[[1]](#footnote-2)

# *Word Frequency Counter* (Dec 2014)

Justin Trotter

*Abstract*—This document records the steps taken in programming a word frequency counter in C using multiple threads for parallel computing. It includes a hashmap as a data structure and radix sorting for alphabetizing. It can handle plain text files as input of nearly any size and account for delimiting common special characters.

*Index Terms*—Alpha Hash, Complexity, Hash Map, FNV PRIME Hash, Insertion Sort, Linked List, Multi-threading, Radix Sort, Special Characters, Tokenizing, UTF-8 Encoding

# INTRODUCTION

THIS document covers my process in designing and compiling program that counts the number of occurrences of words that appear in a plain text file. The program reads from the text file into a buffer, records the input onto several hashmaps (each per thread), sorts the data from the hashmaps out onto various linked lists and merges all the data together to be printing on an output file in alphabetical order.

# Procedure and design

## Design

After careful consideration, I decided to use a hashmap as a data structure to keep track of each word and the number of times it appears in my input textful. This proved to be useful as I could quickly look up how many times a word was used beforehand in constant time. In addition it allowed me to prevent any repeat of words in my list.

The next step in my design was to convert the hashmap to an array of strings. This was as simple as reading through the hashmap array and each of the linked listed attached to the hashmap array and copying them over into a single linear list.

Once everything is merged into a single list, the next plan of action was to sort the list alphabetically.

At this point, this is where I became a bit too ambitious. I wanted my program to compile extremely quickly and efficiency. This is why I tried to use a radix sort for ordering the list alphabetically. This proved to be very challenging, especially in pure C. This took a huge drag on my time, but I finally succeeded in getting it working.

Since the radix sorting would take too long to sort all by itself, I cut off the radix sorting after the first 4 characters of each string. Once semi-sorted, I used the insertion sort algorithm to finish the job quickly.

At this point all was left to do was output the list and data from the hashmap to a file. Then once everything was set, to multithread it.

## Hashmap

The hashmap algorithm is fairly simple. I simply reach each line in the file and delimit by whitespace and most special characters and a few important escape characters. I would take the token pulled from the file and using an alphaHash functi*on* that I designed, converted it into a unique integer b*ased of each of its chara*cter*s. Fo*r example “apple” would be converted into the integer 115151305, where as 'a' = 1, 'p' = 15, 'p' = 15, 'l' = 13, and 'e'= 5. Then I would hash the concatenated integer using the FNV PRIME hashing function to receive an extremely large number which would be modded by the size of the hashmap array. The result of this operation would at last give me the key for my hashmap to find where in the hashmap array to store my values for both the recorded sting and frequency count. The hashmap array structure consisted up of the following: a string, an integer, and a pointer to another hashmap structure. Thus the whole hashmap array structure would be an array of pointers to structures.

## ArrayList

Once the hashmap array was set up, interating through the hashmap was simple enough. I just needed to create a separate structure to hold only the strings and pointers. Then cycle though the hashmap array while making sure to extend though each pointer in each array, while recording the strings encountered onto the arraylist.

## Radix Sort

As mentioned before the radix sort was one of the more challenging parts of this project. The radix sort function I designed relied mostly on a sort helper function. First the radix sort would initial call the sort function on the 4th character in each string. The sort function would look at each strings' forth character and sort them alphabetically accordingly. Once done it would return the list back to the radix sort, which would call it again for the 3rd, 2nd, and finally the 1st characters of each string. This would semi-sort each string sufficiently.

## Insertion Sort

Although the radix sort did a fair job of sorting my list of strings, it would not do a complete job efficiency. I determined it would be more optimal to insertion sort the already semi-sorted list.

## Printing the Sorted List with Values

Once everything was sorted, all I needed to to the print the results was to iterate through the sorted list and check each string that would be printed. Using the string as a key for my hashmap, it was simple and very quick to look their respective values. Just like how I first created the hashmap, I simply converted the string into the integer form that I had mentioned earlier and hashed it using the FNV PRIME hashing function and modding it by the number of hashmap arrays.

## Multi-threading

This is where things began to fall apart. Everything works smoothly in a simple single thread. Adding multiple threads just seems to complicate things and lower efficiency. However, after much spent time I was able to come up with a decent enough scheme for threading my program. First I had to adjust nearly all of my structures including both my hashmap data structure and linked list data structure, to be complied in an array for. Doing this in C was very difficult, but I was finally able to accommodate it using pointers of pointers.

Once the data structures were changed, I had to reconsider how I was reading the file. Part of the project requirements was to not read in the entire fail all at once since there would be a possibility of running out of memory if the file was too large. To manage this I devised a scheme to only read every 1000 lines of the file then build the hashmap from it. Once the hashmap was built from the 1000 lines it would continue to the next 1000 lines and repeat until the entire file was read. As a result no more than 1000 lines of the file was ever read into memory at once.

At this point is where I created the each of the threads. Each thread would have it own personal hashmap assigned to it for writing. This allowed me to not ever have to worry about race conditions as all I needed to do was have each thread build on each of their maps, wait until each thread was done, then repeat for the next 1000 lines until the file was completely read. Not once was there a risk of race conditioning since each thread would only write to their own hashmap.

Once done, I would have a number of hashmaps equal to the number of my threads. All that was left was to merge each hashmap together. This proved to be far too difficult. After an entire day of work, I finally gave up on merging hashmaps together in C and decided the slightly easier route of merging linked lists together.

To make this process as efficient as possible I make sure to have each list sorted before merging them. Once each list was sorted I would merge two at a time. To illustrate this I will refer an example case. First I would have List A and List B passed into my merging function as arguments. I would have created an empty list called List C that would be returned as the merged list. Then I would iterate though each list at same time looking at the first strings at the front of the lists and comparing them alphabetically. First I would check if both strings were the same, if so then I would just copy List A's string down to List C. In addition, I would look up List A's and B's string's frequency number though using their hash keys, add the two values together and record them in List C's structure. If List A's string was less than B's, I would copy A over, increment List A's and loop back to the beginning and vice versa for if List B was less than A. Once one of the Lists ran empty I would attach the remaining strings in one of the lists to the end of List C. This would result in an already sorted List C that also included each of the word frequency, which I would then return to the main function and output.

## Complexity

Although I have not had the time to completely test for efficiency, I have determined the general complexity of my program. I would best guess that the hashmap's complexity would be around O(n), since I'm only going through the file one time to build the hashmap. This is of course assuming I using a large enough hashmap array.

The complexity of the radix sort is where this program shines. Unless I am mistaken, it runs at basically O(n) as well! If you had to be exact, in my program it runs at O(4n), but since 4 is a constant, it is just considered to be O(n). Merging the lists together was not expected when designed this program and may be the bottleneck. I have not yet be able to determine its complexity; however, it does not appear to have that much of an impact, if at all, on performance.

## Compliing and Running

To complie this program, first you need to upzip its package. It is compressed in a simple zip file, so most linux operating systems should be able to extract it by default.

Next step is to navigate to the directory that the program is listed in through the terminal. To complie the program please enter the following command:

gcc frequency.c -lm -w -g -lpthread

This will create an a.out binary file. To run this binary file please enter the following command:

./a.out

Before doing so, make sure that you have a file named input.txt file located in the same directory. I will not run problem with out them. Once you start the program you may see various output printed on the screen. You can observe the program reading the file 1000 lines at a time, then tokenizing them into the hashmap before repeating. Once it if finished, it would output the merged and sorted list for you. Because the list is so hard, it may be difficult to read the beginning part of the list.

Because of this difficulty, I have had the program also print the merged and sorted list to an output file named output.txt. You can view it once the program finishes to see your result. As of right now the program should be able read most text file of nearly any size. However, while the program takes into account most special characters to delimit from, it does not know how to handle very unusual special characters such as obscure utf8-encoding and such. Including these in the program's input text file may result in a segmentation fault.

Lastly, although, this set up in this currently project. With a few tweaks to the source could, it is easily possible to read multiple text files at the same time and receive their word frequency count. This was done earlier as a way from preparing for merging lists with multiple threads. I first make separate hashmaps for each file that would later be merged together.

## Acknowledgments

At this point I would like to thank my peers for their helpful advice with working with C. This program was made most difficult due to the fact that it was necessary to write it entirely in C. Without a fairly decent background in C no one would be able to pull something like this together, especially with the radix sorting my strings.

1. [↑](#footnote-ref-2)