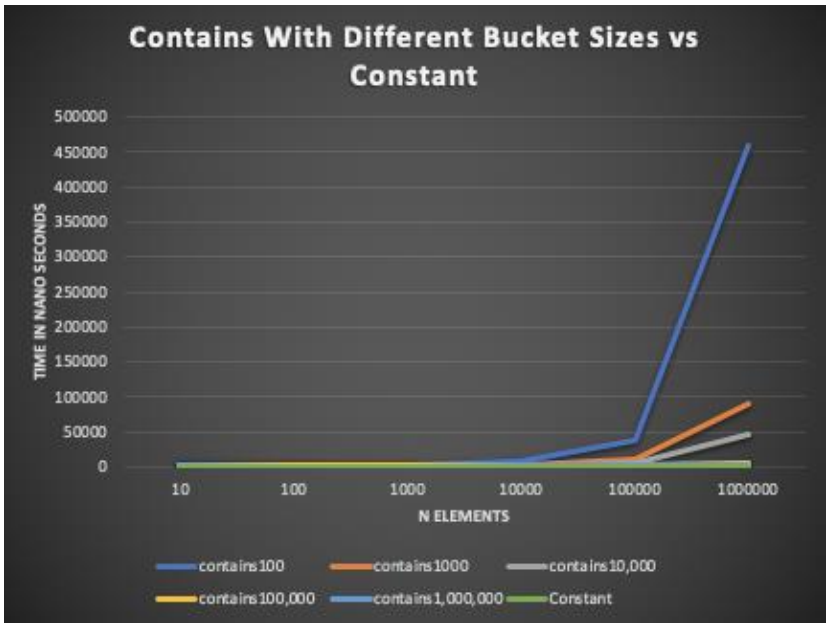

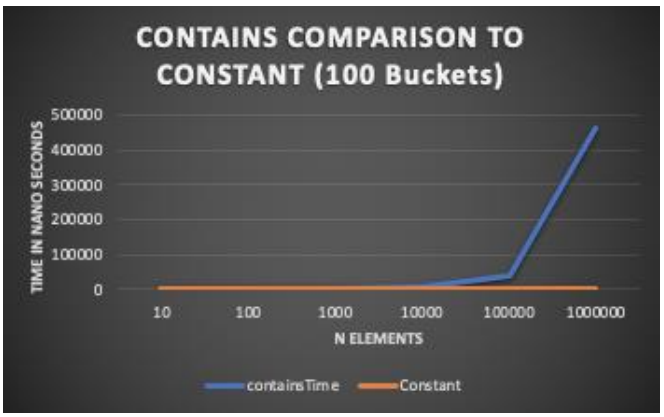

← the # following contains line, is the number of buckets during that test

1. We were constant up until the time we hit 100k elements, that is because we hit the worst case of O(N), this be because the buckets were not being changed or re-hashing our hash table. If we were to modify the way we addressed the buckets, keeping a small amount of element within each bucket, we could have retained O(1), however, since we were just using the default number of buckets, we reached the worst case; which should happen if you don't rehash. The number of buckets does affect the performance of contains. Since we are not rehashing, the number of buckets has to applicable to the size of the number of elements stored in the HashTable. That is why the slope of the linear part decreases when we use more buckets.

2. It looks like all of the elements are stored in the first 14 buckets in a bell curve fashion. No, it is not a good distribution. There are plenty of empty buckets throughout the whole hash table

- It is just one straight line, indicating that all of the elements are stored in one bucket. No, the distribution is terrible and worse than the first one. There's no point in using a hash table if all of them are stored in one bucket.

-It's similar to the first one. However, it's spread out a bit more by covering the first 20 buckets. No, it's not a good distribution for the same reason as the first one. It only utilizes 2% of the buckets available.

-This one looks the same as the first and third one as well, leading us to believe that the distribution is in fact pretty well spread out, but considering the number of buckets there are, we still think that it could be utilized more efficiently by either reducing the number of buckets or spreading out the elements throughout the rest of the buckets

-It's terrible because all of the strings are the same length so when we mod, the string element all return the same number and go into the same bucket. The way we could fix is this is by changing the hash function itself, to where it doesn't just return the length of the string.

-Changing the number of buckets wouldn't fix the distribution for the a08length text file. The reason behind that is because since the hash function is returning the length of the string, every string in that file will be stored in the same bucket no matter how many buckets there are.

-The evil adversary is the mod30 text file. We lessened his scheme by changing the number of buckets to 29, which is a prime number, resulting in the spread of elements more uniform than before. However, it is still not completely uniform, some having 10 more than others, and this could be improved more. We concluded from other tests that when we use prime numbers, the distribution was more uniform rather than something to the bell curve or complete outliers. The advantage is that when we use prime numbers the remainder from the mod function will vary because there won't be the possibility of the mod function over filling one bucket.

-The way we implemented the hash code adding the values of the first half of the string, and then multiplying it by a prime number, which in our case was 3. We think that it does an okay job. Every file looks to be uniform/distributed evenly. However, we found that when the number of buckets isn't a prime number, everything is distributed out in buckets of multiples of 3. As for if our implementation could be improved, we don't think that there would be any good way to improve it without completely changing the method, because the elements would still be distributed in the different buckets of multiples of whichever number used. We might be able to improve it if we could get the number of buckets that the HashTable was using to change the way we multiplied. Overall, we think the sum of char values is better than the way we did it, just because it still distributes elements in all the buckets, instead of ours that only fills the buckets of multiples of the number we multiply by.

```java
@Override
public int hashCode() {
    // converts string to char array
    char[] strChar = str.toCharArray();
    int value = 0;
    for (int i = 0; i < str.length() / 2; i++) {
        // adds the first half of the character's values together
        value += strChar[i];
    }
    // returns the first four values together multiplied by a prime number
    return value * 3;
}
```