Yubin Lim yl2vv
March 7, 2019
CS 2150-103

The big-theta running time of my application comes out to be θ(**r c** (**lambda w**)).
The (**r** x **c)** comes from the quad-nested for loop used in my application. The first for loop iterates through the possible x values of the starting letter, the rows. The second for loop is the possible y value of the starting letter, the columns. The third for loop is the direction of the word, which is limited to be eight directions, a constant. The final for loop is an iteration through the length of the string. In our case, the max length of word was set to be 22, a constant. The latter two are just constants, so what we really care about is the rows x columns.
The (**lambda w**) comes from the average number of elements in a bucket of the given number of words. Inside the for loops, there is a find function. The find function essentially checks if the word is in the hash table. My hash table was a vector of lists, so the collision resolution that I used was separate chaining. The best case for find would be a constant 1, where the number of entries occupied in the hash table equals the number of buckets and there is no collision. The worst case would be linear, where every key is in the same spot. With my hash function, I landed with the average case, where there is an average number of elements in each bucket, and so each iteration through a list is in between the worst case of searching every element and searching once.

The machine of my choice was my personal laptop, a 2014 MacBook Pro. For my timing of the words2.txt file as my dictionary and 300x300.grid.txt file as my input grid, I was able to get a run time average of 1456 milliseconds, or 1.456 seconds. This is a vast improvement from my original application, which had a time of 6626 milliseconds, or 6.626 seconds.
A new hash function I chose to make the performance worse was just 1 mod size. Every time the function mods, because it mods a constant value, it would always be 1. This would put every value inside the same bucket, making searching longer. The average runtime was 74.491 seconds.
A new hash table size to make performance worse was just the number of elements. This performance was worse because I did not make it a prime number, and I did not get to double the size. Therefore, more collision occurred. The average run time was 1.542 seconds.

A couple problems that I ran into was my laptop being really slow. My laptop is pretty old and has very little memory left, so sometimes, the code would run slower than normal. That was why it was important for me to try and find the best average time that is not affected by an outlier. Another problem was my hash function. I did not how to make a good hash function, so I just used the one provided to us in the lecture slide. This was a decent hash function, but left plenty of room for improvement and optimization. I will discuss in detail below.

There were a few optimizations from my prelab that I was able to maintain. The first was the -O2 optimization. This was maintained from when I first learned to make a Makefile, it was included in CXX, so I maintained it in there. Turns out –O2 is very important in making the code run faster and it was a good habit to do. The second thing I was able to maintain was the size of the hash table. I first took the number of elements expected and multiplied that by two. I then turn that number into the next biggest prime number. Prime numbers are very helpful in reducing

collisions, especially in probing, which I did not use. But the bigger size still helped me decrease collision and spread out my values.
A big change I made in my application was changing the hash function. The original hash function was from the lecture slides. It was for each character in the string in position i, multiply string[i] by i to the power of 37. I changed my hash function by replacing the value multiplied at the end to just three instead of i to the power of 37. I think changing the hash function was really effective in many ways. The main reason was that I was able to decrease the number of collisions resulted and spread out the values. This was able to decrease my time running the words2.txt and 300x300.grid.txt by about five seconds. This put my running time in the ideal range of 1-3 seconds.

The original running time (un-optimized) of my application was 6626 milliseconds, and I was able to speed it up with my optimizations to get a final running time of 1456 milliseconds. The speed up therefore, is 6626/1456, approximately 4.55. This shows how important it is to find the best optimizations in coding.