Catching Plagiarists

This assignment presents a real problem that requires a creative software solution. It was initially created by Baker Franke at the University of Chicago.

Problem Statement
A lead Teaching Assistant for an Introduction to Physics class at a Big-Ten University had roughly 800 students in the course and about 20 TAs who led lab sections. The lead TA got word that some students were plagiarizing their lab reports. This situation, however, was not an instance where submissions were being copied outright from the web, nor were students so bold as to turn in another student's work as their own. Instead, the word on the street was the all-too-common digital age occurrence of one friend "helping" another by giving them a copy of a completed report, only to have the friend steal sections, words, and phrases from it and turn it in. The problem was that the T/A had no idea who the plagiarists might be, and all they had were 800 lab reports turned in digitally.

Your task:
Given a folder containing a set of text documents, identify the ones with a large amount of similar text – in other words, catch the plagiarists.

More formally:

> **Input**: from a list, select the folder containing the documents to process
> **Output**: some representation that identifies files with many word sequences in common.

Details
What constitutes plagiarism?
We will search for common *n*-contiguous-word sequences among the documents for our purposes on this assignment, where the *n* is up to the user. For example: if a document contains the sentence:

> In 1492, Columbus sailed the ocean blue.

Then for *n* = 4, the 4-contiguous-word sequences from that sentence would be

> In 1492 Columbus sailed
> 1492 Columbus sailed the
> Columbus sailed the ocean
> Sailed the ocean blue

Keep in mind that collecting all of the *n*-contiguous-word sequences will (likely) require *n* iterations of reading each text file. For example, getting all of the possible 4-word phrases in a text file typically requires scanning through the file four times. The first time through, you grab all of the initial 4-word phrases. The second time around, you throw away the first word and then capture the remaining 4-word phrases. The third time through, you toss the first two words. The fourth time, you discard the first three words.

Half of the task is *parsing* (slicing) up all of the text files into *n*-contiguous-word phrases that will then be compared to the same length phrases in all the other text files.

Input: The documents
Three sets of documents are provided for you to use. One collection will be small (20 or so documents) for testing purposes. The other two sets will be larger (one is about 100 documents, the other over 1300 documents) to test your solution's scalability. (The documents came from www.freeessays.cc, a repository of *really bad* high school and middle school essays on various topics).

Output:
A reasonable way to display the final results would be to list the number of 'hits' between pairs of documents, sorted in descending order. The list would only need to print those results above a certain threshold of 'hits' to limit unneeded data. The Teaching Assistant could then inspect those submissions at the top of the list individually to determine if they were plagiarized.

Getting started and Steps to Take

Step 0: Parsing the Documents

See if you can display all of the *n*-contiguous-word phrases from all of the documents in a given directory according to the following steps:

0.0 Create a list of all of the (sub)directories.

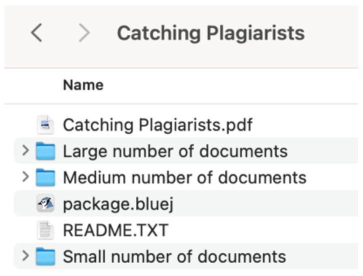
0.1 Be able to access a text file in a chosen directory.

0.2 Produce all of the *n*-contiguous-word phrases from one of the files in the directory.

0.3 Produce all of the *n*-contiguous-word phrases from all of the files in the directory.

The following is some helpful information to help you get started with parsing the text files.

Files and Folders
Every computer (obviously) contains a multitude of files. Each one is stored in a folder (aka directory). Folders can contain other folders (subdirectories). Every file storage medium has a single starting place (e.g., computer hard drive). From there, folders and subfolders branch out in a tree-like data structure. Navigating between the starting folder and the subfolders beneath it is key to the first half of this assignment.



Catching Plagiarists.pdf
Large number of documents
Medium number of documents
package.bluej
README.TXT
Small number of documents

Here is an example of how to read the various subdirectories of a given directory (folder) into an `ArrayList` of `File` objects. (The `File "."` tells Java to look for the subdirectories in <u>this</u> particular folder. It is vital to make this a *relative* reference rather than *explicit* (i.e., Desktop\CompSci\ . . .) so that your code can find the subdirectory no matter what directory your source code is in.)

```
File dir = new File(".");
ArrayList<File> directories = new ArrayList<File>();
for(File folder : dir.listFiles())
   if(folder.isDirectory())
      directories.add(folder);
```

(Recall that `File`, `ArrayList`, and `Scanner` (used later) objects must be imported before you can use them.)

Here is an example of how to read a directory of file names into an `ArrayList`, where `directory` is a `String` representing the name of the directory.

```
File dir = new File(directory);
String[] temp = dir.list();
List<String> files = new ArrayList<String>();
for(int i = 0; i < temp.length; i++)
{
   if(temp[i].endsWith(".txt"))
      files.add(temp[i]);
}
```

Generate a complete file pathname using the directory and file name variables you created. Specifically, the proper pathname for accessing a given file in the **Small number of documents** directory would be something like

```
./Small number of documents/erk185.shtml.txt
```

Here is an example of how to read a certain number of words (`numWords`) from a file, where `pathname` is a `String` representing the file's pathname.

```
Scanner file = new Scanner(new File(pathName));
while(file.hasNext())
{
   String phrase = "";
   // read 'numWords' words from file
   for(int j = 0; j < numWords; j++)
   {
      if(file.hasNext())
         // strip away all punctuation, and set to lowercase
         phrase += file.next().replaceAll("[^A-z]","").toLowerCase();
      else
         phrase = null;  // not enough words at end of file
   }
```

Be sure to get all of the proper permutations for each *n*-contiguous-word phrase. For example, the file `erk185.shtml.txt` begins with the following sentence.

> Esperanza feels lost, alone, and confused. She believes that the only ones who can understand her feelings are trees.

Using the above code example to strip away all punctuation (including spaces), convert it to lowercase, and use 4-word phrases would generate the following strings (including permutations).

esperanzafeelslostalone andconfusedshebelieves thattheonlyones whocanunderstandher
feelslostalongand confusedshebelievesthat theonlyoneswho canunderstandherfeelings
lostaloneandconfused shebelievesthatthe onlyoneswhocan understandherfeelingsare
aloneandconfusedshe believesthattheonly oneswhocanunderstand herfeelingsaretrees

Check that you correctly generate all of the *n*-contiguous-word phrases, using the above example as your guide. Note that this is just for the first sentence in that particular file. You'll need to create all of the *n*-contiguous-word phrases for the entire text file and then do it for each of the files in a given directory.

Of course, *n* will be variable and up to the user to determine when they run the program, so make sure you allow the user to enter a value for *n.*

When you are satisfied with your work, you might consider writing a helper method that, given the name of a file and an *n*, produces an `ArrayList` of all the *n*-contiguous-word sequences. This procedure will likely help you later on.

If this is as far as you are going to go on this project (just completion of Step 0), be sure to print your n-contiguous-word phrases in the terminal window in a similar format to the example above so that I can check your work. (Otherwise, there is no need to print out the n-contiguous-word phrases.)

Step 1: Produce a "hit list" for the small set of documents
Produce a pair-wise list of the number of *n*-contiguous-word sequences each file in a directory has in common.

1.0 You might want to start by creating a custom class that holds two file names and a hit count.

1.1 Create a collection that can contain your custom class.

1.2 Write a method that accepts two `ArrayLists` of word sequences and returns the number they have in common. Use this to create an instance of your custom class and add it to the collection.

1.3 Modify it to sort the output so that the largest number of hits is first (hint: have your custom class implement `Comparable<` *class name* `>`. Then use `Collections.sort` to sort your collection).

1.4 Add the ability to ignore entries less than a given threshold of 'hits'.

To validate your work, the following are the proper results from processing all of the files in the small number of documents directory, searching for 4-word phrases, and ignoring any file parings containing less than 10 duplicates.

```
[jrf1109.shtml.txt,   sra31.shtml.txt]      -> 1,589
[abf0704.shtml.txt,   edo26.shtml.txt]      ->   670
[abf0704.shtml.txt,   abf70402.shtml.txt]   ->   538
[abf70402.shtml.txt,  edo26.shtml.txt]      ->   181
[abf0704.shtml.txt,   edo20.shtml.txt]      ->    10
```

(Note that your specific numeric values may vary slightly from the above, depending on how you handle duplicated phrases in the same document. However, the ranking order of the file parings ought to be the same.)

Step 2: Adding User-Interface
Your program should provide the user with a simple introduction of what your program intends to accomplish. Next, your program should present the user with a numbered menu (list) of the document folders and then ask the user to select one from the list they want to use. Have the user enter the folder number (from the menu) rather than relying on them to type the complete folder name correctly. Your program should then prompt the user for the number of words in a phrase and the cutoff threshold for showing hits. Of course, your code should validate the user entries (e.g., your program won't crash with out-of-range inputs). The user prompts and results should be displayed in an aesthetically pleasing manner. There should be some sort of progress indicator so the user can anticipate how long it will take to complete.

Try it out on the medium and large document sets to see what happens. You will get a passing grade for this project if this step works properly for the medium document set.

Minimum Requirement:
Your primary goal is properly comparing the medium document set in a reasonable amount of time (less than 30 seconds). The program's output must be sorted by the highest hit count first. Ensure your program can locate the document subdirectories no matter where your source code is (i.e., don't hard-code the explicit subdirectory paths). It goes without saying, but it is expected that your program will successfully run after you turn it in.

For this step, I will be comparing your results for the **Medium set of documents** against an answer key of known results. As in the example above, I am more concerned with ordering the file pairings than I am with the exact count of the 'hits'. The 'hits' count variation, however, should be minimal.

Challenge
The real challenge is to catch the plagiarists in the large document set (~1300 documents). It will require some cleverness and ingenuity to do it in a reasonable amount of time, where "reasonable" means under a minute or two. Generally speaking, if your code takes less than two minutes on the **Large set of documents**, it is $O(N^2)$; otherwise, it is $O(N^3)$ or worse.

Turning it in
- Name your project file directory Catching Plagiarists.
- Zip up the entire project file directory by control-clicking on the directory name and then selecting Compress "Catching Plagiarists" from the menu.
- Submit the zipped directory using the link on the Schoology assignment page.

Academic Honesty
- This is an individual assignment and not a group project. In submitting your project, you implicitly state that your creation is not someone else's work (apart from what has been provided to you in these directions).
- Your code should reflect a unique solution to this challenging problem, which will be significantly different from anyone else's solution. You can count on your code being evaluated line by line against the work of others, so please do not be tempted to submit even portions of code that are not yours.
- Submissions deemed plagiarized from others' work will receive a zero on this assignment and cannot be resubmitted for re-evaluation.