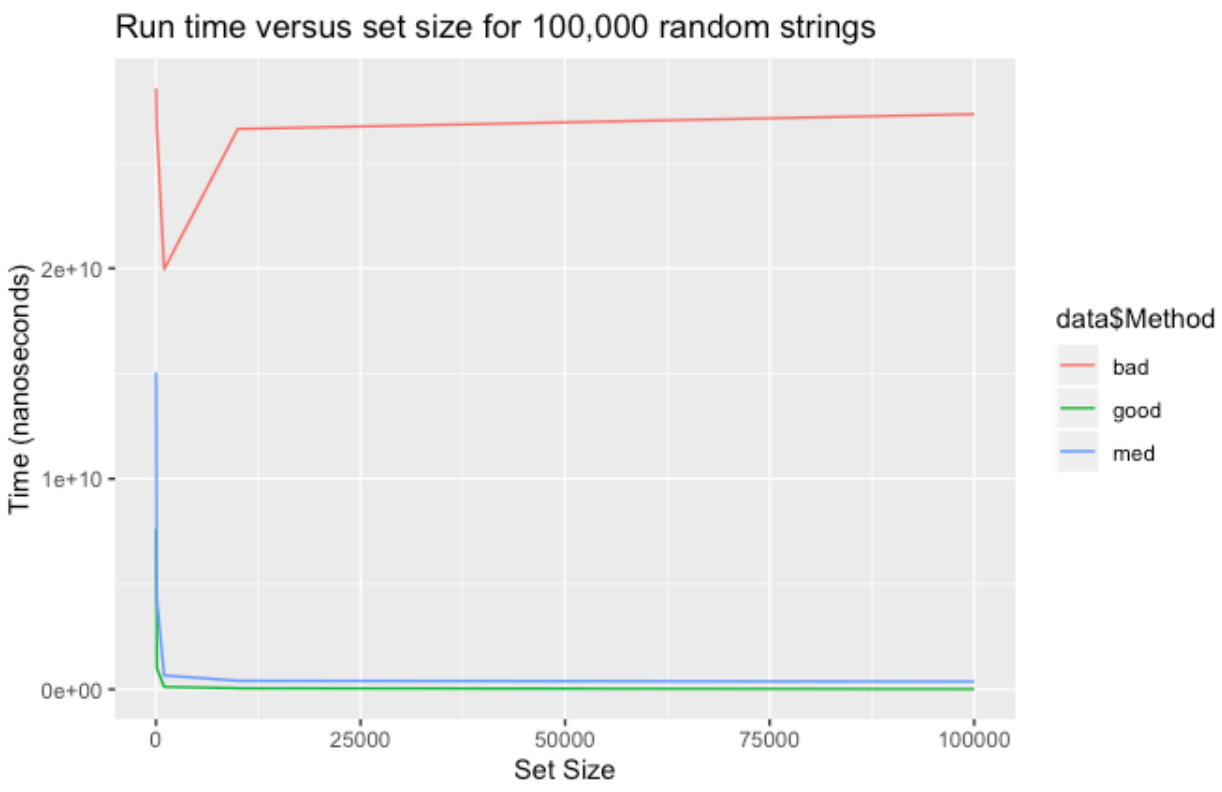
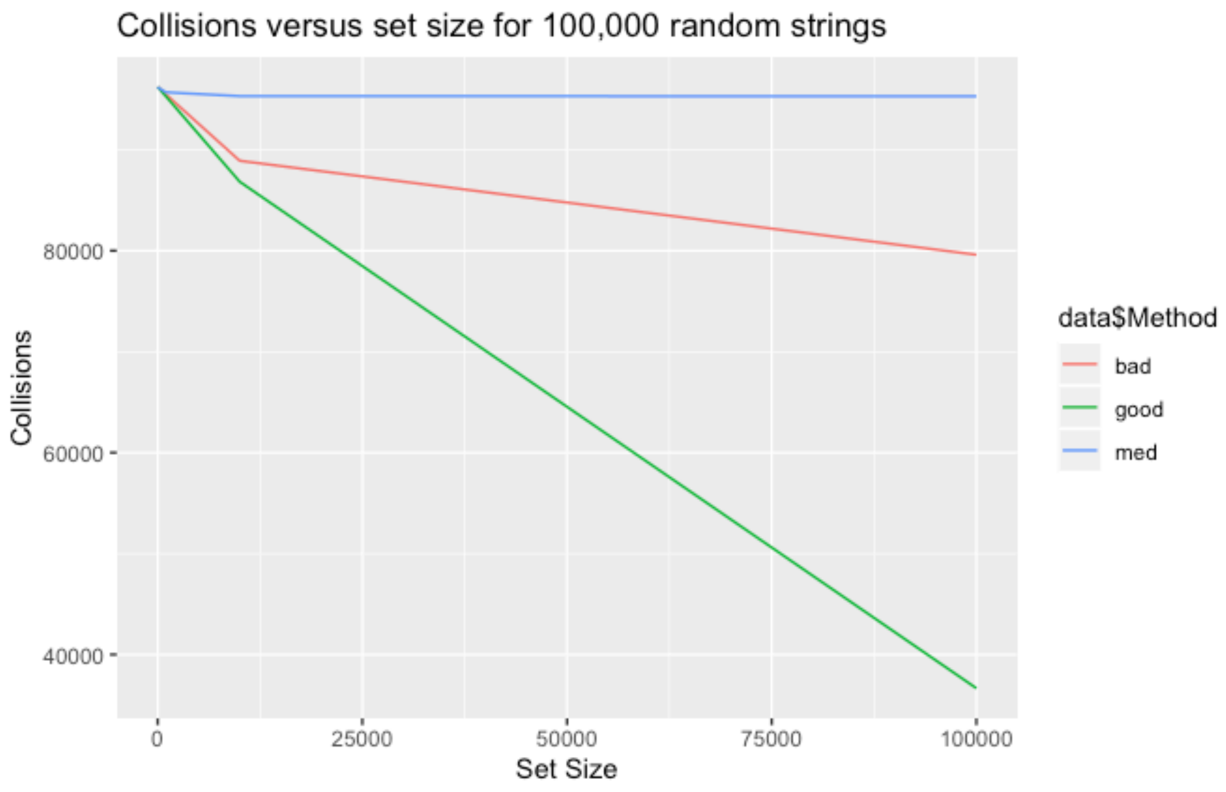
11/26/19 Derek Olson

1. My bad hash functor attempts to recreate the Java hash function. This was originally my good hash functor, but after testing I found that it was very slow so I changed it to be my bad hash functor.This method multiples each character in the string by 31 to the power of the number of characters in the string minus the characters position in the string. These values are then summed to derive the hash value. The exponentiation in this function creates a greater spread of values than the mediocre hash functor, and results in fewer collisions. Although it results in fewer collisions the complex computations of the hashing results in slower performance.
2. My mediocre hash functor sums the values of each character of the string. It does this by putting each character into a character array, and iterating across the character array to derive the sum. It then multiplies the sum by the length of the word. This takes advantage of the character values and word size to create a unique value. As these values are more likely to be unique, they are less likely to result in collisions.
3. My good hash functor replicates the djb2 hash for java. This hash creates the greatest spread of hash values in comparison to my other hash functors. It is capable of taking advantage of large array capacities, and it is efficient to calculate.
4. My experiment times the addAll method for hash tables resulting from each of the functors. The addAll method incorporates the add and contains methods, which both must utilize hashing.
    1. In the figure below the run time for each hash functor was measured across set sizes ranging from 10 to 100,000. The bad hash functor was substantially worse than the other 2 functors, and even increased slightly as the set size increased. Although this increase was somewhat small, it was unexpected. Especially since the number of collisions actually decreased below the level of the mediocre hash functor at large set sizes. The mediocre and good hash functors saw constant time throughout the range of set sizes, which was expected. The mediocre hash functor performed nearly as good as the good functor for all tested set sizes. Perhaps at much larger set sizes the good hash functor would see increased relative performance gains over the mediocre hash functor.



2. In the figure below the good hash functor had many few collisions than the other 2 hash functors. This difference increased with an increase in set size. The bad hash functor had fewer collisions than the mediocre hash functor. However, as described previously, it still had slower runtimes. Perhaps this was due to the complication of calculating the hash.



5. The bad hash functor should have an order N big-O notation. It loops through each letter of the string and performs a calculation. Thus the big-O of N corresponds to the number of characters in the string. I am not quite sure why this hash functor seems to take so much longer to calculate than the other hash functors which are also order N. LIke the bad hash functor, both the mediocre and good hash functors perform a calculation on each letter of the string. This makes them order N as well. And unlike the bad hash functor these 2 performed as expected.