CSE 143E: Computer Programming II — Winter 2022

Take-home Assessment 6: Anagrams — *due February 24, 2022 11:59pm*

This assignment will assess your mastery of the following objectives:

- Implement a well-designed Java class to meet a given specification.
- Implement a recursive backtracking approach to exhaustive search.
- Implement simple optimizations to improve efficiency and avoid unnecessary computation.
- Follow prescribed conventions for code quality, documentation, and readability.

Program Behavior

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 assessment, 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.

We have provided you with a client program, `AnagramMain`, that prompts the user for a phrase and then passes that phrase to your `AnagramSolver`, which will then print all its anagrams.

AnagramSolver

Your `AnagramSolver` class 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).

Your `AnagramSolver` should also implement the following public 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.

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 (n-queens, for example) achieve this separation by using a separate class to manage the state of the current possible solution. (In n-queens, this was the `Board` class that tracked where queens were located and whether or not a placement was safe.)

In this assessment, 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 Assessment 1 turns out to be exactly what we need! You should review the Assesment 1 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, so `subtract` would return a non-null result (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.

Implementation Guidelines

`AnagramSolver` Constructor

You should "pre-process" the dictionary in your constructor to compute the inventories for each word in advance (one 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 be sure to use `HashMap`.

`print` Algorithm

Your `print` method must produce the anagrams in the same format as in the example execution below. 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 print 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. That is, you should examine the words in the same order in which they appear in the dictionary. (Do not make any assumptions about what that order is.)


The given dictionary might not be sorted alphabetically!

Recursive backtracking is inherently inefficient since it has to explore every possible option. To improve on this, you will implement one optimization known as "pruning". For any given phrase, you should reduce the dictionary to a smaller dictionary of relevant words *before* you begin your exhaustive search. A word is relevant if it can be subtracted from the given phrase. In most cases, only a fraction of the dictionary will 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 start by writing `print` without the pruning optimization and ignoring `max` entirely. Once that works, you should then implement a non-zero max, followed by the zero case once the non-zero case is working. Finally, you should add pruning after the rest of the behavior works.

Full Example Walk-Through

Suppose we have the following dictionary (eleven.txt):

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

We ask the program to print all anagrams of the phrase "`eleven plus two`":

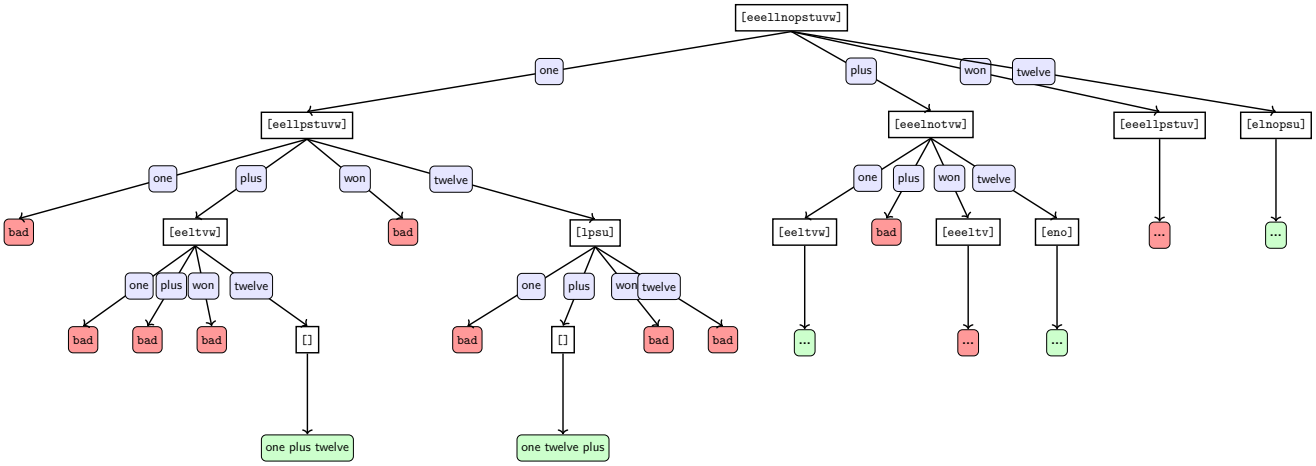
Step 1: Prune The Dictionary

Not all the words in the dictionary are relevant to the phrase "`eleven plus two`." In particular, "potato" and "zebra" contain letters ("a" and "z", respectively) that are not in `eleven plus two`. 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 word choices are (the remaining words in the dictionary.) So, we go through our words recursively, keeping track of the unused letters at each step:



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 When `max` is Zero

You may not make any assumptions about the length of the input or output when handling the case when `max` is 0 in the `print` method. In particular, do NOT try and simply search for all anagrams where the number of words is smaller then some arbitrary number since then there is no guarantee your output will always be correct (no matter how large an artificial limit you choose).

Let Exhaustive Search Do Its Job

Don't make this problem harder than it needs to be. You are doing an exhaustive search of all the possibilities. You have to avoid dead ends, and you have to implement the optimization 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.

Output Limits

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).

Ed also limits the amount of output, but this limit cannot be changed. Therefore, we strongly recommend you test your code in jGRASP before submitting.

Code Quality Guidelines

In addition to producing the behavior described above, your code should be well-written and meet all expectations described in the grading guidelines, Code Quality Guide, and Commenting Guide. For this assessment, pay particular attention to the following elements:

Recursive Approach

Be sure to implement your recursive backtracking following the guidelines discussed in class. Avoid repeating computations that you don't need to and exploring branches that you know will never be printed. Don't create special cases in your recursive code if they are not necessary. Avoid repeated logic as much as possible.

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

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. If you use jGRASP, you may want to change your settings to see which line the warning refers to. Go to `Settings/Compiler Settings/Workspace/Flags/Args` and then uncheck the box next to "`Compile`" and type in: `-Xlint:unchecked`

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.

Exceptions

The specified exceptions must be thrown correctly in the specified cases. Exceptions should be thrown as soon as possible, and no unnecessary work should be done when an exception is thrown. Exceptions should be documented in comments, including the type of exception thrown and under what conditions.

Commenting

Each method should have a header comment including all necessary information as described in the Commenting Guide. Comments should be written in your own words (i.e. not copied and pasted from this spec) and should not include implemenation details.

Running and Submitting

When you are ready to submit, you can submit your work by clicking the "Mark" button in the Ed assessment.

You may submit your work as often as you like until the deadline; we will always grade your most recent submission. Note the due date and time carefully—**work submitted after the due time will use up late days**. You cannot submit work after the late cutoff for the assignment.

Getting Help

If you find you are struggling with this assessment, make use of all the course resources that are available to you, such as:

- Reviewing relevant examples from class
- Reading the textbook
- Visiting office hours
- Posting a question on the message board

Collaboration Policy

Remember that, while you are encouraged to use all resources at your disposal, including your classmates, **all work you submit must be entirely your own**. In particular, you should **NEVER** look at a solution to this assessment from another source (a classmate, a former student, an online repository, etc.). Please

review the full policy in the syllabus for more details and ask the course staff if you are unclear on whether or not a resource is OK to use.