1 Running Time Analysis

In the following analysis the variables r, c, and w will be used to represent the number of rows in the given word puzzle, the number of columns in the puzzle, and the number of words in the wordlist. We disregard constants when calculating big theta running time and so we can ignore any single statements that don't take place in a loop. The first loop occurs when reading in the puzzle and involves iterating over each letter and performing a single assignment into the two dimensional array representing the puzzle. The complexity of this operation is thus $\theta(r \cdot c)$.

Next I read in the wordlist and insert each line into the hashtable. I also insert all prefixes of a given string, so this operation requires two loops. The first is over the number of words in the file, w, and the second is over the substrings of any given word. The length of words is assumed to be some small constant and so this second loop can be ignored. The running time for this operation is thus $\theta(w)$.

The final component of the program is to check every vector in the matrix to see if it's a valid word in the dictionary. This invloves four loops. The first two iterate over r and c in order to access every location in the puzzle as a possible starting letter for a word. The third loop checks each of 8 possible directions, and the fourth loop checks all possible lengths in the given direction. The last two loops are over constants and can be ignored. Within the loops a word is retrieved from the puzzle, a check is made to see if the word is in the hashtable and another check is made to see if it's a prefix. If the word is in the table, it is added into the output vector. Each of these operations requires constant time. The total running time for this step is then $\theta(r \cdot c)$.

Combining the time required for each of the steps we get a total running time of:

$$\theta(r \cdot c + w + r \cdot c) = \theta(2(r \cdot c) + w) \tag{1}$$

We neglect the 2 and thus this is equivalent to:

$$\theta(r \cdot c + w) \tag{2}$$

2 Timing Information

In the following table, Input 1 is the combination of the wordlist 'words2.txt' and the puzzle '300x300.grid.txt.' Input 2 is the combination of the wordlist 'words.txt' and the puzzle '140x70.grid.txt.' Times are given in seconds. The new hash function that I chose to test simply took the length of the key modulo the table size. This is guaranteed to produce many collisions. The smaller wordlist has over 1500 words, each with a length between 3 and 22. The situation is even worse for the larger wordlist which contains over 25,000 words. The table size I chose to test was $(1/2 * w)$ where w is the number of words in the wordlist. If the hash function distributed the keys as efficiently as possible, there would still be at least $w/2$ collisions. Increasing collisions decreases the efficiency of the table.

	Input 1	Input 2
Base	0.175	0.032
Worse Hash Function	0.289	0.052
Worse Table Size	0.213	0.038

3 Optimizations

The following table shows each of the optimizations that I tried and its individual impact on the execution time. The optimizations are explained and justified below the table. All times are for the wordlist 'words2.txt' on the puzzle '300x300.grid.txt.'

Optimization	**Average of Five (seconds)**	**Speedup**
None	9.97086	n/a
Buffered Output	9.82309	1.01504
Replace List with Set	2.22782	4.25802
Store Prefixes in Table	27.9898	0.356231
Triple Table Size	9.78648	1.01884
Remove Check Vector	10.1160	0.98565
Change Hash Function	2.22782	4.47561

1. Buffered Input
   My initial program printed out each element as soon as it found it. I attempted to increase speed by instead appending the word to a vector and then at the end of the program looping through the vector and printing out each word. This did increase the speed but only very minimally.

2. Replace List with Set
   My initial hashtable was an array of pointers to Linked Lists that I had implemented. I changed it to be an array of pointers to sets from the standard library. Sets are typically implemented as binary search trees and thus have a faster lookup time. Additionally, being part of the standard library, they are likely to be implemented more efficiently than my own data attempts. This had a significantimpact on the running time, second only to improving the hash function.

3. Store Prefixes in Table
   As suggested in the lab, I thought that storing the prefixes of all words in the table would improve speed by allowing the program to stop checking in a given directino as soon as it encounters a word that isn't in the dictionary and isn't a prefix of a word in the dictionary. Surprisingly,

instead of increasing speed, this resulted in a significant slowdown. However, removing it from the final code increases the run time from 0.2-0.3 seconds to 2-3 seconds. This leads me to think that either it benefits from being implemented alongside one of the other optimizations, or that I implemented it incorrectly when testing it on its own.

4. Triple Table Size
   Rather than initializing the hashtable with a size of getNextPrime(# of words), I initialized the table with a size of getNextPrime(3 * # of words). By increasing the size of the table I hoped to reduce collisions and improve the average time that it took to retrieve an element. This change had only a minimal positive impact which comes at the cost of increased memory usage.

5. Remove Check Vector
   The way getWordInTable() works is that if you give it a length larger than fits in the table it truncates it to the maximum possible length. Looping over a set range of lengths means that I often ended up checking the same word multiple times. In order not to add the same word to the output string multiple times I would append it to a vector and then every time I found a word that was in the hashtable I would check that it wasn't in this vector. In order to optimize it I refactored the code so that it would never check the same thing multiple times. This also avoided needing to do lookups in the vector, which I assumed was significant source of slowness. Surprisingly, the effect on speed was very minimal and in the opposite direction of what I expected. The only explanation I can come up with for why this might have happened was that the checks I introduced to make sure I wasn't using the same word multiple times might have been slower than the lookups they replaced.

6. Change Hash Function
   I took my hash function, which was simply taking the length of the key, and replaced it with the simple djb2 hash function created by Dan Bernstein. The function is described here. Unsurprisingly, my initial function, which resulted in a maximum of 19 buckets and guaranteed many collisions, was much slower.

Combining all of my optimizations, I achieved an average time of 0.175 seconds. This represents a speedup of 56.97.