



## Assignment:

Your task is very similar to the one described and shown above: find the common word sequences among documents in a closed set. Simply put, your **input** will be a set of plain-text documents, and a number  $n$ ; your **output** will be some representation showing the number of  $n$ -word sequences each document has in common with every other document in the set.

Finally, you should identify “suspicious” groups of documents that share many common word-sequences among themselves but not with others.

## DETAILS:

- *Output:*

You can think of processing everything into an  $N \times N$  matrix (where  $N$  is the number of total documents) with a number in each cell representing the number of “hits” between any pair of documents.

For example: below is a small table showing the comparisons between 5 documents:

	A	B	C	D	E
A	-	4	50	700	0
B	-	-	0	0	5
C	-	-	-	50	0
D	-	-	-	-	0
E	-	-	-	-	-

From this table we can conclude that the writers of documents A, C and D share a high number of similar 6-word phrases. We can probably say A and D cheated with a high degree of certainty.

For a large set of documents, you may only want to print a matrix for those documents with a high number of commonalities above a certain threshold.

Printing an  $N \times N$  matrix may be untenable for large sets. You could instead produce a list of documents ordered by number of hits. For example:

```
700: A, D
50:  A, C
50:  C, D
5:   B, E
4:   A, B
```

You could also produce a graphical representation like the one shown above. If you want to discuss strategies for how to accomplish this please see me.

- *The documents:*

Some sets of documents will be provided. One set will be small (20 or so documents) for testing purposes. The other sets will be larger (one is 100 documents, the other over 1300

## APCS Lab Assignment

---

documents) to test the scalability of your solution. (The documents came from [www.freeessays.cc](http://www.freeessays.cc), a repository of \*really bad\* high school and middle school essays on a variety of topics).

Your program should be able to process all of the documents in a given folder/directory.

A HOW-TO for processing a directory of files can be found on the web page.

- *Strategy:*

How are you going to do this? We'll let it's up to you. The brute force solution (comparing each six-word sequence to all other six-word sequences) gives an  $O(w^2)$  solution – where  $w$  is the *total number of all six-word sequences in all documents*. For a large set of documents  $w^2$  grows very large, very fast. It will work though – it will just take a while. For perspective, if 20 documents takes 1 sec. to process this way: 1300 documents will take about 70 minutes.

There may be a clever way to use a hash table that will, in theory, produce a running time of  $O(w)$ . It will work for the small set of documents. The problem with the hash table strategy is not the time complexity but the space complexity. For a large number of documents the amount of memory required to compute this is too large to hold in memory all at one time. If you want this solution to scale to large sets of documents, you'll have to do even more clever things, probably by creating your own supplementary data files that you can store and load on demand.

One way to gain ultimate control over the processing is to write your own specialized data structures. However, you're free to use anything in the java API.

### **Getting started, Grading, and Milestones:**

Your grade will be composed of points earned for each of three major tasks (milestones) that need to be achieved in order to complete this project. Each milestone should have its own testing class to use as proof that you completed it.

**This lab will be weighted to be worth 3x a normal lab if you do better on it than your average Lab grade and 2x a normal lab if you do worse.**

### **Milestone I :: Processing the Documents (50 points)**

This is very similar to processing the files for Labs 20 and 21 – it's a combination of the two actually.

Proof: You should have a class called FileProcDemo that accepts two command line arguments: the directory path to the files, and the word sequence length. In other words I should be able to call `main()` on a class called FileProcDemo with the arguments `{"path/to/text/files", "6"}` and see a printing of all 6-word sequences of all the files contained in the folder "path/to/text/files".

NOTE: you should have supporting classes that do the actual work of processing the files in a way that's convenient for your whole program. The testing class FileProcDemo should simply call on those other methods.

### **Milestone II :: Produce a “hit list” for a small set of documents (50 points)**

One of the document sets provided has only 25 documents in it. Your program should at least be able to print to the console the number of word-sequence matches between each pair of documents. You may do this however you like. NOTE: for 25 documents this will require over 300 lines of output.

Proof: Your project should contain a class called “SimpleHitList” which has a main method that accepts the same two arguments as milestone I. Running the main method should just print a bunch of lines to the console that indicate the number of common word sequences between a pair of files. For example, the output might be:

```
[FileA.txt, FileB.txt] -> 247
[FileA.txt, FileC.txt] -> 7
...
[FileY.txt, FileG.txt] -> 876
```

You will get all the points for this by showing every pair and having an accurate hit count for each.

### **Milestone III :: Final Submission (100 points).**

This is the final product you wish to produce.

The minimum here is essentially a synthesis of Milestones I and II plus some processing of the findings to reveal the most “suspicious” cases in the set of documents. A nice product to produce would be a simple console application that accepted 3 command-line arguments {“directory of files”, “word-sequence-size”, “noise threshold”) and would print out a list of the documents that shared the most number of word sequences above the threshold.

**At a MINIMUM:** Your project should contain a class called “CatchPlagiarists” with a main method that accepts three arguments. For example providing the args: {“path/to/docs”, “6”, “200”} would produce a list of all the pairs of files in path/to/docs that shared more than 200 six-word sequences in common. It should also be ordered so that the highest hit counts are listed first. It should be able to process the small document set in a reasonable amount of time. Doing this will earn **70 points**.

**The real challenge** is to catch the plagiarists in the large document set (~1300 documents). This will require some cleverness and ingenuity in order to do it without running out of heap space and in a reasonable amount of time. Being able to do this will earn the full **100 points**. NOTE: let “reasonable amount of time” be a few minutes. If the program takes more than a few seconds though it should frequently output some sort of progress indicator so that the user doesn’t think the program has stalled.

**You can make up points in milestone III in other ways, without having to tackle the large document set:**

## APCS Lab Assignment

---

- A more dynamic user interface: allow the user to repeatedly re-process the documents by allowing the user to change the word-sequence length and “noise” level. Basically a pretty-ification of a console-based application. **DO NOT LOSE** the command line args. If no args are given then enter interactive mode. **10 points.**
- Create GUI for user input. The GUI **must** use JFileChooser to allow the user to select the document directory, rather than typing it by hand. More points will be earned for a graphical representation of the results rather than just text-based – think bar charts, line graphs, images, etc. If you make a GUI the class with the main method **must be called** “CatchPlagiaristsGUI” and be separate from console-based version of Milestone III. **20 points**

Lastly, with Milestone III you **must write into the project README** about what your program does, how to use it, what works, what doesn’t work and any other features, bugs I should know about when I’m looking at your code. The README should contain the following sections:

```
PROJECT TITLE: Catching Plagiarists
AUTHOR: You
DATE:
DESCRIPTION:
    (A few sentence only.  What does this program do?)
HOW TO RUN THIS PROJECT:
    (Where’s the main method, what are the arguments?)
OTHER USER INSTRUCTIONS:
    (You’ve told the user how to run the program, but what should
    they expect to see?  And what does it mean?  This section should
    also include any instructions for the user if you have some
    interactive UI, or GUI.)
DESCRIPTION OF YOUR ALGORITHM:
    (This description should be Grandma-Proof)
ESTIMATED RUNNING TIME OF YOUR ALGORITHM:
    (Should be a tight, worst-case big-oh statement.  You may also
    include space usage if that’s important).
```

### **NO README = 0 POINTS for Milestone III.**

Some final words on grading. What I’m really looking for is:

- **Quality** of final product including program design, strategy, output and cleanliness of code and comments.
- **Amount** of work done after Milestone II – doing the minimum after milestone II will net about 35 points. If your output isn’t that impressive, but you tried something very complex and/or innovative, this counts as work and it will be recognized.
- **Effort.** I consider effort to be the level to which your program matches your potential as a programmer. Your project will not be graded “against” other students’ projects but rather against your own expectations and abilities. You have a pretty good idea of what you’re capable of at this point, and if it’s clear that you really extended yourself to do good work, it will also be evident to me.