N/B:I HAVE SCREENSHOTS, TOGETHER WITH MY WORK TO ILLUSTRATE MY VALUES. THE FILES ARE NAMED AS FOLLOWS:
1.Brute and Brute1 for testing Brute.
2.MapNew and MapNew2 for testing map.
3.BadHash code1 and BadHashcode2 for testing hashcodes.
4.TreeMap1 and TreeMap2 for testing treemap values.

PART A:BRUTE VS MAP:

BRUTE FORCE:

(I). The length of the training text: Inside the for loop, for (int i = 0; i < length; i++), there is a while loop,

i.e. while (num != text.size() + 1) {
num = text.indexOf(seed, num) + 1;…….

This means that, each time the for loop runs, text.indexOf(seed,num) runs up to the point where that particular seed doesn't exist in the training text.If you look at the implementation of indexOf in training text, i.e."public int indexOf(NGram seed, int startPos) {
for (int i = startPos; i < size(); i++) {
if (get(i).equals(seed)) {
return i;
}
}
return size();", the method takes O(m) time(assuming m is length of training text),since it goes through the entire training text.Thus, as the length of the training text increases, the more the runtime of the while loop in BruteGenerator increases and vice versa.

(II). Length of the random text:If the length of the random text is n, then the runtime would be expressed as O(n) i.e. the runtime is dependent on n.Thus, as the length of the random text that we are supposed to produce increases,the runtime increases and vice versa. This is expressed in my Brute Generator class as "for (int i = 0; i < length; i++)", meaning that the for loop will go from 0 up to, but not including length.

(III). The k value i.e. the size of the ngram does not affect the runtime. This is because, in generate text method, we go through the entire training text, regardless of the size of the gram.Also, we shift only one character/word to go to the next gram, hence what

determines the number of grams that we'll have to go over is determined by the size of training text and not the size of the gram.

MAP:

(I). We'll have to go through the entire training text in the train method, and hence the runtime of this operation is dependent on the size n of the training text. The larger the size, the longer the runtime.
(**II).**In the generate text method, we'll generate text length times, so the length of the random text will affect the runtime propotionaly to n.

(II). The k'th value will also not affect runtime for the same reasons that I stated in brute.

Map however takes a shorter timeto run than brute because we go through the training text and length separately, unlike in brute where they are nested together, and if the run time for the while and for loop are n and m respectively, runtime for brute becomes O(nm), while that of map generator is n+m which will evaluate to the leading term. Also, accessing a random item in a hash map takes O(1) time.

WITH DATA:
All my hypothesis are supported with the data, except from a few outliers that may have resulted from factors beyond my control. I felt my hypotheses was pretty solid.

PART B:
Hypothesis:

1.With a constant hash code, the time taken increases by something close to O(n). This is because, the hash codes of the grams will hash to the same bucket which results into the creation of a linked list, in order to store the rest of the values. Hence, at worst case, accessing a key from the hash map will take O(n), instead of the usual O(1). Runtime becomes slower as n/keys increases since the runtime now depends on n.

2.With the good hash code,the number of keys are evenly distributed in the buckets, hence accessing and inserting items into the hash map takes O(1). The number of keys will thus not affect runtime.

3.A treemap however uses the comparator interface to sort the keys in order. Adding an item to the tree map(at worst case) will thus take O(log n) as implementation uses binary search trees. Accessing and adding grams to the tree map will thus take O(log n), hence the more keys we have, the longer the program will take and vice versa.

WITH DATA:

(i).Like I said, the runtime with the bad hash code is dependent/ directly proportional to n, hence, we can express the mathematical statement that runtime=cn, where c is a constant and n is the size of the training text/number of unique keys. If we were to take 3 random numbers i.e. Plugging in the values of runtime and unique keys gives me constants of 5.66*10^-5,5.03*10^-5 for the first two values in the table and 5.84494*10^-6. for runtime is 0.402845 and 68922 keys. The rest of the constant values are in the range of approximately 2.4-8^-5. I concluded that 3 values are enough to give me a line of best fit, with other values close enough to the line of best fit. Thus my conclusion that a bad hash code makes the runtime be affected by O(n) appears to be true.

(ii).My hypothesis was that the number of unique keys does not affect the runtime of the program, as much since adding and searching for n grams take constant time O(1). Between unique keys 2694-7499,the run time is approximately 0.02,hence proving my hypothesis that runtime for an efficient hash code isn't affected by the number of unique keys. They're increases in the runtime as the keys increase thereafter, but that increase is not as large, and I can attribute factors such as processor speed e.t.c. for the outliers.

(iii). The runtime as I stated earlier is now directly proportional to log (n). Calculating the regression of the line using a calculator bends towards this, though not accurately. Some values are way off the line but most are within the line of best fit in the calculator.

PS: NEVER HAD MUCH TIME TO DO ANALYSIS, HENCE I JUST LOOKED OVER DATA AND GRAPHED ON A CALCULATOR.