Solution strategy:

First I deconstructed the criteria into pseudocode, and split the program into functions. I then determined that I would use a **string pool** for the storing of words, with **parallel arrays** determining the start of each word and the count of each word. The text file is read in line by line into a process function, which breaks down the line into lowercase, punctuation-free words, which are stored by the store function. Store compares the first char of each word in the array with the word being input, and if the char matches then it continues to the second char, until the final char is compared, and if matching, will be added to the corresponding count. If no matches are found then it is added onto the end of the string pool. Then it is sorted by an **insertion sort,** which iterates through all words, and if a count is less than the count at the index before, it swaps. If a count is the same as the index before, it checks to see if the words are in alphabetical order, and if they aren't, then it swaps.

Data structures

String pool

Where: Line 17.

Reasons: The string pool allows me to effectively make comparisons without having to compare an entire word. It also means that I am not wasting memory on dynamic data.

Parallel arrays

Where: Line 19.

Reasons: Allows for logical navigation between related data. Also easy on memory.

Algorithms

Insertion sort

Where: Line 135.

Why: The insertion sort allows me to iterate through each word and sort by count and alphabetically in one for loop. For all words, while count[n-1] > count[n], swap count and word. Following that, while count[n-1] == count[n], swap word.