CS 0445
Algorithms and Data Structures 1
Assignment 4

Online: Monday, March 25, 2024
Due: All assignment materials: 1) All source files of program, 2) All input data files 3) All output files and 4) Assignment Information Sheet. All materials must be zipped into a single .zip file and submitted **via the course submission site** by **11:59 PM on Friday, April 12, 2024**
Late Due Date: 11:59PM on Monday, April 15, 2024

Note: Do not submit a NetBeans or Eclipse (or any other IDE) project file. If you use one of these IDEs make sure you extract and test your Java files WITHOUT the IDE before submitting.

Background:
Word scrambles (or anagrams) are fun puzzles that challenge our minds and our patience. The general idea of an anagram is that a string of letters is rearranged to produce a valid word or phrase. Originally the game was to unscramble a meaningless string of letters to generate a meaningful word or phrase. More recently (thanks to computers) it is also used to "create" phrases from other words or phrases. For input consider a string of letters which may or may not contain blanks (the blanks will be ignored in the anagram game, but are allowed to make the input easier for the user). Each solution **must use all of the letters in the input string** (except blanks) but the solution can have an arbitrary number of words in it. The output will be a collection of valid resulting words or phrases, sorted from the phrase with the fewest words to the one with the most words. Within groups with the same number of words, the listings should be alphabetical. For example, given the letters:

```
raziber
```

your anagram program should generate the following list of words:

```
bizarre
brazier
raze rib
rib raze
```

(Note: Answers will vary depending on the underlying dictionary used. The answers above are based on the dictionary used in this assignment). For some more examples of input and output see the test files and output posted on the course Web site and Canvas site.

The basis for forming anagrams is being able to generate arbitrary permutations of the letters that were provided by the user. Generating permutations of a list of characters that can be repeated (i.e. the same character can be used in more than one location) can be done with a fairly simple recursive / backtracking algorithm. To see how this would be done, see the handout Perms.java. Note how each letter is tried and how the recursion and backtracking are done. The process is a bit more complicated if each letter can be used only one time. To do this, you must be able to do the following:

- Create some type of iterable set or list of the characters in the original string
- Iterate on the set of characters within a single recursive call – each iteration will represent the character chosen in a given position – making another recursive call for each case
- Remove the current letter from the set that is passed into the next recursive call (so that the next call will not consider the letter that was just used).
- Add the current letter back to the set (after backtracking) so that it will again be available on the next recursive call (since a different letter will be the "current" letter on that call).

Consider (for now) building a *single word* anagram of a string containing K characters. Since any solution must be an actual word in the English language, one way of approaching this problem is to generate all permutations of the K letters and to test each permutation to see if it is a valid word in an English dictionary. We know (using discrete math) that for K characters there are K! distinct permutations. We also know that K! grows very quickly, so as the number of characters in our original string increases, the number of permutations that we will have to test increase greatly (ex: 8! = 40320, 9! = 362,880, 10! = 3628800, etc). Note that most permutations of letters will not actually be words, so with this approach we will be generating and testing a lot of strings that will not be valid anagrams – consuming a lot of time. Searching for a word in a dictionary can also take some time, depending upon how the dictionary is stored.

One effective way to eliminate many invalid permutations before using all of the characters is to check the current string *as it is being built* to **see if it is a prefix of any word in the English language.** A prefix of a word is the characters that begin the word. For example, for the word bizarre, some prefixes are b, bi, biz and biza. Consider again single word anagrams of K letters, and assume that at some point in the process M letters have been used (with K-M letters remaining). If it can be determined that the M-character string that has been built is not a prefix of any English word, there is no need to try any additional letters, since they could not possibly result in a solution. We could thus immediately backtrack, eliminating all of the recursive calls involving the remaining (K-M) letters.

For example, using the input above ("raziber"), if we consider the letters in the order given, we can test in the following fashion (note each prefix corresponds to a recursive call):

r → this is a prefix to a word in English so recurse to the next letter
ra → this is a prefix to a word in English so recurse to the next letter
raz → this is a prefix to a word in English so recurse to the next letter
razi → this is NOT a prefix to any word in English (i.e. no words in our dictionary begin with the letters “razi”). Thus, adding any more characters to it could not possibly result in a solution, so backtrack at this point.

To understand the savings in this approach consider that there are 3 more characters remaining, which have 3! = 6 possible permutations. Thus, backtracking due to the prefix test saves not one but rather 6 possible recursive calls. The benefit is even greater for longer strings when the prefix test fails early in the process. For example, consider a progression in the test of the above input, where the first call has progressed to the letter z. Assume the characters are configured such that the following occurs:

z → this is a prefix to a word in English so recurse to the next letter
zr → this is NOT a prefix to any word in English. Adding any more characters to it could not possibly result in a solution, so backtrack at this point.

In this case there are 5 characters remaining in the string and 5! = 120 possible permutations that have been avoided. Consider the many other strings that would not be prefixes to words (ex: zb, bz, rb, rz, rr – just to name a few of the 2-character strings) and it is easy to see how this check can save a lot of recursive calls.

In order to incorporate prefix testing into our anagram solution, we need an efficient way of testing whether a string is or is not a prefix of a word in our dictionary. If this check takes too long it will diminish the benefit of the test in our solution. Luckily, there is a data structure that accomplishes this task very efficiently – the multiway trie. The multiway trie can also efficiently determine if a string is itself found in the dictionary, which is also necessary for our anagram algorithm, since all of our solutions must contain only valid words. We have not discussed the details of this data structure in this course – but you may discuss it in CS 1501. Luckily, due to data abstraction we do not need to know the implementation details in order to effectively use a multiway trie. Rather, we simply need to know the functionality – which is fairly simple.

Consider the class TrieSTNew in file TrieSTNew.java. This class is a modification by me of a class provided in the Sedgewick CS 1501 textbook. Of interest are the following methods:

```
// The first argument (String key) is the means by which to search for
// the second argument (arbitrary type Value). After this method succeeds
// the key and value can be accessed using the get() and
// searchPrefix() methods. For our purposes, both arguments will be the
// same String - it will be both the key and the val - so the generic
// type Value will actually be String when we use this class.
public void put(String key, Value val)


// This method will return one of four possible values:
// 0 if the key is neither found in the TrieSTNew nor a prefix to
//      a string in the TrieSTNew
// 1 if the key is a prefix to some string in the TrieSTNew but it
//      is not itself in the TrieSTNew
// 2 if the key is found in the TrieSTNew but it is not a prefix to
//      a longer string within the TrieSTNew
// 3 if the key is found in the TrieSTNew and is also a prefix to a
//      longer string within the TrieSTNew
public int searchPrefix(String key)
```

Note that based on the return value of the searchPrefix() method, we can determine several results with a single call, as described above. All of the possible results will be very important in our anagram solver algorithm. To see how the TrieSTNew can be initialized and used, see the program TrieSTTest.java. Look at the program, read the comments, run it and see the output. Make sure you understand what this program does and what the methods above accomplish.

A multiway trie can perform both of the methods above (put() and searchPrefix()) in time proportional to the length of the key String – independent of the number of Strings stored in the data structure (in other words, the size of the dictionary does not impact the time to search for a given String). This very efficient access allows the multiway trie to be an effective tool in our anagram solver algorithm. Consider creating a new TrieSTNew object and inserting (via put()) all of the words in an English dictionary into it – in particular the words in file dictionary.txt. Now the TrieSTNew can be utilized via the searchPrefix() method to check as anagrams are being generated, thereby saving a lot of unnecessary recursive calls.

Your anagram solver program must be a recursive backtracking algorithm as described above, and you are required to use a TrieSTNew object (also as described above) with your algorithm in order to eliminate impossible permutations early in the generation process.

At this point we can specify the Assignment 4 requirements more precisely. Your assignment is to implement an anagram solver program, called **Assig4.java** in the following way:

1) Read a dictionary of words in from file dictionary.txt and generate a **TrieSTNew** (as described above) of these words.
2) Read input strings from an input file (one string per line, which may contain spaces) and calculate the anagrams of those strings and output them as described above. Note that your anagram solutions must use all of the letters that were input (ignoring the spaces) and the solutions may have more than one word. **Your output must go to an output text file.** The user should be able to enter both the input file name and the output file name on the command line. Your program should read the strings one at a time from the input file until it reaches the end of the file. For each input string your program must output **all of** the anagram strings to the output file in the order specified above (from fewest words to most words, and alphabetical for phrases with the same number of words). It must also output the number of solutions found for each input string, and within that answer it must output the number of solutions for each K where K is the number of words in a solution. **For more specifics on the requirements for generating the anagrams, see the description above and the comments below.** See the CS 0445 Assignments page and Canvas page for the input files that you must use and for some sample output files. **Your output must match the output in the sample output files.**

For an example of an anagram-finding program, see the following link:
https://www.ssynth.co.uk/~gay/anagram.html

Use this program for reference and to help you in developing your solution, but note a few important things:

– The dictionary used for this program is different from the one you will be using, so in many cases the solutions will not be the same
– The solutions there are not listed in the same order that you are required to follow and not all solutions are shown

Important Requirements, Notes and Hints:

– I recommend initially developing the anagram algorithm using a small words file (i.e. not dictionary.txt) and a known test file with small strings. This will allow you to trace the execution by hand if necessary, and will help with the debugging. Once you think your program is working, try it using the dictionary.txt file and the provided input files.
– **The anagram generation algorithm will be non-trivial, especially for multiple-word anagrams.** To enable you to get more partial credit for the anagram algorithm, you **may** want to consider the algorithm in **two parts**:
    1) Determine the valid anagrams (if they exist) using ALL of the letters in the input in a **single word** (if there were spaces in the input, remove them first) (ex: for the example above the solutions would be `bizarre` and `brazier`). This process can be done as described above, utilizing a recursive backtracking algorithm together with the TrieSTNew to eliminate some impossible permutations early in the process. Note that before getting a solution we may end up having to backtrack all the way to the first character, as can be seen in this example. The original string was "raziber" and the two solutions that use all of the letters are "bizarre" and "brazier", so clearly our initial attempt that starts with "r" will not be successful. Also note that since we want ALL solutions, the algorithm must backtrack even after a solution has been found.
    2) Once you can find the single-word anagrams using all of the letters, add code that allows you to generate multiple-word anagrams as well (ex: for the above example all of the solutions AFTER `bizarre` and `brazier`). This is tricky since there are different ways of approaching the algorithm and the recursion / backtracking is more complicated. Even storing the solutions requires some thought in this case. Think carefully about how you would approach this before coding it (try it with a pencil and paper). In particular note that in this case if **a string is BOTH a prefix AND a word** (return value 3 in the searchPrefix() method), you will have to consider both of these possibilities in your recursive process. Don't forget to sort the solutions from fewest words to most words (and alphabetically for solutions with the same number of words) and don't forget to output the number of solutions overall and for each K where K is the number of words in a solution.
– If a letter (or letters) appear more than once in the initial string, your algorithm will likely produce some anagrams (possibly many) more than once. Think about why this is the case. However, **duplicates should NOT appear in your solutions**, so be sure to eliminate them before printing your results. There are different ways of removing the duplicates so consider carefully how to do this. Note also that different solutions can have identical words in different orderings – the order of the words is part of the solution, so solutions with the same words in different positions should be kept. See the example output for a demonstration of this point.

– Your output should match that shown on the CS 0445 Web site and Canvas site for the input files shown. This includes both the solutions found and the order in which they are shown. Think about the requirements and how you can present the solutions in the required order. See Assignment 4 on the web site or Canvas site for the output files.

– If you search the Web you will find the anagram program indicated in the link above and others as well. If you search hard enough you can probably find source code to one or more of these programs. I strongly urge you to resist trying to find this code. If you use code found from the Web for this project and you are caught, you will receive a 0 for the project (following the cheating guidelines as stated in the Course Policies).

- Even with the described strategy, this program will require a lot of computation – to form the anagrams, to store and sort the results and even to check for duplicates. For test files test1.txt up through test5.txt, the total run-time should be just a few seconds. However, due to the length of the source string and the number of solutions, test file test6.txt may take quite a lot of time (minutes) to complete. Your times will vary, but if your program is taking a very long time for any of the test files test1.txt through test5.txt, you should check to make sure you are forming and checking the anagrams correctly. **Very slow run-times on the smaller test input will result in a score deduction.**

- Be sure to thoroughly document your code, especially any code whose purpose is not clear / obvious.

- Make sure you **submit ALL** of the following in your **single .zip file** to get full credit:
  Assig4.java (your main program file)
  Any other source code files that you have written
  TrieSTNew.java
  dictionary.txt
  All input files (test1.txt through test6.txt)
  Your completed Assignment Information Sheet

Extra Credit:

The TrieSTNew class as defined utilizes String values for the keys in the multiway trie. This works well but as you are building your anagrams you will need to add and remove characters – likely utilizing a StringBuilder in this process. Thus, in order to search the TrieSTNew for a String, you will first need to convert your StringBuilder into a String (ex: by using the toString() method in the StringBuilder class). This is simple to do but it requires the overhead of building a String out of the StringBuilder each time you call the searchPrefix() method. The overall process could be improved if the TrieSTNew class could be formed of StringBuilder objects rather than String objects and could thus be searched directly with a StringBuilder. As an extra credit exercise do the following:

1) Write a second version of TrieSTNew, called TrieST2, in which the String objects are replaced by StringBuilder objects. Make sure to test this class so that it works. The changes should be very simple and you don't really need to know the full logic of the class in order to make this new version.
2) Write a second version of your main program that utilizes TrieST2 for your prefix tests rather than TrieSTNew. Note that this program will be identical to your original program except that TrieST2 will be substituted for TrieSTNew and you will not have to cast your StringBuilders into Strings before using the TrieST2.
3) Time both versions of your main program on the larger test files (ex: test4.txt, test5.txt, test6.txt) and see if there is a non-trivial difference in the run-times. Any difference would be likely be the most pronounced with test6.txt. Try several runs of each and average your results.
4) Submit a very short essay (1 or 2 paragraphs) indicating your results. Include a chart of your run-times with your essay.