CSE143—Computer Programming II
Programming Assignment #5
due: Saturday, 8/3/13, 9 pm

This program will give you practice with recursive backtracking. Turn in a file named `AnagramSolver.java` from the Homework section of the course web site. You will need the support files `AnagramMain.java`, `LetterInventory.class`, and various dictionary text files from the Homework section of the course web site; place them in the same folder as your class. All files can be downloaded together in `hw5.zip`.

Program Description:

An *anagram* is a word or phrase made by rearranging the letters of another word or phrase. For example, the words "midterm" and "trimmed" are anagrams. If you ignore spaces and capitalization and allow multiple words, a multi-word phrase can be an anagram of some other word or phrase. For example, the phrases "Clint Eastwood" and "old west action" are anagrams.

In this assignment, you will create a class called `AnagramSolver` that uses a dictionary to find all anagram phrases that match a given word or phrase. You are provided with a client program `AnagramMain` that prompts the user for phrases and then passes those phrases to your `AnagramSolver` object. It asks your object to print all anagrams for those phrases. Below is a sample log of execution (user input is 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)? Wesley Clark
Max words to include (0 for no max)? 2
[creaky, swell]
[seller, wacky]
[swell, creaky]
[wacky, seller]

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

Implementation Details:

Your `AnagramSolver` class must have the following public constructor and methods:

`public 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 and that it contains no duplicates.

`public void print(String phrase, int max)`

This is the method that will use recursive backtracking to find combinations of words that have the same letters as the given phrase. It should print to `System.out` all combinations of words from the dictionary that are anagrams of `phrase` and that include at most `max` words (or unlimited number of words if `max` is 0). For example, if your anagram solver is using the dictionary corresponding to `dict1.txt` and this method is passed a phrase of `"hairbrush"` and a max of 2, your method should produce the following output:

```
[briar, hush]
[hush, briar]
```

You 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 above. The easiest way to do this is to build up your answer in a `List` or `Stack` or `Queue`. Then you can simply `println` the structure and it will have the appropriate format. If you use a `Stack` or `Queue`, you are not restricted to the short list of methods that were available for homework 2 and on the midterm.

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

The solution to the 8-queens problem provides a good example to follow for your own code. The primary difference in the recursion is that in 8-queens, we stopped as soon as we found an answer, whereas in the anagrams program you are to produce all answers.

LetterInventory Class:

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 the assignment 1 specification 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.

Efficiency:

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. Don't continue to explore branches that you know will never be printed. In addition, 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. We don't need to keep the keys in sorted order for this program, so you should use the faster `HashMap` implementation.
- 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.

Some people try to use their `Map` in place of their dictionary. There is some space efficiency to this in that the `Map` will store all of the dictionary words as keys of the `Map`, but this isn't really an appropriate use of the `Map`. Plus, the dictionary might have a different order than what the `Map` uses and you are required to explore the possibilities in dictionary order.

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.

Development Strategy and Hints:

We provide you with four dictionary files. We suggest that you start testing with the smallest dictionary (or create your own) and then progress to the larger files when you feel confident in your implementation. The dictionaries are `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).

One difficult part of this program is limiting the number of words that can appear in the anagrams through the `max` parameter. We suggest you do this part last, initially printing all anagrams regardless of the number of words.

While developing your program, 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 add `LetterInventory.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).

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.

Style Guidelines and Grading:

Part of your grade will come from appropriately utilizing recursive backtracking to implement your algorithm as described previously. We will also grade on the elegance of your recursive algorithm; don't create special cases in your recursive code if they are not necessary or repeat cases already handled. Redundancy is another major grading focus; some methods are similar in behavior or based off of each other's behavior. You should avoid repeated logic as much as possible. Your class may have other methods besides those specified, but any other methods you add should be `private`.

You should follow good general style guidelines such as: making fields `private` and avoiding unnecessary fields; declaring collection variables using interface types and generics (e.g. `List<String>`); appropriately using control structures like loops and `if/else`; properly using indentation, good variable names and types; and not having any lines of code longer than 100 characters.

Comment your code descriptively in your own words at the top of your class, each method, and on complex sections of your code. Comments should explain each method's behavior, parameters, return, pre/post-conditions, and exceptions.