//Yujian Li(yl7kd) 03/18/16 section 102

In this lab, the big theta runtime is r*c*w*n+w^2. Where r is the number of rows in the grid file, c is the number of columns in the grid file, w is the number of words and n is a constant. At the very beginning, it asked to take w words from the dictionary files and hash them into the hashtable. The average run-time for extracting all words in the dictionary file is linear hence w and the average run-time for inserting all together will also be linear since we are not sure the number of collision may occurred. Therefore the total average run-time for reading the file should be w^2 The main body of the word search method is consisted of four nested for loops and each loop's run-time is linear because of the nature of hash table. Due to the structure of the grid file, the run-time is at first mainly influenced by the number of columns and rows which is r*c because the code needs to go through every part of the grid file to search. Then it is also influenced by the total number of words that inside the grid file and also inside the dictionary file w. In the end, the search run-time is also influenced by the direction and the maximum word size. However, since both are considered as a small constant, we assume the product of these two will also be a constant n. Therefore, the total run-time over all is r*c*w*n+w^2.

In the application process, I used words2.txt file for my dictionary file and 300x300.grid.txt file for my grid file. I ran the program on my own laptop which is 64 bit thinkpad laptop and I compiled the program using clang++ and the -O2 for the optimization. . At the initial run, my average run-time result was 1924 milliseconds. Later I choose a different hash function (version 2) which takes product of the second index of the string input and 33 divided by the table size as shown below:

```
int val = 0;
        val=inpu[1]*33;
        return val % this->siz;
```

this gives me a much longer average running time which is 5063 milliseconds. Therefore I wrote a code to check the collision occurred during the code. The collision number is 1529 comparing to 1003 in the original function. Because of the increasing number of collisions, the code had to take longer time to hash all of the elements. On the other hand, when I take the size of the hashtable and reduce by 2, the

run-time becomes 1971 milliseconds and the collisions times also increases from 1003 to 1070. By reducing the amount of size in hashtable it actually increases the chance of collision which slows down the hashtable's speed.

Since my code's initial run-time is 1924 milliseconds at the beginning (below 2-3 seconds), it can be considered as efficient. Prior to the optimization process, I have already kept a value “len” for the maximum size of the word in the hashtable. After all the words in the dictionary file was hashed into the table, I can then use the value len as a constant in the search for loop which helps to decrease the amount of time it takes to search the word. However, there are still many spaces that can be improved. When searching the word, we put the cout statement inside the nested loop to print out all the results. However, this is actually not a part of the performance process and creating a buffer to store all the value and return it after the timer stops will save a lot of time. Among different data structures, the unique structure of stack makes me to choose to be the buffer since it has a O(1) run-time when insert a element at the top of the stack. After the timer has stopped, the stack will return the top value and pop it until the stack is empty. This helps to decreases the run-time from initially 1924 ms to 1850 ms. Later, I created two different version of hashing function to optimize the run-time. As mentioned before, a good hashtable can help to avoid collisions when inserting and save time when finding since the chained list will be shorter with less collisions.

For the first hash function, I took the add every value of the string input and times the 2nd power of corresponding index and then divided by the table size :

```
int val = 0;
for(int i =0; i<inpu.length();i++){
        val+=inpu[i]*pow(i,2)+90;
        }
        return val % this->siz;
```

Obtaining the result of the code, we get a collisions of 317 and an average run-time of 1.719. It seems the “ax+b” method does help to make the values evenly distributed in the hashtable and avoid many unnecessary collisions. However, it still seems not to be a very significant change for optimization.

Later I used the bitshiffing method shown below:

```
int val = 5381;
for(int i=0;i<inpu.length();i++){
        val+=(inpu[i]<<5)+val;
}
return val % this->siz;
```

The collisions number becomes 269 and the average run-time is 1625 ms. Comparing to the previous method, this hashing function fits better with the given inputs and the code. In the end, incrementing the size of the hashtable is also a mean of optimization. As the table size becomes larger(within certain range), it can help to avoid collisions and make the run-time faster. After a few trails of experimenting, I believe that when I multiply the size of table by the factor of 11, it will generally a run-time of 1574 ms and a collision of 226. Comparing to my final run-time y to my initial run-tim x, I can see the optimization process makes the program 1924/1574 about 1.22 times faster.

The greatest problem I encountered in this process is to find the best hashing function. Hashing function is the core of the hash-table. Finding a good hash function enables us to run the code faster. At first I assumed that creating a more complex function will help to decrease the run-time. After a few failing attempt, I realized that a good hash function is the find the balance between the complexity of computing while tried to generate a more unique value for each input keys. In the end, I was able to use certain combinations of operators to generate a more efficient hash function. Aside from the existing optimizations, I believed that implementing a linear probing or even a quadratic probing method will help to make the search method more efficient since it does not requires to search through each elements in the linked-list when collision. For the data structure of separating chaining, I chose the linked-list structure because comparing with other structures like array or vectors, linked-list is more memory efficient.