Maria Hito
3/15/12
Post-lab 6

The Big-Theta running complexity of my program can be calculated as follows:

> The scanning of the dictionary file and the insertion of the words into the hashTable are a linear function of W (words). The reading and mapping of the grid file is a linear function of R and C (rows and columns). The brute-force search through every combination of grid position (R * C) and word length (small constant) can be considered a complexity of R*C*8 (the number of directions), which reduces to R*C.
>
> This makes the final run-time complexity of my program W + RC + RC = W + 2RC = Theta (W + RC).

In terms of optimization of performance, I made three improvements in the program: delayed all console outputs, optimized the hash function, and increased the number of buckets.

The most important improvement was to minimize iterations on the nested for-loops. Functionally my algorithm seemed to stop finding certain edge-cases because of an improper middle condition in one of my for-loops. This lead to improper wordCounts, for the bigger text files, upon program termination and not finding some of the words. I was erroneously checking that the length was smaller than the rows-currRow, but that was missing large words going in directions other than East. I decided to store the maximum word size of the hashTable's values as a public data member, and iterated my length from 3 to maxWordSize in order to reduce extra getWordInTable lookups. This ended up making my algorithm run faster and reducing my time for words2.txt/300x300.grid.txt files from 51 seconds to 2.36. This was the optimization that was keeping all of my searches so slow all along, as I was doing 1000's of extra calls to hashTable::contains(). This fixed the output for all of the grids to match the given sample output.

In a smaller scale, I decided to use a stringstream rather than piping to cout and printed the resulting string only after all of the searching was done and the timer stopped. Reducing the amount of cout statements in my code increased average speed to about 10% for processing a word puzzle. I did not know how computationally expensive console logging could be, but it makes sense, as altering pixels on the display is much more involved than bit manipulation at the memory level.

The hash function was improved by reducing repetition that was added due to factoring. I initially had a hash function which made a generic hash and then a myHash function to scale it down to the correct size to fit into the hashTable. I replaced this by merging the two and was able to save a few repeated calculations.

In the constructor of my hashTable, before the primeness of the size argument is checked, I inserted a statement that would multiply it by two by using a left bitwise shift. This would lead to more buckets which reduces collisions and helps to verify that each bucket has a smaller average size.

Condition (words.txt // 50x50.grid.txt)	**Time (s)**
Original application	0.099
Slow hash (hash = str[1] * 78)	3.17
Bad size (no check for prime, not large enough)	0.11

For the slow hash condition, the hash function is terrible because it leads to a lot of collision. First of all, it only checks one character of the string, so many strings can get similar results. Also, it doesn't even multiply by a prime constant, so there are multiple factors that can lead to the same hashValue product, leading to yet more collisions.

For the bad size, I simply removed my bit shift and also the check for primeness. Without a prime number of buckets, the modulus function is repeated often, for example all of the factors of a number evaluate to 0. This causes the increment of collisions. The removal of the bit shift just leads to way fewer buckets, which increases the need for separate chaining and the load factor.

In the end, my output for words2.txt/300x300.grid.txt ended up being 2.36398 seconds, which is a significant increase from the 51 seconds it took my code to complete prior to my optimizations. This is a speed increase by a factor of 52/2.36 ~22x! This shows a great amount of algorithmic simplification.