1. Determine (from running Benchmark.java) how long it takes for BaseMarkov to generate
2,000, 4,000, 8,000, 16,000, and 32,000 random characters using the default file and an order 5 Markov Model.
Include these timings in your report. The program also generates 4,096 characters using texts that increase
in size from 487,614 characters to 4,876,140 characters (10 times the number). In your analysis.txt file include
an explanation as to whether the timings support the O(NT) analysis. Use the fact that for some runs N is fixed and
T varies whereas in the other runs T is fixed and N varies.

time	source	#chars
0.106	487614	1000
0.175	487614	2000
0.286	487614	4000
0.700	487614	8000
1.396	487614	16000
2.866	487614	32000
5.592	487614	64000
0.364	487614	4096
0.760	975228	4096
1.104	1462842	4096
1.467	1950456	4096
1.783	2438070	4096
2.174	2925684	4096
2.530	3413298	4096
2.920	3900912	4096
3.286	4388526	4096
3.660	4876140	4096

For the first set of runs, the size of the file is kept constant at 487614 characters (N), while the number of random
characters being generated is doubling (T). The O(NT) analysis seems to be supported because as the characters double,
like if we have (2T characters,N file size) and (4T characters, N file size), the difference in runtime is approximately by a
factor of two, where the set of (4T characters, N file size) take twice as long. This is because 4T*N = 2*2T*N, so we should
expect a runtime of O(4T*N) to take twice as long as O(2T*N)> For instance, going from 8000 characters
to 1600 characters approximately doubled the runtime.

On the second set of runs, characters generated (T) is held constant, while source size (N) varies. For example,
(975228*4096) is twice as large as (487614*4096), and thus has about twice as much runtime (0.760 vs 0.364). The same follows for the rest of the numbers.

The runtime of O(NT) makes sense based on the BaseMarkov code because each time we generate a random character, the model
scans the entire training text (of N characters) to find the characters that follow it when calling getFollows (since .getFollows has a runtime complexity of O(N)). If we do this for T random characters, we have O(NT).

2. Determine (from running Benchmark.java) how long it takes for EfficientMarkov to generate
2,000, 4,000, 8,000, 16,000, and 32,000 random characters using the default file and an order 5 Markov Model.
Include these timings in your report. The program also generates 4,096 characters using texts that increase
in size from 487,614 characters to 4,876,140 characters (10 times the number). In your analysis.txt file
include an explanation as to whether the timings support the O(N+T) analysis.

time	source	#chars
0.161	487614	1000
0.113	487614	2000
0.111	487614	4000
0.066	487614	8000
0.075	487614	16000
0.068	487614	32000
0.117	487614	64000
0.085	487614	4096
0.151	975228	4096
0.254	1462842	4096
0.333	1950456	4096
0.421	2438070	4096
0.504	2925684	4096
0.639	3413298	4096
1.143	3900912	4096
1.324	4388526	4096
1.526	4876140	4096

Yes, this supports the O(N+T) analysis because the time complexity follows more of a linear pattern than one
with multiplication, like O(N*T). This is apparent, as 487614 * 4096 is much greater than 487614 + 4096, and so the runtime with source = 487614 and #chars = 4096 is much greater for a O(N*T) complexity model (time = 0.364) than a O(N+T) model (time = 0.085).

I noticed For instance, in the first section where the source (N) is held constant, when doubling
the number of characters generated, we do not see that the time has increased by a factor of two. In fact, because adding 2000 (T) to

a source value of 487614(N) for a complexity of O(N+T) is not that different comparatively to adding 4000 (T) to a source value of 487614(N) when the complexity of O(N+T). This is why the time taken to run these two is approximately the same. This is the same for the rest of the values, which is why there isn't clear variation in time complexity, because the changes in T are so comparatively small to N (487614), so N+T won't change too much with these small changes in T.

In fact, the numbers for the first section are so low that it is normal to have dips in runtime. (In fact, each time I run benchmark, the numbers are different, but they still follow an O(N+T) pattern. It could be due to different factors such as Java performing garbage collection, or another program on my computer. I tried running Benchmark with less applications open on my computer, and the numbers all changed, but still followed an O(N+T) pattern). This dip makes sense because even 64000 is comparatively less than 487614, so adding something that is almost 10 times smaller than itself to itself will not give that much of a change in result, and the runtime is affected by many factors.

On the second part, when number of characters generated (T) is held constant, while file size (N) increases, we can more easily see the O(N+T) pattern. For instance, by adding 487614 + 4096 vs 975228+ 4096, we are approximately doubling O(N+T), and we get approximately double the runtime (.085 vs .151). When we add 975228 + 4096 vs 1462842 + 4096, we are approximately increasing N+T by a factor of 1.5 because 1462842/975228 = 1.5. (I disregarded the 4096 because it is considerably smaller than the value that is being varied (N)). We also see a similar increase in runtime : .254/.151 = 1.681. This pattern is continued, as (4876140 + 4096) / (4388526 + 4096) is approximately 1.11, and 1.526/1.324 is approximately 1.15. However, if these values were multiplied,
they would be much larger and thus increasing faster (in terms of objective time increase) (at a rate proportional to the change in source).

Our result makes sense based on the way the code for Efficient Markov was written, because we rewrote EfficientMarkov.getFollows to be an O(1) operation, by iterating through the source (of size N) once, and then saving each substring in a map (as a key) and adding each character that follows that key (substring) as a value. This means generating T random characters by calling getFollows T times will be an O(T) operation. Thus, since we iterate through the map once O(N), and we call getFollows T times, we will have a total complexity of O(N+T).