Noah French (njf5cu)
Lab 6
8/20/17
Filename: postlab6.pdf
File Description: Post-lab Report for Lab 6

Big-theta runtime/explanation:

The big-theta running time of my application is n^2. My application performs the actual word search in a quad-nested for loop. The first loop iterates through the number of rows in the grid. The second loop iterates through the number of columns in the grid. The third loop iterates over the the eight possible directions (north, northeast, east, southeast, south, southwest, west, and northwest). The final for loop iterates over the possible lengths of words. Because the smallest words that we're looking for are three letters long and because the largest possible word is 22 letters long, this loop goes from 3 to 22. Thus, two of these for loops iterate a constant number of times when either of the three variables (number of rows, number of columns, or number of words) are changed. These for loops are the one for direction (0 to 7) and the one for length (3 to 22). The two for loops dependent on rows and columns each increase their number of iterations linearly with number of rows and number of columns respectively. Thus, when nested, their combined big-theta running time is n^2. These loops make calls to the find function, which gets slower the greater the number of words there are. But this rate of growth is nowhere near n^2. Therefore, the overall big-theta running time is n^2.

Timing Tests:

All my timing tests were performed with the words2.txt file as the dictionary and the 140x70.grid.txt file as the grid. They were performed in the VirtualBox on my personal Windows laptop. The results were recorded with timer.h/.cpp using the timer placement as we used in lab. My original implementation submitted in the prelab (with the addition of the -O2 flag) completed the search of the 140x70 grid with the words2 dictionary in 2.112 seconds. This implementation used the

example hash function from the slide set, wherein the ASCII value of each character is multiplied by 37 to the power of that character's position in the string. These values are all added together and modded by the table size get a hash value. I replaced this hash function with a clearly inferior solution: simply adding the ASCII values of each of the letters and then modding the result by the table size. This hash function presumably results in many more collisions, which would make the find operation slower. The data supports this; the runtime was 10.599 seconds, over five times slower than the runtime with my original hash function

My prelab solution used the next highest prime number after twice the dictionary size as the hash table size. This ensured a load factor of around .5, which – though not perfect for conserving memory – is fairly fast. I changed the table size to be the next highest prime number after the dictionary size. Thus, the load factor would be almost 1, and there would be many collisions. The slow runtime of 11.802 seconds supports this hypothesis. Because there are more collisions when creating the hash table, more linear probing is necessary when the program checks to see if the string of characters from the grid is present in the hash table.

Optimizations and resultant runtimes:

All of the runtime tests in this section were performed with words2.txt as the dictionary file and 300x300.grid.txt as the grid file. My original solution from the prelab (with the addition of the -O2 flag) solved the crossword puzzle in 119.301 seconds. This is obviously quite slow, but I thought it was reasonable enough to submit for the prelab. My original solution used linear probing, a load factor of about .5, and the hash function from the slides, as described in the previous section.

My first optimization proved incredibly helpful. The head TA who guest lectured on hash tables mentioned to us the the pow() function takes a lot of time to compute. My hash table implementation called pow() to calculate a hash value every time a potential word was inputed to my find method. My solution called pow() thousands of times for a single run! I added an array called exponent as a private

member variable in my hash table class. Then, in my constructor implementation, I filled the array with powers of 37 that I would be using in my hash function. In my hash function, I just picked from the array rather than computing the power of 37 again. After this change, my program ran in 2.926 seconds. The speedup was about 40.8. This technically fulfills the maximum time requirement, but it could still be better.

After that, I decided to play with the load factor. In class, Floryan mentioned that Java's hash table maintains a load factor of about .75. I decided to raise my load factor from around .5 to .75 to see how that would affect the runtime. I changed my calculateTableSize() method to return the next highest prime number after 4/3 times the dictionary size. Funnily enough, the runtime after this change was 4.101 seconds. It got slower! This makes sense, because increasing the load factor (for the same number of inputs) decreases the table size. A smaller table size increases collisions on finds, slowing the program down. Then, I decided to decrease my load factor. What if the size of the hash table were many times larger than the number of items? I kept increasing the dictionary size and monitoring the runtime. I found that there's a point of diminishing returns once the table size is about 20 times the number of elements. But using that much memory is obnoxious, so I settled for making the hash table size the next prime number after three times the dictionary size. That makes the load factor about .33, not crazy low, and the runtime drops to 2.595 seconds. The speedup is about 1.13.

Next, I decided to buffer the input like the postlab instructions suggest. This a far more frustrating optimization to implement than I had expected, because, evidently, there is no simple way to convert an int to a string in vanilla C++. I ended up using the most common method, which involves streams. Unfortunately, I'm fairly certain that streams are pretty slow, so this optimization could probably be implemented more efficiently. I created a vector and used the push_back() method to add the strings to the vector instead of immediately outputting them to the console. I probably could have instantiated the vector with a certain size and used a loop to input the strings, but that would have

required me restructuring most of wordPuzzle.cpp, so I just stuck with this simpler (possibly slower) strategy. With this change, the runtime is 1.826 seconds, a speedup of 1.42.

With these optimizations, I had an overall speedup of 65.33. It was originally able to complete the 300x300 word search (with words2 dictionary) in 119.301 seconds. After optimization it runs on my machine in a mere 1.826 seconds.