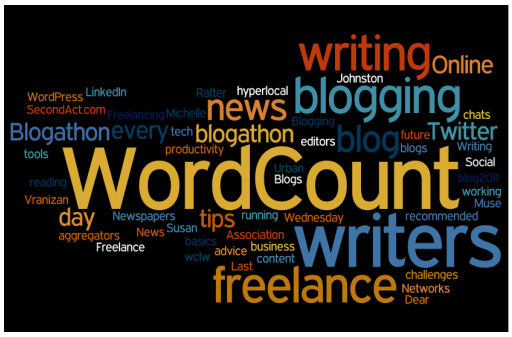
Homework Assignment 6
CS161, Spring 2017
Term Frequency



Have you ever wondered how Google ranks the web pages that it returns when you do a search? Not everything is known of course, but two things are: 1: How many pages link to a page and in which way, and 2: How many times words appear on the webpage. For example, suppose an Internet citizen searches for a webpage containing the word "Spock" (warp 9 sir). Whichever page mentions Spock the most will rank higher than pages that rarely mention Spock (in combination with page links). This is pretty straight forward, except that really common words, like "the" and "it," have to be weeded out. So words that are common on all webpages are given lower ranking when weighing words on a page. This is called "term frequency - inverse document frequency," and if you are interested, here is the Wikipedia page: http://en.wikipedia.org/wiki/Tf–idf.

Assignment Description
Write a program that counts the words in a text document. Then report what the most used word in the file is (the word with the highest frequency) and how many times it appears, and the least used word in the text file (least frequency) and how many times it appears. There might be more than one word that appears once, so you can report any one of them. Though less likely, there might be more than one "most-used" word, and you can report any one of them. Two testing text documents will accompany this document in the assignments folder.

Assignment Requirements
You can either use 2 parallel arrays, or an array of structs to do this assignment. You may not use the vector class, and do not use any C++ 11 containers (or pairs, or tuples, or any other class or structure that is not covered in chapters 1 through 10 of the textbook). Do not use any global variables (global constants are OK). Therefore all of your variables should be local to main or some other function. If a variable in one function is needed in a different function, then send it as a parameter.
If you use 2 parallel arrays, one will hold the words and the other will hold the counts. So the arrays might be declared as:

```
int counts[ARRAY_SIZE]; // remember, no literals,
string words[ARRAY_SIZE]; // use constant globals.
```

Notice that I'm using ARRAY_SIZE, which is a constant global. If you choose to use an array of structs, it might look like the struct to the right in the textbox. You may add more items to the struct if you need them. If you choose an array of structs, then to declare an array, you would use:

```
wordInfo words[ARRAY_SIZE];
```

```
struct wordInfo {
    string word;
    int count;
};
```

When opening the text file, make sure that the file exists before trying to read from it. You can use the input stream method is_open() to check that the file is open. If the file does not open, use a loop to continue to ask for a file that exists. If the user decides not to open a file, then allow the user to quit. If a file is successfully opened, then use an eof-controlled loop to read every word from the file. Each word from the file will then be stored and counted using the methods described below.

Case should not be a factor in counting words, so create a function to convert a string to lowercase (or uppercase, whichever you prefer). The function header might look like:

```
string convertCase(string word)
```

For this strategy, you can convert the word first, and then add it to the array. Remember that a string is a collection of chars, so you can convert each individual char to lowercase or uppercase using tolower() or toupper(), inside of a for-loop. toupper() and tolower() are part of the <cctype> library.

To search for a particular word, create a function `search` that will take at least 3 arguments: the array of words, the word to search for, and the number of words in the array. If a word is found in the array, then return the index of that word. If the word is not found, then return `NOT_FOUND`, which is a constant global:

```
const int NOT_FOUND = -1;
```

I chose -1 for NOT_FOUND, because indices for arrays always start at 0. Therefore -1 is not a valid index and can safely be used to signal that an error has occurred. If you use a struct array, then the heading for the search function might look like:

```
int search (wordInfo words[], string searchWord, int totalWords)
```

or for parallel arrays,

```
int search (string words[], string searchWord, int totalWords)
```

After you have loaded and counted all of the words in a file, you will need to call a function to find the word with the lowest frequency (used the least) and a function to find the word with the highest frequency. The headers for these functions might look like:

```
int findIndexOfMost (int counts[], int totalWords);
int findIndexOfLeast (int counts[], int totalWords);
```

Both of these functions are for use with parallel arrays. If you choose to use an array of structs, the headers would be:

```
int findIndexOfMost (wordInfo words[], int totalWords);
int findIndexOfLeast (wordInfo words[], int totalWords);
```

Make sure that you do not run out of array space, or have an index out-of-bounds. Terminate the program if you run out of room in your array. To do this, compare your loop counter and/or index to ARRAY_SIZE (which is the constant global you set up when you declared the array). If totalWords (or other index value you may have) is equal to ARRAY_SIZE, then there is no more room in the array.

Strategies

Suppose you have set up two arrays, one to store the words and the other to store the counts:

```
string words[ARRAY_SIZE];
int counts[ARRAY_SIZE];
```

Then we will need a variable to keep track of the number of unique words:

```
int totalWords = 0;
```

Initialize the array or arrays by setting all locations in the count array to zero and the string array to the empty string (or the values in the structs, if you use an array of structs). To begin with, totalWords should be initialized to zero (because no words have been loaded yet). Once you read a word from the file, search the word array for it by calling the search function. Suppose you have set up another variable called index and a string to store a word that you have loaded from the file:

```
int index;
string searchWord;
inFile >> searchWord; // inFile will be set up elsewhere.
```

Then you could call search like this:

```
index = search(words, searchWord, totalWords);
```

If searchWord is found in the words array, then index will contain the index location. All you have to do then is increase the count of that word:

```
counts[index]++;
```

or

```
counts[index] = counts[index] + 1;
```

If the word is not found in the words array, then add it to the words array at the next available location, and keep track of how many times we have seen it (once):

```
words[totalWords] = searchWord;
counts[totalWords] = 1;
```

Then, since we added a brand-new word, increment totalWords: `totalWords++;`
The variable totalWords will only be incremented when our program encounters a brand-new word, but not for repeated words.

This is kind of tricky; let's take a look at a specific example.
Suppose the first word you read from the file is "the." This is the first word, so it won't be found in the words array. So the word "the" will be placed at the first index of the words array, and the first counts index will be set to 1. Then totalWords will be incremented. So the state of our arrays and totalWords is:

```
totalWords = 1    // We've seen one unique word.
counts[0] = 1     // The first word has been encountered once.
words[0] = "the"  // The first word encountered is "the".
```

Then suppose the next word is "person." This word will not be found either, so the process will be repeated: add this word to the words array at index 1, set the count array at index 1 to 1, and increment totalWords. So the new state of our variables is:

```
totalWords = 2
counts[0] = 1, counts[1] = 1
words[0] = "the", words[1] = "person"
```

Remember that parallel arrays use the indices to associate information about things. So counts[0] is the count of the word at words[0], counts[1] is the count of the word at words[1], etc.
Then suppose you read another "the" from the file. This time when you search for "the," it will be found at index 0, so increment the count at index 0 (from 1 to 2). This time we didn't see a new word, so totalWords will not be incremented. So at this point, we have seen two unique words ("the" and "person"), the word "the" was counted twice, and "person" was counted once. So the state of the arrays and totalWords would be:

```
totalWords = 2  // We have encountered two unique words.
counts[0] = 2, counts[1] = 1 // First word read twice, second word once.
words[0] = "the", words[1] = "person"
```

You might notice that totalWords is doing double duty: it contains both the next index for a new word, and it tells us how many words are already in the array. It starts out at zero when there are no words in the array, and the first index should be zero. After you have added a word, increment totalWords so that it will equal 1. Now we know that there is one word in the array, and that the next word should be placed at index 1.

In the body of search, use a loop to compare all of the words in the array to searchWord. Stop searching the array when there are no more words (when your loop counter is equal to or greater than totalWords). Let's write some pseudo code for that:

```
int search (string words[], string searchWord, int totalWords) {
    int index = NOT_FOUND; // Assume that searchWord is not in words.
    // Set up a loop that stops at totalWords and has counter i.
    // If searchWord == words[i], then index = i and break.
    // End of loop.
    // Return index.
}
```

After the function search returns, index will either be NOT_FOUND, or it will contain the index of the word in the words array. If index is equal to NOT_FOUND, then you know you should place the new word at location totalWords: `words[totalWords] = searchWord;` Then set the counts array at totalWords to 1, and then increment totalWords.

Otherwise, if index is not equal to NOT_FOUND, it will contain the location of the searchWord in the words array. If this is the case, simply increment counts at index: `counts[index]++;`

When you reach the end of the file, you will have loaded and counted all of the words in the file. So then it will be time to find the least and the most used words. These functions will be identical except for the relation (> or <), so let's use pseudo code to describe one of them:

```
int findIndexOfMost (int counts[], int totalWords) {
    int most; // a locally declared variable.
    // Assume that the first location of counts is most.
    most = counts[0] // We have to start somewhere...
    // Set up a loop that stops at totalWords.
    // Suppose your loop counter is called i.
    // Compare most to all other locations: if (counts[i] > most)
    // If true, then set most equal to the new value.
    most = counts[i];
    // End of loop.
    // return i, which is the location of the most used word.
}
```

Then all you have to do is output the most used word and how many times it was used:

```
cout << words[i] << counts[i]; // With more info of course.
```

To make sure you don't run out of array space, set the ARRAY_SIZE constant to a large number, such as 1000: `const int ARRAY_SIZE = 1000;` Keep track of the current array location when you add words, either using totalWords or some other index counter. If the current new word is being added to the array at the location that is equal to ARRAY_SIZE, report that the program ran out of memory and terminate the program.

There are text files available in the assignments folder on D2L along with this document, but I would advise you to use short files that you create yourself for initial testing. Create a file that contains only a few words with a few repeats, such as: `one two one three`, or something like that. Once you get your code working with a short text file, try the long ones that you can download from D2L.

Extra Credit

For 2 points of extra credit, create a function to sort the arrays, or array of structs, and then output the top 5 most frequent words and the top 5 least frequent words. You must create a function to do the sorting, and you may not use any sorting algorithm found in a library. In other words, create a custom sorting algorithm. Any 5 of the least frequent words will be fine (meaning there are probably more than 5 words with a frequency of 1). Selection sort is one of the easier sorting algorithms to understand, and is described in the 6th edition book in chapter 8 on page 530 (you will have to search your textbook for selection sort for the other revisions, but revision 7 should be close to the same page).

For an additional 1 point of extra credit, create a function to remove punctuation from the file before you count the words. You can do this by writing another intermediate file, or by deleting all punctuation in memory. There is a function in <cctype> called ispunct() that will help with this.

So there are a total of 3 extra credit points available with this assignment.

Submit your assignment by uploading your source code file as usual. Place your name and sources line at the top of the source code file using comments. Follow the style guide for indentation, naming conventions, empty lines, etc.