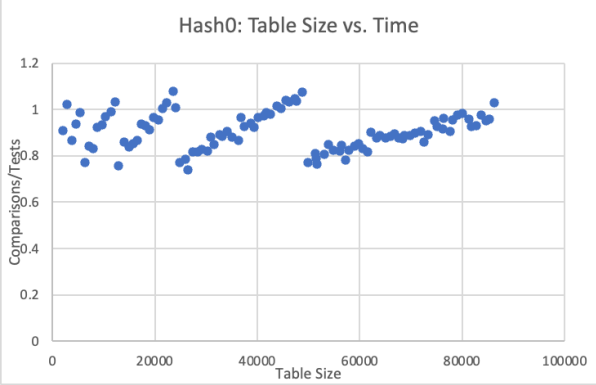
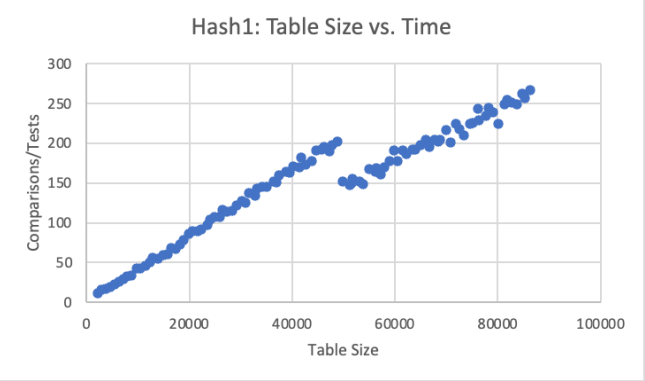
Project 3: Hash Table (Analysis Part)

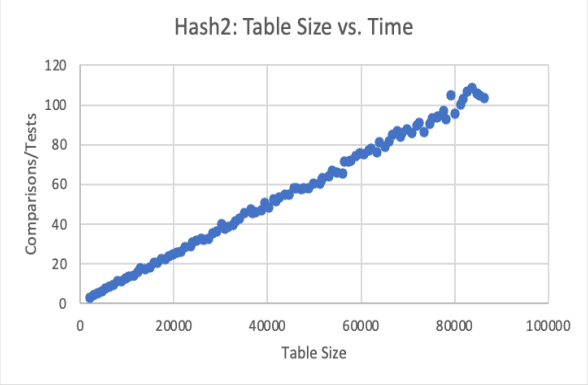
A. Data Collection and Plotting

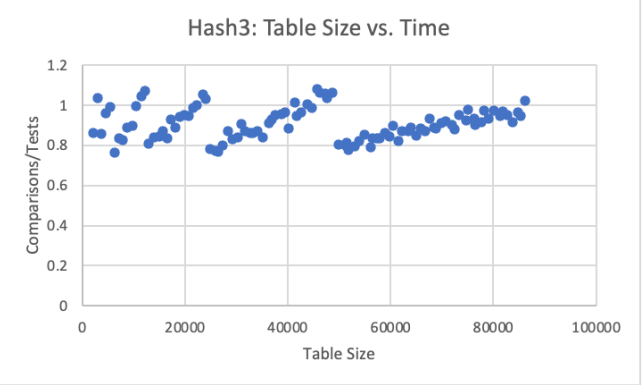
When designed correctly, a hash table should have O(1) time complexity for insert, contains, and remove. The first step in confirming this claim is by collecting information about our implemented hash table.

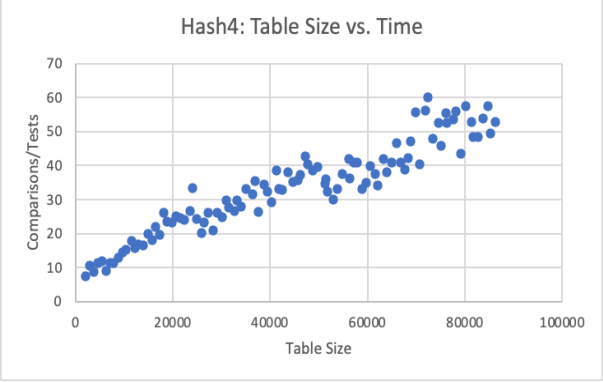
For five different Hash functions, here is the result of the number of comparisons per test for various hash table sizes.

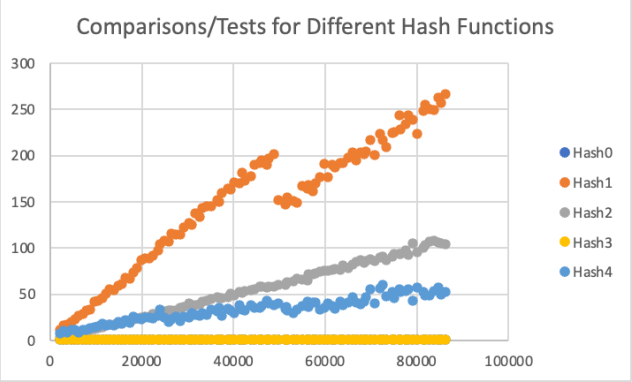












B. Explanation of Scatterplot Features

From our plots, there are a few conclusions we can draw based on the fact that we are using chaining as our method of combatting collisions. In many of our graphs, we see trends when the graph appears to be linear but has breaks that drop the number of comparisons per test down. This is likely due to load factors that require a doubling of table size. When doubling the table size, we must reinsert every value in the hash table; with good hash functions, this should redistribute current values across the entire table, thus reducing the average number of comparisons needed for every test. We see this happening often with hash0 and hash3, and looking at the comparison graph, these are the two hash functions that have a constant time complexity for contains. We can conclude that these two functions are the most efficient hash functions of the five that were tested.

The other three functions have at most one break for the given number of tests and sizes. Looking at the magnitude of the average number of comparisons for these tests, the bigger table sizes can require anywhere from 60x to 300x. These hash functions are clearly not as efficient. We can assume that these functions do a worse job at spreading the values across the entire hash table; instead, they chain more values together. In particular, hash1 appears to be the worst hash function of the batch even with the drop in complexity at around a table size of 50,000. Throughout the range of table sizes, hash1 has the highest comparison/test complexity compared to the four other hash functions.

C. String Searching

Think about functions we might want to do with strings, something we might want to do is see if a given string is a substring of another string. Given that the substring has length m and the string to search from has length n, the brute force approach to this problem would be to check the first m letters of the original string to see if it matches the substring. If it doesn't match, slide one character over and check again.

This has a worst-case Big-O time complexity of **O(nm).** This is because string comparisons are not constant; every time we compare substrings to see if they match, that is an **O(m)** operation because you must compare every letter. In the worst case, you will have to slide over **(n-m) times** and check m letters again because this will get you to the last position that m would fit if it were in the string. This

assumes m is less than n because it was larger, we already know that it is not a substring of the original string. Therefore, we would get the total complexity of O(m+m(n - m)) = **O(m + nm – m²)**. Because n > m, the only term that matters in Big-O is O(nm). It could be noted that as m approaches n, the last two terms will cancel each other out, giving a time complexity of **O(m) for this specific case.**

If instead we use hash codes to compare strings, we may see some speedup. If we take the hash of the pattern and the hash of the section to test, they will be the same code if they match. If they don't match, we can bypass the string comparison, slightly speeding up our time. However, this method does come with its own problems.

We must still directly compare the pattern and substring even if their hash codes match because there is a slight chance that different combinations of letters hash to the same code.

Another, more prominent issue with this idea is that a good hash function will use every letter of the string; the hash function still depends on the size of the substring meaning it is not a constant time operation. Our current method of comparing requires us to calculate the hash of the new substring every time, slowing us down. We instead want to use a rolling hash update equation that has a constant time complexity every time we slide to the next substring. Let's use the hash function given below.

$$h_j = \Big(\big((s[j] \times 31 + s[j+1]) \times 31 + s[j+2]\big) \times 31 + s[j+3]\Big)$$

This equation is for a substring of size 4. The more general case is a sum with m many terms, with each term multiplied by a power of 31. When sliding through the string, the term with the highest power of 31 will get subtracted because this term holds the information of the first character of the string. This character is no longer considered because we have slid past it. No matter what, this term is the letter considered multiplied by 31^{m-1} because the power of 31 for each term is determined by the number of terms in the string. But after we subtract this term, every other term should be multiplied by a factor of 31. This increases the factor of 31 on every term by 1, effectively "sliding over the position" of each character in our hash calculation. Finally, we can add the hash of the next letter in the sequence, completing the hash function in three steps.

$$h_{j+1} = \Big(\big(h_j - \big(s[j] \cdot 31^{(m-1)}\big)\big) \times 31\Big) + s[j+m]$$

It may look like this equation still depends on m, but luckily, we know that m will stay constant throughout our program. Therefore, we can perform the calculation of our power of 31 before we enter the loop, remember the result of the calculation, and simply multiply that every time we want to subtract the largest term.

Let's now analyze our new function. No matter what, we still need to calculate our first hash code, which involves **O(m)** calculations. However, this only happens once because from here, we can have a constant time operation calculate the rest of the hash codes. But how many constant time operations are there? Well, each slide requires constant time operation, and in the worst case, we will need **(n-m) slides** until we get to the end of the string. This new rolling hash has a Big-O time complexity of **O((n-m) + m)** which simplifies to **O(n)**. Because the substring could theoretically be anywhere in the string, the average complexity would have our (n-m) term divided by two, but in terms of Big-O, we will still get **O(n)** as our simplified average complexity.