Introduction

This introduction will present the general summary of my project. I decided to do distance 0, 1 and 2. This is how I defined distance:

- Two sentences are **distance 0** if they are identical.
- Two sentences α and β are **distance 1** if len(α) = 1+ len(β) and *beta* is an ordered substring of α.
- Two sentences α and β are **distance 2** if len(α) = 2+ len(β) and β is an ordered substring of α, if len(α) = len(β) and β remove a word is an ordered substring of α, or if α and β are identical.

I used Python ver. 2.7.5 to write the code for each distance. The code was ran on a Red Hat 64-bit operating system with 5 GB of RAM running as a virtual machine under a Windows 10 64-Bit Core i7-4700MQ CPU at 2.40 GHz with a SSD. I was able to achieve $O(n)$ time for distance 1, while distance 0 and distance 2 are close to being linear.

Algorithm

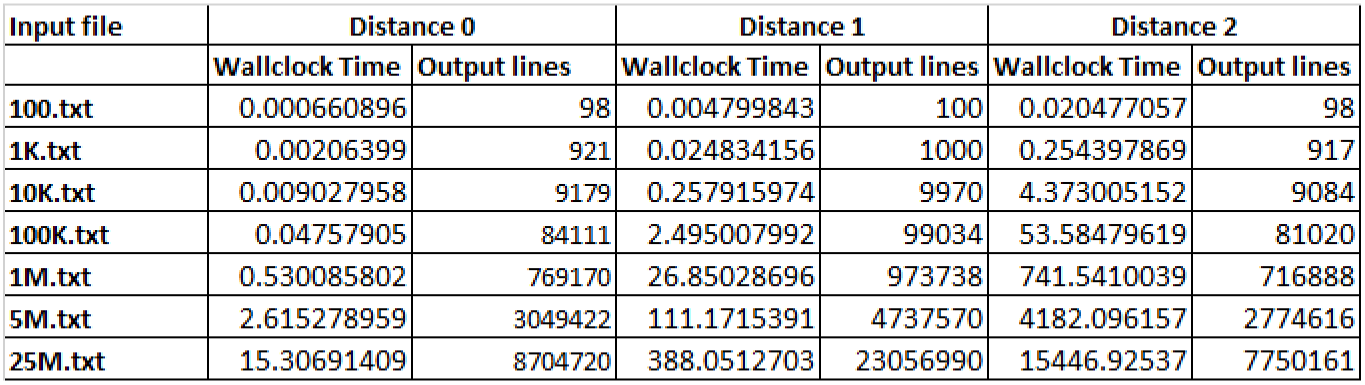
For distance 0, I simply read the lines (sentences) for each file one by one and saved them into a set using the *set*() function in Python. This function uses a hash table to get rid of duplicate sentences.

For distance 1, I read in the lines for each file but this time I stored them in a list. Then I used the *Counter*() function from the *Collections* library to count how many times was each sentence repeated. Then I created a default dictionary (storing values in a set) with keys being all the different lengths for the sentences and the values being the associated sentences of the appropriate lengths. Then, starting with the largest key (length of the largest sentence), for each sentence, if the lower key was length one less, I found all possible combinations (using the *combinations* function from *itertools*) of ordered substrings after removing one word and used the *set*().*remove*() function to remove those substrings from the previous key (words of length one less.) Now, since in my definition of distance 1 I don't include distance 0, when it was time to write the sentences to a file, I used the previously defined counter for each sentence to print the remaining sentences as many times as they were duplicated in the original text file.

For distance 2, I used the same default dictionary/sentence length key to create key value pairs. And as before, if keys were distance two apart (if the lengths were distance 2 apart), starting from the lartgest key (sentence length), for each sentence, I found all possible ordered substrings and deleted them from the appropriate distance 2 apart key. Since I am using a default dictionary which reads values into a set, this already took care of identical sentences. Now, memory issues starting to arise for finding sentences of the same length which when removed a word the resulting ordered string is a substring of the other word. Unfortunately, Python isn't as good as *C* for memory allocation. In my first attempted code (which I will include commented out in the code for distance 2), for each key (sentence length), I created a new dictionary (call it d2) were one by one, the keys were all possible ordered substrings of length one less. Just as the *set*() function, for each following word, if an ordered substring of one less length for the word was in the set of keys, I discarded the word from the original dictionary. If no substring was in the set of keys, I added the substrings as new keys. This approach, unfortunately, led to high memory usage and also a little over counting of words. After talking with Xiu Khun to see how we could solve the memory usage issue, he showed me his tuple approach which was very similar to my original approach in that it compared substrings but using tuples instead. He also showed me where I could clear my variables to make sure the memory allocation wasn't an issue. I included this portion of his code final submission. Note that the importance of using sets, tuples, and dictionary keys, is that this objects are **hashable**, meaning that comparing sentences is more efficient than comparing word by word.

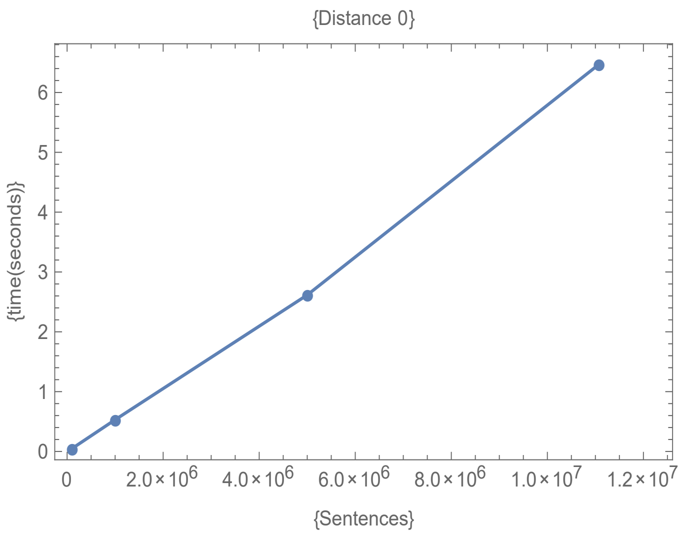
Results

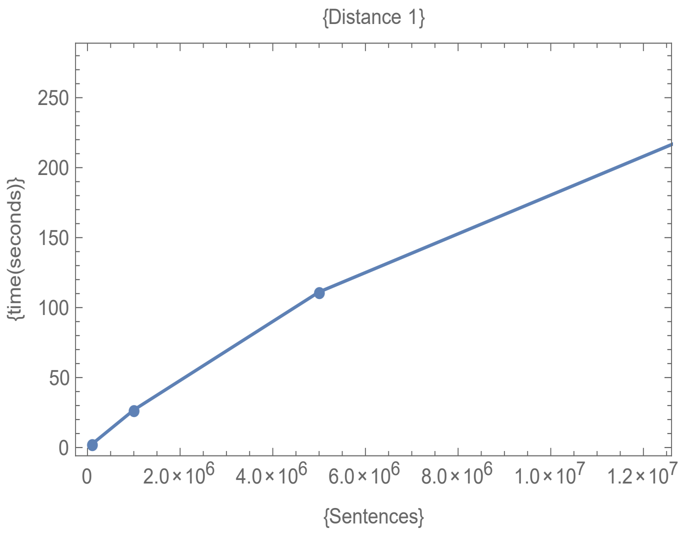
As previously mentioned, linear time was achieved for distance 1 while almost linear times were achieved for distance 0 and 2. I believe this is due to using the *combinations* function which meant I needed to check $\binom{k}{2}$ possible substrings for each sentence, where k is the length of the sentence when I wanted to compare sentences of length 2 apart. For distance 0, I am unsure why the slope was varying (although it was varying in the thousandth or millionth places). Finally, I noticed that for both distance 1 and distance 2, the slope decreased when comparing the 5M file with the 25M file. It seems the algorithm became more efficient for the 25M file. Below is a table summarizing my results for each text file of sentences:

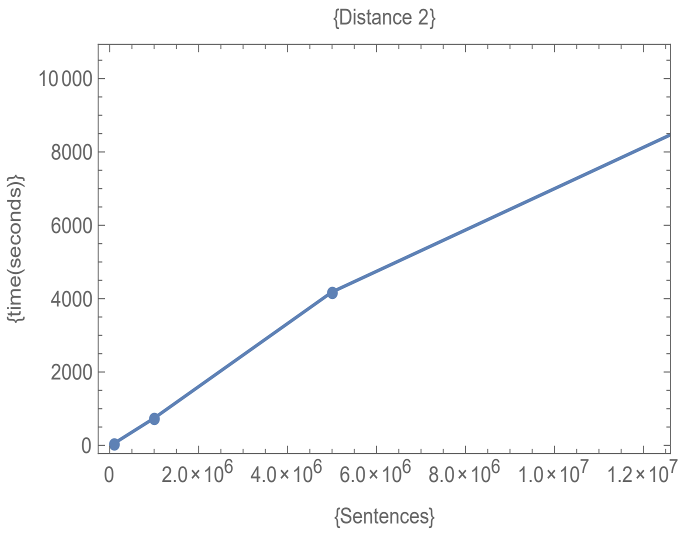


Input file	Distance 0		Distance 1		Distance 2	
	Wallclock Time	Output lines	Wallclock Time	Output lines	Wallclock Time	Output lines
100.txt	0.000660896	98	0.004799843	100	0.020477057	98
1K.txt	0.00206399	921	0.024834156	1000	0.254397869	917
10K.txt	0.009027958	9179	0.257915974	9970	4.373005152	9084
100K.txt	0.04757905	84111	2.495007992	99034	53.58479619	81020
1M.txt	0.530085802	769170	26.85028696	973738	741.5410039	716888
5M.txt	2.615278959	3049422	111.1715391	4737570	4182.096157	2774616
25M.txt	15.30691409	8704720	388.0512703	23056990	15446.92537	7750161

Additionally, I also graphed a few of the points for each distance so the slopes would be visualized easily:


{Distance 0}
{time(seconds)}
0
1
2
3
4
5
6
0
2.0×10^6
4.0×10^6
6.0×10^6
8.0×10^6
1.0×10^7
1.2×10^7
{Sentences}


{Distance 1}
{time(seconds)}
0
50
100
150
200
250
0
2.0×10^6
4.0×10^6
6.0×10^6
8.0×10^6
1.0×10^7
1.2×10^7
{Sentences}


{Distance 2}
{time(seconds)}
0
2000
4000
6000
8000
10 000
0
2.0×10^6
4.0×10^6
6.0×10^6
8.0×10^6
1.0×10^7
1.2×10^7
{Sentences}