CSE 143: Computer Programming II — Winter 2019

HW6: AnagramSolver (due Thursday, Feb 28, 2019 11:30pm)

This assignment focuses on recursive backtracking. Turn in the following files using the link on the course website:

- `AnagramSolver.java` – A class that uses a dictionary to find all combinations of words that have the same letters as a given phrase.

You will need the support files `AnagramsMain.java` and the dictionary files; place these in the same folder as your program or project. The code you submit must work properly with the unmodified versions of the provided files.

AnagramSolver

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 print all anagram phrases of a given word or phrase. You will use *recursive backtracking* to implement your algorithm.

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.

`AnagramSolver` should have the following constructor:

`public AnagramSolver(List<String> dictionary)`
This constructor should initialize a new `AnagramSolver` object 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. You should "preprocess" the dictionary in your constructor to compute all of the inventories in advance (once per word).

`AnagramSolver` should also implement the following method:

`public void print(String text, int max)`
This method should use recursive backtracking to find combinations of words that have the same letters as the given string. It should print all combinations of words from the dictionary that are anagrams of `text` and that include at most `max` words (or an unlimited number of words if `max` is 0) to `System.out`. You should throw an `IllegalArgumentException` if `max` is less than 0.

Implementation Details

Using `LetterInventory`

An important aspect of the recursive backtracking solutions is separation of the recursive code from the code that manages low-level details of the problem. Many of the problems we've done so far (8 Queens, see section) have this separation.

In this assignment, you will follow a similar strategy. In the anagrams problem, the low-level details involve keeping track of various letters and figuring out when one group of letters can be formed from another group of letters. Luckily, the `LetterInventory` class we implemented in HW1 turns out to be exactly what we need! You should review the HW1 specification to remind yourself of the available methods, but you should use our provided implementation of `LetterInventory.class` or `LetterInventory.jar`.

The "subtract" method of the `LetterInventory` class is the key to solving this problem. For example, if you have a `LetterInventory` for the phrase "george bush" and ask if you can subtract the `LetterInventory` for "bee", the answer is yes, because every letter in the "bee" inventory is also in the "george bush" inventory. Since `null` is not returned, you need to explore this possibility. 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.

`AnagramSolver` Constructor

There is no reason to convert dictionary words into inventories more than once. You should "pre-process" 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. As usual, a map will give you fast access. In this problem, we don't care about the order of the words in our map, but we *do* care about speed; so, you should make sure to use the `HashMap` implementation.

`print` Algorithm

Your `print` method must produce the anagrams in the same format as in the example execution below (as well as the diff tool). The easiest way to do this is to build up your answer in a `List` or `Stack`. Once you build up a complete answer in your data structure, 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 should write a recursive method that builds up an answer one word at a time. On each recursive call, you should 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 should be explored in dictionary order. To clarify, 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 given dictionary might not be sorted!

Recursive backtracking is inherently highly inefficient since it has to explore every possible option. We are asking you to implement one optimization to speed up the process. For any given phrase, you should 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 should do this once before the recursion begins–not on each recursive call. You *may* continue to prune this smaller dictionary on each recursive call, but keep in mind that it is not required and it will make the code more difficult to write. If you decide to prune on each recursive call, clearly document it.

Development Strategy

We recommend that you first start by writing print without the pruning optimization and ignoring max entirely. Once that works, you should first make it limit to the non-zero max, then deal with the zero case after you get the non-zero case working. After that, you should then worry about pruning.

Full Example Walk-Through

Suppose we have the following dictionary, and we ask the program to print all anagrams of "`eleven plus two`":

eleven.txt

```
zebra
one
plus
won
potato
twelve
```

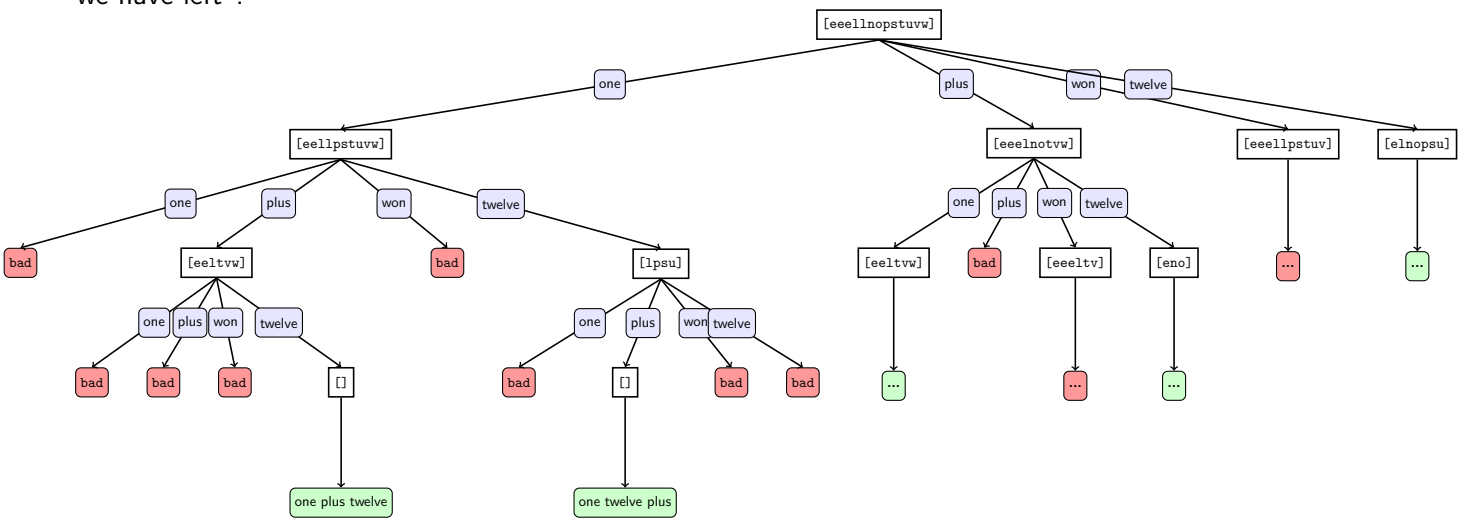
Step 1: Prune The Dictionary

The letters in "`eleven plus two`" do not support all the words in the dictionary. In particular, "potato" and "zebra" contain letters ("a" and "z", respectively) that are not in `elevenplustwo`. So, we prune the dictionary to the following:

```
one
plus
won
twelve
```

Step 2: Find The Words

Now that we've pruned our dictionary, we know what our choices are at each step. (Namely, they're the remaining words in the dictionary.) So, we go through our words recursively, keeping track of "the letters we have left":



Example Execution

```
>> Welcome to the cse143 anagram solver.
>>
>> What is the name of the dictionary file? eleven.txt
>>
>> phrase to scramble (return to quit)? eleven plus two
>> Max words to include (0 for no max)? 0
>> [one, plus, twelve]
>> [one, twelve, plus]
>> [plus, one, twelve]
>> [plus, twelve, one]
>> [twelve, one, plus]
>> [twelve, plus, one]
>>
>> phrase to scramble (return to quit)?
```

Hints and Tips

Use The Provided 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. 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.

Handling `max` is zero

You may not make any assumptions about the length of the input or output when handling the "`max` is 0" case in the `print` method. In particular, do NOT try and simply search for all anagrams where the number of words is smaller then some arbitrarily number since then there is no guarantee your output will always be correct. Instead, try and handle the "`max` is 0" case in a way that has meaning.

Follow The Recursive Backtracking Pattern

Don't make this problem harder than it needs to be. You are doing an 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 the example execution you will see that one solution for "eleven plus two" is [one, plus, twelve]. Because this is found as a solution, you know that every other permutation of these words will also be included ([one, twelve, plus], [plus, twelve, one], etc.). But you don't have to (and should not) write any special code to make that work. This 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.

Testing Pruning

While developing your program, you can verify that pruning is working by printing the size of the original dictionary and the pruned dictionary. This should be doable by hand for the `eleven.txt` dictionary. And, for example, when processing "george bush" on `dict1.txt`, you go from a dictionary size of 56 to a pruned size of 31.

Using `LetterInventory` With Eclipse

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.

jGRASP Output Limit

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).

Style Guidelines and Grading

In terms of external 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.

Part of your grade will come from appropriately utilizing recursive backtracking to implement your algorithm as described previously. Be careful not to compute something twice if you dont need to and dont continue to explore branches that you know will never be printed.

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. Redundancy is another major grading focus; 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.

Avoid Redundancy

Create "helper" method(s) to capture repeated code. As long as all extra methods you create are `private` (so outside code cannot call them), you can have additional methods in your class beyond those specified here. If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code.

Generic Structures and Interfaces

You should always use generic structures. If you make a mistake in specifying type parameters, the Java compiler may warn you that you have "unchecked or unsafe operations" in your program. You should also 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>

Data Fields

Properly encapsulate your objects by making data your fields `private`. Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place. Fields should always be initialized inside a constructor or method, never at declaration.

Java Style Guidelines

Appropriately use control structures like loops and if/else statements. Avoid redundancy using techniques such as methods, loops, and factoring common code out of if/else statements. Properly use indentation, good variable names, and types. Do not have any lines of code longer than 100 characters. Please refer to the Style Guide and the General Style Deductions.