**Part A: Brute Force vs Map**

Form a hypothesis about how each of the following three factors should affect the runtime of BruteGenerator and MapGenerator in big-O notation and explain your reasoning by referencing segments of your code.

i. the length of the training text
ii. the k-value or length of the word
iii. the length of the random text

Solution:

**Theoretical Analysis**

Let's say the length of the training text - m;
the k-value - k;
the length of the random text - n;

The running time for BruteGenerator is O(n*m*(1/k)logk); while the running time for MapGenerator is O(n*m*exp(-k)).

For the BruteGenerator: (analysis in red letter)

the TrainingText to form the textMode, prove running time is proportional to exp(-k).

```
for(int i=0; i<length; i++){//generate  new text with length "length"
            this "for loop" prove the running time is proportional to
            to n
                  ArrayList<NGram> followstate = new
ArrayList<NGram>();//the compiled list of the following states after
"seed"
                  for (int j=0; j<new_TT.size();j++){//last NGram has
not "followstate" this "for loop" prove the running time is
proportional to to m
                        int index=new_TT.indexOf(seed, j);
                        if(index==(new_TT.size()-1)||
index==new_TT.size()){break;}
                        //index=new_TT.size()-1, then, the following
NGram is null; if index=new_TT.size(), no found seed, then break.
                        followstate.add(new_TT.get(index+1));//add the
next NGram after "seed" to the compiled list
                        j=index;
                  }//end of creating the compiled following states for
the "seed"
                  output.append(seed.toString());
                  seed =
followstate.get(rnd.nextInt(followstate.size()));//update seed with
one random new following states
            }
```

For the MapGenerator: **(analysis in red letter)**

the TrainingText to form the textMode, prove running time is proportional to exp(-k).

```
for(int i=0; i<length; i++){//generate  new text with length "length"
this “for loop” prove the running time is proportional to to n
                random1 = rnd.nextInt(stateMap.get(seed).size());//
random number under the size of list of the value in map the size of
sateMap is proportional to m prove the running time is proportional
to m
                output.append(seed.toString());
                seed = stateMap.get(seed).get(random1);
            }
```

**Empirical Analysis**

For the BruteGenerator: (analysis in red letter)

```
Varying k, using random text length 100 and file length 152145
(alice.txt)
k: 1   mean: 1.091622   stddev 0.007755       ci: [1.076422, 1.106822]
k: 2   mean: 1.034834   stddev 0.000116       ci: [1.034607, 1.035062]
k: 3   mean: 1.150517   stddev 0.000058       ci: [1.150404, 1.150631]
```

We can see, when k change from 1 to 2, 3, the running time increase slowly, which is proportional to (1/k)logk.

```
Varying text length, using k 5 and file length 152145 (alice.txt)
text length: 20   mean: 0.243674   stddev: 0.000045      ci: [0.243586,
0.243762]
text length: 40   mean: 0.498932   stddev: 0.000140      ci: [0.498658,
0.499207]
text length: 60   mean: 0.748870   stddev: 0.000187      ci: [0.748504,
0.749236]
```

We can see, when text length (n) change from 20 to 40, 60, the running time decrease from almost proportionally. Therefore, the running time is proportional to text length (n).

```
Varying file length, using k 5 and text length 100
unique keys: 4439      mean: 0.034171   stddev 0.000002       ci:
[0.034167, 0.034175]
unique keys: 4823      mean: 0.037135   stddev 0.000003       ci:
[0.037130, 0.037140]
unique keys: 5953      mean: 0.046078   stddev 0.000006       ci:
[0.046067, 0.046089]
```

We can see, when text length (m) change from 2694 to 2982, 3939, the running time decrease from almost proportionally. Therefore, the running time is proportional to text length (m).

For the MapGenerator: (analysis in red letter)

```
Varying k, using random text length 100 and file length 152145
(alice.txt)
k: 1   mean: 0.000339   stddev 0.000000       ci: [0.000339, 0.000340]
k: 2   mean: 0.000083   stddev 0.000000       ci: [0.000083, 0.000083]
```

We can see, when k change from 1 to 2, the running time decrease from 0.000339 to 0.000083.

And, 0.000083=0.00039*exp(-2)*0.5. However, when k is even bigger, the running time generally is slowly decreasing, which fits to exp(-k)

```
Varying text length, using k 5 and file length 152145 (alice.txt)
text length: 20   mean: 0.000018   stddev: 0.000000      ci: [0.000018,
0.000018]
text length: 40   mean: 0.000032   stddev: 0.000000      ci: [0.000032,
0.000032]
text length: 60   mean: 0.000056   stddev: 0.000000      ci: [0.000056,
0.000056]
```

We can see, when text length (n) change from 20 to 40, 60, the running time decrease from almost proportionally. Therefore, the running time is proportional to text length (n).

```
Varying file length, using k 5 and text length 100
unique keys: 2694      mean: 0.000022   stddev 0.000000       ci:
[0.000022, 0.000022]
unique keys: 2982      mean: 0.000026   stddev 0.000000       ci:
[0.000026, 0.000026]
unique keys: 3939      mean: 0.000030   stddev 0.000000       ci:
[0.000030, 0.000030]
```

We can see, when text length (m) change from 2694 to 2982, 3939, the running time decrease from almost proportionally. Therefore, the running time is proportional to text length (m).

**Part B: HashMap VS. TreeMap**

In these questions, the goal is to compare the different kinds of Maps, so you will only run the MapGenerator through Benchmark. You will examine the performance of HashMap both with a good hash function and bad one, and a TreeMap on different number of keys from the training text. Note that for this section only data from the last segment of Benchmark output is relevant.

i. HashMap with the default hashCode function (always return a constant)
ii. HashMap with the hashCode function you wrote
iii. TreeMap

Solution:

**Theoretical Analysis:**

the running time constant hashCode will be O(m), m is the text length. And, it will take more time compared to a good hashCode function, since it will has more collision.

the running time for the hashCode I wrote will be O(m), m is the text length.

the running time for TreeMap will be O(m), m is the text length. But, it will take longer time compared the previous two hashCode cases, since the train method will take longer time to create the TreeMap.

**Empirical Analysis:**

By changing the HashCode to a constant (3333) in NGram class, the last segment of Benchmark output is:

```
Varying file length, using k 5 and text length 100
unique keys: 2694      mean: 0.000150   stddev 0.000000       ci:
[0.000150, 0.000150]
unique keys: 2982      mean: 0.000130   stddev 0.000000       ci:
[0.000130, 0.000130]
unique keys: 3939      mean: 0.000146   stddev 0.000000       ci:
[0.000146, 0.000146]
unique keys: 7499      mean: 0.000175   stddev 0.000000       ci:
[0.000175, 0.000175]
unique keys: 7777      mean: 0.000184   stddev 0.000000       ci:
[0.000184, 0.000184]
```

while, for the hashCode function I wrote, the last segment of Benchmark output is:

```
Varying file length, using k 5 and text length 100
unique keys: 2694      mean: 0.000022   stddev 0.000000       ci:
[0.000022, 0.000022]
unique keys: 2982      mean: 0.000026   stddev 0.000000       ci:
[0.000026, 0.000026]
unique keys: 3939      mean: 0.000030   stddev 0.000000       ci:
[0.000030, 0.000030]
unique keys: 7499      mean: 0.000051   stddev 0.000000       ci:
[0.000051, 0.000051]
unique keys: 7777      mean: 0.000042   stddev 0.000000       ci:
[0.000042, 0.000042]
```

the running time for constant hashCode is longer than the hashCode I wrote; while, the running time for the constant hashCode is increasing slower than the increase of the second case.