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).**

The hashing function that is used for this assignment; returns the length of the item as its integer. Returning the length of the item leads to collisions because all words that are different but have the same length would collide. In code something like str “apple” and “Brian” would have the same int value (its hashcode) when they are two different strings.

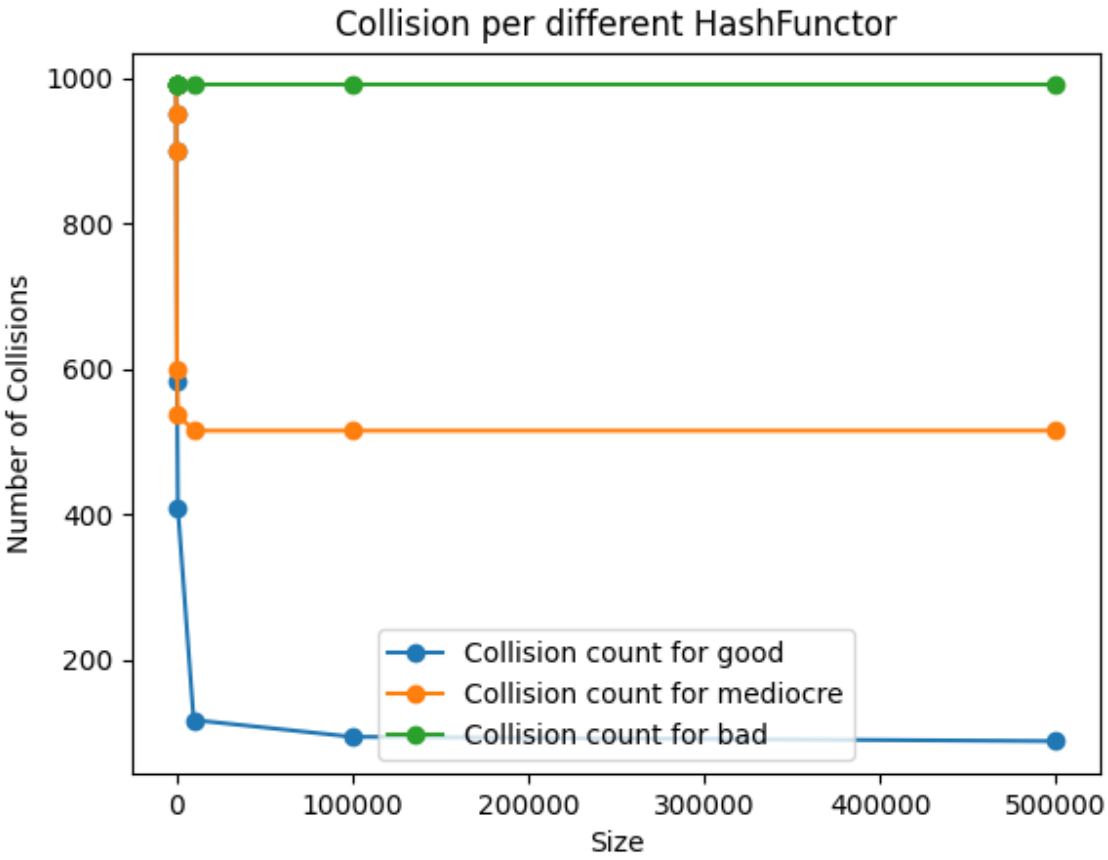
2. **Explain the hashing function you used for MediocreHashFunctor. Be sure to discuss why you expected it to perform moderately (i.e., result in some collisions).**

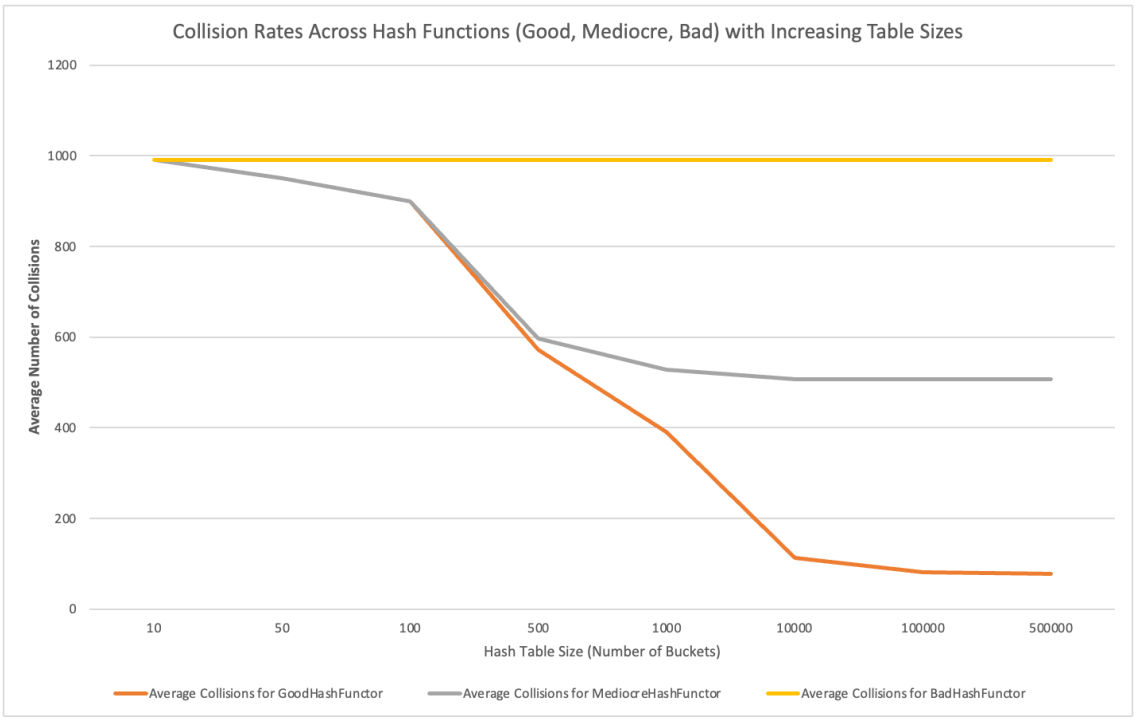
The MediocreHashFunction that is used for this assignment; returns an int based on the sum of the char values from all chars from the string. As an example of possible collisions strings like abc and cba, bbb (all of these three have a total of 294 from their ASCII would still collide. It is expected to perform better than the BadHash function because different words that share the same length could still have different unique integer values assigned to them.

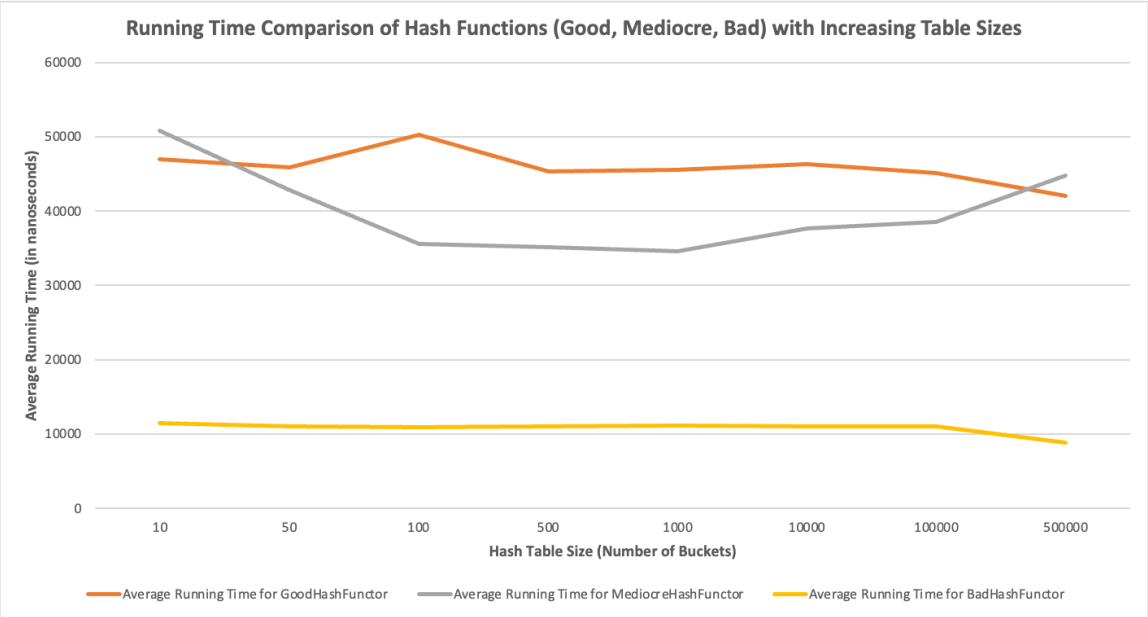
3. **Explain the hashing function you used for GoodHashFunctor. Be sure to discuss why you expected it to perform well (i.e., result in few or no collisions).**

TheGoodHashFunction that is used for this assignment; returns instead of char values of the sum in the String; it iterates over each char of the string; multiplies the current hash by 31 (Prime number) and adds the ASCII value to each char as its integer key. It performed well compared the other two functions because the non-linear manipulation of the hash integer improves more diversity for different strings; it’s a more even distribution across the different strings.

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.**


Collision per different HashFunctor
Number of Collisions
Size
1000
800
600
400
200
0
100000
200000
300000
400000
500000
Collision count for good
Collision count for mediocre
Collision count for bad


Collision Rates Across Hash Functions (Good, Mediocre, Bad) with Increasing Table Sizes
Average Number of Collisions
Hash Table Size (Number of Buckets)
1200
1000
800
600
400
200
0
10
50
100
500
1000
10000
100000
500000
Average Collisions for GoodHashFunctor
Average Collisions for MediocreHashFunctor
Average Collisions for BadHashFunctor



The conducted experiment consisted in following the same steps as previously for size and measuring time using System.nanoTime() for precise time measurements. The calculated collisions were made by iterating through the hash table components and assessing how many strings had the same index in the full string collection. We used this method to get the total of collisions private static int measureCollisions(HashFunctor hashFunctor, String[] strings, int tableSize) {

```
// Map to store the count of strings at each hash code

Map<Integer, Integer> hashTable = new HashMap<>();



// Insert each string into the hash table and count collisions

for (String str : strings) {

    int hash = Math.abs(hashFunctor.hash(str)) % tableSize;

    hashTable.put(hash, hashTable.getOrDefault(hash, 0) + 1);

}
```

```
// Count total collisions in the hash table
int collisions = 0;
for (int count : hashTable.values()) {
    if (count > 1) {
        collisions += count - 1;
    }
}

return collisions;
```

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)?**

Both the MediocreHashFunctor and GoodHashFunctor have linear time complexity O(N) because both iterate through each character, and the time complexity grows proportionally with the number of characters in the string, making their cost increase linearly with the length of the input string. The BadHashFunctor maintains a constant time complexity O(1), irrespective of the input string length because the hash function simply returns the length of the string, resulting in a constant time complexity.