CSE 143, Summer 2016
Programming Assignment #5: Anagrams (30 points)
Due Thursday, August 4, 2016, 11:30 PM

This program focuses on recursive backtracking. Turn in a file named `Anagrams.java` from the Homework section of the course web site. You will need support files `AnagramMain.java` and various dictionary text files from the Homework section of the course web site; place them in the same folder as your class.

Anagrams:

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 `Anagrams` 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 `Anagrams` object. It asks your object to print all anagrams for those phrases. Below is a sample log of execution, wrapped into two columns (user input is underlined):

```
Welcome to the CSE 143 anagram solver.
What is the name of the dictionary file? dict1.txt

Phrase to scramble (Enter to quit)? barbara bush
All words found in "barbara bush":
[abash, aura, bar, barb, brush, bus, hub, rub, shrub, sub]

Max words to include (Enter for no max)?

[abash, bar, rub]
[abash, rub, bar]
[bar, abash, rub]
[bar, rub, abash]
[rub, abash, bar]
[rub, bar, abash]


Phrase to scramble (Enter to quit)? john kerry
All words found in "john kerry":
[he, her]

Max words to include (Enter for no max)?



Phrase to scramble (Enter to quit)? hairbrush
All words found in "hairbrush":
[bar, briar, brush, bus, hub, huh, hush, rub, shrub, sir, sub]

Max words to include (Enter for no max)? 2

[briar, hush]
[hush, briar]


Phrase to scramble (Enter to quit)? george bush
All words found in "george bush":
[bee, beg, bog, bogus, bough, brush, bug, bugs, bus, egg, ego,
erg, go, goes, gorge, gosh, grub, gush, he, her, here, hog,
hose, hub, hug, rub, she, shrub, shrug, sub, surge]

Max words to include (Enter for no max)? 0

[bee, go, shrug]
```

(continued on right)

```
[bee, shrug, go]
[bog, he, surge]
[bog, surge, he]
[bogus, erg, he]
[bogus, he, erg]
[bug, erg, hose]
[bug, goes, her]
[bug, her, goes]
[bug, hose, erg]
[bugs, ego, her]
[bugs, go, here]
[bugs, her, ego]
[bugs, here, go]
[bus, erg, go, he]
[bus, erg, he, go]
[bus, go, erg, he]
[bus, go, he, erg]
[bus, gorge, he]
[bus, he, erg, go]
[bus, he, go, erg]
[bus, he, gorge]
[egg, hose, rub]
[egg, rub, hose]
[ego, bugs, her]
[ego, grub, she]
[ego, her, bugs]
[ego, she, grub]
[erg, bogus, he]
[erg, bug, hose]
[erg, bus, go, he]
[erg, bus, he, go]
[erg, go, bus, he]
[erg, go, he, bus]
[erg, go, he, sub]
[erg, go, sub, he]
[erg, goes, hub]
[erg, he, bogus]
[erg, he, bus, go]
[erg, he, go, bus]
[erg, he, go, sub]
[erg, he, sub, go]
[erg, hose, bug]
[erg, hub, goes]
```

. . . (full output is on web site)

Implementation Details:

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

`public Anagrams(Set<String> dictionary)`

In this constructor you should initialize a new anagram solver over the given dictionary of words. Do not modify the set passed to your constructor.

You should throw an `IllegalArgumentException` if the set passed is `null`.

`public SortedSet<String> getWords(String phrase)`

In this method you should return a set containing all words from the dictionary that can be made using some or all of the letters in the given phrase, in alphabetical order. For example, if your anagram solver is using the dictionary corresponding to `dict1.txt` and you are passed the phrase `"Barbara Bush"`, you should return a set containing the elements `[abash, aura, bar, barb, brush, bus, hub, rub, shrub, sub]`.

You should throw an `IllegalArgumentException` if the string is `null`.

`public void print(String phrase)`

In this method you should use recursive backtracking to find and print all anagrams that can be formed using all of the letters of the given phrase, in the same order and format as in the example log on the previous page. For example, if your anagram solver is using the dictionary corresponding to `dict1.txt` and you are passed the phrase `"hairbrush"`, your method should produce the following output:

```
[bar, huh, sir]
[bar, sir, huh]
[briar, hush]
[huh, bar, sir]
[huh, sir, bar]
[hush, briar]
[sir, bar, huh]
[sir, huh, bar]
```

You should throw an `IllegalArgumentException` if the string is `null`. An empty string generates no output.

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

In this method you should use recursive backtracking to find and print all anagrams that can be formed using all of the letters of the given phrase and that include at most `max` words total, in the same order and format as in the example log on the previous page. 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]
```

If `max` is 0, print all anagrams regardless of how many words they contain. For example, if using the same dictionary and passed a phrase of `"hairbrush"` and a max of 0, the output is the same as that shown earlier on this page with no max.

You should throw an `IllegalArgumentException` if the string is `null` or if the max is less than 0. An empty string generates no output.

The provided `AnagramMain` program calls your methods in a 1-to-1 relationship, calling `getWords` every time before calling `print`. But you should not assume any particular order of calls by the client. Your code should still work if the methods are called in any order, any number of times.

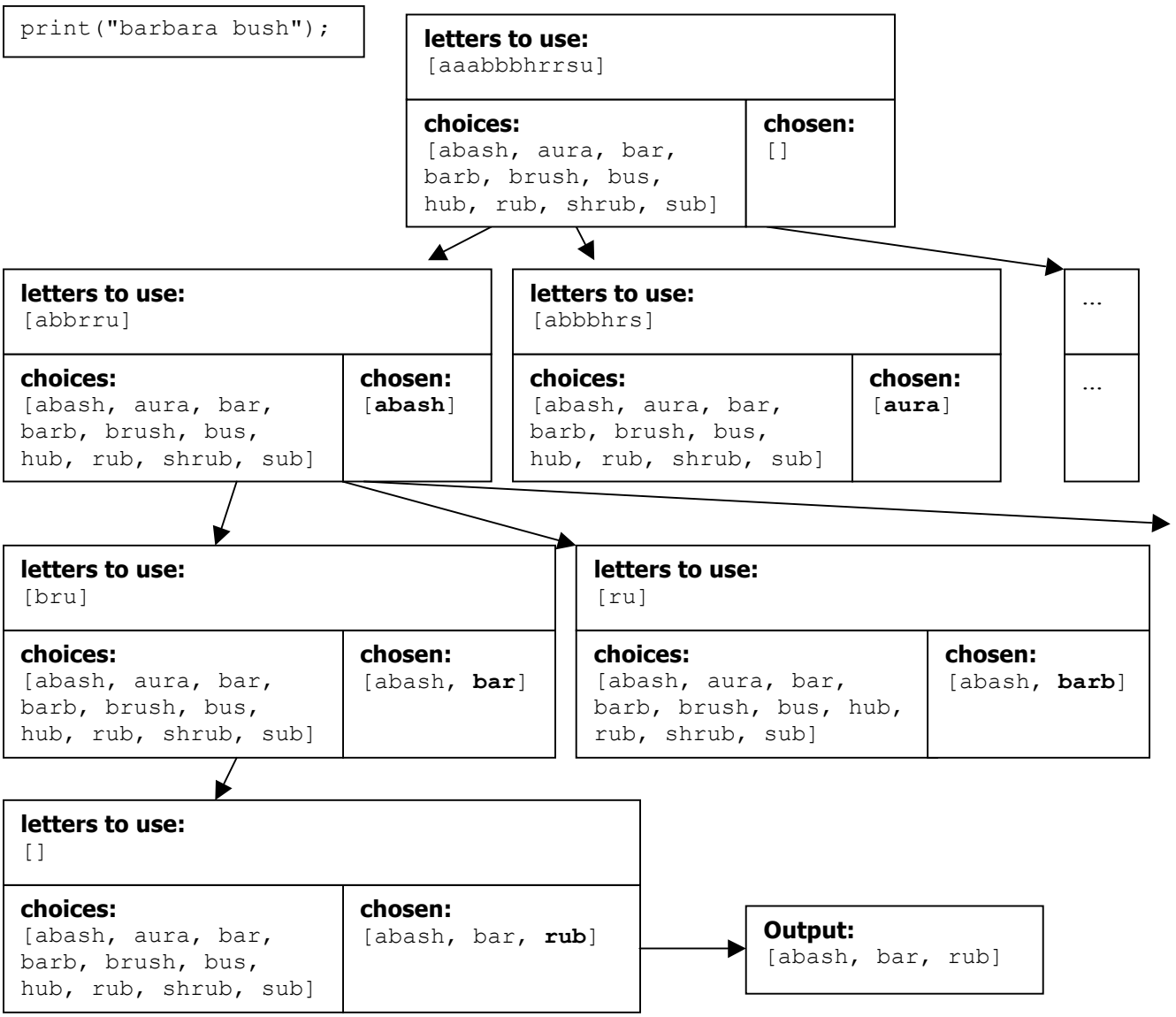
Recursive Algorithm:

Generate all anagrams of a phrase using recursive backtracking. Many backtracking algorithms involve examining all combinations of a set of choices. In this problem, the choices are the words that can be formed from the phrase. A "decision" involves choosing a word for part of the phrase and recursively choosing words for the rest of the phrase. If you find a collection of words that use up all of the letters in the phrase, it should be printed as output.

Part of your grade will be based on efficiency. One way to implement this program would be to consider every word in the dictionary as a possible "choice." However, this would lead to a massive decision tree with lots of useless paths and a slow program. Therefore for full credit, to improve efficiency when generating anagrams for a given phrase, you must first find the collection of words contained in that phrase and consider only those words as "choices" in your decision tree. You should also implement the second `print` method so that it backtracks immediately once it exceeds the max.

The following diagram shows a partial decision tree for generating anagrams of the phrase `"barbara bush"`. Notice that some paths of the recursion lead to dead ends. For example, if the recursion chooses `"aura"` and `"barb"`, the letters remaining to use are `[bhs]`, and no choice available uses these letters, so it is not possible to generate any anagrams beginning with those two choices. In such a case, your code should backtrack and try the next path.

One difference between this algorithm and other backtracking algorithms is that the same word can appear more than once in an anagram. For example, from `"barbara bush"` you might extract the word `"bar"` twice.



`LetterInventory` Class:

An important aspect of simplifying the solution to many backtracking problems is the separation of recursive code from code that manages low-level details of the problem. We have seen this in several of our backtracking examples, such as 8 queens (recursive code in `Queens.java`, low-level code in `Board.java`). You are required to follow a similar strategy in this assignment. The low-level details for anagrams involve keeping track of letters and figuring out when one group of letters can be formed from another. Recall that this is exactly what the `LetterInventory` keeps track of.

Review the Assignment #1 write up to remind yourself of the available methods.

An instructor provided `LetterInventory.class` is available on the course website if you don't use your own Assignment 1. For those using jGRASP, put your LetterInventory.class in the same folder as your `Anagrams.java`. 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.

Development Strategy and Hints:

Your `print` method must produce the anagrams in the same format as in the log. The easiest way to do this is to build up your answer in a list, stack, or other collection. Then you can `println` the collection and it will have the right format.

The provided `AnagramMain` program can read its input from different dictionary files. It is initially set to use a very small dictionary `dict1.txt` to make testing easier. But once your code works with this dictionary, you should test it with larger dictionaries such as the provided `dict2.txt` and `dict3.txt`. You can find other larger dictionaries here:

- http://www.puzzlers.org/dokuwiki/doku.php?id=solving:wordlists:start

One difficult part of this program is the second version of `print` that limits the number of words that can appear in the anagrams. We suggest you do this part last, initially printing all anagrams regardless of the number of words.

Style Guidelines and Grading:

Part of your grade will come from appropriately utilizing recursive backtracking to implement your algorithm as described previously. 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. In particular, you are required to implement the following optimization.

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 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; 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. For reference, our solution is around 85 lines long including comments and blank lines (36 "substantive" lines).