Assignment 07 Analysis Document:

1. Explain the hashing function you used for BadHashFunctor. Be sure to discuss why you expected it to perform badly (i.e., result in many collisions).
   BadHashFunctor:

   a. **Why it performs badly**: Hashes based only on string length, leading to many collisions for strings of the same length.
   b. **Expected performance**: High number of collisions, poor performance overall, especially for larger problem sizes.

2. Explain the hashing function you used for MediocreHashFunctor. Be sure to discuss why you expected it to perform moderately (i.e., result in some collisions).

   MediocreHashFunctor:

   c. **Why it performs moderately**: Hashes based on the sum of character values, which helps distribute keys better than `BadHashFunctor`, but still causes some collisions for strings with similar character sums.
   d. **Expected performance**: Moderate number of collisions, performs better than `BadHashFunctor`, but still suboptimal.

3. Explain the hashing function you used for GoodHashFunctor. Be sure to discuss why you expected it to perform well (i.e., result in few or no collisions).

   GoodHashFunctor:

   e. **Why it performs well**: Uses a multiplicative method with a prime constant (31), ensuring that changes in the string lead to different hash values, and spreading the values more evenly across the hash table.
   f. **Expected performance**: Few collisions, very efficient, especially as problem size increases. This is the most optimal of the three hash functions. However, during timing experiments the GoodHashFunctor performed worse than my BadHashFunctor probably due to the overhead in the hash function itself.

4. Design and conduct an experiment to assess the quality and efficiency of each of your three hash functions. Briefly explain the design of your experiment. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plot(s), as well as, your interpretation of the plots is important. A recommendation for this experiment is to create two plots: one that shows the number of collisions incurred by each hash function for a variety of hash table sizes, and one that shows the actual running time required by various operations using each hash function for a variety of hash table sizes.

Setup of Timing Experiment:

1. **Problem Sizes (n)**: The table size (n) is incremented from 1,001 to 10,001 in steps of 1,000. These values represent the number of strings that are inserted and removed from the hash table. Each time, a set of random strings is generated and added to the table, and then the strings are removed.
2. **Setup**: For each value of n:
    - A new `ChainingHashTable` is created for each of the three hash functions (Good, Mediocre, and Bad).
    - Each hash table starts with a size of 2n (to accommodate the insertions).
    - The experiment runs for each of these hash functions with a fixed set of random strings of length proportional to n.
3. **Operations**:
    - **`addAll()`**: Adds all n random strings to the hash table.
    - **`removeAll()`**: Removes the same n strings from the hash table.
    - The operations are timed, and the total time for each hash function at each table size is measured.
4. **Timing**:
    - Each experiment runs several iterations to ensure the results are statistically meaningful.

Setup of Collision Experiment:

This experiment tests the performance of three different hash functions (Bad, Mediocre, and Good) by counting the number of collisions that occur when inserting random strings into a hash table. The experiment involves the following steps:

1. **Problem Sizes**: The experiment tests five different hash table sizes (problem sizes), which are:
    - 100
    - 500
    - 1000

    - 5000
    - 10000 These sizes represent the number of elements the hash table can hold.
2. **Hash Functions**: The experiment uses three different hash functions to test their performance:
    - **Bad Hash**, **Mediocre Hash** and **Good Hash**.
3. **Random Strings**: For each problem size, a list of random strings is generated. The length of each string is approximately equal to the size of the hash table (i.e., the number of entries n).
4. **Experiment Process**:
    - For each problem size, the random strings are inserted into three separate hash tables, each using one of the three hash functions.
    - The number of collisions that occur during the insertion process is recorded for each hash function.
    - This is repeated for several iterations.
5. **Collisions**:
    - The number of collisions is recorded at each problem size and is used as a measure of the efficiency of the hash function.

5. What is the cost of each of your three hash functions (in Big-O notation)? Note that the problem size (N) for your hash functions is the length of the String, and has nothing to do with the hash table itself. Did each of your hash functions perform as you expected (i.e., do they result in the expected number of collisions)?

Timing experiment Analysis:

- **BadHashFunctor Time**: This function starts with very low time (even better than `GoodHashFunctor` for small n), and the times grow slowly as n increases.
- **GoodHashFunctor Time**: `GoodHashFunctor` has noticeably higher times in the small values of n (like 1001) compared to `BadHashFunctor`. This indicates that `GoodHashFunctor` might be doing something more complex, leading to overhead or extra computations.
- **MediocreHashFunctor Time**: The times fall between the other two hash functions, which suggests that `MediocreHashFunctor` is somewhere in between `Good` and `Bad` in terms of efficiency.

Collision experiment Analysis:

- All three hash functions exhibit **linear growth in collisions** with respect to the problem size. This suggests that in terms of Big-O notation, the collision count grows **linearly**

with the number of elements being inserted into the hash table: **O(n)** for all three hash functions.

- However, the key difference is the **constant factor** (the number of collisions per element), which is much smaller for the **Good Hash Functor** than for the **Bad** or **Mediocre** Hash Functors. The **Good Hash Functor** distributes the keys much more evenly across the hash table, resulting in fewer collisions overall.
- **Bad Hash Functor**: **O(n)**, with a **high constant factor** for collisions.
- **Mediocre Hash Functor**: **O(n)**, with a **moderate constant factor** for collisions.
- **Good Hash Functor**: **O(n)**, with a **low constant factor** for collisions. It performs the best in terms of minimizing collisions.