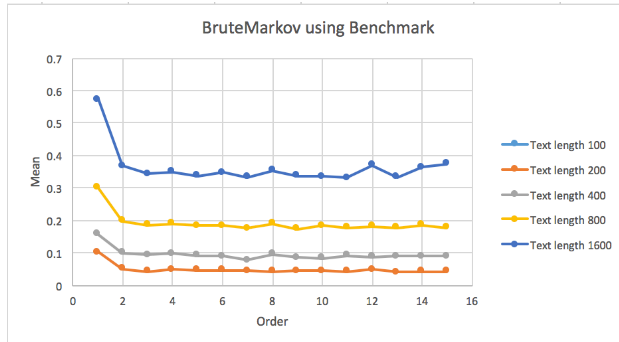
Summer Smith
CS 201
October 5, 2016

Markov Analysis

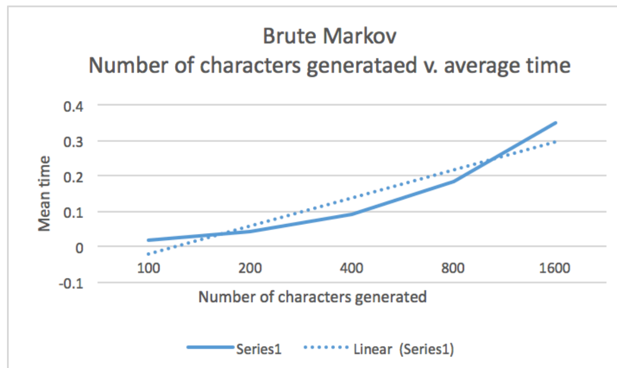
BruteMarkov Hypotheses

Partial data for first graph:

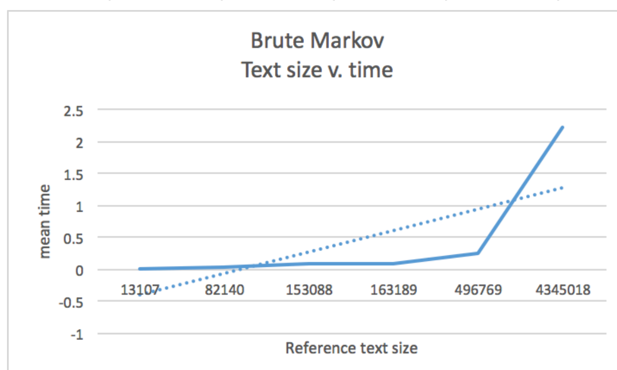
Order (k)	Mean time
2	0.027346
3	0.020172
4	0.021455
5	0.019161
6	0.02177
7	0.020933



Characters generated (T)	Mean time
100	0.021003929
200	0.044803786
400	0.090039357
800	0.183286929
1600	0.348610214



Source Text size (N)	Mean time
13107	0.006586
82140	0.04002
153088	0.08339
163189	0.079156
496769	0.248852
4345018	2.214265



In the getFollows() method in BruteMarkov, the program scans through the entire text one character at a time, looking for any occurrences of the key. The length of the entire text is represented by N. As we can see in the third graph above, the runtime has a linear relationship with N, meaning that the getFollows() method has a run time of O(N) because it takes longer to look through a longer text when you go one letter at a time. While the graph does not perfectly overlap with the linear line of best fit, you can tell by the values in the table that they are close to being linear; the discrepancies in time could be due in part to the activity of the garbage collector or other programs running on the computer at the same time.

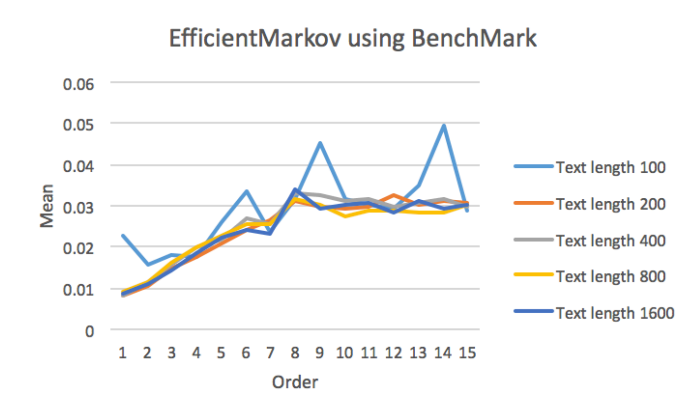
In the getRandomText() method in BruteMarkov, it generates t random characters. As seen on the second graph above, the relationship between the characters generated and the mean time is linear, due to the for loop in the method. If double the characters are produced, it doubles the time to produce them, making the run time for this method O(T).

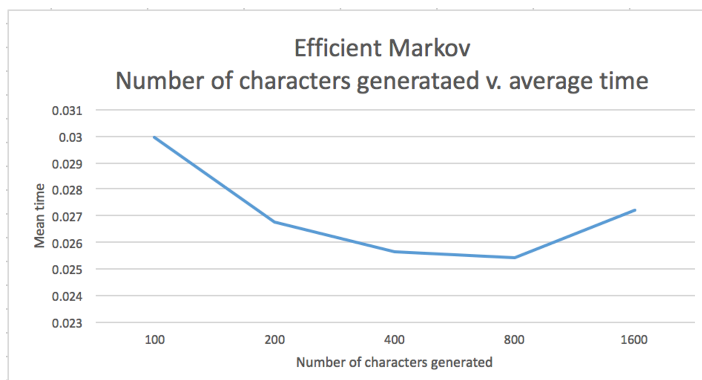
The first graph shows that the order (length of the key) does not affect the run time of the class. Each line on the graph represents the same amount of generated text. The x-axis shows the order and they y-axis shows the mean time. No matter what the order of a test that generates the same number of characters, the run time is the same, meaning it is O(1). Given the information above, it can be concluded that the total run-time of the BruteMarkov class is O(NT).

EfficientMarkov Hypotheses

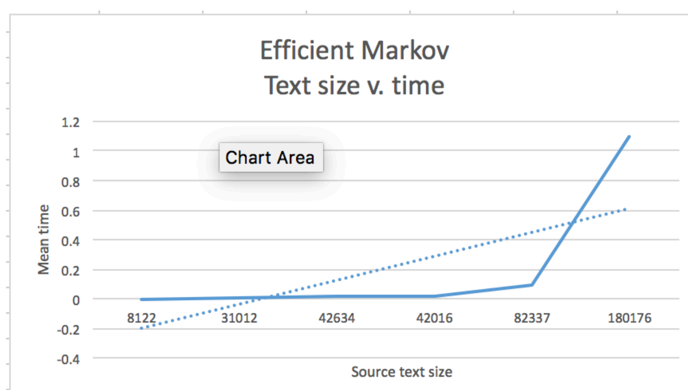
Partial data for graph 1:

Order (k)	Mean time
2	0.012207
3	0.016066
4	0.020124
5	0.024457
6	0.027354





Characters generated (T)	Mean time
100	0.029944571
200	0.026744929
400	0.025668143
800	0.025411
1600	0.0272265



Source text size (N)	Mean time
8122	0.001428
31012	0.011583
42634	0.020872
42016	0.021896
82337	0.091253
180176	1.098967

The getFollows() method in EfficientMarkov runs at constant time O(1), independent of the source text size, N, because it is referencing a key in a map by calling myMap.get(), something that takes the same amount of time no matter how many keys the map contains. This is only true if we are ignoring the time it takes to create the map. The third graph shows the run time in respect to the size of the source text; however, this graph slopes up at the end due to the amount of time it takes for the map to be created when the source size is noticeably larger. The last point is an outlier, and without it the line would be close to constant.

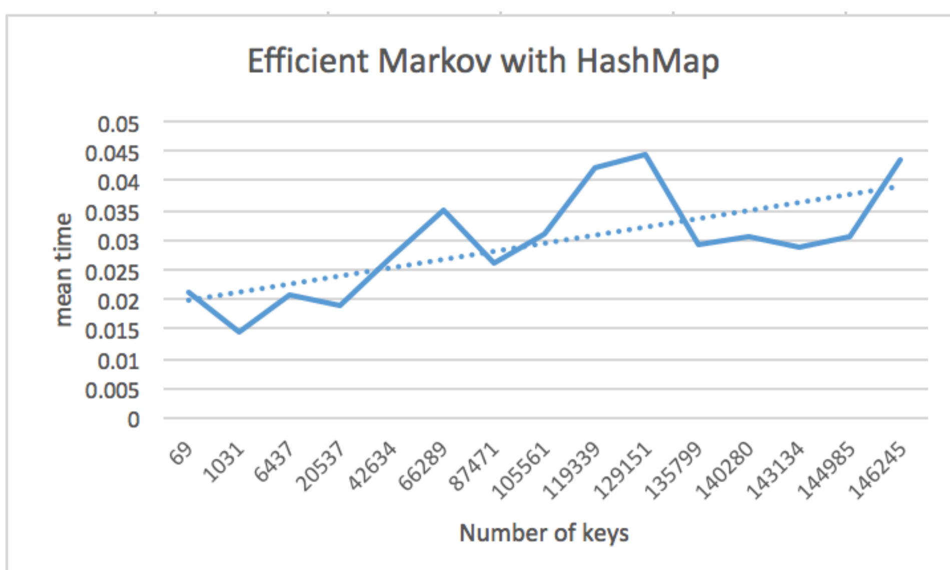
In order to generate more random characters, the method .getFollows() must be called more and more times. If the program is generating 200 characters, it must call .getFollows() 200 times, however, if the program is generating 400 characters, it should call .getFollows() 400 times. These additional calls change the runtime of the method, making the runtime O(T). However, my graph above does not show this. No matter how many characters are generated, the runtime only differs within a few hundredths of a second, it is not a linear relationship.

The runtime is independent of k, where k is the order of the Markov process. This is because the map is made the same way every time, no matter what the order is. Also the total length of the text – k is still equal to the total length of the text as k is negligible. This can be seen on the first graph above where the lines are flat, besides the first few points which are discrepancies. Given this information, the runtime of the EfficientMarkov class should be O(T) if the training time is ignored, or O(T+N) if the training time is factored in.

Map Time Hypotheses

Efficient Markov with HashMap

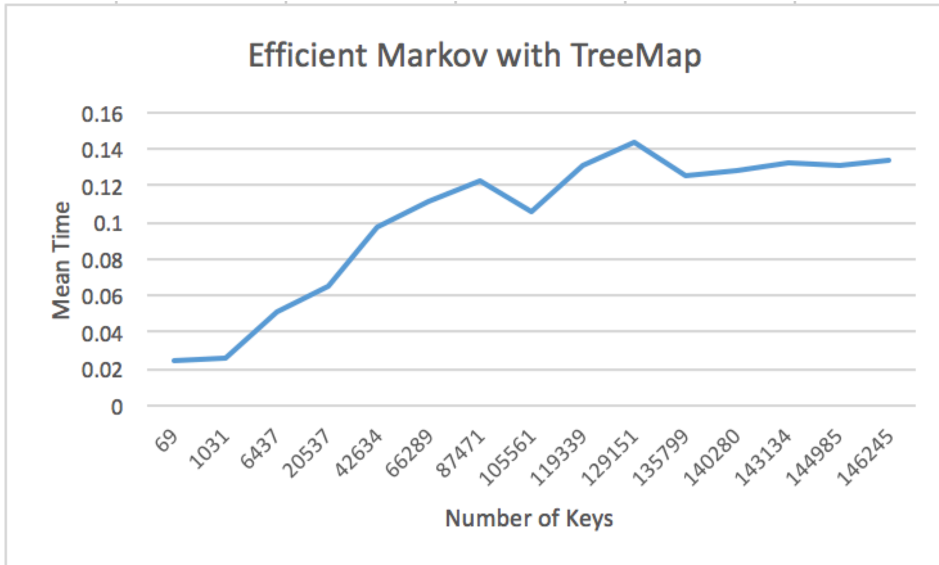
Keys	Mean time
69	0.021131
1031	0.014636
6437	0.020786
20537	0.018932
42634	0.026979
66289	0.034908
87471	0.026125
105561	0.031028
119339	0.042114
129151	0.044404
135799	0.029173
140280	0.030414
143134	0.028727



144985	0.030704
146245	0.043545

Efficient Markov with TreeMap

Keys	Mean time
69	0.024696
1031	0.025495
6437	0.051594
20537	0.064535
42634	0.096577
66289	0.1112
87471	0.121868
105561	0.105848
119339	0.131586
129151	0.142887
135799	0.124725
140280	0.127814
143134	0.132232
144985	0.130614
146245	0.133865



The MapTime hypothesis states that the number of unique keys affects the runtime of the program, stating that a HashMap should have a runtime of O(U), where U is the number of unique keys, and TreeMap should have a runtime of O(U log U). This difference is because TreeMap is a sorted Map, whereas HashMap is not. My graph for HashMap runtime is not linear, but the line of best fit shows that it could be if there is noise in the data. My map of EfficientMarkov shows a trend that resembles O(U log U), with number of unique keys on the x-axis and mean time on the y-axis. These graphs prove that the Map Time hypothesis is accurate regarding EfficientMarkov.

In order to get these graphs, I added a print statement in the setTraining() method in the Efficient Markov class. This statement prints out the number of keys in each map when the benchmark program is run. I also changed the declaration of the map to make it a TreeMap.