AnagramMaker.java

Objective: To use recursion to construct anagrams.

Background:
An anagram is a word, phrase, or name formed by rearranging the letters of another. For example, “debit card” can be “bad credit”, “dormitory” can be “dirty room”, and “Clint Eastwood” can be “Old West action”. Anagrams do not need to make sense, they just have to be a reordering of the letters into words or names. For example, “running with the bulls” can be rearranged into the anagram “bell whining untruths”.

More anagrams:

“Donald Trump” —> “odd rum plant” and “damp old turn”
“Cupertino” —> “rope tunic” and “poetic run”
“Greenstein” —> “teen singer” (how did it know?) and “tense reign”
“Computer Science” —> “cue comic serpent”

For this project, we will create an anagram-maker which takes a **phrase**, reorders the letters, and comes up with every possible anagram in dictionary order. Different word orderings are treated as different anagrams. For example, “unbid dollars” and “dollars unbid” are two different anagrams.

On the next page is an outline of the algorithm for making anagrams of the word “digit”. A graphical bracket represents a call to the anagram-making method. Nested brackets are recursive calls to the same method. Each nested method tries to make anagrams of a smaller **phrase** (collection of letters). When the method is passed an empty **phrase** (length is 0), then the **ArrayLIst anagram** contains an anagram. Study this algorithm before you attempt to write the code.

Once your anagram-making code is written, your program will need some controls. Inputting a long phrase will produce a large number of anagrams. For example, the phrase “once upon a time” will produce thousands of anagrams and the program will take a long time to complete. It is important to restrict the number of anagrams printed so it completes in a reasonable amount of time.

You are to implement two controls. First, allow the user to dictate exactly how many words are formed in each anagram. Using the word “teaching”, a list of one-word anagrams would include “cheating”, a list of two-word anagrams would include “night ace”, a list of three-word anagrams would include “get a chin”, and so on. Second, the user should be able to control the maximum number of anagrams printed. For example, “resistance” has over 80,000 three-word anagrams. The user would specify the number output, say 1,000, to save time.

Below is an example algorithm for finding anagrams of “digit”. Each graphical bracket “[“ is a recursive call to the anagram maker method.

What is (are) the base case (cases)? Which parameters get smaller each recursive call? Which parameters get bigger?

- Input phrase, create ArrayList to hold anagram.
  - **`phrase = "digit"`** **`anagram = [ ]`**
- Remove non-alphabetic characters in phrase.
  - **`phrase = "digit"`**
- Is phrase empty? If no, continue finding anagram of phrase (“digit”).
- Find all words that can be made from phrase (“digit”).
  - **`allWords = [ "dig", "digit", "it" ]`**
- Select first word (“dig”) from allWords and add to anagram words.
  - **`anagram = [ "dig" ]`**
- Remove letters of selected word (“dig”) from phrase (“digit”).
  - **`newPhrase = "it"`**
- Is (new) phrase empty? If no, continue finding anagram of (new) phrase (“it”).
- Find all words that can be made from phrase (“it”).
  - **`allWords = [ "it" ]`**
- Select first word (“it”) from allWords and add to anagram.
  - **`anagram = [ "dig", "it" ]`**
- Remove letters of selected word (“it”) from phrase (“it”).
  - **`newPhrase = "" (empty string)`**
- Is (new) phrase empty? If yes, then print anagram and return.
  - **Print anagram “dig it” and return**
- Remove last word (“it”) from anagram.
  - **`anagram = [ "dig" ]`**
- No more words in this level of allWords, so return
- Remove last word (“dig”) from this level of anagram
  - **`anagram = [ ]`**
- Select second word (“digit”) from allWords at this level and add to anagram words.
  - **`anagram = [ "digit" ]`**
- Remove letters of selected word (“digit”) from this level’s phrase (“digit”).
  - **`newPhrase = "" (empty string)`**
- Is (new) phrase empty? If yes, then print anagram and return.
  - **Print anagram “digit” and return**
- Remove last word (“digit”) from this level of anagram.
  - **`anagram = [ ]`**
- Select third word (“it”) from allWords at this level and add to anagram words.
  - **`anagram = [ "it" ]`**
- Remove letters of selected word (“it”) from this level’s phrase (“digit”).
  - **`newPhrase = "dgi"`**
- Is (new) phrase empty? If no, continue finding anagram of (new) phrase (“dgi”).
- Find all words that can be made from phrase (“dgi”).
  - **`allWords = [ "dig" ]`**
- Select first word (“dig”) from allWords and add to anagram.
  - **`anagram = [ "it", "dig" ]`**
- Remove letters of selected word (“dig”) from this level’s phrase (“dgi”).
  - **`newPhrase = "" (empty string)`**
- Is (new) phrase empty? If yes, then print anagram and return.
  - **Print anagram “it dig”**
- Remove last word (“dig”) from this level of anagram.
  - **`anagram = [ "it" ]`**
- No more words in this level of allWords, so return
- Remove last word (“it”) from this level of anagram.
  - **`anagram = [ ]`**
- No more words in this level of allWords, so return

Assignment:

1. Download **AnagramMaker.zip** from Mr Greenstein's web site and unzip. It will create the directory **AnagramMaker** and do all of your work in this directory. The directory contains **AnagramMaker.java**, **WordUtilitiesExtraCode.txt**, and **randomWords.txt**.

   AnagramMaker.java has all of the methods except for the anagram-making method.
   WordUtilitiesExtraCode.txt contains three important methods for your **WordUtilities**.
   randomWords.txt is the word database.

2. Copy the following files into the **AnagramMaker** directory: **WordUtilities.java**, **Prompt.java**, **FileUtils.java**, and your latest **SortMethod.java** that sorts an **ArrayList** of **String**s.

3. Add the three methods from **WordUtilitiesExtraCode.txt** (**wordMatch**, **allWords**, and **sortWords**) to your **WordUtilities.java**. The method **wordMatch** is a helper method to **allWords**. The method **allWords** will be used by your anagram algorithm to find all words in the database that match some or all of a group of letters. **sortWords** uses your **SortMethod**'s **mergeSort** method on the database so all the anagrams come out in dictionary order.

4. Inside **AnagramMaker.java**, create a recursive method to find all the anagrams of a phrase provided by the user. Use the algorithm on the previous page as a guide.

5. The performance of your **AnagramMaker** is also considered in grading. The program should take less than a few seconds to produce the sample runs below. Tune your algorithm to get the desired performance.

Here is a sample run output.

```
% java AnagramMaker

Welcome to ANAGRAM MAKER

Provide a word, name, or phrase and out comes their anagrams.
You can choose the number of words in the anagram.
You can choose the number of anagrams shown.

Let's get started!

Word(s), name or phrase (q to quit) -> cupertino
Number of words in anagram -> 2
Maximum number of anagrams to print -> 10

cento puir
centro piu
certi upon
certo puni
ciento pur
cierto pun
cio pentru
cio purent
cire punto
citer upon

Stopped at 10 anagrams

Word(s), name or phrase (q to quit) -> monta vista
```

```
Number of words in anagram -> 2
Maximum number of anagrams to print -> 10

amant visto
attains vom
avait monts
avant moist
avant omits
avant somit
avions matt
mast tavoin
mat vastoin
matins ovat

Stopped at 10 anagrams

Word(s), name or phrase (q to quit) -> computer science
Number of words in anagram -> 3
Maximum number of anagrams to print -> 10

cc ceinture poems
cc censure tiempo
cc centimes proue
cc centuries mope
cc centuries poem
cc ceremonie puts
cc ceremonies put
cc ciento presume
cc ciento supreme
cc come peintures

Stopped at 10 anagrams

Word(s), name or phrase (q to quit) -> q

Thanks for using AnagramMaker!
```