Name: Keith Pham

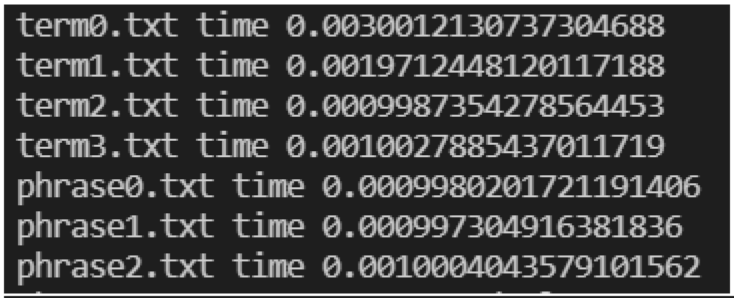
SID: 32507133

Project 3

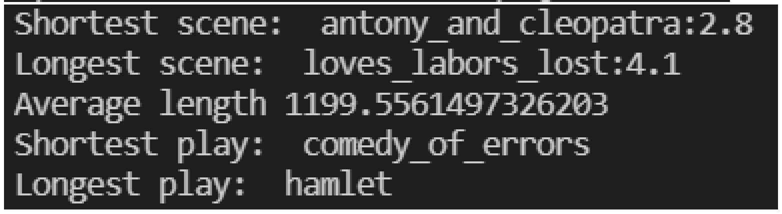
1. Indexing: line 25-85
   Evaluation: line 87-232
2. Indexing Data Structure:
   - InvertedIndex: Map<Term, PostingList>
   - PostingList: Map<docId, Position>
   - Position: int[ ]
   - docId: sceneId
   - PlayId: Map<docId, playId> - PlayId Map is used to get PlayId when having docId

   Design:
   - For comparing Frequency, the program call function ***findGreaterScene(term1, term2)***.
     - Each ***terms*** is a string[ ] which contains words to accumulate frequency. For **term0.txt**, term1 = ['thee', 'thou'], term2 = ['you']
     - Function returns Set<sceneId> that satisfies total frequency from term1 > total frequency from term2
     - If a sceneId exists in first term set, but does not exist in the second term set, still count, example: freq1 = 10, freq2 = 0. As a result, …
     - … this function can also be used to find sceneId that contains **terms** by simply leave **term2** empty because freq2 is 0 in this case
   - For finding playId,
     - Initially, I use ***docId.split(':')[0]*** to get playId. However, there is no set rule for playId and sceneId. As a result, I store a map between docId and playId.
     - Instead, I create a new variable ***playId***: Map<docId, playId>. Thus, retrieving playId is retrieving sceneId and convert it to playId through Map ***playId***.
   - For finding phrase, the program call ***PhraseBase Class***. This class constructor takes InvertedIndex. We can also use this Class to find Terms but only in exact input order.
     - First, the program will find all document that contains all terms in phrase by calling ***findSceneIds***
     - Then the program will perform ***Intersecting*** and ***MatchingWindow*** similar to pseudo in slides

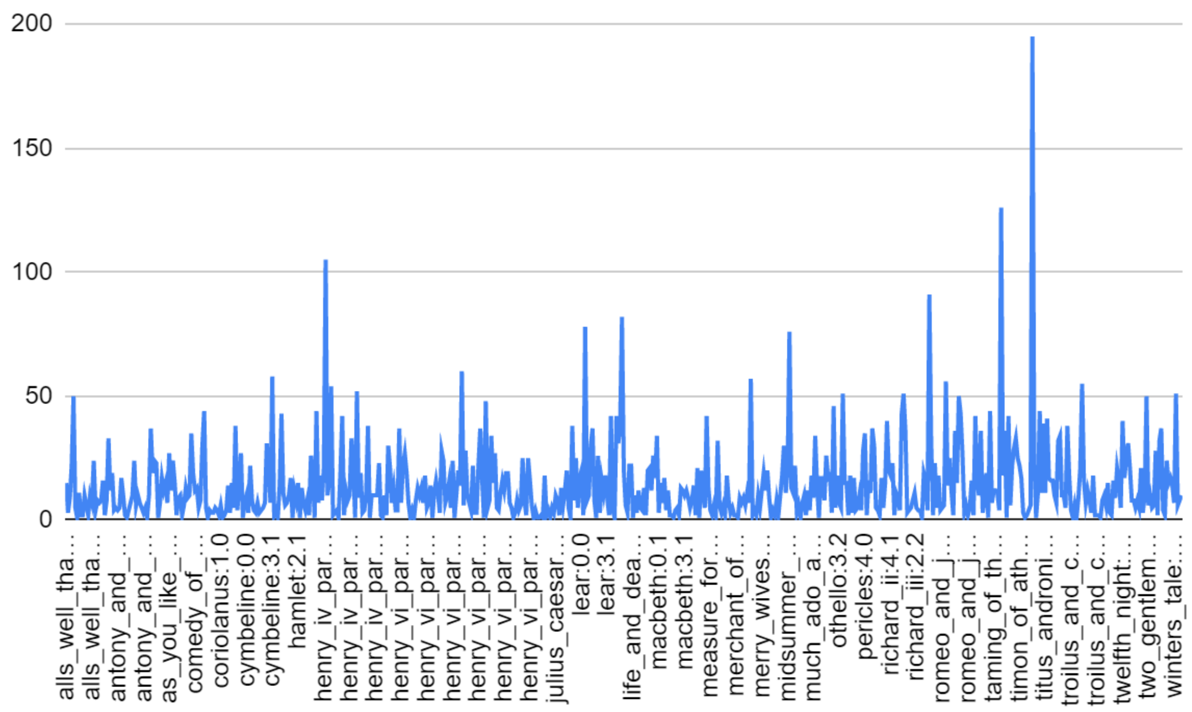
3. The program imports JSON library to read JSON file; time library to time the runtime; collections library to implement orderedDictionary. The rest of the code is handcrafted by me
4. While word count is a good indicator for scene relevance. However, there are several cases that word count might be misleading. For example, if our *Query* is "tropical fish", our *article 1* is: "this article mention fish, but the fish are not in tropical", and our *article 2* is: "1 tropical fish please". If we rank scene relevance by word count, then article 1 is ranked higher (3 "fish" and 1 "tropical") compared to article 2 (1 "fish" and 1 "tropical). One way to fix this issue is to store index of term and implement intersection to check article relevance
5. The Query that is the longest and contains common words takes the longest to execute. The explanation is that the intersection and matching window algorithms have to recursively go deeper to each term indexed position and compare distance of each term indexed position; As a result, run time is proportional to the length of Query and Query's terms commonality



```
term0.txt time 0.0030012130737304688
term1.txt time 0.0019712448120117188
term2.txt time 0.0009987354278564453
term3.txt time 0.0010027885437011719
phrase0.txt time 0.0009980201721191406
phrase1.txt time 0.0009973049163818360
phrase2.txt time 0.0010004043579101562
```

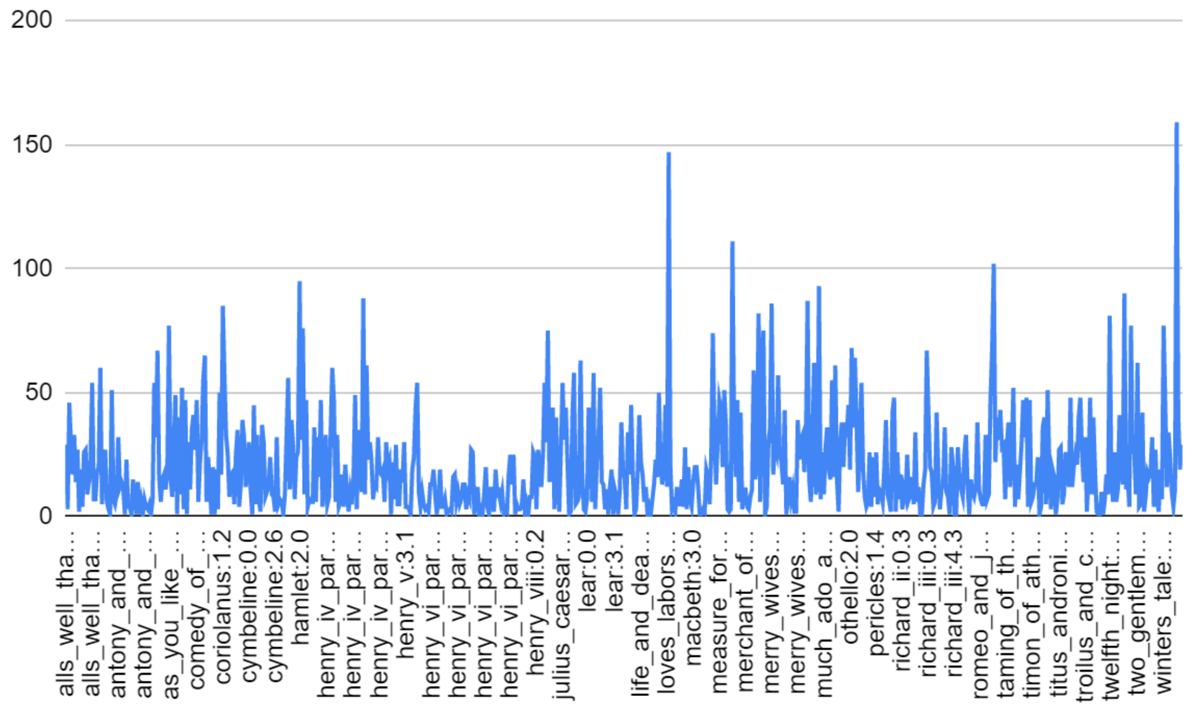
6. 

```
Shortest scene:  antony_and_cleopatra:2.8
Longest scene:  loves_labors_lost:4.1
Average length 1199.5561497326203
Shortest play:  comedy_of_errors
Longest play:  hamlet
```



7.

1. "thee" or "thou"



2. "You"