PSA 2: Measuring File Similarity

Due: 11:59pm, Sunday 1/21/2018

People use programs to help extract new information. This assignment asks you to implement programs that count the number of words from a file. While that task alone can earn full credit, what's more interesting is using the word counter to create new information. The extra credit asks you to build a program that can tell you how similar two files are. Interesting files we may want to compare include tweets from Twitter bots, essays, and even PSA2 writeups from different quarters. We also introduce unit testing and test-driven development (TDD), which we suggest you do for this assignment. TDD is a systematic approach towards programming **modular** and **readable** code.

Helpful Information:

- Setting Up the PSA
- Online Communication: Using Piazza, Opening Regrade Requests
- Getting Help from Tutor, TA, and Professor's Hours
  - Lab and Office Hours (Always refer to this calendar before posting.)
- Academic Integrity: What You Can and Can't Do in CSE 8B
- How to Use Vim
- Style Guidelines
- Submitting on Vocareum and Grading Guidelines

Table of Contents:

Part 1: Vim and Unix Commands [7 Points]

In `COMMANDS.md`, answer the following short questions with short answers.

Questions about Vim:
1. What are two ways to switch from command mode to insert mode?
2. How do you move the cursor to the end of a line in vim without using the arrow keys or mouse?
3. How do you copy 8 lines of code in Vim?

Questions about Linux:
1. How do you change to the home directory, no matter what directory you are currently in?
2. How do you make a new directory from the command line?
3. How do you show the path to the directory you are currently in?
4. How do you copy a directory and all of its contents?
5. How do you copy files from your personal computer to the ieng6 server?
6. How do you copy files from the ieng6 server to your personal computer?

Introduction to Test-Driven Development

In PSA1, you learned to write testers for your code, whether it was inserting print statements, checking the print statements, testing various outputs for some input, or perhaps even using Exceptions. Regardless of technique, the important goal in testing your code is to determine whether your code has the correct behavior.

We now expand that idea with unit testing and test-driven development. Instead of writing one very large program and testing it, break it down into several smaller methods that have the same behavior as that one very large program. The smaller methods will be easier to test. Therefore, you might find it beneficial to write your own helper methods. Then test both the helper methods and the methods assigned. This is where test-driven development (TDD) becomes very useful. TDD is the technique of writing unit tests for methods **before** you write your methods. Writing tests ahead of time is a way of expressing what characteristics your program is supposed to have.

Why is this useful? When you first run your tests, all of them will fail because none of the methods are written. As you write methods one-by-one, some tests may begin to succeed. When all tests succeed, then you have accomplished your implementation goal as set by your tests. (Of course, the better the tests, the more likely the program will be correct.)

Example

```
/* This method returns the sum of first and second if and only if flag is true. Otherwise,
return null
Note: This method is not defined yet! */
public Integer addIfTrue (int first, int second, boolean flag);
```

If you were testing a method **`addIfTrue()`**, create a test **`testAddIfTrue()`**. In the test, include a method call to the function you want to test, even though the function has not been implemented yet. Your tests should not focus on what's inside of the method, but what output/result is expected given an input. As in this example, the programmer already knows what this method should do: It takes the two arguments (`first` and `second`), adds them together, and returns the sum. However, it will do so if and only if the third argument, `flag`, is true. Otherwise, just return null. Since the output's expected value can be easily guessed, the programmer can write a tester method for it *before writing the method itself*. Notice that **`testAddIfTrue`** was fully implemented *before* **`addIfTrue`** was.

```
public boolean testAddIfTrue () {
	if ( addIfTrue ( 5  , 7 , true )   !=  12   ) return false;
	if ( addIfTrue ( 3  , 2 , true )   !=  5    ) return false;
	if ( addIfTrue ( 0  , 0 , true )   !=  0    ) return false;
	if ( addIfTrue ( -1 , 5 , false )  !=  null ) return false;
	if ( addIfTrue ( 7  , 8 , false )  !=  null ) return false;
	if ( addIfTrue ( 999, 1 , true )   !=  1000 ) return false;
	if ( addIfTrue ( -6 ,-9 , false )  !=  null ) return false;
	return true; // Reaching this statement means all passed.
}
```

After **`addIfTrue()`** is implemented, the programmer will be able to run the unit tests on **`addIfTrue()`** and then modify its implementation from there.

Exercise (optional):

Let's practice unit testing. Neither of these files will be submitted for points. You will be writing tests and debugging a method that finds the derivative of a math expression.

Part A: Writing unit tests

In **`ExampleClass.java`**, you will find the following method, which is not defined yet:

```
	public String derivative(int multiplier, char variableName, int power) {}
```

Do not write the method body for this method yet. Do the following:

1. Read the method header of derivative() to understand what this method is supposed to do.
2. Go to **`UnitTestEx.java`**, write down **your own** tests methods to test derivative(). You may refer to **`exampleTestUsingPrinting()`** and **`exampleTestUsingExceptions()`** as to how to write your own tests.
3. Compile and run **`UnitTestEx`**. Observe that all the tests fail, because derivative() has not been defined yet.

Part B: Using unit tests to debug buggy code

Now that we have some test methods, we can now proceed with derivative(). Instead of writing this method from fresh, we have provided you with a partially correct version of this method. Your goal is to debug this method until it works correctly. To do so:

1. Go to **`ExampleClass.java`**, comment out the first **`derivative()`** in ExampleClass.java. (If you did not modify this file, it should be line 74)
2. Scroll down, rename **`derivative2()`** to be **`derivative()`**
3. Compile and run **`UnitTestEx`**. Observe that some tests still fail, proceed to debug **`derivative()`** until all tests pass.
4. Give yourself a pat on the back. You have just learned the essence of unit testing.

Part 2: WordCountList [70 Points]

Overview

Your task here is to define a set of methods in **`WordCountList.java`**. Write your well-documented unit tests for the WordCountList methods in the provided file called **`WordCountListTester.java`**

Your personal tests should focus on testing one method at a time with different arguments to ensure each method produces expected result. Once you have finished writing a unit test method, call it in the main method of **`WordCountListTester.java.`** For unit testing to be effective, you should test your code with your pre-written tests after each method is written. If your method fails the unit tests, you can use these failures to improve/debug your method's functionality. Once you've made the correction, test the method again. Refer to A-1, B-1, C-1, D-1 below on how to write unit test for each method.

To promote proper TDD, **there will be an optional early submission checkpoint. You will earn 2 points of extra credit for turning these unit tests in early by Thursday 1/18 at 11:59 PM.** Students will also submit this file at the **final deadline** to show improvements made to the testers.

We provided the full code for **`WordCount.java`**. Do not modify **`WordCount`**. Implement the following methods in **`WordCountList.java`**. The intended use for **`WordCountList`** is:

1. Read in the words from a file.
2. Strip out any common words (i.e., the, a, an) (the file with common words is given to you).
3. Display the most occurring words in the input file to another output file or on console.
4. Be able to know which words occurred the most.

Each of the methods below corresponds to one of the steps above. We have also provided some more detailed explanations on various components in the **Appendix** section.

Do not modify any method signatures, or the grading scripts won't be able to call the methods. You may add helper method implementations, but you may not change given starter code.

A. Read words from file into your program. (20 points)

A-1. Unit Testing

Before writing the implementation for reading from a file, write a test in WordCountListTester.java that will determine whether or not `getWordsFromFile()` is working correctly. Create your own small files to read words from. Your test should:

1. Create a new instance of a `WordCountList` object.
2. Load the words from a file that you created.
3. Verify that the correct words appear the `WordCountList`'s ArrayList.
4. Verify that the `WordCount` objects have the correct counts.

Run the tester before writing the implementation to make sure that it fails. You are free to add additional test cases as in order to ensure that your method program is correct.

A-2. Implementation

`void getWordsFromFile( String filename ) throws IOException`

This method populates the **ArrayList** containing **WordCount** objects for each word in the file. The **throws IOException** clause is there because you will deal with file I/O in this method. Your algorithm will be:

```
for every word in the file
    search for the word in the arrayList
        if the word is present
```

```
            increment its count
    if word is not already in the arrayList
        add word to the arrayList
```

Important:
Make sure that you do not ignore punctuations while saving your words in `WordCount`. For example,

- "Apple" and "Apple!" should be counted as different words in the ArrayList (with different entries)
- Similarly "state", "state," and "state." should be counted as different words

(Although this may not be logically correct, it will make it much simpler for you to implement)

Note:

- Adding a word into the ArrayList should be **case insensitive** and should just keep the String in the list the same as its first occurrence.
- All the input files should be put in the same location as the .java files. You should use relevant path while creating your File object.

B. Remove a given set of common words from your program (10 points)

B-1. Unit Testing

Before writing the implementation for reading from a file, write a test in WordCountListTester.java that will determine whether or not `removeCommon()` is working correctly.

1. Create a new instance of a `WordCountList` object.
2. Load the words from a file that you created.
3. Remove common words using a file that you created.
4. Verify that the correct words were removed from the WordCountList object's ArrayList.
5. Verify that other words were not removed.

B-2. Implementation

`void removeCommon(String omitFilename) throws IOException`

This method will read **each word** from the specified file and remove that word from the ArrayList. Your method need not be efficient (nested for loops is fine). Also, removing a word is case insensitive like in the method before.

C. Define your toString(), writing words to file (15 points)

C-1. Unit Testing

Before writing the methods in part C, write a tester that verifies the correctness of the `WordCountList's toString()` method. You do not need to write a tester for `outputWords()`, but you should still verify its correctness manually.

C-2. Implementation

`public String toString()`

This method is called to format the words in the `ArrayList` of `WordCounts` into a string. If a file contains three occurrences of "this", two occurrence of "is" and one occurrence of "a", then one valid output of the `toString()` is "**`this(3) a(1) is(2) `**". Ordering does not matter, but you will be graded on the correct formatting.

`public void outputWords(boolean printToFile) throws IOException`

This method is called to print the words in the `Arraylist of WordCounts`. This method takes in a `boolean printToFile`. If `printToFile` is false, it should print the result of `toString` on the screen. If `printToFile` is true, it should output to a file `myOutput.out`. **The output file should be written into the current path, and not using an absolute path but a relative path. By default, if you don't provide any path when creating a file for output, Java will create the file in the folder where your java files are.**

Also, you will be graded for formatting of the outputted text. **If your formatting is wrong, up to 5 points will be deducted.** Make sure that spaces and parentheses are consistent with what is provided in the sample output. Check the **"Testing Using WordCountListTester (Sample Output)"** section for more details.

Note: Print a newline after printing all the words in the console or the file.

Things to Test: Make sure the correct action is chosen for the corresponding printToFile boolean. Then, make sure the content is correct. It is possible for the Tester method to open the file that was just written (through the outputWords method call). Make sure to check for consistency between the wordList and the printed words. You may need to verify the print statements to terminal manually.

D. Find n most frequent words (25 points)

D-1. Unit Testing

Before writing the `topNWords` method in part D, write a test in WordCountListTester.java that checks if `topNWords` functions properly (i.e. verify that the method returns the correct words and their respective counts). Your tester should:

1. Create a new instance of a `WordCountList` object.
2. Load the words from a file you created.
3. Call the `topNWords` method.
4. Verify that the WordCount objects in the returned ArrayList contain the correct words.
5. Verify that those words have their correct respective counts.

D-2. Implementation

`public ArrayList<WordCount> topNWords(int n)`

This method will find the top n occurring words as determined by their frequency in the arraylist and returns this list of words as well as their counts as an ArrayList**.** **In the event of a tie, use the first occurring word with that count.**

Hint:
Here is one way to get the top n words from an unsorted Arraylist (we refer to it as the original list here). You can iterate through the list looking for the word with the highest count. Once you have saved this word and its count into another ArrayList, you can make its count negative of its original in the original list so it won't be the one with the highest rank again in the next round of search. You repeat this process until you have located the top n words. When you are done finding the top n words in the original list, you will need to iterate through the original array list again and change the negated counts back to their original values. For instance, if the original arraylist is the following (with word and its count) and the length parameter passed is 1.

```
dog 5
cat 222
snake 10
dophin 200
```

We want to locate the top 2 words in the list (n=2). In the first round of search, your code will locate cat which has the highest count, and make its count negative of its original count int the array list. It becomes

```
dog 5
cat -222
snake 10
dophin 200
```

In the second round of search, the list becomes the following.

```
dog 5
cat -222
snake 10
dophin -200
```

The last step is to go through the original array list again to make all the counts positive again. So the array list changes back to its original form.

You are required to be able to have the method topNWords execute more than once and produce the same results (ArrayList should have the same values). In the case of a tie (two words are equally frequent), you should select the word which occurred first in the original text file.

*****Note 1: You are free to implement any other helper algorithm/method as well to get the topNwords.**

*****Note 2: topNWords method does not print/display anything.**

*****Note 3: If the number of words is less than n, return all those words. For example n = 10 but you have only 5 words, return those 5 words.**

Things to Test: Think about what needs to be tested to ensure correctness. Be aware that this includes Note 1 above.

Part 3: Summary [3 Points]

In **README.md**, write about your test-driven development of your WordCountList. Tell us about how you approached this different angle of development. Provide a program description for each file you worked on. For example, **`WordCountList.java`** is a file you worked on, but not **`WordCount.java`**. However, you can mention WordCount when talking about WordCountList.

The audience of the summary is anyone who knows **no programming or computer science**. As a rule of thumb, write as if you were writing for an impatient 5-year old boy or a non-computer scientist grandmother. This means use absolutely no Java or CSE terms, but high-level terms are fine. The scripts will warn you if you use a word that's too technical and the checks will be including in the grading scripts.

Style Guidelines (Link) [20 Points]

We will grade #1 through #10 in the style guidelines. Two points each.

Part 4: Similarity Detector [+8 Points]

The following methods will be defined in **`Similarity.java`**.

Write a simple plagiarism checker that uses Jaccard Similarity in order to determine how similar two texts are. The two methods you will be writing are the following:

```
public static WordCountList findMax(WordCountList a, WordCountList b)
public static WordCountList findMin(WordCountList a, WordCountList b)
```

They should find the maximum and minimum number of occurrences of each word. For example, if the word “hello” appears 3 times and "hi" appears 1 times in file A, and "hello" appears 2 times and "hi" appears 4 times in file B, the WordCountList returned by findMax will have a WordCount for “hello” with a count of 3 and for "hi" with a count of 4. Likewise, the WordCountList returned by findMin will have a WordCount for “hello” with a count of 2 and for "hi" with a count of 1. If a word occurs in one file, but not the other, then the word occurs in the other file 0 times.

You should use the two above methods to write the following method to find the percent similarity between two documents. This can be found by dividing the sum of all minimum counts by the sum of all maximum counts and multiplying by 100. Remember to use floating point division. Continuing the example, the similarity between document A and B should be 3/7 (42.8571429%) because the sum of all the minimum counts is 3 and the sum of all the maximum counts is 7.

```
public static double getSimilarity(WordCountList a, WordCountList b)
```

We want to check for this because a document that has every word in another document may not be considered similar based on the percentage generated by getSimilarity, but these documents should be similar so we should give our users a warning. Continuing the example, if document A is a subset of document B, this should return True.

```
public static boolean isStrictSubset(WordCountList a, WordCountList b)
```

When you are finished, write a main method in Similarity.java that checks for plagiarism between two documents given at the command line, where the first and second arguments are the names of the files you want to check similarity between. Don’t forget to have the program strip common words from both given files. For example, you can run Similarity.java in the following way:

```
java Similarity odyssey_shortened.txt odyssey_original.txt
Warning! odyssey_shortened.txt is a subset of odyssey_original.txt
The files have a 27.958191100504486% similarity.
```

This is by no means a sophisticated plagiarism detector because it does not consider words with similar meanings, but it does a good job determining file similarity if the files were directly copied.

See **Similarity Checker (Sample Output)** for other cases.

Submitting the Assignment (Link)

Early Submission:

- **`WordCountListTester.java`**

Submission Files (Make sure your submission contains all of the files and that they work on the ieng6 lab machines!)

- **`WordCountListTester.java`** (For early checkpoint and normal submission)
- **`WordCountList.java`** (With methods defined in Part 2)
- **`README.md`** (Program summary)
- **`COMMANDS.md`** (Answers to linux/vim questions)
- **`Similarity.java`** (Extra Credit program)

Maximum Score Possible: 110/100 Points

Read the submission scripts!

Appendix

toString

Every class in Java has access to a method called `toString()`. The purpose of the `toString()` method is to return a string representation of an object.

Suppose you create a class called `Coordinate.java,` in which each Coordinate object has an x-value and a y-value. You write two getter methods for this class: `getX()` and `getY()`. You then write the following code in your main method:

```
Coordinate c1 = new Coordinate(10, 20);
System.out.println(c1);
```

This prints the *memory address* of the object. An example of something you may see on your screen after compiling and running this code is:

```
Coordinate@7852e922
```

However, this seemingly random/"garbage" series of letters and numbers isn't very helpful for us. We can remedy this problem by defining a method called `toString()` in the `Coordinate` class:

```
public String toString() {
      return "The coordinates are: " + this.getX() + ", " + this.getY();
}
```

Now, when we call `System.out.println()` on a Coordinate object, its x and y coordinates will be printed out. For instance, `System.out.println(c1)` will now print:

```
The coordinates are: 10, 20
```

Scanner

Use a Scanner object to read words from the file.
The Scanner Javadocs: http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html
The basic framework to use a Scanner in this PSA is:

```
// Construct a Scanner that reads in words from a file
Scanner input = new Scanner( new File(fileName));
while ( input.hasNext() )  // while there are more words to be read in
{
      word = input.next();  // reads next string
      …
}
```

WordCount

This class is just a pairing of a String (a word) with an integer (the number of its occurrences) for use in your WordCountList class. Your ArrayList will store WordCount objects. Applicable methods are provided.

WordCountListTester

This file contains some default testers for you to test the validity of the methods in your WordCountList.java file. It can simply be run by compiling along with the other files on the command line and running it.

If you are intent on completing the unit testing extra credit portion of the PSA, consider using this file as a reference in thinking about how you can write unit tests for individual methods as you develop the PSA.

ArrayLists

ArrayLists are dynamically resizing arrays. In Java, standard arrays are initialized to a fixed size when first created. However, with ArrayLists, there is an initial size but when it becomes full, it automatically enlarges. When an element is removed, it automatically gets smaller.

Go to the Useful Links section on the course website for a link to Java API where you can find information about ArrayLists and the methods they have. Some suggested methods to use are : get, size, add, and remove.

Testing Using WordCountListTester (Sample Output)

To use WordCountListTester, run a line like:

```
java WordCountListTester nameOfInputFile.txt numberOfTopNWords {file|console}
```

The following is an example of using the WordCountListTester on the provided file (harry_potter.txt), printing the top 10 words that start with the letter 's' on the console. Here the first argument is the name of the file. The second argument is numberOfTopNWords (number of most frequent words to be printed). The third argument is console (printing it to the console). The fourth argument is char indicating that we will be finding top numberOfTopNWords which begin with a given char value. The last argument is the value of the char i.e 's'

General format of printing the words with their counts is : **`word(count)<space>`**
You need not print every word on a separate line.

1. The example below shows the output on the console. Note your display might be different from the following based on the width of your console.

```
> java WordCountListTester inputFiles_DoNotSubmit/harry_potter.txt 10 console
Reading in File: inputFiles_DoNotSubmit/harry_potter.txt
Removing common words
Printing the top 10 words  on console
Dursley(37) Mr.(29) Mrs.(19) --(13) people(12) cat(8) he'd(8) Dudley(7) owls(7) called(6)
```

2. The example below shows the output printing to a file called myOutput.out

```
> java WordCountListTester inputFiles_DoNotSubmit/harry_potter.txt 10 file
Reading in File: inputFiles_DoNotSubmit/harry_potter.txt
Removing common words
Printing the top 10 words  in file a named myOutput.out
```

And a file called myOutput.out should be created with the following contents:

```
Dursley(37) Mr.(29) Mrs.(19) --(13) people(12) cat(8) he'd(8) Dudley(7) owls(7) called(6)
```

Note: If you don't include any arguments, you may simply use the terminal command **`java WordCountListTester`** in order to run your own testers.

Similarity Checker (Sample Output)

```
javac Similarity.java
This program takes exactly 2 arguments!

java Similarity odyssey_original.txt odyssey_shortened.txt
Warning! odyssey_shortened.txt is a subset of odyssey_original.txt
```

```
The files have a 27.710866853757405% similarity.
```

java Similarity odyssey_shortened.txt odyssey_original.txt

```
Warning! odyssey_shortened.txt is a subset of odyssey_original.txt
The files have a 27.710866853757405% similarity.
```

java Similarity starwars.txt harry_potter.txt

```
The files have a 1.3054830287206267% similarity.
```

java Similarity warandpeace.txt warandpeace.txt

```
Warning! warandpeace.txt and warandpeace.txt have the same contents!
The files have a 100.0% similarity.
```