System Programming: Assignment 3

Program Profiling & Optimization

학 과: 시스템경영공학과

학 번: 2022310853

이 름: 박연우

교 과 목: 시스템프로그램_SWE2001_42

담당 교수: 엄영익 교수님

2022310853 박연우

< Assignment 3: Program Profiling & Optimization >

A Bigram Analyzer is a tool which can be used to count the frequency of two words that occur consecutively in a given sample of text. Bigram analysers come in useful in many different fields including but not limited to language modeling, text prediction, and cryptanalysis. Bigrams, or any other n-grams, may come in more useful compared to a unigram because it holds more contextual information. In this assignment, I will be creating a bigram analyzer and use it as my target program to optimize.

1. Target program Implementation, Testing and Validation

The general structure of the bigram analyzer I have implemented follows the description in page 601 of the textbook. The basic overview of the implementation of the target program is as follows:

1. **Read:** The input text file is read. Each word is separated based on whitespace.
2. **Lower:** Converts all strings to lowercase.
3. **Remove Punctuation:** Removes any punctuation from a given string.
4. **Insert:** Hashes the bigrams into the hashtable, and organizes collisions into a linked list.
5. **Hash to Array:** Gets the bigrams from the hash table and stores them in an array of Nodes.
6. **Sort:** Sorts the bigrams by frequency
7. **Etc. Helper functions:** swap, hash_function, and more. Library functions such as strlen are implemented with wrapper functions.

The function to change all letters to lowercase was the same as the `lower1` function in Figure 5.7 of the textbook. The hash function—sum of all ascii codes % hash bucket size—and organization of the hash buckets as a linked list also followed the outlines written in the textbook; the original hash bucket size was set to 1201. Lastly, the sorting method initially used insertion sort as mentioned in the textbook as well. An additional feature I added that was not mentioned in the textbook was a function to remove all punctuation. This feature was implemented in order to accurately compare bigrams (also, Google's n-gram analyzer removes all punctuation too).

The target program was tested using a text file of all the works of Shakespeare provided by MIT OpenCourseWare[1]. Texts that were not literary work were removed (e.g. license agreements, information, etc.). The total number of words in this text are 899,595 words, which is significantly less compared to the 965,028 word text that the textbook used. Despite the difference in the amount of total text in Shakespeare's work, the overall results turned out to be similar. The bigram analyzer presents a total of **357,945 bigrams**, which is less than the number of bigrams in the textbook (363,039 bigrams). This is due to the fact that the Shakespeare text I have used has significantly less text compared to that of the textbook. Similar to the textbook, the **most common bigram was "i am," occurring 1853 times** (1,892 times in the textbook), and one of Shakespeare's most well-known bigram **"to be" occurs 971 times** (1,020 times in the textbook), being the 9th most common bigram. Given similar bigram values, it appears the bigram analyzer is working as intended, and some differences in frequency is likely due to the difference in the original text file of Shakespeare literature used. From these results, it can be concluded that the target program is working as intended.

[1] Shakespeare, William. "MIT OpenCourseWare | Free Online Course Materials." MIT OCW, ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt. Accessed 24 Nov. 2023.

2. Compiler Settings

The target program was profiled using **gprof,** and the **optimization level was set to `-Og`**. The following three steps were used to profile:

1. **Compiling and linked for profiling (bigram.c is the file name)**
```
gcc -Og -pg bigram.c -o prog
```
2. **Execution of the program as usual**
```
./prog
```
3. **Invoking gprof to analyze the data in gmon.out**
```
gprof prog
```

There are **no inputs required** for the bigram analyzer. In order to change the text file, you merely have to **change the `FILE_NAME` String constant at line 6** of the code (as shown below).

```
#define FILE_NAME "shakespeare.txt"
```

The output will print out the **total number of unique bigrams** found as well as the **top 10 most common bigrams**. The actual output is printed below.

```
total bigrams: 357945
Top 10 bigrams:
#1: i am 1853
#2: of the 1712
#3: my lord 1651
#4: in the 1642
#5: i have 1615
#6: i will 1563
#7: to the 1430
#8: it is 1075
#9: to be 971
#10: that i 926
```

3. Initial Program Profiling

[1] The flat profile and call graph for the initial target program are shown below:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 97.21    136.06   136.06        1   136.06   136.10  insertion_sort
  2.13    139.04     2.98   899594     0.00     0.00  insert
  0.39    139.58     0.54 32199537     0.00     0.00  string_length
  0.36    140.08     0.50                             main
  0.04    140.13     0.05  1799188     0.00     0.00  remove_punctuation4
  0.03    140.17     0.04        1     0.04     0.04  hash_to_array
  0.02    140.20     0.03 150118698     0.00     0.00  string_compare
  0.02    140.23     0.03  1799188     0.00     0.00  lower_case1
  0.01    140.25     0.02                             frame_dummy
  0.00    140.25     0.00  4497970     0.00     0.00  string_copy
  0.00    140.25     0.00   899594     0.00     0.00  hash_function
  0.00    140.25     0.00        1     0.00     3.63  read_file_and_hash
```

```
                                  Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.01% of 140.25 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.50  139.73                 main [1]
              136.06    0.04       1/1           insertion_sort [2]
                0.00    3.63       1/1           read_file_and_hash [3]
-----------------------------------------------
              136.06    0.04       1/1           main [1]
[2]     97.0  136.06    0.04       1         insertion_sort [2]
                0.04    0.00       1/1           hash_to_array [9]
-----------------------------------------------
                0.00    3.63       1/1           main [1]
[3]      2.6    0.00    3.63       1         read_file_and_hash [3]
                2.98    0.22  899594/899594      insert [4]
                0.05    0.19 1799188/1799188     remove_punctuation4 [6]
                0.03    0.16 1799188/1799188     lower_case1 [7]
                0.00    0.00  899594/4497970     string_copy [12]
-----------------------------------------------
                2.98    0.22  899594/899594      read_file_and_hash [3]
[4]      2.3    2.98    0.22  899594         insert [4]
                0.00    0.16  899594/899594      hash_function [8]
                0.03    0.00 1799188/32199537     string_length [5]
                0.03    0.00 150118698/150118698     string_compare [10]
                0.00    0.00 1799188/4497970     string_copy [12]
-----------------------------------------------
                0.03    0.00 1799188/32199537     insert [4]
                0.16    0.00 9370241/32199537     hash_function [8]
                0.16    0.00 9615460/32199537     lower_case1 [7]
                0.19    0.00 11414648/32199537     remove_punctuation4 [6]
[5]      0.4    0.54    0.00 32199537         string_length [5]
-----------------------------------------------
                0.05    0.19 1799188/1799188     read_file_and_hash [3]
```

```
[6]      0.2    0.05    0.19 1799188         remove_punctuation4 [6]
                0.19    0.00 11414648/32199537     string_length [5]
                0.00    0.00 1799188/4497970     string_copy [12]
-----------------------------------------------
                0.03    0.16 1799188/1799188     read_file_and_hash [3]
[7]      0.1    0.03    0.16 1799188         lower_case1 [7]
                0.16    0.00 9615460/32199537     string_length [5]
-----------------------------------------------
                0.00    0.16  899594/899594      insert [4]
[8]      0.1    0.00    0.16  899594         hash_function [8]
                0.16    0.00 9370241/32199537     string_length [5]
-----------------------------------------------
                0.04    0.00       1/1           insertion_sort [2]
[9]      0.0    0.04    0.00       1         hash_to_array [9]
-----------------------------------------------
                0.03    0.00 150118698/150118698     insert [4]
[10]     0.0    0.03    0.00 150118698         string_compare [10]
-----------------------------------------------
                                                 <spontaneous>
[11]     0.0    0.02    0.00                 frame_dummy [11]
-----------------------------------------------
                0.00    0.00  899594/4497970     read_file_and_hash [3]
                0.00    0.00 1799188/4497970     remove_punctuation4 [6]
                0.00    0.00 1799188/4497970     insert [4]
[12]     0.0    0.00    0.00 4497970         string_copy [12]
-----------------------------------------------
```

4. First Profiling Analysis

The **total runtime for the initial profiling was 140.25 seconds**. Even from just taking a glance, it can clearly be observed that **`insertion_sort` is taking the most time to complete**, amounting to 136.06 cumulative seconds and 97.21% of the total running time. The function `insertion_sort` is called only once throughout the entire program by the main function, and only makes one call to another function called `hash_to_array`. The called function takes only around 0.04 seconds to complete and amounts to only 0.03% of the runtime. Hence, **improving `insertion_sort` is our first priority**.

The insertion sort algorithm itself runs on O(n^2) time. Therefore, when the array size increases, its run time begins to increase drastically as well. In order to solve this problem, quick sort can be used instead since it runs in O(n·logn) time on average. **In conclusion, the code will be optimized through replacing `insertion_sort` with `qsort`,** the quick sort library function in C**.**

Optimization Changes:

In order to implement `qsort`, the `hash_to_array` function was moved to the main function instead of being called inside the sorting function. Because library functions cannot be seen in the gprof profiling, a function called `quick_sort` was implemented to hold `qsort`. That `quick_sort` function was called in the main, and a new variable to denote the sorted array of the bigram called `array_size` was made. Lastly in order to pass a compare function as a parameter to `qsort`, a `compare` function was made. Codes related to `insertion_sort` were deleted.

In sum, the following were the changed portions:
Main function:

```
int main(){
   //same...
   int array_size = 0;
   hash_to_array(hashtable, sorted_bigrams, &array_size);

   quick_sort(sorted_bigrams, array_size, sizeof(Node*), compare);
   //same...
}
```

Quick sort function:

```
void quick_sort(Node** sorted_bigrams, int array_size, size_t size,
           int (*compare)(const void *, const void *)){
   qsort(sorted_bigrams, array_size, sizeof(Node*), compare);
}
```

Compare function:

```
//compare function for qsort
int compare (const void * a, const void * b) {
   Node* node_a = *(Node**)a;
   Node* node_b = *(Node**)b;
   return (node_b->count - node_a->count);
}
```

[2] The flat profile and call graph after the `quick_sort` implementation are shown below:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 65.24      2.51     2.51   899594     0.00     0.00  insert
 15.66      3.11     0.60 32199537     0.00     0.00  string_length
 12.27      3.58     0.47                             main
  2.35      3.67     0.09  1799188     0.00     0.00  remove_punctuation
  1.04      3.71     0.04  1799188     0.00     0.00  lower_case
  1.04      3.75     0.04        1     0.04     0.04  hash_to_array
  0.78      3.78     0.03 150118698     0.00     0.00  string_compare
  0.78      3.81     0.03   899594     0.00     0.00  hash_function
  0.52      3.83     0.02                             compare
  0.26      3.84     0.01  4497970     0.00     0.00  string_copy
  0.26      3.85     0.01                             frame_dummy
  0.00      3.85     0.00        1     0.00     0.00  quick_sort
  0.00      3.85     0.00        1     0.00     3.31  read_file_and_hash
```

```
=================================================================================
```

```
                     Call graph (explanation follows)
granularity: each sample hit covers 2 byte(s) for 0.26% of 3.85 seconds
```

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]     99.2    0.47    3.35                 main [1]
                0.00    3.31       1/1           read_file_and_hash [2]
                0.04    0.00       1/1           hash_to_array [8]
                0.00    0.00       1/1           quick_sort [13]
-----------------------------------------------
                0.00    3.31       1/1           main [1]
[2]     85.9    0.00    3.31       1         read_file_and_hash [2]
                2.51    0.27  899594/899594      insert [3]
                0.09    0.22 1799188/1799188     remove_punctuation [5]
                0.04    0.18 1799188/1799188     lower_case [6]
                0.00    0.00  899594/4497970     string_copy [11]
-----------------------------------------------
                2.51    0.27  899594/899594      read_file_and_hash [2]
[3]     72.2    2.51    0.27  899594         insert [3]
                0.03    0.17  899594/899594      hash_function [7]
                0.03    0.00 1799188/32199537     string_length [4]
                0.03    0.00 150118698/150118698     string_compare [9]
                0.00    0.00 1799188/4497970     string_copy [11]
-----------------------------------------------
                0.03    0.00 1799188/32199537     insert [3]
                0.17    0.00 9370241/32199537     hash_function [7]
                0.18    0.00 9615460/32199537     lower_case [6]
                0.21    0.00 11414648/32199537     remove_punctuation [5]
[4]     15.6    0.60    0.00 32199537         string_length [4]
-----------------------------------------------
                0.09    0.22 1799188/1799188     read_file_and_hash [2]
[5]      8.0    0.09    0.22 1799188         remove_punctuation [5]
                0.21    0.00 11414648/32199537     string_length [4]
                0.00    0.00 1799188/4497970     string_copy [11]
-----------------------------------------------
                0.04    0.18 1799188/1799188     read_file_and_hash [2]
[6]      5.7    0.04    0.18 1799188         lower_case [6]
                0.18    0.00 9615460/32199537     string_length [4]
-----------------------------------------------
                0.03    0.17  899594/899594      insert [3]
[7]      5.3    0.03    0.17  899594         hash_function [7]
                0.17    0.00 9370241/32199537     string_length [4]
-----------------------------------------------
                0.04    0.00       1/1           main [1]
[8]      1.0    0.04    0.00       1         hash_to_array [8]
-----------------------------------------------
                0.03    0.00 150118698/150118698     insert [3]
[9]      0.8    0.03    0.00 150118698         string_compare [9]
-----------------------------------------------
                                                 <spontaneous>
[10]     0.5    0.02    0.00                 compare [10]
-----------------------------------------------
                0.00    0.00  899594/4497970     read_file_and_hash [2]
                0.00    0.00 1799188/4497970     remove_punctuation [5]
                0.00    0.00 1799188/4497970     insert [3]
[11]     0.3    0.01    0.00 4497970         string_copy [11]
-----------------------------------------------
                                                 <spontaneous>
[12]     0.3    0.01    0.00                 frame_dummy [12]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[13]     0.0    0.00    0.00       1         quick_sort [13]
-----------------------------------------------
```

The **total runtime decreased drastically to 3.85 seconds**, which is around 36.4 times faster than the run time of the original code. Fortunately, the runtime for sorting has gone down immensely, and `quick_sort` is second to last in the flat profile, no longer serving as a bottleneck.

5. Second Profiling Analysis

From the second profiling (the most recent gprof result), we can observe that the **`insert` function is the new bottleneck**, taking up 65.24% of the total runtime. The `insert` is a function that takes two given words and inserts, or hashes, them into the hashtable. The function first creates a new node for the given bigram, and adds the bigram as a node to the hashtable. If a node already exists at that hashtable, traverses the hashtable to check if that bigram already exists, and if it does not, the bigram is appended to the end of the list.

The only loop that exists in this function is the while loop to traverse the linked list. Hence, the bottleneck is an issue arising from the traversal of the linked list. In order to solve this problem, we have to reduce the length of the linked lists. First, I will try **increasing the `BUCKET_SIZE`** from the initial value of 1021 to **15331** (a much larger prime number) in hopes that it will distribute the bigrams more uniformly in the hashtable.

```
#define BUCKET_SIZE 15331
```

[3] The resulting flat profile and call graph from increasing the `BUCKET_SIZE` are as follows:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 59.96      2.13     2.13   899594     0.00     0.00  insert
 17.17      2.75     0.61 32199537     0.00     0.00  string_length
 13.23      3.22     0.47                             main
  2.53      3.31     0.09 144113729     0.00     0.00  string_compare
  1.97      3.38     0.07  1799188     0.00     0.00  lower_case
  1.97      3.45     0.07  1799188     0.00     0.00  remove_punctuation
  1.41      3.50     0.05                             compare
  1.13      3.54     0.04   899594     0.00     0.00  hash_function
  0.84      3.57     0.03        1     0.03     0.03  hash_to_array
  0.00      3.57     0.00  4497970     0.00     0.00  string_copy
  0.00      3.57     0.00        1     0.00     0.00  quick_sort
  0.00      3.57     0.00        1     0.00     3.02  read_file_and_hash
```

===

```
                    Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.28% of 3.57 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     98.6    0.47    3.05                 main [1]
                0.00    3.02       1/1           read_file_and_hash [2]
                0.03    0.00       1/1           hash_to_array [10]
                0.00    0.00       1/1           quick_sort [12]
-----------------------------------------------
                0.00    3.02       1/1           main [1]
```

```
[2]     84.6    0.00    3.02       1         read_file_and_hash [2]
                2.13    0.34  899594/899594      insert [3]
                0.07    0.22 1799188/1799188     remove_punctuation [5]
                0.07    0.18 1799188/1799188     lower_case [6]
                0.00    0.00  899594/4497970     string_copy [11]
-----------------------------------------------
                2.13    0.34  899594/899594      read_file_and_hash [2]
[3]     69.4    2.13    0.34  899594         insert [3]
                0.04    0.18  899594/899594      hash_function [7]
                0.09    0.00 144113729/144113729     string_compare [8]
                0.03    0.00 1799188/32199537     string_length [4]
                0.00    0.00 1799188/4497970     string_copy [11]
-----------------------------------------------
                0.03    0.00 1799188/32199537     insert [3]
                0.18    0.00 9370241/32199537     hash_function [7]
                0.18    0.00 9615460/32199537     lower_case [6]
                0.22    0.00 11414648/32199537     remove_punctuation [5]
[4]     17.1    0.61    0.00 32199537         string_length [4]
-----------------------------------------------
                0.07    0.22 1799188/1799188     read_file_and_hash [2]
[5]      8.0    0.07    0.22 1799188         remove_punctuation [5]
                0.22    0.00 11414648/32199537     string_length [4]
                0.00    0.00 1799188/4497970     string_copy [11]
-----------------------------------------------
                0.07    0.18 1799188/1799188     read_file_and_hash [2]
[6]      7.1    0.07    0.18 1799188         lower_case [6]
                0.18    0.00 9615460/32199537     string_length [4]
-----------------------------------------------
                0.04    0.18  899594/899594      insert [3]
[7]      6.1    0.04    0.18  899594         hash_function [7]
                0.18    0.00 9370241/32199537     string_length [4]
-----------------------------------------------
                0.09    0.00 144113729/144113729     insert [3]
[8]      2.5    0.09    0.00 144113729         string_compare [8]
-----------------------------------------------
                                                 <spontaneous>
[9]      1.4    0.05    0.00                 compare [9]
-----------------------------------------------
                0.03    0.00       1/1           main [1]
[10]     0.8    0.03    0.00       1         hash_to_array [10]
-----------------------------------------------
                0.00    0.00  899594/4497970     read_file_and_hash [2]
                0.00    0.00 1799188/4497970     remove_punctuation [5]
                0.00    0.00 1799188/4497970     insert [3]
[11]     0.0    0.00    0.00 4497970         string_copy [11]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[12]     0.0    0.00    0.00       1         quick_sort [12]
-----------------------------------------------
```

The runtime decreased to 3.57 seconds, which is 1.08 times faster than the previous runtime. While the runtime changed for the better, the difference is minimal. While a larger hash bucket size has helped distribute the bigrams more uniformly, it didn't make too much of a difference. `insert` is still first on the flat profile amounting to 59.96% of the total runtime, suggesting that the `BUCKET_SIZE` can only do too much in improving the distribution in the hash table. **The `insert` function will try to be optimized again in the next profiling**.

6. Third Profiling Analysis

This time, I will try **changing the hash function to a better hash function** from djb2 algorithm[2] designed by Daniel J. Bernstein. The djb2 hash function uses exclusive or in a simple manner with a straightforward implementation. It also has good distribution properties, and hence I chose this algorithm as my hash function. The modified `hash_function` code is shown below. The return value was changed to an `unsigned int` in case we go beyond the range of integers.

New hash function:

```
unsigned int hash_function(char* word1, char* word2){
   unsigned int hash = 5381;

   while(*word1){
       hash = (hash*33) ^ *word1++;
   }
   while(*word2){
       hash = (hash*33) ^ *word2++;
   }

   return hash % BUCKET_SIZE;
}
```

[4] The flat profile and call graph while using a better hash function is as shown below:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
 40.92      0.49     0.49                             main
 33.40      0.89     0.40 22829296     0.00     0.00  string_length
 10.02      1.01     0.12   899594     0.00     0.00  insert
  5.43      1.08     0.07  1799188     0.00     0.00  lower_case
  5.43      1.14     0.07  1799188     0.00     0.00  remove_punctuation
  2.51      1.17     0.03                             compare
  1.67      1.19     0.02        1    20.04    20.04  hash_to_array
  0.84      1.20     0.01   899594     0.00     0.00  hash_function
  0.00      1.20     0.00  7097206     0.00     0.00  string_compare
  0.00      1.20     0.00  4497970     0.00     0.00  string_copy
  0.00      1.20     0.00        1     0.00     0.00  quick_sort
  0.00      1.20     0.00        1     0.00   661.41  read_file_and_hash
```

```
===============================================================================

                   Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.83% of 1.20 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     97.5    0.49    0.68                 main [1]
                0.00    0.66       1/1           read_file_and_hash [2]
                0.02    0.00       1/1           hash_to_array [8]
                0.00    0.00       1/1           quick_sort [12]
```

[2] "Hash Functions." Homepage, www.cse.yorku.ca/~oz/hash.html. Accessed 24 Nov. 2023.

```
-----------------------------------------------
                0.00    0.66       1/1           main [1]
[2]     55.0    0.00    0.66       1         read_file_and_hash [2]
                0.07    0.20 1799188/1799188     remove_punctuation [4]
                0.07    0.17 1799188/1799188     lower_case [5]
                0.12    0.04  899594/899594      insert [6]
                0.00    0.00  899594/4497970     string_copy [11]
-----------------------------------------------
                0.03    0.00 1799188/22829296     insert [6]
                0.17    0.00 9615460/22829296     lower_case [5]
                0.20    0.00 11414648/22829296     remove_punctuation [4]
[3]     33.3    0.40    0.00 22829296         string_length [3]
-----------------------------------------------
                0.07    0.20 1799188/1799188     read_file_and_hash [2]
[4]     22.1    0.07    0.20 1799188         remove_punctuation [4]
                0.20    0.00 11414648/22829296     string_length [3]
                0.00    0.00 1799188/4497970     string_copy [11]
-----------------------------------------------
                0.07    0.17 1799188/1799188     read_file_and_hash [2]
[5]     19.5    0.07    0.17 1799188         lower_case [5]
                0.17    0.00 9615460/22829296     string_length [3]
-----------------------------------------------
                0.12    0.04  899594/899594      read_file_and_hash [2]
[6]     13.5    0.12    0.04  899594         insert [6]
                0.03    0.00 1799188/22829296     string_length [3]
                0.01    0.00  899594/899594      hash_function [9]
                0.00    0.00 7097206/7097206     string_compare [10]
                0.00    0.00 1799188/4497970     string_copy [11]
-----------------------------------------------
                                                 <spontaneous>
[7]      2.5    0.03    0.00                 compare [7]
-----------------------------------------------
                0.02    0.00       1/1           main [1]
[8]      1.7    0.02    0.00       1         hash_to_array [8]
-----------------------------------------------
                0.01    0.00  899594/899594      insert [6]
[9]      0.8    0.01    0.00  899594         hash_function [9]
-----------------------------------------------
                0.00    0.00 7097206/7097206     insert [6]
[10]     0.0    0.00    0.00 7097206         string_compare [10]
-----------------------------------------------
                0.00    0.00  899594/4497970     read_file_and_hash [2]
                0.00    0.00 1799188/4497970     remove_punctuation [4]
                0.00    0.00 1799188/4497970     insert [6]
[11]     0.0    0.00    0.00 4497970         string_copy [11]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[12]     0.0    0.00    0.00       1         quick_sort [12]
-----------------------------------------------
```

By implementing a new hash function, the run time was 2.98 times faster, having a **total runtime of 1.20 seconds**. The `insert` function was improved, by moving down to 3rd in both the flat profile and 10th on the call graph. The insert function still amounts to 10.02% of the entire time, which is not little, but a definite improvement since the beginning of the second profiling analysis. Since the `insert` function only takes up a small portion, I will try optimizing other areas of the code and return to back to optimizing the `insert` function if necessary.

7. Fourth Profiling Analysis

In the profiling analysis [4], it appears that the `main` function is contributing most to the total runtime. However, since the `main` function is making many different calls to other functions, it is hard to pinpoint the exact portion of code that needs to be optimized. Therefore, **I will instead optimize the fourth and fifth functions on the flat profile**, which are the **`remove_punctuation` and `lower_case` functions**.

From examining the code, it can be observed that both functions call `string_length`—a wrapper function for `strlen`—in the condition portion of the loop, while iterating over each word. The excessive calls to `string_length` can be shown in the profiling analysis. `string_length` takes 33.40% of the entire runtime, taking a total of 0.40 self-seconds. From the third section of the call graph, it can be seen that the `string_length` function is called 22829296 times by functions `remove_punctuation` and `lower_case`.

In order to reduce the number of calls to `string_length`, we can **apply code motion to both `remove_punctuation` and `lower_case`**: create a variable for the string length and reduce the number of calls of `string_length` to once in each function.

The following are the modified code for `remove_punctuation` and `lower_case` functions:
Function to remove punctuation:

```
void remove_punctuation(char* word){
   char no_punct[strlen(word)+1];
   int index = 0;
   int i = 0;
   int len = strlen(word);

   for(i = 0; i < len; i++){
       if(!ispunct(word[i])){
           no_punct[index] = word[i];
           index++;
       }
   }

   no_punct[index] = '\0';
   strcpy(word, no_punct);
}
```

Function to convert to lowercase:

```
void lower_case(char* s){
   int len = strlen(s);
   for(long i = 0; i < len; i++){
       if(s[i] >= 'A' && s[i] <= 'Z'){
           s[i] -= ('A' - 'a');
       }
   }
}
```

[5] The flat profile and call graph after applying code motion to `remove_punctuation` and `lower_case` is as shown below:

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 53.45      0.48     0.48                             main
 14.48      0.61     0.13   899594     0.00     0.00  insert
 13.36      0.73     0.12  7196752     0.00     0.00  string_length
  5.01      0.78     0.05  1799188     0.00     0.00  remove_punctuation
  4.45      0.82     0.04        1    40.09    40.09  hash_to_array
  3.90      0.85     0.04  1799188     0.00     0.00  lower_case
  3.34      0.88     0.03                             compare
  1.11      0.89     0.01  4497970     0.00     0.00  string_copy
  1.11      0.90     0.01   899594     0.00     0.00  hash_function
  0.00      0.90     0.00  7097206     0.00     0.00  string_compare
  0.00      0.90     0.00        1     0.00     0.00  quick_sort
  0.00      0.90     0.00        1     0.00   350.75  read_file_and_hash
```

```
                   Call graph (explanation follows)
granularity: each sample hit covers 2 byte(s) for 1.11% of 0.90 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     96.7    0.48    0.39                 main [1]
                0.00    0.35       1/1           read_file_and_hash [2]
                0.04    0.00       1/1           hash_to_array [7]
                0.00    0.00       1/1           quick_sort [12]
-----------------------------------------------
                0.00    0.35       1/1           main [1]
[2]     38.9    0.00    0.35       1         read_file_and_hash [2]
                0.13    0.04  899594/899594      insert [3]
                0.05    0.06 1799188/1799188     remove_punctuation [5]
                0.04    0.03 1799188/1799188     lower_case [6]
                0.00    0.00  899594/4497970     string_copy [9]
-----------------------------------------------
                0.13    0.04  899594/899594      read_file_and_hash [2]
[3]     19.3    0.13    0.04  899594         insert [3]
                0.03    0.00 1799188/7196752     string_length [4]
                0.01    0.00  899594/899594      hash_function [10]
                0.00    0.00 1799188/4497970     string_copy [9]
                0.00    0.00 7097206/7097206     string_compare [11]
-----------------------------------------------
                0.03    0.00 1799188/7196752     lower_case [6]
                0.03    0.00 1799188/7196752     insert [3]
                0.06    0.00 3598376/7196752     remove_punctuation [5]
[4]     13.3    0.12    0.00 7196752         string_length [4]
-----------------------------------------------
                0.05    0.06 1799188/1799188     read_file_and_hash [2]
[5]     12.1    0.05    0.06 1799188         remove_punctuation [5]
                0.06    0.00 3598376/7196752     string_length [4]
                0.00    0.00 1799188/4497970     string_copy [9]
-----------------------------------------------
                0.04    0.03 1799188/1799188     read_file_and_hash [2]
[6]      7.2    0.04    0.03 1799188         lower_case [6]
                0.03    0.00 1799188/7196752     string_length [4]
-----------------------------------------------
```

```
                0.04    0.00       1/1           main [1]
[7]      4.4    0.04    0.00       1         hash_to_array [7]
-----------------------------------------------
                                                 <spontaneous>
[8]      3.3    0.03    0.00                 compare [8]
-----------------------------------------------
                0.00    0.00  899594/4497970     read_file_and_hash [2]
                0.00    0.00 1799188/4497970     remove_punctuation [5]
                0.00    0.00 1799188/4497970     insert [3]
[9]      1.1    0.01    0.00 4497970         string_copy [9]
-----------------------------------------------
                0.01    0.00  899594/899594      insert [3]
[10]     1.1    0.01    0.00  899594         hash_function [10]
-----------------------------------------------
                0.00    0.00 7097206/7097206     insert [3]
[11]     0.0    0.00    0.00 7097206         string_compare [11]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[12]     0.0    0.00    0.00       1         quick_sort [12]
-----------------------------------------------
```

The **total runtime decreased to 0.90 seconds**, which is 1.33 faster than the third profiling. The total self-seconds of `string_length` went from 0.40 seconds to 0.12 seconds, showing great improvement in performance. It also went from being second on the flat profile to third. From the call graph we can see that the total number of calls to `string_length` went from 22829296 to 7196752, which is three times less calls. Furthermore, by optimizing the number of calls to `string_length`, the self seconds of `remove_punctuation` and `lower_case` improved from 0.07 seconds to 0.05 seconds and 0.04 seconds respectively.

8. Fifth Profiling Analysis

Despite the increased performance, I speculated **both `remove_punctuation` and `lower_case` function could be further improved by having `string_length` completely removed.**

For both functions the loop was **changed into pointer indexing instead**. This method allows us to completely remove the call to `string_length` since we no longer need to compute the length for the condition of the for-loop. In addition, for the `remove_punctuation` function, `string_copy`—a wrapper function for `strcpy`—is no longer needed to copy the string with removed punctuation into the original string. Furthermore, array indexing using a pointer is generally faster compared to using an array index operator[3]. This is due to the fact that indexing requires additional calculations compared to pointer indexing.

The following are the modified code for `remove_punctuation` and `lower_case` functions:
Function to remove punctuation:

```
void remove_punctuation3(char* word){
   char* no_punct = word;
```

[3] “Array Indexing vs Pointer Arithmetic Performance.” Stack Overflow, 1 Feb. 1962, stackoverflow.com/questions/34256067/array-indexing-vs-pointer-arithmetic-performance.

```
    for(; *word != '\0'; word++){
        if(!ispunct(*word)){
            *no_punct++ = *word;
        }
    }
    *no_punct = '\0';
}
```

Function to convert to lowercase:

```
void lower_case3(char* s){
  for(; *s != '\0'; s++){
       if(*s >= 'A' && *s <= 'Z'){
           *s -= ('A' - 'a');
       }
    }
}
```

[6] The flat profile and call graph after changing to pointer indexing are shown below:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 58.47      0.49     0.49                             main
 19.09      0.65     0.16   899594     0.00     0.00  insert
  4.77      0.69     0.04  1799188     0.00     0.00  string_length
  4.77      0.73     0.04        1    40.09    40.09  hash_to_array
  3.58      0.76     0.03   899594     0.00     0.00  hash_function
  3.58      0.79     0.03                             compare
  1.79      0.81     0.02  1799188     0.00     0.00  lower_case
  1.19      0.82     0.01  7097206     0.00     0.00  string_compare
  1.19      0.83     0.01  1799188     0.00     0.00  remove_punctuation
  1.19      0.84     0.01                             frame_dummy
  0.60      0.84     0.01  2698782     0.00     0.00  string_copy
  0.00      0.84     0.00        1     0.00     0.00  quick_sort
  0.00      0.84     0.00        1     0.00   270.62  read_file_and_hash
```

```
=================================================================================
```

```
                    Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 1.19% of 0.84 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     95.2    0.49    0.31                 main [1]
                0.00    0.27       1/1           read_file_and_hash [2]
                0.04    0.00       1/1           hash_to_array [5]
                0.00    0.00       1/1           quick_sort [13]
-----------------------------------------------
                0.00    0.27       1/1           main [1]
[2]     32.1    0.00    0.27       1         read_file_and_hash [2]
                0.16    0.08  899594/899594      insert [3]
                0.02    0.00 1799188/1799188     lower_case [8]
                0.01    0.00 1799188/1799188     remove_punctuation [10]
                0.00    0.00  899594/2698782     string_copy [12]
-----------------------------------------------
                0.16    0.08  899594/899594      read_file_and_hash [2]
[3]     29.0    0.16    0.08  899594         insert [3]
                0.04    0.00 1799188/1799188     string_length [4]
                0.03    0.00  899594/899594      hash_function [6]
                0.01    0.00 7097206/7097206     string_compare [9]
                0.00    0.00 1799188/2698782     string_copy [12]
-----------------------------------------------
                0.04    0.00 1799188/1799188     insert [3]
[4]      4.8    0.04    0.00 1799188         string_length [4]
-----------------------------------------------
                0.04    0.00       1/1           main [1]
[5]      4.8    0.04    0.00       1         hash_to_array [5]
-----------------------------------------------
                0.03    0.00  899594/899594      insert [3]
[6]      3.6    0.03    0.00  899594         hash_function [6]
-----------------------------------------------
                                                 <spontaneous>
[7]      3.6    0.03    0.00                 compare [7]
-----------------------------------------------
                0.02    0.00 1799188/1799188     read_file_and_hash [2]
[8]      1.8    0.02    0.00 1799188         lower_case [8]
-----------------------------------------------
                0.01    0.00 7097206/7097206     insert [3]
[9]      1.2    0.01    0.00 7097206         string_compare [9]
-----------------------------------------------
                0.01    0.00 1799188/1799188     read_file_and_hash [2]
[10]     1.2    0.01    0.00 1799188         remove_punctuation [10]
-----------------------------------------------
                                                 <spontaneous>
[11]     1.2    0.01    0.00                 frame_dummy [11]
-----------------------------------------------
                0.00    0.00  899594/2698782     read_file_and_hash [2]
                0.00    0.00 1799188/2698782     insert [3]
[12]     0.6    0.01    0.00 2698782         string_copy [12]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[13]     0.0    0.00    0.00       1         quick_sort [13]
-----------------------------------------------
```

The **overall runtime decreased to 0.84 seconds**, which is a 0.06 second difference from the previous profiling's runtime. Although the overall running time had not much of a difference, the **optimized functions' self-seconds showed a noticeable difference**. The `remove_punctuation` function especially showed a lot of improvement: while `remove_punctuation` had a self-second of 0.05 seconds, it decreased to 0.01 seconds. `lower_case` had a reduced self-second time of 0.02 seconds, which is a 0.02 second improvement compared to profiling analysis [5]. Furthermore, according to the call graph, the number of calls to `string_length` decreased to 1799188 and is only called by the `insert` function.

9. Sixth Profiling Analysis

Although the convert to lowercase and remove punctuation functions have shown great improvement compared to the initial version, I would like to make one last optimization attempt.

Both functions can be potentially improved **by removing the if-statement inside the for-loop**. Since if-statements make branch prediction more unpredictable, it is better to **change the coding style from imperative to functional** by adding a ternary operator (as mentioned in chapter 5 of the textbook). With a ternary operator, there is a better chance that the code will be compiled into a `cmov` instruction instead of a combination of `cmp` and `jmp` instructions.

The modified code is as shown below:
Function to remove punctuation:

```
void remove_punctuation4(char* word){
   char* no_punct = word;

   for(; *word != '\0'; word++){
       *no_punct = (!ispunct(*word)) ? *word : *no_punct;
       no_punct += (!ispunct(*word)) ? 1 : 0;
   }

   *no_punct = '\0';
}
```

Function to convert to lowercase:

```
void lower_case4(char* s){
  for(; *s != '\0'; s++){
       *s = (*s >= 'A' && *s <= 'Z') ? *s - ('A' - 'a') : *s;
   }
}
```

[7] The flat profile and call graph after changing to functional style are shown below:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
 61.87      0.50     0.50                             main
 18.56      0.65     0.15   899594     0.00     0.00  insert
  8.66      0.72     0.07  1799188     0.00     0.00  string_length
  3.71      0.75     0.03        1    30.07    30.07  hash_to_array
  3.71      0.78     0.03                             compare
  2.47      0.80     0.02  1799188     0.00     0.00  remove_punctuation
  1.24      0.81     0.01        1    10.02   250.56  read_file_and_hash
  0.00      0.81     0.00  7097206     0.00     0.00  string_compare
  0.00      0.81     0.00  2698782     0.00     0.00  string_copy
  0.00      0.81     0.00  1799188     0.00     0.00  lower_case
  0.00      0.81     0.00   899594     0.00     0.00  hash_function
  0.00      0.81     0.00        1     0.00     0.00  quick_sort
```

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 1.23% of 0.81 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     96.3    0.50    0.28                 main [1]
                0.01    0.24       1/1           read_file_and_hash [2]
                0.03    0.00       1/1           hash_to_array [5]
                0.00    0.00       1/1           quick_sort [12]
-----------------------------------------------
                0.01    0.24       1/1           main [1]
[2]     30.9    0.01    0.24       1         read_file_and_hash [2]
                0.15    0.07  899594/899594      insert [3]
                0.02    0.00 1799188/1799188     remove_punctuation [7]
                0.00    0.00 1799188/1799188     lower_case [10]
                0.00    0.00  899594/2698782     string_copy [9]
-----------------------------------------------
                0.15    0.07  899594/899594      read_file_and_hash [2]
[3]     27.2    0.15    0.07  899594         insert [3]
                0.07    0.00 1799188/1799188     string_length [4]
                0.00    0.00 7097206/7097206     string_compare [8]
                0.00    0.00 1799188/2698782     string_copy [9]
                0.00    0.00  899594/899594      hash_function [11]
-----------------------------------------------
                0.07    0.00 1799188/1799188     insert [3]
[4]      8.6    0.07    0.00 1799188         string_length [4]
-----------------------------------------------
                0.03    0.00       1/1           main [1]
[5]      3.7    0.03    0.00       1         hash_to_array [5]
-----------------------------------------------
                                                 <spontaneous>
[6]      3.7    0.03    0.00                 compare [6]
-----------------------------------------------
                0.02    0.00 1799188/1799188     read_file_and_hash [2]
[7]      2.5    0.02    0.00 1799188         remove_punctuation [7]
-----------------------------------------------
                0.00    0.00 7097206/7097206     insert [3]
[8]      0.0    0.00    0.00 7097206         string_compare [8]
-----------------------------------------------
                0.00    0.00  899594/2698782     read_file_and_hash [2]
                0.00    0.00 1799188/2698782     insert [3]
[9]      0.0    0.00    0.00 2698782         string_copy [9]
-----------------------------------------------
                0.00    0.00 1799188/1799188     read_file_and_hash [2]
[10]     0.0    0.00    0.00 1799188         lower_case [10]
-----------------------------------------------
                0.00    0.00  899594/899594      insert [3]
[11]     0.0    0.00    0.00  899594         hash_function [11]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[12]     0.0    0.00    0.00       1         quick_sort [12]
-----------------------------------------------
```

The **runtime decreased to 0.81 seconds**, which is a small difference to the previous runtime of 0.84 seconds. If we look at the self-seconds of `lower_case` were reduced to 0.00 seconds and the self-seconds of remove_punctuation increased to 0.02 seconds. However, since the runtime for each of these functions were very small in the first place, an increase or decrease of a few centiseconds cannot ensure a significant optimization. However, based on the contents of chapter 5 in the textbook, functional coding style should be more effective. Thus, I will be keeping the modifications of the two functions.

10. Seventh Profiling Analysis

The **`main` function is taking a large portion of the runtime, totaling to 61.87% of the runtime**. If we examine the main function, it has **several loops assigning `NULL` to the `hashtable` and `sorted_array`**. This is a process which **can be removed**.

Furthermore, the `sorted_array` does not need to be created to a size of `MAX_BIGRAMS 100000000`. Instead, I will introduce a **new variable, `num_words`, to count the number of words** (it will be passed to the `read_file_and_hash` function). With the number of words obtained, the `sorted_array` will now be initialized to the size of words, reducing any potential of unnecessary memory allocations (max number of bigrams possible will be number of words-1).

Below is the modified code:
`main` function:

```
int main(){
   //initialize hash table, an array of pointers to nodes
   Node** hashtable = (Node**)malloc(sizeof(Node*) * BUCKET_SIZE);

   int num_words = 0;
   read_file_and_hash(hashtable, &num_words);

   Node** sorted_bigrams = (Node**)malloc(sizeof(Node*) * num_words);

   int array_size = 0;
   hash_to_array(hashtable, sorted_bigrams, &array_size);

   quick_sort(sorted_bigrams, array_size, sizeof(Node*), compare);

   printf("Total bigrams: %d\n", array_size);
   printf("Top 10 bigrams: \n");
   for(int i = 0; i < 10; i++){
       printf("#%d: %s %s %d\n", i+1, sorted_bigrams[i]->word1, sorted_bigrams[i]->word2,
sorted_bigrams[i]->count);
   }
   return 0;
}
```

`read_file_and_hash` modified in order to count the number of words:

```
void read_file_and_hash(Node** hashtable, int* num_words){
   // same...
   int word_count = 0;

   //scan first word
   if(fscanf(input_file, "%99s", first_w) == 0){
       printf("Error: File is empty\n");
       exit(1);
   }

   while(fscanf(input_file, "%99s", second_w) == 1){
       lower_case(first_w);
       lower_case(second_w);
       remove_punctuation(first_w);
       remove_punctuation(second_w);

       insert(hashtable, first_w, second_w);

       string_copy(first_w, second_w); //change the first word to the second word
       word_count++;
   }
   *num_words = word_count;
}
```

[8] The flat profile and call graph after changing the main function are shown below:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 51.40      0.20     0.20   899594     0.00     0.00  insert
 12.85      0.25     0.05                             compare
  7.71      0.28     0.03  1799188     0.00     0.00  lower_case
  7.71      0.31     0.03        1    30.07    30.07  hash_to_array
  5.14      0.33     0.02  1799188     0.00     0.00  string_length
  5.14      0.35     0.02   899594     0.00     0.00  hash_function
  5.14      0.37     0.02        1    20.05   310.71  read_file_and_hash
  2.57      0.38     0.01  7097206     0.00     0.00  string_compare
  2.57      0.39     0.01  1799188     0.00     0.00  remove_punctuation
  0.00      0.39     0.00  2698782     0.00     0.00  string_copy
  0.00      0.39     0.00        1     0.00     0.00  quick_sort
```

```
                   Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 2.56% of 0.39 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     87.2    0.00    0.34                 main [1]
                0.02    0.29       1/1           read_file_and_hash [2]
                0.03    0.00       1/1           hash_to_array [6]
                0.00    0.00       1/1           quick_sort [12]
-----------------------------------------------
                0.02    0.29       1/1           main [1]
[2]     79.5    0.02    0.29       1         read_file_and_hash [2]
                0.20    0.05  899594/899594      insert [3]
                0.03    0.00 1799188/1799188     lower_case [5]
                0.01    0.00 1799188/1799188     remove_punctuation [10]
                0.00    0.00  899594/2698782     string_copy [11]
```

```
-----------------------------------------------
                0.20    0.05  899594/899594      read_file_and_hash [2]
[3]     64.1    0.20    0.05  899594         insert [3]
                0.02    0.00 1799188/1799188     string_length [7]
                0.02    0.00  899594/899594      hash_function [8]
                0.01    0.00 7097206/7097206     string_compare [9]
                0.00    0.00 1799188/2698782     string_copy [11]
-----------------------------------------------
                                                 <spontaneous>
[4]     12.8    0.05    0.00                 compare [4]
-----------------------------------------------
                0.03    0.00 1799188/1799188     read_file_and_hash [2]
[5]      7.7    0.03    0.00 1799188         lower_case [5]
-----------------------------------------------
                0.03    0.00       1/1           main [1]
[6]      7.7    0.03    0.00       1         hash_to_array [6]
-----------------------------------------------
                0.02    0.00 1799188/1799188     insert [3]
[7]      5.1    0.02    0.00 1799188         string_length [7]
-----------------------------------------------
                0.02    0.00  899594/899594      insert [3]
[8]      5.1    0.02    0.00  899594         hash_function [8]
-----------------------------------------------
                0.01    0.00 7097206/7097206     insert [3]
[9]      2.6    0.01    0.00 7097206         string_compare [9]
-----------------------------------------------
                0.01    0.00 1799188/1799188     read_file_and_hash [2]
[10]     2.6    0.01    0.00 1799188         remove_punctuation [10]
-----------------------------------------------
                0.00    0.00  899594/2698782     read_file_and_hash [2]
                0.00    0.00 1799188/2698782     insert [3]
[11]     0.0    0.00    0.00 2698782         string_copy [11]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[12]     0.0    0.00    0.00       1         quick_sort [12]
-----------------------------------------------
```

The run time decreased to **0.39 seconds**, which is 2.08 times faster than the previous profiling analysis run time. The main function was well optimized as it cannot be even seen in the flat profile.

11. Eighth Profiling Analysis

Now, the insert function is again taking up the largest portion of the runtime, totaling to 51.40% of the total run time and taking 0.20 seconds to complete. Hence, I will try to **optimize the `insert` function** once again. In order to optimize the `insert` function, I **created a new while-loop to check if the bigram already exists before creating a new node**. This way, the program can avoid creating an unnecessary node.

Below is the modified code for the `insert` function:

```
void insert(Node** hashtable, char* first_w, char* second_w){
   unsigned int hash_value = hash_function(first_w, second_w);
   Node* temp = hashtable[hash_value];
```

```
    //check if the node exists
    while(temp != NULL){
        // if the bigram already exists, increment the count
        if(strcmp(temp->word1, first_w) == 0 && strcmp(temp->word2, second_w) == 0){
            temp->count++;
            return;
        }
        temp = temp->next;
    }

    //if it doesn't exist create a new node
    Node* new_node = (Node*)malloc(sizeof(Node));

    //same…
}
```

[9] The flat profile and call graph after optimizing the `insert` function are shown below:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 51.96      0.14     0.14   899594     0.00     0.00  insert
 18.56      0.19     0.05  1799188     0.00     0.00  remove_punctuation
 11.14      0.22     0.03                             compare
  7.42      0.24     0.02        1    20.04    20.04  hash_to_array
  3.71      0.25     0.01  1799188     0.00     0.00  lower_case
  3.71      0.26     0.01   899594     0.00     0.00  hash_function
  3.71      0.27     0.01   715890     0.00     0.00  string_length
  0.00      0.27     0.00  4183846     0.00     0.00  string_compare
  0.00      0.27     0.00  1615484     0.00     0.00  string_copy
  0.00      0.27     0.00        1     0.00     0.00  quick_sort
  0.00      0.27     0.00        1     0.00   220.48  read_file_and_hash
```

```
                    Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 3.70% of 0.27 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     88.9    0.00    0.24                 main [1]
                0.00    0.22       1/1           read_file_and_hash [2]
                0.02    0.00       1/1           hash_to_array [6]
                0.00    0.00       1/1           quick_sort [12]
-----------------------------------------------
                0.00    0.22       1/1           main [1]
[2]     81.5    0.00    0.22       1         read_file_and_hash [2]
                0.14    0.02  899594/899594      insert [3]
                0.05    0.00 1799188/1799188     remove_punctuation [4]
                0.01    0.00 1799188/1799188     lower_case [7]
                0.00    0.00  899594/1615484     string_copy [11]
-----------------------------------------------
                0.14    0.02  899594/899594      read_file_and_hash [2]
[3]     59.3    0.14    0.02  899594         insert [3]
                0.01    0.00  899594/899594      hash_function [8]
                0.01    0.00  715890/715890      string_length [9]
                0.00    0.00 4183846/4183846     string_compare [10]
                0.00    0.00  715890/1615484     string_copy [11]
-----------------------------------------------
                0.05    0.00 1799188/1799188     read_file_and_hash [2]
[4]     18.5    0.05    0.00 1799188         remove_punctuation [4]
-----------------------------------------------
                                                 <spontaneous>
[5]     11.1    0.03    0.00                 compare [5]
-----------------------------------------------
                0.02    0.00       1/1           main [1]
[6]      7.4    0.02    0.00       1         hash_to_array [6]
-----------------------------------------------
                0.01    0.00 1799188/1799188     read_file_and_hash [2]
[7]      3.7    0.01    0.00 1799188         lower_case [7]
-----------------------------------------------
                0.01    0.00  899594/899594      insert [3]
[8]      3.7    0.01    0.00  899594         hash_function [8]
-----------------------------------------------
                0.01    0.00  715890/715890      insert [3]
[9]      3.7    0.01    0.00  715890         string_length [9]
-----------------------------------------------
                0.00    0.00 4183846/4183846     insert [3]
[10]     0.0    0.00    0.00 4183846         string_compare [10]
-----------------------------------------------
                0.00    0.00  715890/1615484     insert [3]
                0.00    0.00  899594/1615484     read_file_and_hash [2]
[11]     0.0    0.00    0.00 1615484         string_copy [11]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[12]     0.0    0.00    0.00       1         quick_sort [12]
-----------------------------------------------
```

The **total runtime was 0.27 seconds**, which is around 1.44 times faster than the previous profiling analysis's runtime. The insert function was optimized so that it only takes 51.96% of the total runtime. Since we check if the bigram exists before appending it to the hashtable, the number of `string_copy` functions being called has decreased to 1615484 from 2698782.

12. Ninth Profiling Analysis

To make **another attempt at optimizing the `insert` function,** I tried changing the way nodes are appended. In the current code, the nodes are appended at the end of the linked list. The modified code will **instead append the node to the front of the linked list**.

Here is the modified code for `insert`:

```c
void insert(Node** hashtable, char* first_w, char* second_w){
   //same...
   if(hashtable[hash_value] == NULL){
       hashtable[hash_value] = new_node;
   }
   else{
       new_node->next = hashtable[hash_value];
       hashtable[hash_value] = new_node;
   }
}
```

[10] The flat profile and call graph after optimizing the `insert` function are shown below:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
 63.55      0.26     0.26   899594     0.00     0.00  insert
  9.78      0.30     0.04        1    40.09    40.09  hash_to_array
  7.33      0.33     0.03  1799188     0.00     0.00  remove_punctuation
  7.33      0.36     0.03                             compare
  4.89      0.38     0.02  1799188     0.00     0.00  lower_case
  4.89      0.40     0.02   715890     0.00     0.00  string_length
  2.44      0.41     0.01        1    10.02   340.74  read_file_and_hash
  0.00      0.41     0.00  4183846     0.00     0.00  string_compare
  0.00      0.41     0.00  1615484     0.00     0.00  string_copy
  0.00      0.41     0.00   899594     0.00     0.00  hash_function
  0.00      0.41     0.00        1     0.00     0.00  quick_sort
```

```
                    Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 2.43% of 0.41 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     92.7    0.00    0.38                 main [1]
                0.01    0.33       1/1           read_file_and_hash [2]
                0.04    0.00       1/1           hash_to_array [4]
                0.00    0.00       1/1           quick_sort [12]
-----------------------------------------------
                0.01    0.33       1/1           main [1]
[2]     82.9    0.01    0.33       1         read_file_and_hash [2]
                0.26    0.02  899594/899594      insert [3]
                0.03    0.00 1799188/1799188     remove_punctuation [5]
                0.02    0.00 1799188/1799188     lower_case [7]
                0.00    0.00  899594/1615484     string_copy [10]
-----------------------------------------------
                0.26    0.02  899594/899594      read_file_and_hash [2]
[3]     68.3    0.26    0.02  899594         insert [3]
                0.02    0.00  715890/715890      string_length [8]
                0.00    0.00 4183846/4183846     string_compare [9]
                0.00    0.00  899594/899594      hash_function [11]
                0.00    0.00  715890/1615484     string_copy [10]
-----------------------------------------------
                0.04    0.00       1/1           main [1]
[4]      9.8    0.04    0.00       1         hash_to_array [4]
-----------------------------------------------
                0.03    0.00 1799188/1799188     read_file_and_hash [2]
[5]      7.3    0.03    0.00 1799188         remove_punctuation [5]
-----------------------------------------------
                                                 <spontaneous>
[6]      7.3    0.03    0.00                 compare [6]
-----------------------------------------------
                0.02    0.00 1799188/1799188     read_file_and_hash [2]
[7]      4.9    0.02    0.00 1799188         lower_case [7]
-----------------------------------------------
                0.02    0.00  715890/715890      insert [3]
[8]      4.9    0.02    0.00  715890         string_length [8]
```

```
-----------------------------------------------
                0.00    0.00 4183846/4183846     insert [3]
[9]      0.0    0.00    0.00 4183846         string_compare [9]
-----------------------------------------------
                0.00    0.00  715890/1615484     insert [3]
                0.00    0.00  899594/1615484     read_file_and_hash [2]
[10]     0.0    0.00    0.00 1615484         string_copy [10]
-----------------------------------------------
                0.00    0.00  899594/899594      insert [3]
[11]     0.0    0.00    0.00  899594         hash_function [11]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[12]     0.0    0.00    0.00       1         quick_sort [12]
-----------------------------------------------
```

The **runtime increased to 0.41 seconds** when the node was added to the front of the linked list. The total self-seconds for the insert function increased to 0.26 seconds, taking 63.55% of the entire runtime. Hence, I will **revert to adding the node to the end of the linked list**. Although the `insert` function still takes a significant amount of the runtime, it would appear to be difficult to further optimize the traversal portion of the code, since there is a limit on how much we can decrease the time in traversing the linked list. Instead, a way to reduce the number of calls that insert makes to `string_length`, `string_copy`, and `string_compare` should be inspected.

13. Tenth Profiling Analysis

In this portion, I will be discussing several small changes I made in an attempt to further optimize my code. While I inspected `hash_to_array` and `read_file_and_hash` functions, they were pretty efficiently written. There may have been room for improvement, but I was not able to find any areas that could be improved. I also looked into the `hash_function` to see if loop unrolling can be applied. However, since the string values and hash values are both changing, the loop could not be unrolled in a way that several calculations could be done in parallel.

I also tried to **optimize the `compare` function** by computing the compare results in a single return line instead of assigning local variables. Since no local variables need to be made, I hoped it would improve the runtime. Below is the modified version of the `compare` function:

```
int compare (const void * a, const void * b) {
   return (*(Node**)b)->count - (*(Node**)a)->count;
}
```

The order of `remove_punctuation` and `lower_case` was also changed in the `read_file_and_hash_function`. If **punctuation is removed first**, then there are less letters to check for lowercase. Hence, the order was switched in such manner:

```
void read_file_and_hash(Node** hashtable, int* num_of_words){
   //same...
   while{
      remove_punctuation4(first_w);
      remove_punctuation4(second_w);
      lower_case3(first_w);
```

```
        lower_case3(second_w);
        //same...
    }
}
```

Lastly, in order to avoid having to constantly `malloc` to add the strings to the node structure, I changed the arrays which store the strings of the bigram from **dynamic arrays to static arrays**. Since the `MAX_WORD_SIZE` is set to 100, we do not have to worry about not having enough memory to allocate the array. Hopefully, this change can reduce the runtime for `insert` as well as the number of calls it makes to `string_length`. The modified structure is as follows:

```
typedef struct Node{
   char word1[MAX_WORD_SIZE];
   char word2[MAX_WORD_SIZE];
   int count;
   struct Node* next;
} Node;
```

[11] The flat profile and call graph after applying all the changes mentioned above are below:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
 33.41      0.06     0.06   899594     0.00     0.00  insert
 22.27      0.10     0.04                             compare
 16.70      0.13     0.03  1799188     0.00     0.00  remove_punctuation
 16.70      0.16     0.03        1    30.07    30.07  hash_to_array
  5.57      0.17     0.01  1799188     0.00     0.00  lower_case
  5.57      0.18     0.01   899594     0.00     0.00  hash_function
  0.00      0.18     0.00  4183846     0.00     0.00  string_compare
  0.00      0.18     0.00  1615484     0.00     0.00  string_copy
  0.00      0.18     0.00        1     0.00     0.00  quick_sort
  0.00      0.18     0.00        1     0.00   110.24  read_file_and_hash
```

```
                   Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 5.54% of 0.18 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     77.8    0.00    0.14                 main [1]
                0.00    0.11       1/1           read_file_and_hash [2]
                0.03    0.00       1/1           hash_to_array [6]
                0.00    0.00       1/1           quick_sort [11]
-----------------------------------------------
                0.00    0.11       1/1           main [1]
[2]     61.1    0.00    0.11       1         read_file_and_hash [2]
                0.06    0.01  899594/899594      insert [3]
                0.03    0.00 1799188/1799188     remove_punctuation [5]
                0.01    0.00 1799188/1799188     lower_case [7]
                0.00    0.00  899594/1615484     string_copy [10]
-----------------------------------------------
```

```
                0.06    0.01  899594/899594      read_file_and_hash [2]
[3]     38.9    0.06    0.01  899594         insert [3]
                0.01    0.00  899594/899594      hash_function [8]
                0.00    0.00 4183846/4183846     string_compare [9]
                0.00    0.00  715890/1615484     string_copy [10]
-----------------------------------------------
                                                 <spontaneous>
[4]     22.2    0.04    0.00                 compare [4]
-----------------------------------------------
                0.03    0.00 1799188/1799188     read_file_and_hash [2]
[5]     16.7    0.03    0.00 1799188         remove_punctuation [5]
-----------------------------------------------
                0.03    0.00       1/1           main [1]
[6]     16.7    0.03    0.00       1         hash_to_array [6]
-----------------------------------------------
                0.01    0.00 1799188/1799188     read_file_and_hash [2]
[7]      5.6    0.01    0.00 1799188         lower_case [7]
-----------------------------------------------
                0.01    0.00  899594/899594      insert [3]
[8]      5.6    0.01    0.00  899594         hash_function [8]
-----------------------------------------------
                0.00    0.00 4183846/4183846     insert [3]
[9]      0.0    0.00    0.00 4183846         string_compare [9]
-----------------------------------------------
                0.00    0.00  715890/1615484     insert [3]
                0.00    0.00  899594/1615484     read_file_and_hash [2]
[10]     0.0    0.00    0.00 1615484         string_copy [10]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[11]     0.0    0.00    0.00       1         quick_sort [11]
-----------------------------------------------
```

The final runtime of the optimized code is 0.18 seconds. All calls to `string_length` have been removed. I will be ending the optimization at this stage.

14. Conclusion

The objective of this assignment was to optimize a target program. A Bigram Analyzer was used as the target program, and a collection of Shakespare's work was used as the testing file. The program was tested using a `gcc` compiler with optimization settings set to `-Og`.

The **initial runtime of the program was 140.25 seconds**, and the **final runtime of the optimization was 0.18 seconds**. The optimized version of the program was **779.2 times faster** than its initial runtime. It is important to note that the runtime may differ each time the program is run. The lowest I have recorded was 0.16 seconds. However, in the worst case scenario, the runtime is around 0.29 seconds. On average, it seems that the runtime is around 0.18 to 0.24 seconds.

When the wrapper functions were removed and the library functions were in place, the runtime decreased even more due to reduced overhead. The lowest runtime I have recorded was 0.09 seconds (although this appears to be a unique case). The average is a little lower with most runtimes being constantly under 0.20 seconds. The removal of wrapper functions is not reflected in my final code since it is a fairly simple procedure to change the wrapper functions to the library functions.

A **summary of the optimizations** I have made:

- Initial runtime: 140.25 seconds. (Run time after each change is written in parenthesis).

1. Changed sorting method from insertion sort to quick sorting (3.85)
2. Increased `BUCKET_SIZE` to 15331 from 1201. (3.57)
3. Changed the hash function to djb2's algorithm. (1.20)
4. Applied code motion to lowercase and remove punctuation from the strings. (0.90)
5. Changed the `lower_case` and `remove_punctuation` for-loops from array indexing to pointer indexing. (0.84)
6. Changed the `lower_case` and `remove_punctuation` coding style from imperative to conditional. (0.81)
7. Removed `hashtable` and `sorted_array`'s loops to initialize its nodes to `NULL`. (0.39)
8. Added a while loop to `insert` to check if the bigram exists before creating a new node. (0.27)
9. Tried adding a new bigram to the beginning of the linked list instead of the end. (Did not improve; worsened). (0.41)
10. Removed local variables from `compare`, changed the order of `lower_case` and `remove_punctuation` in `read_file_and_hash`, and changed the `Node` structure to hold strings of the bigram as static array instead of dynamic arrays. (0.18)

In conclusion, the significant performance improvement achieved, with a final runtime of 0.18 seconds compared to the initial 140.25 seconds, demonstrates the effectiveness of the applied optimizations. I made the decision to conclude the optimization efforts after realizing that the top three functions consumed nearly equal portions of the runtime, and extensive analysis across various areas of the code revealed no further opportunities for optimization. While further optimization might be possible, the implemented changes have maximized the program's efficiency, addressing bottlenecks and resulting in a highly optimized Bigram Analyzer.

15. Work Cited

[1] Shakespeare, William. “MIT OpenCourseWare | Free Online Course Materials.” MIT OCW, ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt. Accessed 24 Nov. 2023.

[2] “Hash Functions.” Homepage, www.cse.yorku.ca/~oz/hash.html. Accessed 24 Nov. 2023.

[3] “Array Indexing vs Pointer Arithmetic Performance.” Stack Overflow, 1 Feb. 1962, stackoverflow.com/questions/34256067/array-indexing-vs-pointer-arithmetic-performanc e.