortedMap interface and the TreeMap implementation for the grammar solver. For that program, it was important to keep the keys in sorted order. That is not true for this program, so you should instead use the more generic Map interface and the slightly faster HashMap implementation (do not use a LinkedHashMap).

- For any given phrase, you can reduce the dictionary to a smaller dictionary of "relevant" words. A word is relevant if it can be subtracted from the given phrase. Only a fraction of the dictionary will, in general, be relevant to any given phrase. So reducing the dictionary before you begin the recursion will allow you to speed up the searches that happen on each recursive invocation. To implement this, you should construct a short dictionary for each phrase you are asked to explore that includes just the words relevant to that phrase. You'll do this once before the recursion begins, not on each recursive call. Students who want to prune the dictionary on each recursive call are allowed to do so, but keep in mind that it is not required and it might make the code more difficult to write. If you decide to prune on each recursive call, clearly document that you are doing so.

In terms of correctness, your class must provide all of the functionality described above. In terms of style, we will be grading on your use of comments, good variable names, consistent indentation and good coding style to implement these operations. Remember that you will lose points if you declare variables as data fields that can instead be declared as local variables. You should also be making correct use of generics and you should declare fields and variables using interfaces when possible (e.g., using List<String> for your variable even if the object is of type ArrayList<String>). You should also avoid extraneous cases (e.g., don't make something into a special case if it doesn't have to be) and should be careful to make your code as efficient as possible.

The constructor for your class is passed a reference to a dictionary stored as a List of String objects. You can use this dictionary within your class as long as you don't change it (that is expressly forbidden in the specification). In other words, you should *not* make your own independent copy of the dictionary.

Don't make this problem harder than it needs to be. You are doing a fairly exhaustive search of the possibilities. You have to avoid dead ends and you have to implement the optimizations listed above, but otherwise you are exploring every possibility. For example, in one of the sample logs you will see that one solution for "Barbara Bush" is [abash, bar, rub]. Because this is found as a solution, you know that every other permutation of these words will also be included ([abash, rub, bar], [bar, abash, rub], [bar, rub, abash], and so on). But you don't have to write any special code to make that work. That is a natural result of the exhaustive nature of the search. It will locate each of these possibilities and print them out when they are found. Similarly, you don't need any special cases for words that have already been used. If someone asks you for the anagrams of "bar bar bar", you should include [bar, bar, bar] as an answer.

You should name your file AnagramSolver.java and you should turn it in electronically from the "assignments" link on the class web page. A collection of files needed for the assignment is included on the web page as ass6.zip. You will need to have AnagramMain.java and LetterInventory.class in the same directory as AnagramSolver.java in order to run AnagramMain. The folder also contains four dictionary files called dict1.txt (short dictionary of 56 words that has lots of matches for the Bush family—appropriate for testing), dict2.txt (medium sized dictionary of approximately 4 thousand words), dict3.txt (large dictionary of approximately 20 thousand words), and dict4.txt (large dictionary in a different order).

Your program is to produce exactly the same output in exactly the same order as in the sample logs of execution that are on the class web page. The output comparison tool available from the class web page includes the sample executions if you want to compare your results to the expected output.

We strongly recommend that you ignore the map and pruning while you are first writing your program. Construct LetterInventory objects whenever you need them and use the entire dictionary, even though that approach is inefficient. That way you can focus on the backtracking first and make sure that you have a correct solution. Then modify your program to use the map to avoid recomputing LetterInventory objects and introduce pruning to limit the number of words explored. You can verify that pruning is working by printing the size of the original dictionary and the pruned dictionary. For example, when processing "george bush" on dict1.txt, you go from a dictionary size of 56 to a pruned size of 31.

For those using eclipse, the zip file includes LetterInventory.jar. To add the jar file to your project, select your project, go to the Projects menu and select Properties, then Java Build Path, Libraries and select "add External JARs." When you addLetterInventory.jar to the build path, everything should work.

Sometimes this program produces a lot of output. When you run it in jGRASP, it will display just 500 lines of output. If you want to see more, go to the Build menu and select the "Run in MSDOS Window" option. Then when the window pops up, right-click on the title bar of the window, select Properties, and under the "Layout" tab you should be able to adjust the "Screen Buffer Size" Height to something higher (like 9999 lines).

Sample execution (user input underlined)

```
Welcome to the cse143 anagram solver.

What is the name of the dictionary file? dict3.txt

phrase to scramble (return to quit)? Howard Dean
Max words to include (0 for no max)? 2
[ahead, drown]
[don, warhead]
[drown, ahead]
[hadron, wade]
[head, onward]
[nod, warhead]
[onward, head]
[wade, hadron]
[warhead, don]
[warhead, nod]

phrase to scramble (return to quit)? Donald TRUMP
Max words to include (0 for no max)? 2
[doldrum, pant]
[mud, portland]
[pant, doldrum]
[portland, mud]

phrase to scramble (return to quit)?
```