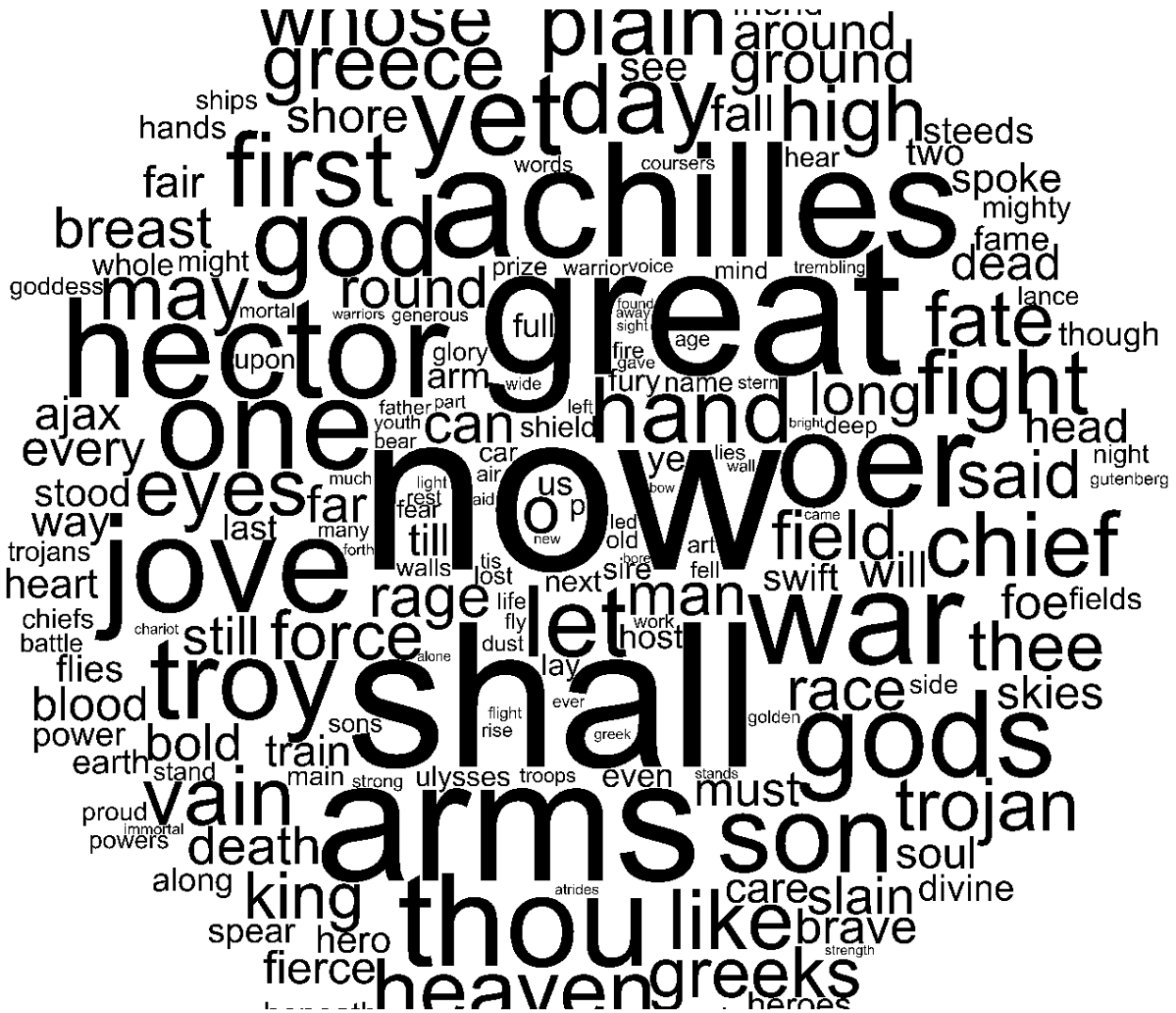
Project 3: What's the word?

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

(Version 2, Last updated April 30, 2019)



The illiad in 200 words (and not quite enough pixels)

1 Introduction

One of the most interesting areas of ongoing research in the field of machine learning is models of human text. There are many algorithms, which work many different ways to achieve various interesting goals. At the heart of all of these algorithms, however, is the ability to write a program to read large quantities of written text, and produce numerical summaries of this information. Since text documents such as books can contains hundreds of thousands of words, it is very important that algorithms for this task are efficient.

In this Project we will be exploring the very basics of large document summarization. Put simply, we will write a program that can, efficiently, read a document containing hundreds of thousands of words in less than a second, and produce a count of how many times every word in the document appeared. From there we will sort this list and produce a list of the 100 most common words.

While this is not, strictly speaking, the end of the assignment, it is mostly the end of the programming. That said, It wouldn't' be a fun assignment if we stop there. Therefore, included with the other starter code for this assignment is a class named `WordlGenerator`. This class contains all of the necessary code for generating a basic word cloud. A word cloud is a simple, but fun, visualization of word use frequencies. While research has shown they are not more effective than more traditional visualizations, it's hard to deny that they are a heck of a lot of fun!

1.1 Learning Goals

This assignment is designed with a few learning goals in mind:

- Implement a binary search tree based datastructure
- Solve a large, realistic problem.
- Create a word cloud

2 Theory: Text summarization

A common problem in traditional text summarization is that there are some words that are common to all text. Examples of these words are "of, the, a" these are words that fill a syntactic role in the English language. As such, these words are going to be very common in every body of text. While it may be true to say the most common word in the Illiad is "the", followed by "and", "of", and "to" it is not particularly enlightening to our understanding of what makes the story unique.

In Text summarization theory these words are called stop words. While not always true, commonly these words are simply ignored. We will adopt this strategy and likewise ignore stop words in our summary. Included in the provided files for this program you have been provided a file "StopWords.txt" that contains a list of stop words. Your program should read this file and ignore every word on this list.

2.1 Data files

No good e-book based assignment would be complete without a trip to Project Gutenburg. Project Gutenburg (`https://www.gutenberg.org/`) is a website that collects of e-book copies of public domain books in a variety of formats. While you aren't going to find the newest game of thrones on there, it is a fantastic collection of classic English literature (Mostly older works that are no longer covered by copyright protections). For this project you are expected to find and download the Plain Text UTF-8 format copy of a book of your choice. This book will serve as the primary document you will summarize.

3 Provided Code

I will be providing two java classes to help you in this process. roughly speaking these handle the more annoying (and more irrelevant to the learning goals) parts of the program at the very beginning and end of the process.

3.1 WordIterator

Reading a text file word-by-word turns out to be an annoying process, and not one that is particularly enlightening. Therefore we will be providing an the `WordIterator` class. This class implements the Java `Iterator<String>` interface allowing you a standard way to read a file one word at a time. The constructor takes the name of a text file. After that the object behaves as a standard iterator object, reading in the given file one word at a time.

3.2 WordlGenerator

This class contains code to generate a word cloud based on the Wordl algorithm. This code is not perfect. Among other things it is rather slow. (This is primarily a flaw with the algorithm from what I can tell, as the algorithm runs in time somewhere between $O(n^2)$ and $O(n^3)$). The code will work with the model objects you have created and will create an internal representation of a word cloud image. After creating this internal representation the save method can be called to create a png file containing the word cloud image. This class has several methods that allow some partial control over the image that is generated which you may want to play around with. You should look over the class file itself for further information on how to use this class.

You are allowed to change this file if you so wish, so long as your changes lead to more interesting word clouds. (For example, the code `at.concatenate(AffineTransform.getRotateInstance(Math.rando` can be placed on line 127 to make each word randomly rotated!

3.3 Standard Java Library

Unlike Project 2, you are allowed to use some of the classes in java's standard library. In particular for part of your assignment you will be asked to return a `List`, which should be some

implementation of the standard Java list interface with either `ArrayList` or `LinkedList` being reasonable. Additionally, there are several utility methods defined on class `Collections` which you are free to use. Please consult the formal java documentation (`https://docs.oracle.com/javase/8/docs/api/?java/util/Collections.html`) for more information on this class.

4 Software Setup

You can use whatever programming environment you wish for this assignment, however the recommended programming environment is IntelliJ. The reason this is worth mentioning is that each programming environment may have slightly different requirements on where to put ebook files and how to name those files. In IntelliJ, if you put an ebook file directly in the top level folder of your IntelliJ project (not under `src/`, but directly in the project folder, so in the same folder as `src/`) then you can access this file simply by file name. If you wish to put your ebook file somewhere else, or you are using a different environment you may need to open the file with a complete file path. I.E. instead of calling the constructor for `WordIterator` with `pg6130.txt` you might need to call it with `C:\Users\Daniel Kluver\Documents\2019-1Spring\CSCI1913s2019\Project3\Solution\pg6130.txt`. Getting this right can be rather difficult in practice, therefore it's recommended you simply place the file carefully in IntelliJ.

5 Implementation

You will be required to implement three classes for this project.

- `WordCount` - this class will represent a single word, and the number of times it has been seen.
- `WordCountTree` - this is a custom purpose data structure with some properties of the Map data structure, and some properties of the Set data structure. You will implement this using a BinarySearchTree. To the program, this data structure represents the count of how many times any large number of words has been seen. It therefore should have efficient methods for adding to the count for a word, getting the count for a word, and checking if a word has been seen before (has a non-zero count).
- `BookStats` This class will have a variety of useful methods for summarizing a book, as well as having the main method for your assignment.

5.1 WordCount

Class `WordCount` should implement the interface `Comparable<WordCount>`. It should track a single word, and the number of times that word has been seen. It should have the following constructors and public methods. Take careful note of the exact spelling and capitalization of these words as we may use automated grading on this assignment.

- `public WordCount(String word, int count)`
  Constructor. Create a new word count object for a given word with a given initial count.

- `public WordCount(String word)`
  Constructor. Create a new word count object for a given word, with a default count of 0.

- `public String getWord()`
  Getter. Get the word for this object.

- `public int getCount()`
  Getter. Get the current count for this word.

- `public void addCount()`
  Increment the count for this word.

- `public String toString()`
  Print a description of this object. The format should be as follows: `"cheese (42)"` (assuming the object represents that the word cheese has been seen 42 times).

- `public int compareTo(WordCount other)`
  This is the method for the comparable interface. A WordCount should sort on it's count, so a word count should be considered greater than another word count if it's count is greater than the other wordCount's count. While this may seem counter intuitive, it ends up being useful later.

5.2 WordCountTree

Class `WordCountTree` will track the word counts for many words. It will do this with a customized Binary Search Tree. It should have the following public methods:

- `public WordCountTree()`
  Constructor. This should create an empty word count tree. This method should run in $O(1)$ time.

- `public void count(String word)`
  This should increase the count associated with a given word. If this is the first time seeing the given word this will likely mean adding a new node to the tree and setting it's count to 1. Otherwise this will involve incrementing the count at a given tree node. This method should run in $O(log(n))$ average-case time (where n is the number of words in the tree).

- `public int getUniqueWordCount()`
  This should get the number of distinct words that are tracked by this tree. Since the tree needs to track every word with a non-zero count, this should be equal to the

number of distinct strings on which count has been called, and likewise it should equal the number of words with a non-zero count. This method should run in $O(1)$ time.

- `public int getTotalWordCount()`
  This should get the total number of words counted by this tree. This should be equal to the number of times count has been called. This method should run in $O(1)$ time.

- `public boolean contains(String word)`
  This should check if a given word is counted in this tree, I.E. has a non-zero count. If this word is in the tree it should return true If this word is not in the tree it should return false. This method should run in $O(log(n))$ average-case time (where n is the number of words in the tree).

- `public int getCount(String word)`
  This should return the current word count for the input word. This should be equal to the number of times the count method has been called. If the word is not in the tree, it should return 0. This method should run in $O(log(n))$ average-case time (where n is the number of words in the tree).

- `public List<WordCount> wordsInOrder()`
  This should return a `java.util.List` of `WordCount` objects representing every word in the tree. Furthermore, this list should be sorted such that the first element is the most common word, the second element is the second most common word etc. **note - this should not be sorted in word order**, it should be sorted based on count with larger words earlier in the list.

To make these methods possible the `WordCountTree` class should have an nested class named `WordBSTNode`.

5.3 WordCountTree.WordBSTNode

The class `WordCountTree.WordBSTNode` (that is, the class `WordBSTNode` nested in class `WordCountTree` should have three instance variables, an instance of `WordCount` storing the word and count for this node, and two references to `WordBSTNode` left and right. The requirements on this class are somewhat looser than the others as it has no public behavior. You will likely want one or two constructors, getters and setters.

To make the `WordCountTree.count`, `WordCountTree.contains`, `WordCountTree.getCount`, and `WordCountTree.wordsInOrder` methods possible you may need recursive methods on this class, these methods may also be on the `WordCountTree` class, or be non-recursive (so long as the implement appropriate binary search tree actions)

5.4 BookStats

The `BookStats` class is essentially the class for the main method of this program. However, to make this easier to grade, we specify three specific static methods it should implement

(in addition to a main method). Taken together these will make the main method simple to implement, (and easier to test).

- `public static WordCountTree readBook(String fileName, WordCountTree ignoreWords)`
  This method should use the `WordIterator` class to iterate over each word in a file and count them. The second parameter can be null or not null. If the second parameter is null the method should simply count all words in the file and return a `WordCountTree` that stores those counts. However, if the second parameter is non-null, every word in the ignoreWords tree should be ignored. In this way the same method can be used to load stop words and the e-book itself. As a side-effect, this function should also print the amount of time it took to read the document as measured in milliseconds. The static method `System.currentTimeMillis()` can help you with this by returning a `long` containing the current time whenever called.

- `public static void render(WordCountTree wc, int max_words)`
  This should use the `WordlGenerator` to generate a word cloud based on the given `WordCountTree`. As word cloud generation is a time consuming task, we would not want to plot every word in a long document, therefore the second parameter says how many words should be drawn. You are free to configure the generator however you want (within reason - don't make font be size 1 to 3, for example, since that's unreadable)

- `public static void summarize(WordCountTree wc)`
  The summarize method should print a brief summary of the input Word counts. In particular it should print the total number of words, the number of distinct words, and the top 25 words by word count.

- `public static void main(String[] args)`
  The main method should read the stop words file, and then read a book of your choice (ignoring stop words). It should print a summary of the e-book of your choice, and then it should render a word cloud of your book of choice.

6 Limitations

There are a few limitations for this project. First and foremost, as usual this is NOT a collaborative project. You should not show your code to anyone, or look at any other code for this problem, nor should you closely discuss solution details with anyone else. If you need help on any part of this project please email the course staff and we can provide assistance.

Secondly, Binary search trees, and word counting projects in general are not particularly uncommon programming assignments. You should be very careful if you search for assistance with this project not to search for implementation of these classes. You are allowed to reference any code from the textbook or written in class, but to maximize your own personal learning you should try to do as much of this as is possible independently. Looking at ANY implementation of these classes or related classes outside of the textbook and lecture notes

will be considered a breach of academic integrity. If you need help on any part of this project please email the course staff and we can provide assistance.

7 Testing and incremental development

- The requirements on the word cloud your code generates is very loose. So long as it can generate a word cloud I'm going to be happy. I challenge you, however, to try to make as interesting and artistic word cloud as possible.
- We will be using, at least partially, automated tests. Therefore it is very important you pay close attention to naming and capitalization.
- A test file for the WordCount class and the WordCountTree class has been posted online.
- You should plan time for personal testing of your code. This could make many forms, it could be as simple as writing a main method for each class for testing purposes, or as complicated as creating a custom text file with known word counts.

As a tool to help immediate testing and to document expected formats for `toString` on class `WordCount` and the formatting for the various `BookStats` methods, below is my program's output when given The Iliad of Homer by Homer

```
Read StopWords.txt in 12ms
Read pg6130.txt in 380ms
total words: 111356
distinct words: 13034
top 25 words
now (576)
shall (511)
great (475)
arms (448)
achilles (418)
hector (372)
jove (367)
oer (365)
war (360)
one (351)
thou (350)
gods (329)
yet (316)
troy (314)
god (312)
son (304)
```

```
first (293)
let (280)
day (279)
hand (264)
chief (263)
fight (261)
plain (260)
heaven (253)
like (246)
```

8 Deliverables

For this project you should submit the following files:

- BookStats.java
- WordCount.java
- WordCountTree.java
- WordlGenerator.java (optional - include if you end up making changes to this file in the interest of superior word clouds)
- your ebook of choice (It's OK to redistribute these ebooks yay!)
- An image file containing the word cloud for your ebook of choice. This should be rendered by your code.

As well as the file for any helper classes you may write. We should be able to run your code given only the files you submit, so do not forget to include any necessary file.

I've been told that canvas does allow multiple file submissions, However, to make sure there are no errors on our part in managing these files, we ask that you submit a zip file containing these files. Please email or attend office hours if you need assistance submitting a zip file.

Please name your submission `<YourName>_Project3.zip` before uploading it to Canvas. (We will handle unzipping the files and renaming the file for testing as needed.) This will help us resolve any issues with identifying files during automated testing that might arise. Your work must be submitted on Canvas by Monday, May 6th at 6:00PM.