In this project, we will do some interesting analysis on some of the most famous literature works while honing our skills working with strings, arrays, hash tables, trees, and more!

Project Gutenberg (<http://www.gutenberg.org/>) is a massive online effort to digitize and archive popular books as a way to encourage the creation and distribution of eBooks. Over 57,000 of the world’s greatest literature have been digitized.

Many of these digitized works have been added to the GITenberg github repo (<https://github.com/GITenberg/>).

**Before you start, create a new GitHub repo. This will be where you push all your code and final analysis to for us to grade. For submission we will be asking you to submit the link to this Github repo. We expect to see at least 1) A PDF document that summarizes your findings as outlined in step 8 and 2) At least one Java / Python class which contains all the methods specified in step 9 below**

**If you are unfamiliar with GitHub and how to set up and push a project, we have outlined the steps here :** [**https://courses.codepath.com/courses/advanced\_software\_eng/pages/submitting\_assignments**](https://courses.codepath.com/courses/advanced_software_eng/pages/submitting_assignments)

1. **Choose the novel that you want to work with on this project. Be sure to pick a novel that has clearly defined Chapters (ie, somewhere in the text it says Chapter 1 , Chapter 2, etc) as we will need to analyze things by chapters in future steps:**
   1. You can browse the most popular novels here if you have no idea how to choose from the massive collection: <https://www.gutenberg.org/browse/scores/top>
   2. Once you’ve selected a novel, find its corresponding github repo through <https://github.com/GITenberg/> and clone the repository. For example, if I selected Thoreau’s Walden, I would search for ‘Walden’ from the GITenberg home page and then download the repo here <https://github.com/GITenberg/Walden-and-On-The-Duty-Of-Civil-Disobedience_205>
   3. Notice that the repositories all have at least one .txt file. This is the file we’ll be working with to do our analysis. Some repositories may have more than one .txt file. In this case, each .txt file corresponds to a different version of the work. It shouldn’t matter which specific .txt file you chose to work with.
2. **Implement your own algorithms to perform text analysis on the novel** 
   1. Once you’ve selected and downloaded the text you want to work with, the next step is figuring out how to process the .txt file. **Create a method that will take in a .txt file and return the number of words in the file. This method should be named getTotalNumberOfWords() .** Note that the text file you downloaded may contain a lot more than just text from the book (for example, the index, author’s notes, or introductions). You can manually edit the text file to take out extra text that’s not part of the book.
   2. Now that we’ve figured out how to process the contents of the file, we can start doing some interesting analysis on the words we see. **Create a new method that returns the number of UNIQUE words in the novel. This method should be named getTotalUniqueWords() .** (For example, in the sentence “The blue cat hopped over the blue dog, there are 6 unique words. We don’t count ‘the’ and ‘blue’ twice)
   3. Now that we’ve warmed up in working with text files and strings, let’s dig into something more a bit complicated. **Implement an algorithm that will return the 20 most frequently used words in the novel and the number of times they were used.** **This method should be named get20MostFrequentWords()** Your method should take in a .txt file and return an array of words and the number of times they were used. For example, if my book was Harry Potter and the Prisoner of Azkaban, my method may return something like : [ [“Harry”, 526], [“owl”, 256], [“Hogwarts”, 135], … ] . Notice that each element in the array is another array of size 2, where the first element is a string and the second is an int.
   4. You may start to realize that common english words like “the” or “and” appear very frequently. It wouldn’t be very interesting if our list just returned the 20 most common english words. **Implement a new algorithm that filters the most common 100 English words and then returns the 20 most frequently used words and the number of times they were used. This method should be named get20MostInterestingFrequentWords()** There’s a compiled list of the top 1000 most commonly used English words here : <https://gist.github.com/deekayen/4148741>. Since the list gives us 1000 words, feel free to tune your algorithm to filter the most common 100, 200, or 300 words and see how it affects the outcomes.
   5. **Implement a new algorithm that returns the 20 LEAST frequently used words and the number of times they were used. This method should be named get20LeastFrequentWords() .** There are probably going to be more than 20 words that are seen only once. Feel free to just pick whichever 20 words you’d like from that set.
3. **Analyze the progression of the words used on a chapter-by-chapter basis**
   1. We’ve analyzed the text as a whole, but some of the most interesting analysis comes when we break things down by chapter. **Implement a method that can take in a word and return an array of the number of the times the word was used in each chapter. This method should be named getFrequencyOfWord()**For example, if my work of literature was Harry Potter and the Sorcerer’s Stone, and I inputted the word “Voldemort” into my method, it should return something like [1, 0, 0, 2, 0, 0, …. , 10, 12, 20, 34, 2]. The size of the array should be equal to the number of chapters in the book. With this output, we can see that “Voldemort” was mentioned infrequently in the beginning parts of the book, but towards the end he becomes more central to the story.
   2. The hard part about this step is finding something interesting to present about. You may have to run this method of multiple different words before finding an interesting pattern. For example, in Moby Dick, the word “death” appears rarely in the beginning parts of the book, but very frequently towards the end. One way to choose what words to analyze could be to look at the list of the 20 most common words you found in step 2. Character names, settings, and words that describe emotion are also common in exhibiting interesting frequency patterns.
4. **Implement a way for us to find out what chapter a certain quote from the book can be found in**
   1. Your method take in a string (the quote) and return a number (the chapter number) **and be named getChapterQuoteAppears() .** If the quote cannot be found in the book, your method should return -1.
      1. For example, if we were looking at Harry Potter and the Sorcerer’s Stone, and we wanted to find which chapter the quote “Harry yer a wizard” was in, we would return 4 (<http://www.hogwartsishere.com/library/book/7107/chapter/4/>)
5. **Write a sentence in the author’s voice by implementing a method named generateSentence()**
   1. Many writers have a unique type of writing style that can be easily recognized based on the types of words they use and their sentence structures. For this part of the project, we want to generate a sentence in the author’s writing style. In order to do this, we will generate a sentence word by word. We will start off our sentence with the word ‘The’. To generate the rest of the sentence, we will parse through the book, look for all the instances of the word ‘the’, store the word that comes after ‘the’, then randomly pick one of the words. We repeat this process 20 times until we have randomly picked 19 other words to complete our sentence.
      1. To get a better understanding of how to approach this, take a look at this guide here : <https://docs.google.com/document/d/11jfZCa_iHiwZjF26R7MfEi5HiIUTNO8hvV23ubdSkTs/edit?usp=sharing>
      2. The sentence you generate may not make a lot of sense. Try running this method several times until you get a sentence that you like.
6. **Sentence Completion / Prediction with Tries (OPTIONAL)**
   1. We want to create a simple autocomplete-like feature where we can input one or more words, and be returned a list of all sentences that can start with our input. **Implement a method List<String>   
      getAutocompleteSentence(String startOfSentence) that takes in a string, and returns a list of strings that start with the input s.** This will require you to first build out a trie with the available words. If you want some guidance on how to approach this, this article may be helpful <https://medium.com/@dookpham/predictive-text-autocomplete-using-a-trie-prefix-tree-data-structure-in-javascript-part-1-6ff7fa83c74b>
      1. Researching and Googling around for “How Autocomplete Works” or “Building Autocomplete with Tries” should also help you move forward
7. **Finding closest match (OPTIONAL)**
   1. Often times, people will try to search for a vaguely accurate quote from a book on Google and be able to find the exact quote. For this challenge, your task is to implement a method defined as **String** **findClosestMatchingQuote(String s)**. This method will take in a quote, and be able to return the chapter this quote is found in. The catch is that the method could take in a misquoted quote , but still be able to find it. For example, the famous Harry Potter quote ‘Yer a wizard, Harry!” is actually “Harry - yer a wizard” in the book.
      1. This is left open-ended so you can do your own research and determine how to best approach this problem. Feel free to use any data structures you seem fit.
8. **Compile your analysis and findings in a document named final\_analysis.pdf**
   1. A simple way to document your analysis would be to place it all in a text file. If you want to get creative and personalize this, visual infographics are a great way to present data. A word document with just plain ol’ text will be very sad to see. If you don’t want to go all out on making an infographic, it would be nice to at least give the text some formatting and include a picture of your favorite cover of the book you analyzed. **Your document should contain at LEAST the following information:**
      1. **Title of the novel you analyzed**
      2. **Total number of words in the novel**
      3. **Total number of unique words in the novel**
      4. **20 most frequent words in the novel including the number of times they were used**
      5. **20 most INTERESTING frequent words in the novel including the number of times they were used**
      6. **20 least frequent words in the novel including the number of times they were used**
      7. **From Step 3, a specific word you found that had an interesting pattern usage through the book. Include the word and it’s frequency of usage for each chapter in the novel. Write a one sentence summary about why it’s pattern was worth noting.**
      8. **An interesting quote from the book and the chapter it’s found in.** If you don’t know what quote to choose, try Googling for quotes from the book: ie “Quotes from Harry Potter”, and chose your favorite one from the list.
      9. **The randomly generated sentence from your generateSentence() method**
   2. **Save the document in PDF form and push it to your GitHub repo. The file should be named final\_analysis.pdf**

Here is an example final\_analysis document:

[**https://docs.google.com/document/d/1z6TFJ2roWxNCNWPyaU2MekJ5fbVHK6GX5CsvKrSOl2Q/edit?usp=sharing**](https://docs.google.com/document/d/1z6TFJ2roWxNCNWPyaU2MekJ5fbVHK6GX5CsvKrSOl2Q/edit?usp=sharing)

Infographic ideas for inspiration:(<https://moz.com/blog/10-tools-for-creating-infographics-visualizations>) (<https://venngage.com/templates>) (<https://www.good.is/infographics/sponsored-infographic-7-things-you-didn-t-know-about-the-golden-gate-bridge#open>)

1. **SUBMISSION**
   1. What we will be looking for in your submission:
      1. A **PDF file named final\_analysis.pdf** of interesting tidbits of analysis as outlined in step 7.
      2. The code you used to come up with these analysis. **Your code should include at LEAST the following methods (possibly more):** 
         1. **getTotalNumberOfWords()**
         2. **getTotalUniqueWords()**
         3. **get20MostFrequentWords()**
         4. **get20MostInterestingFrequentWords()**
         5. **get20LeastFrequentWords()**
         6. **getFrequencyOfWord()**
         7. **getChapterQuoteAppears()**
         8. **generateSentence()**