Lab 5 – Hash Tables with Chaining and Linear Probing

Jonatan M. Contreras

University of Texas at El Paso

**Introduction**

In lab five, Hash Tables with Chaining and Linear Probing were implemented, along Search and B Trees, to store close to 370,000 words and, using these data structures, calculate the similarities of the words given by their embeddings. The program asks the user to choose their data structure implementation (out of the four data structures) of choice in order to store the words and retrieve the similarities of the words desired.

**Proposed Solution Design and Implementation**

*Part 1: Hash Table with Chaining Implementation:*

To implement the Hash Table with Chaining solution, I began by creating the six hash functions described by the lab handout. My hash function of choice takes the ascii value of the first character of the word being hashed and multiples it times the prime number 3. Then, I modified the starter code of the Hash Table with Chaining so it can work with WordEmbedding objects. This meant modifying the Insert() and Find() functions so that the user can choose the hash function used within the Insert() and Find() functions. Then, the program simply reads the glove.6B.50d.txt file, line by line. If the first character of the line that was read is not alphabetical, it moves onto the next line. Else, it creates a WordEmbedding object by passing the first item in the list created by reading the line as the word, and the rest of the list as the embedding attribute. Once this is done, it uses the Insert() function to insert the WordEmbedding object into a Hash Table with Chaining. The program does this until every line has been read, and thus, every word in the .txt file has been stored as a WordEmbedding object in the Hash Table.

Once the words are stored in the Hash Table, a .txt file containing pairs of words is read line by line. As the pairs of words are read, it creates a list out of the words on the line. The function then searches for the words in the Hash Table and retrieves the WordEmbedding object that contains that word. The method then calculates the similarities of the two words. It stores these similarities in a list and continues to the next line. Once it is done storing every similarity, it returns the list of similarities.

*Part 2: Hash Table with Linear Probing Implementation:*

To implement the Hash Table with Linear Probing, I also created the six hash functions required by the lab handout in the Hash Table with Linear Probing starter code. I also modified the starter code so it can work with WordEmbedding objects. The program then reads the glove.6B.50d.txt file, line by line, creates a WordEmbedding object by passing the first item in the list created by reading the line as the word, and the rest of the list as the embedding attribute, stores it in the Hash Table with Linear Probing. The program does this until every line has been read, and thus, every word in the .txt file has been stored as a WordEmbedding object in the Hash Table with Linear Probing.

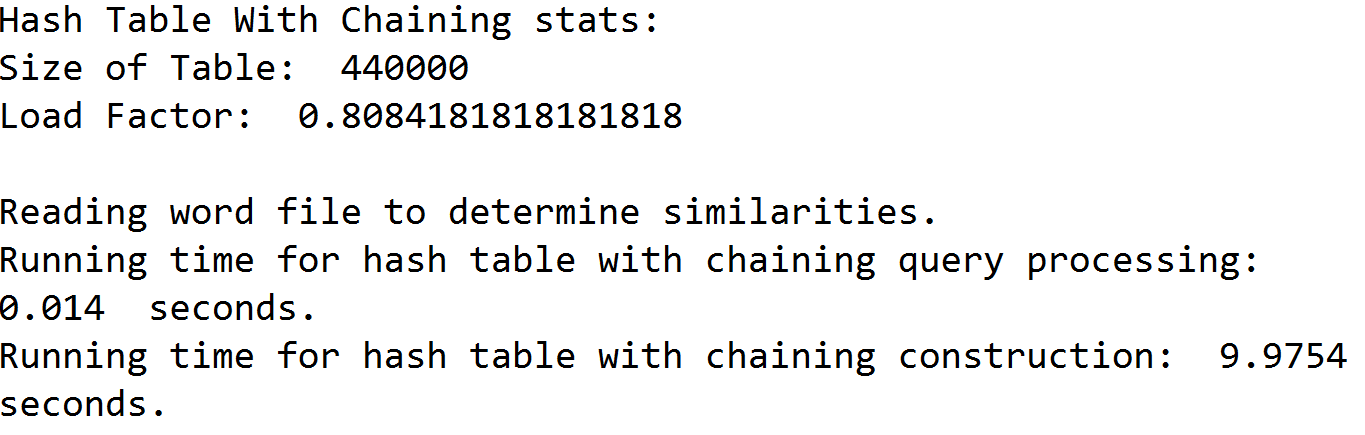
Once the words are stored in the Hash Table, pairs of words are read, and a list is created out of the words on the line. The function then searches for the words in the hash table and calculates the similarities of the two words. It stores these similarities in a list and continues to the next line. Once it is done storing every similarity, it returns the list of similarities.

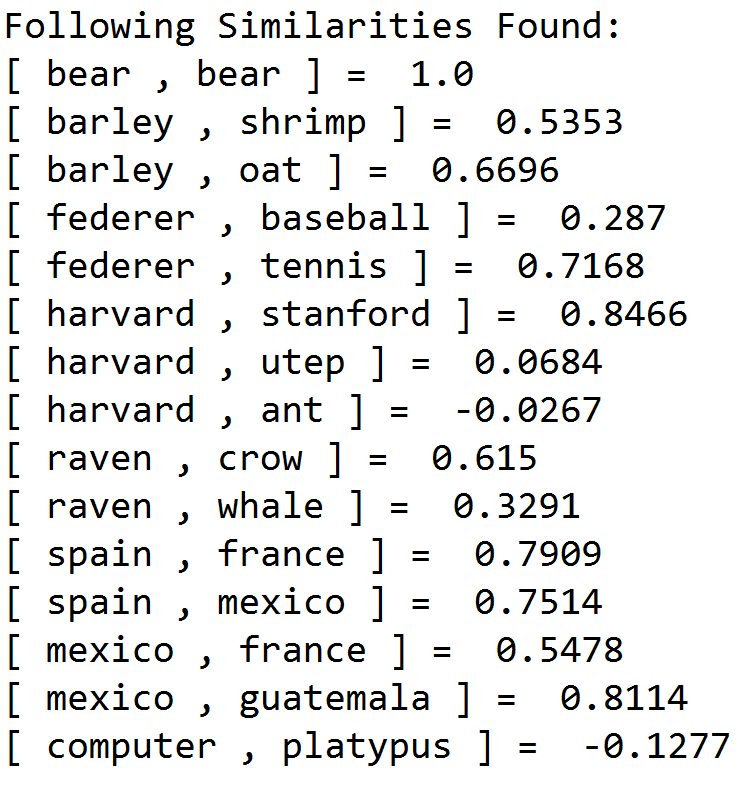
**Experimental Results**

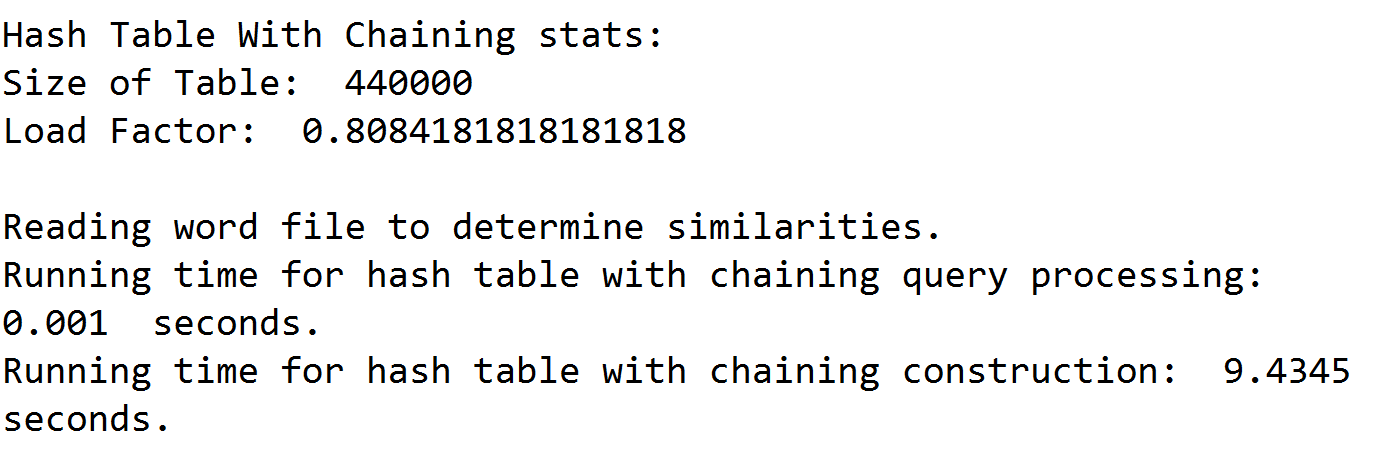
*Part 1: Hash Table with Chaining Implementation:*

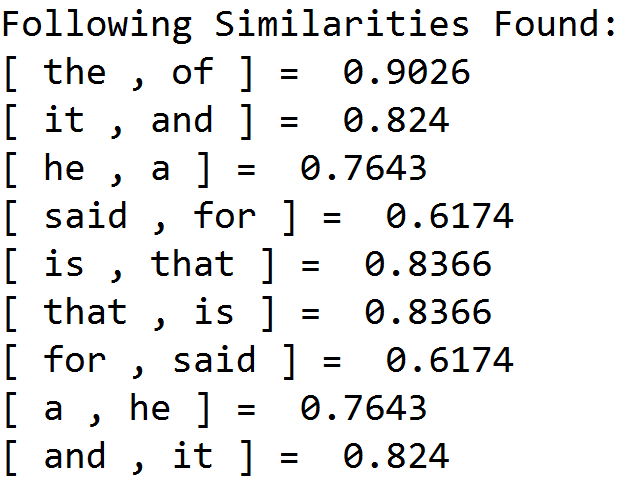
The following are screenshots of the output at varying sizes of compared words. Note: Hash function used for all running times is the recursive function because it was the only hash function that returned acceptable times for construction and query processing.

**Similarities for Example in Lab Handout:**

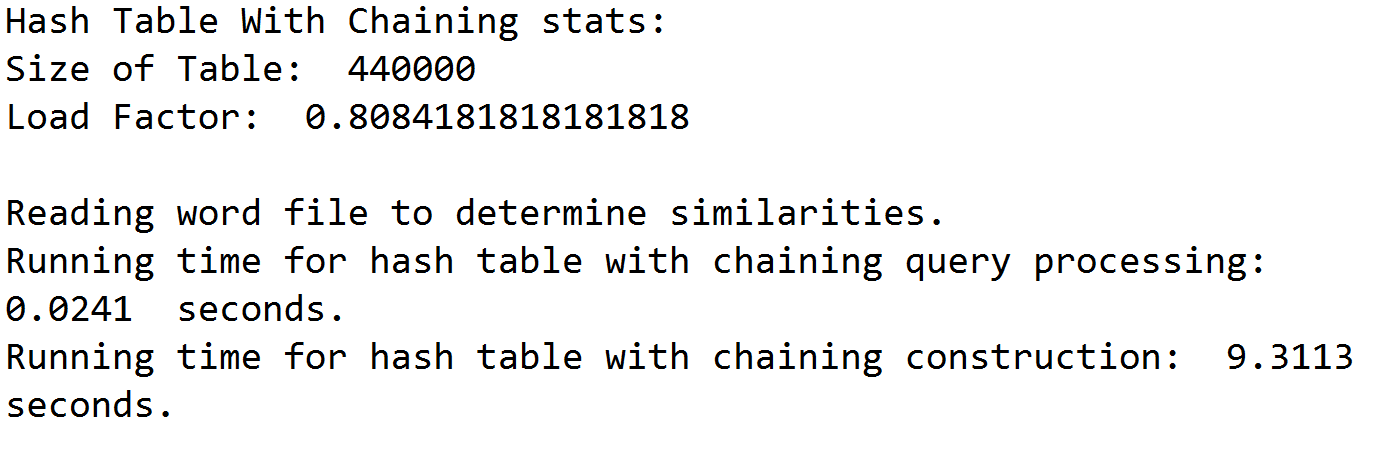
**

****

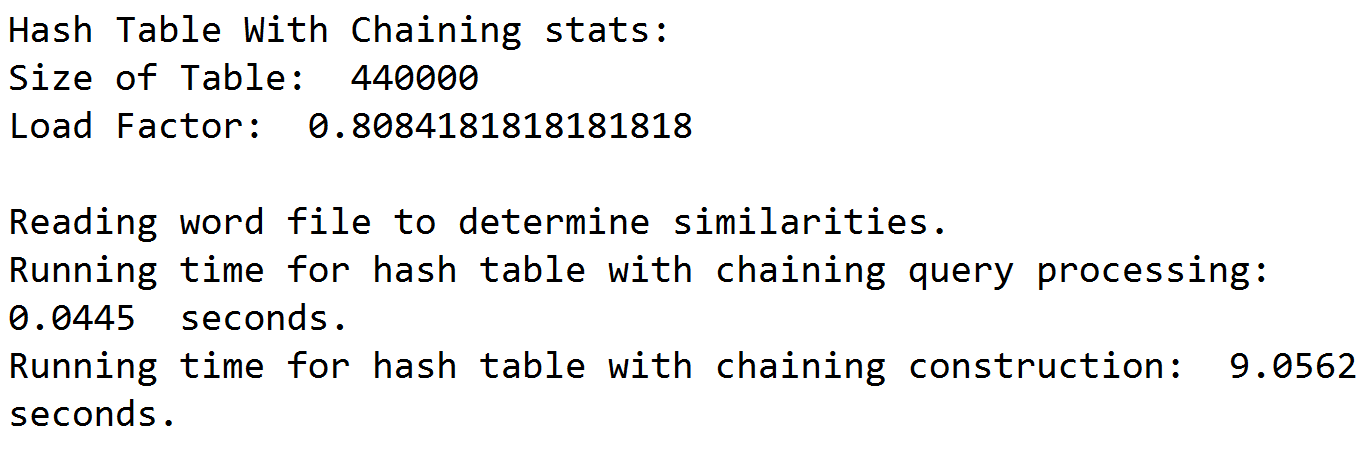
**List Size = 10 [Similarities only printed for this list size]:**  


****

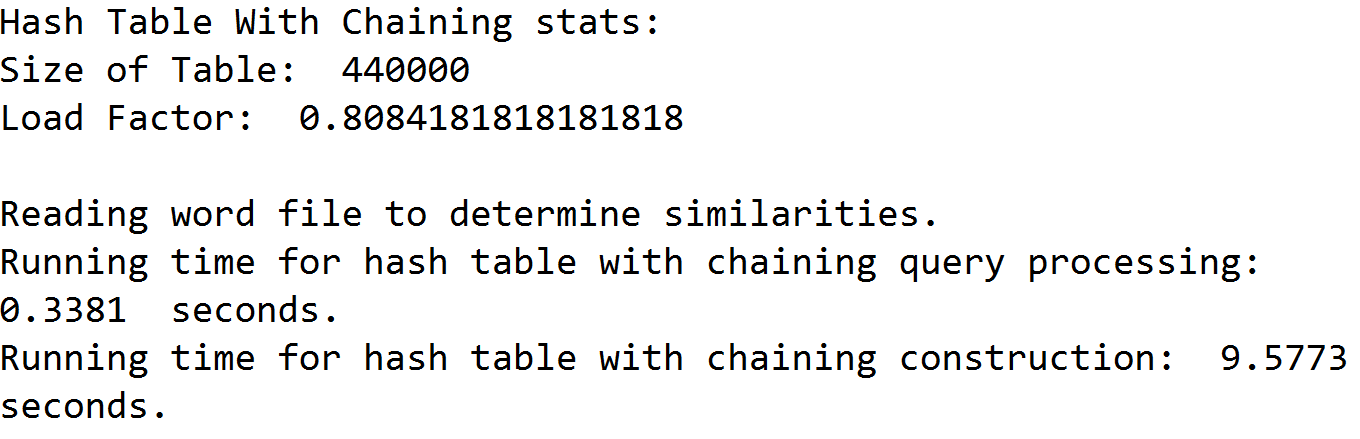
**List Size = 100:**



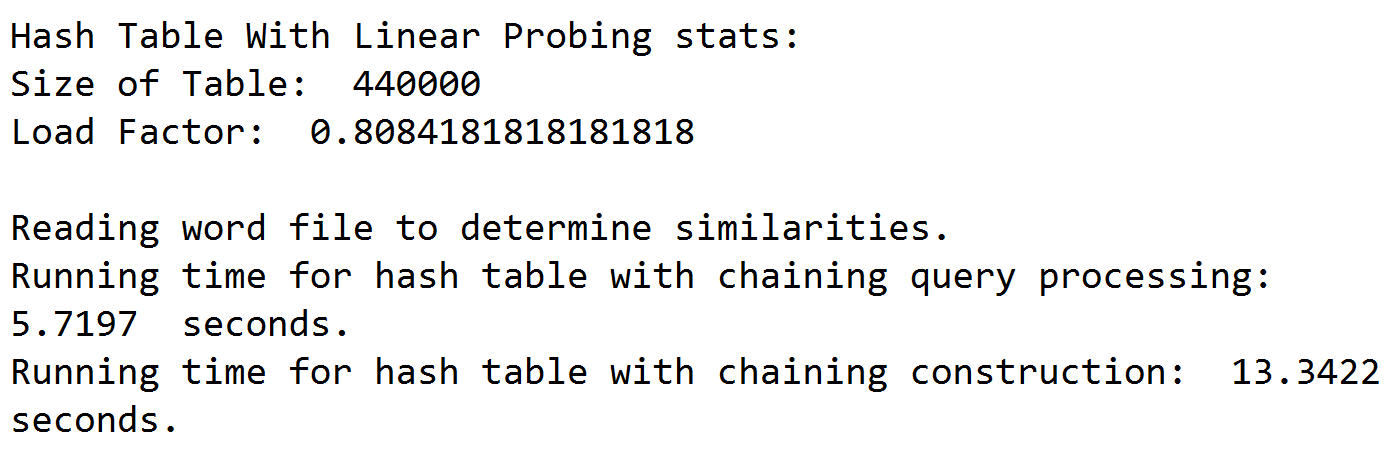
**List Size = 1,000:**

****

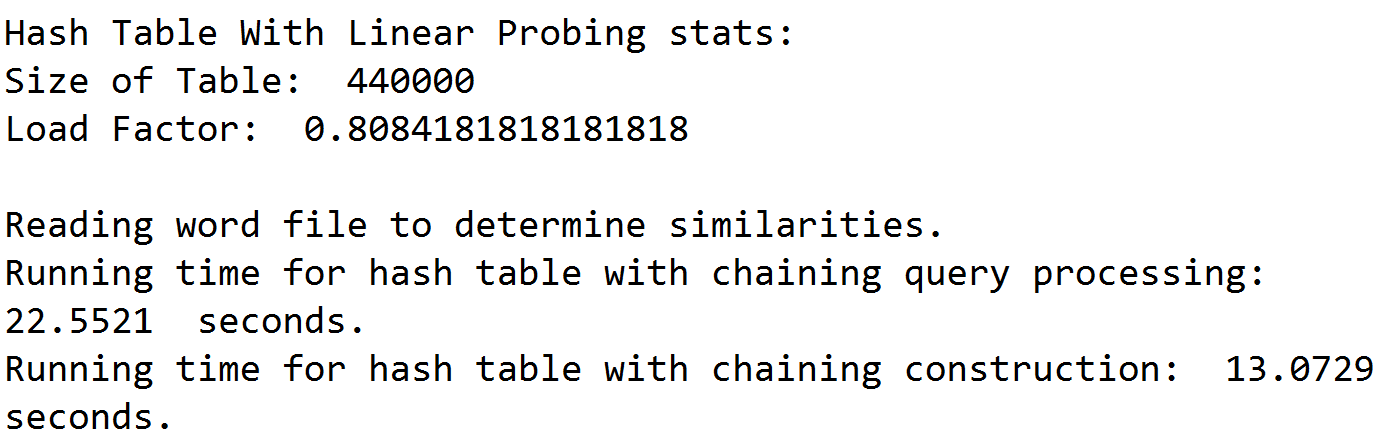
**List Size = 10,000:**

****

**List Size = 100,000:**



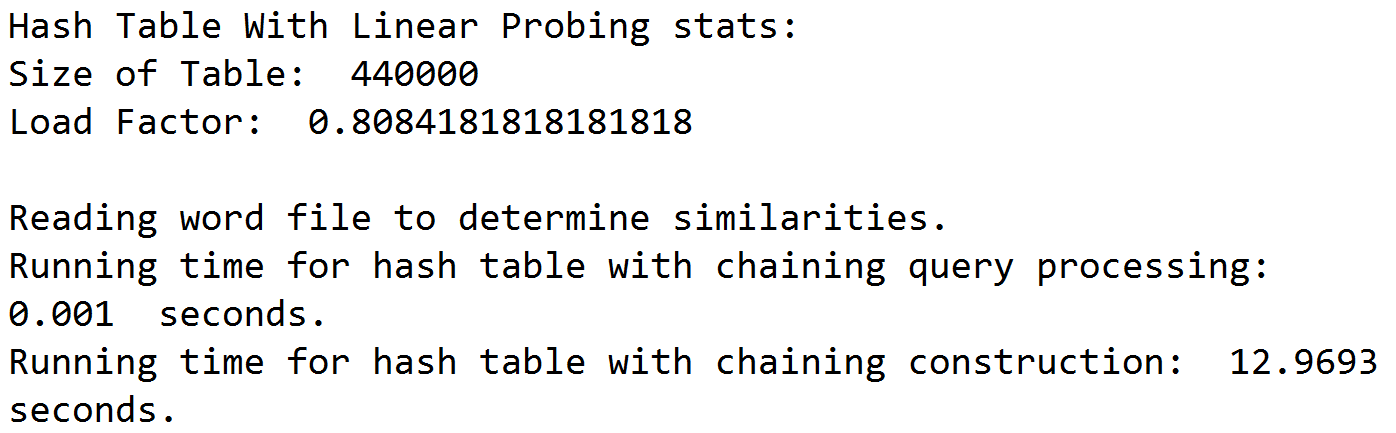
**List Size = (Close to 355,000):**

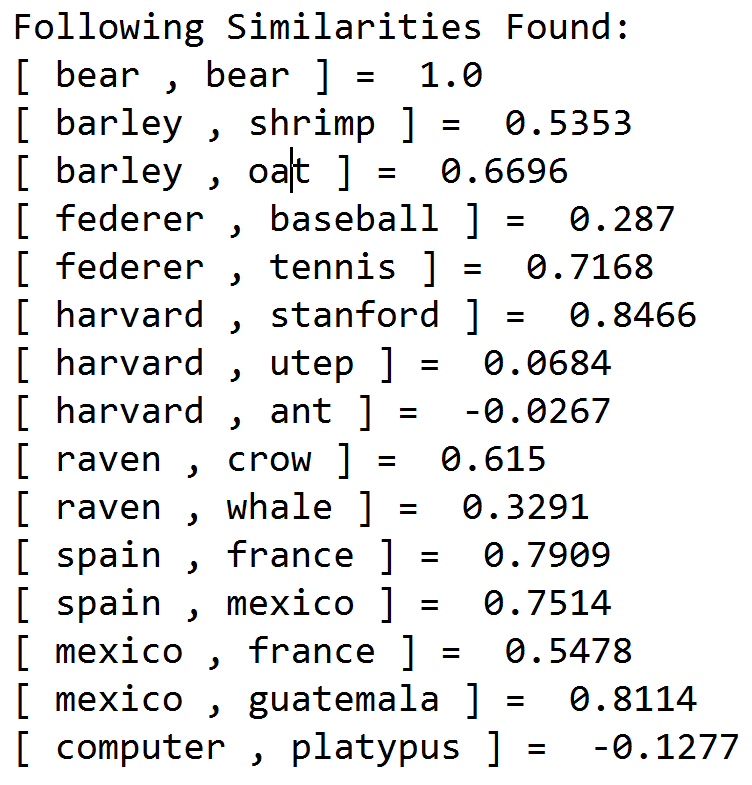
**

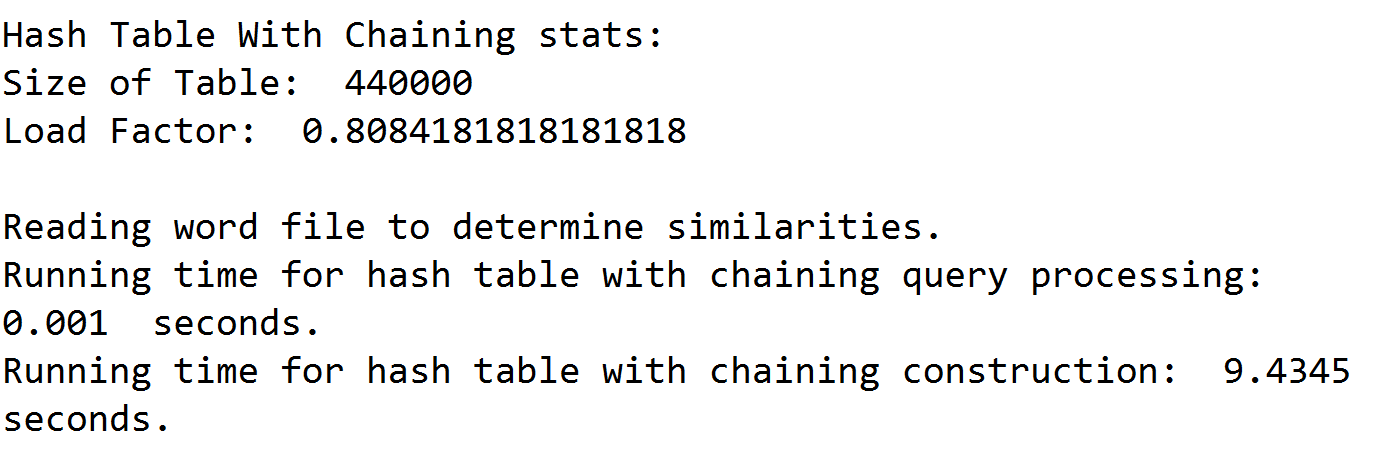
*Part 2: Hash Table with Linear Probing Implementation:*

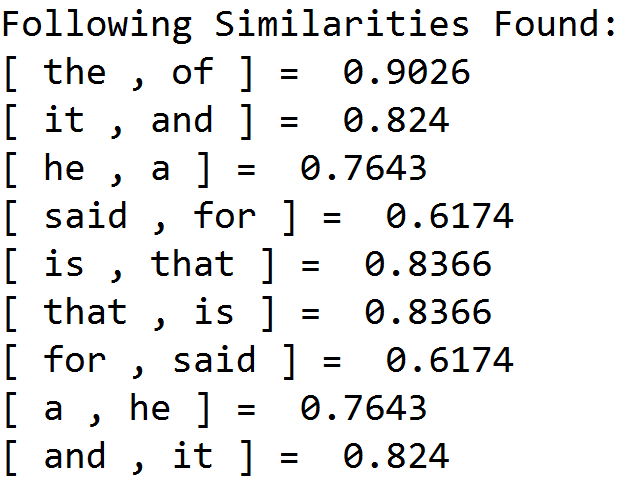
The following are screenshots of the output at varying sizes of compared words:

**Similarities for Example in Lab Handout:**

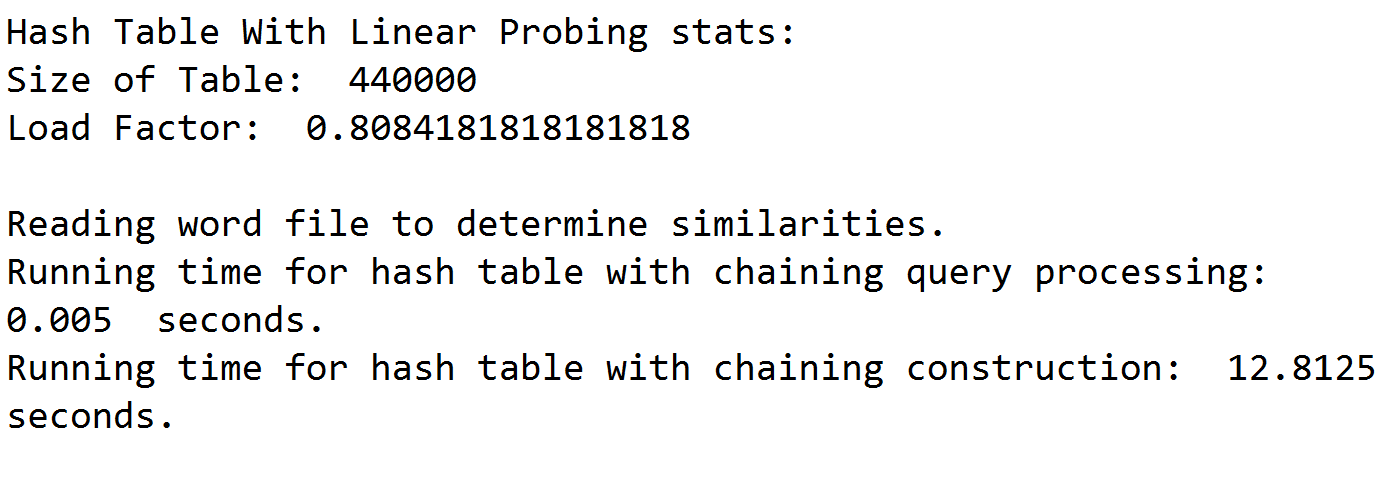
****

****

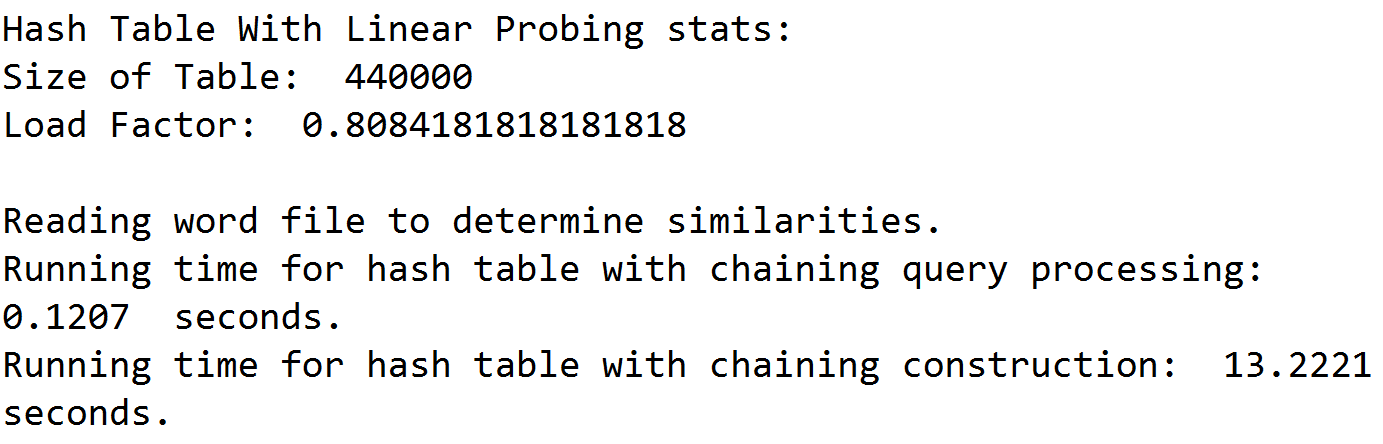
**List Size = 10 [Similarities only printed for this list size]:**  
****



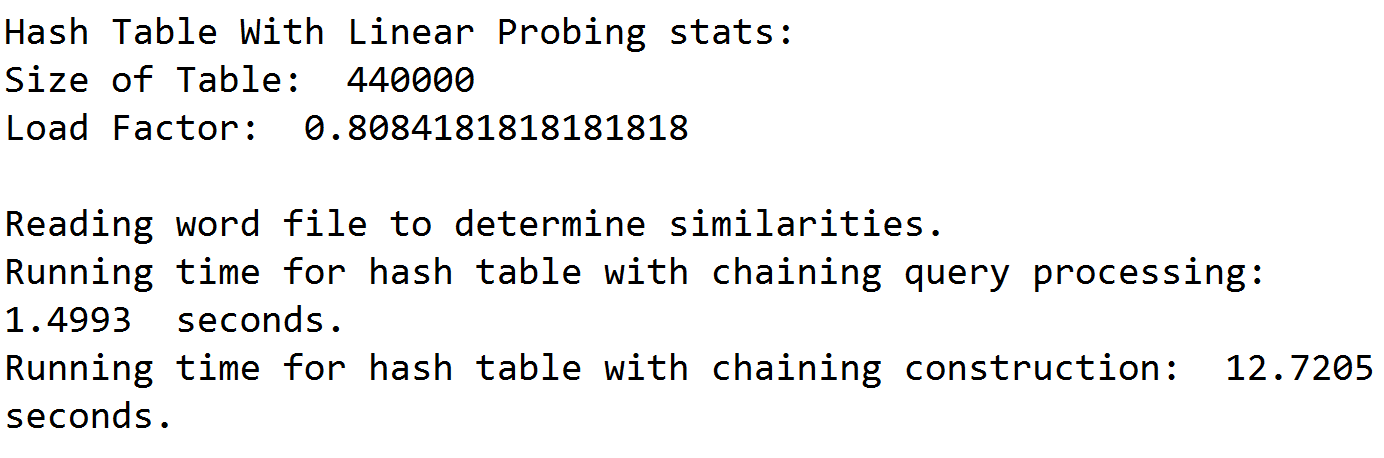
**List Size = 100:**



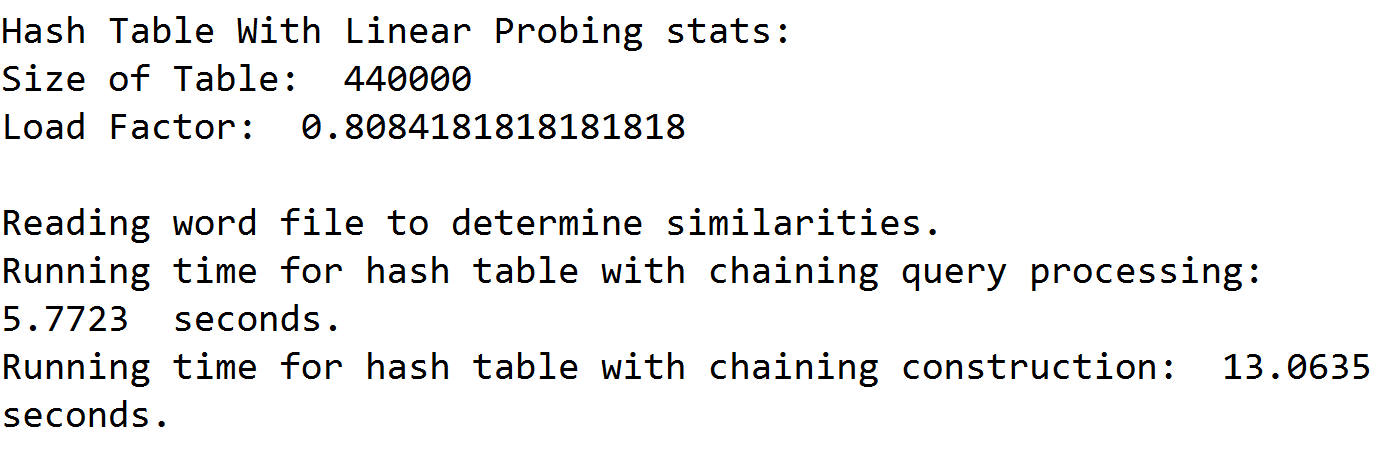
**List Size = 1,000:**

****

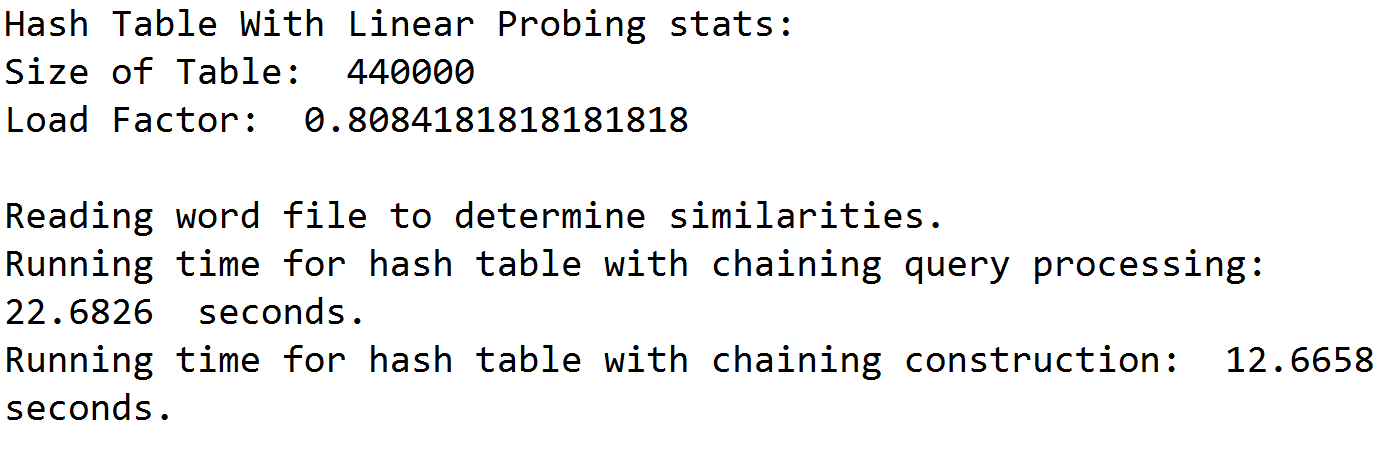
**List Size = 10,000:**

****

**List Size = 100,000:**

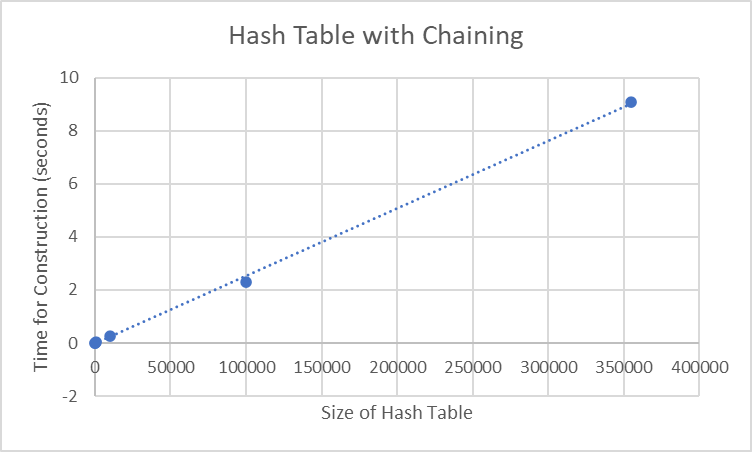


**List Size = (Close to 355,000):**

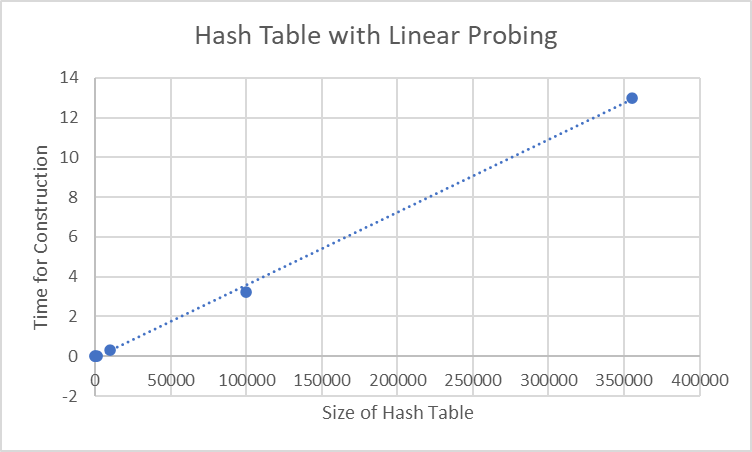


*Part 3: Construction Times for Both Hash Tables:*

*Hash Table with Chaining Implementation:*

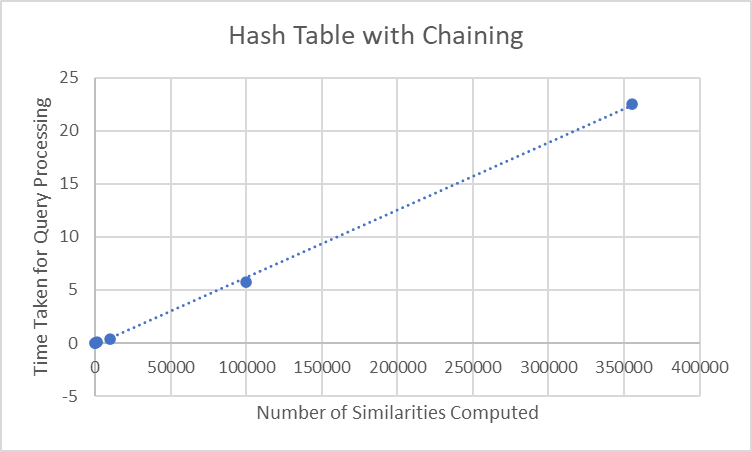


*Hash Table with Linear Probing Implementation*

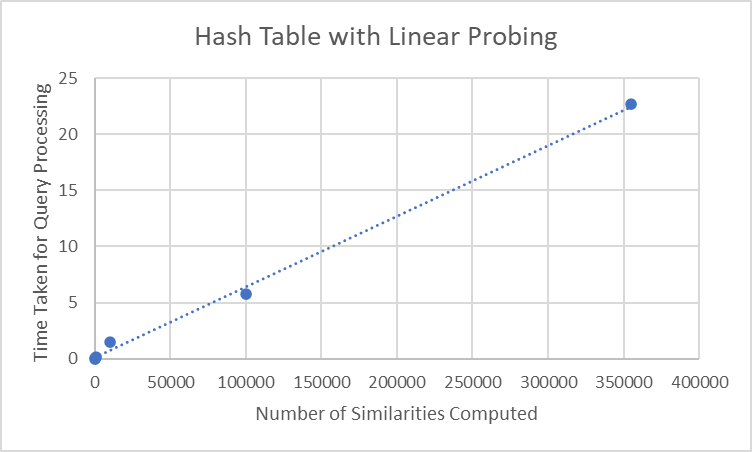


*Part 4: Query Processing Running Times:*

*Hash Table with Chaining Implementation:*



*Hash Table with Linear Probing Implementation:*

**

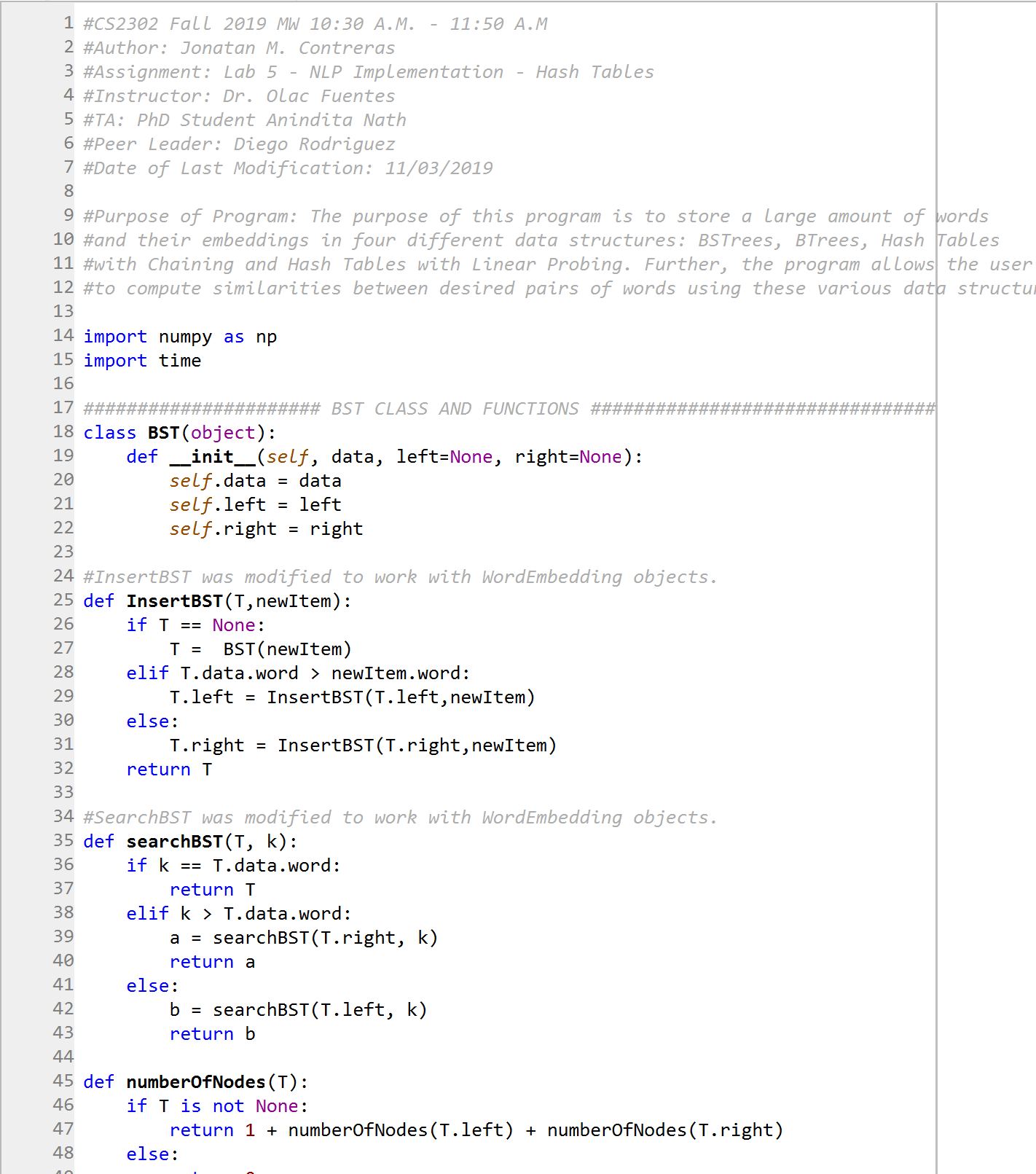
**Efficiency of Methods**

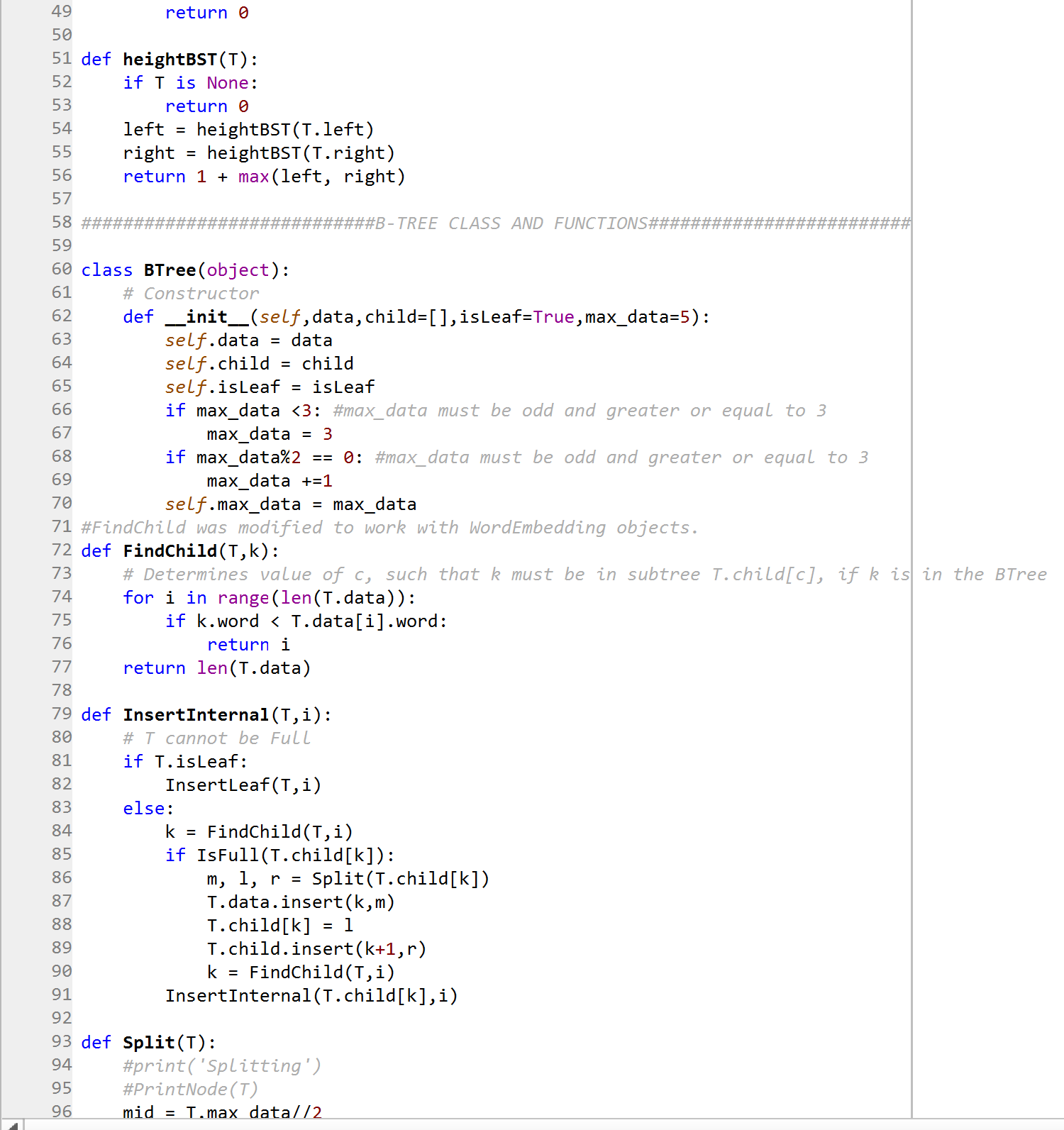
All but one hash function were inefficient for both Hash Tables. This is because ascii values are going to be pretty similar since I am only focusing on alphanumerical characters so there will be a lot of collisions. The recursive hash function worked well and outperformed all both the BSTrees and Btrees.

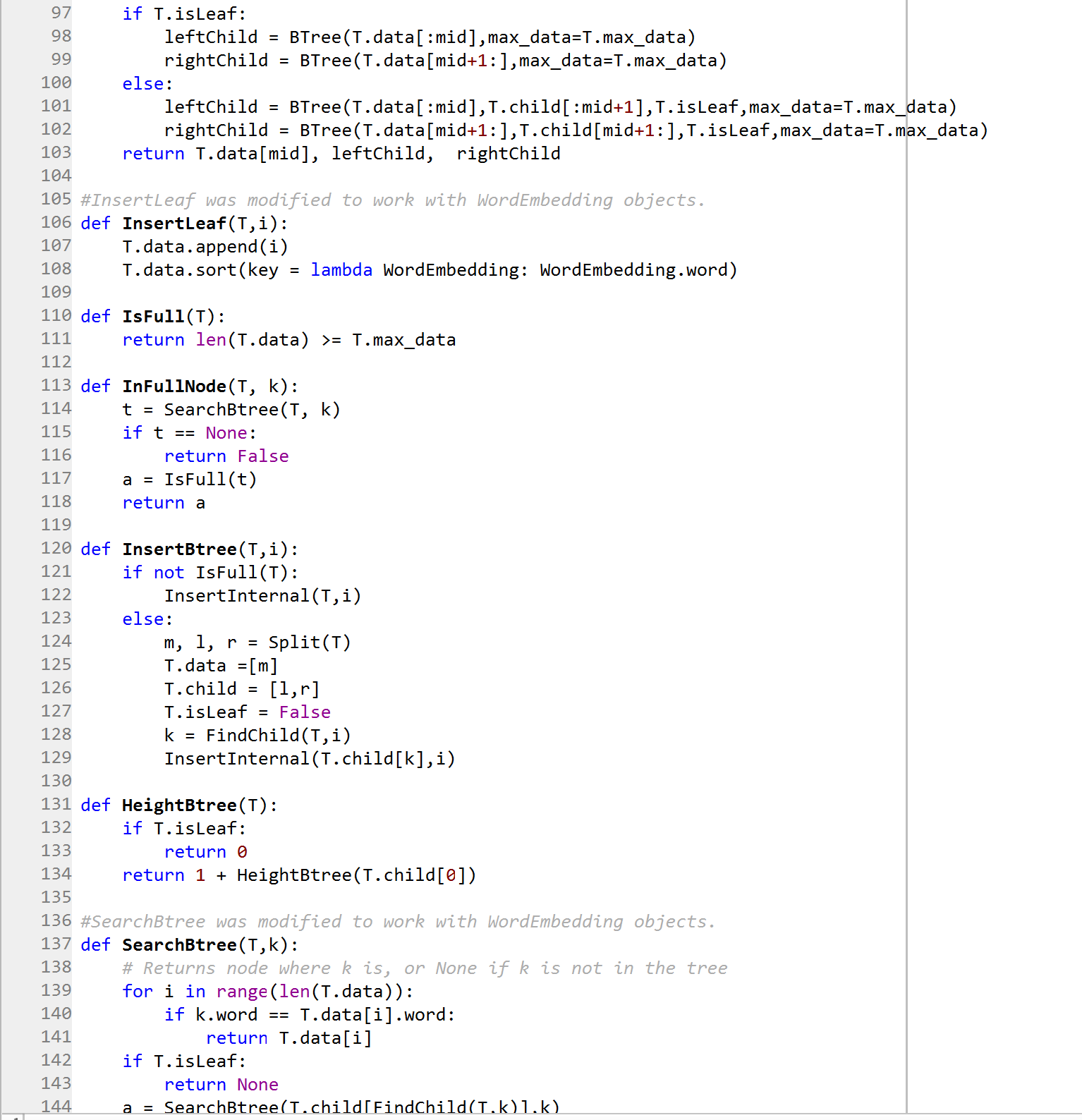
**Conclusion**

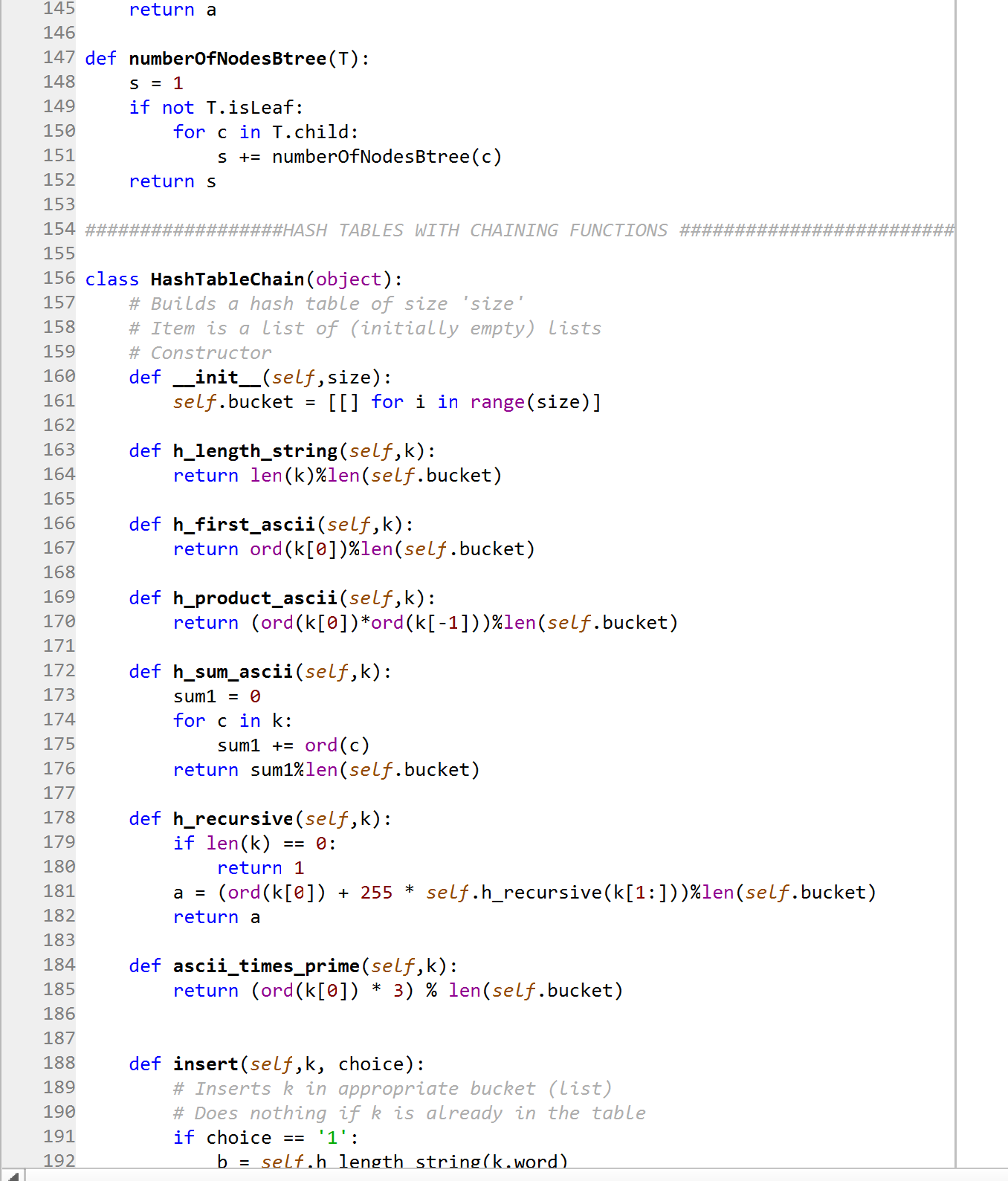
I learned how to implement Hash Tables with chaining and linear probing with different data types. I also learned the importance of a good hash function. Finally, I also learned about the differences and similarities between BSTrees/Btrees and Hash Tables with Linear Probing and Chaining.

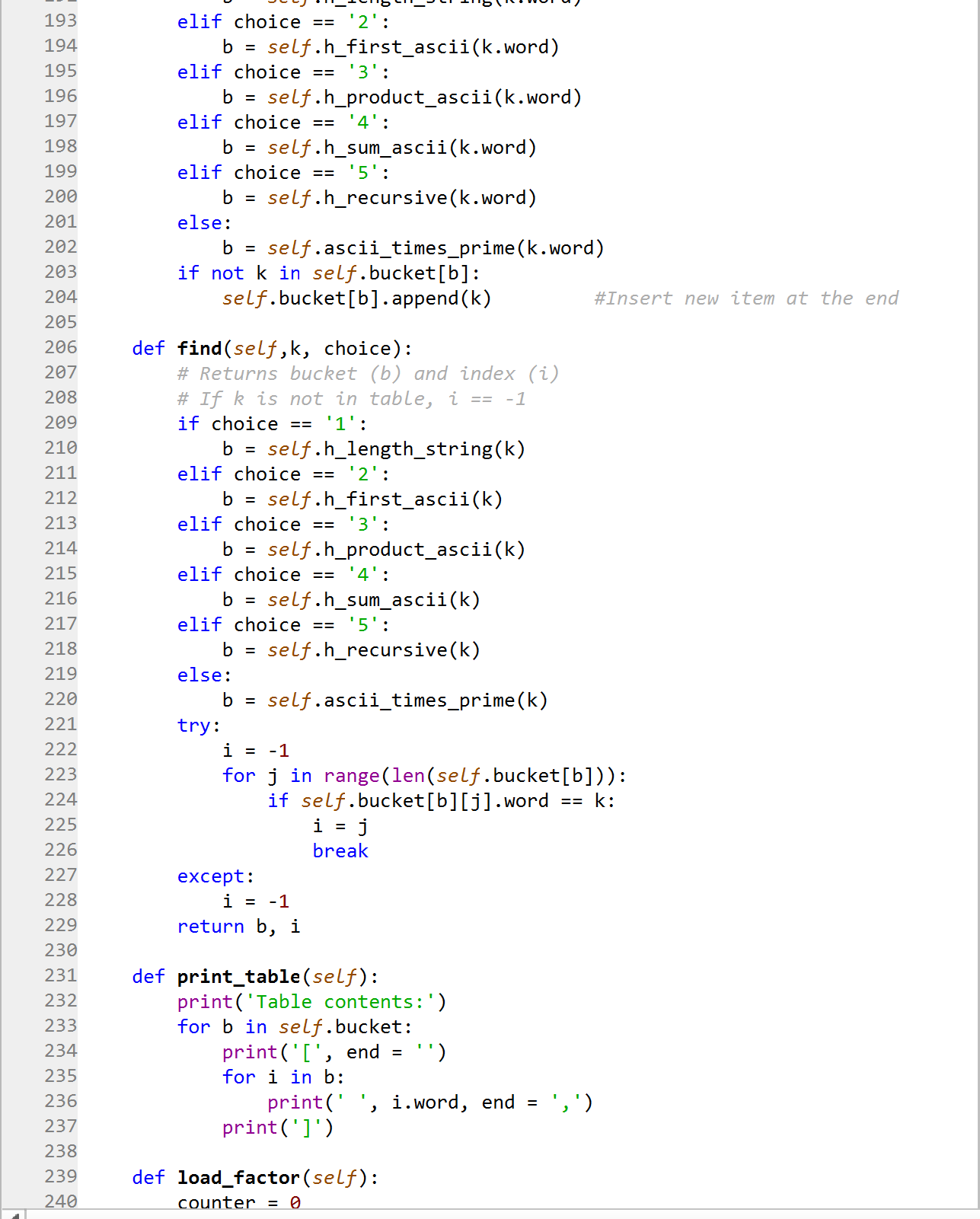
**Appendix – Source Code**

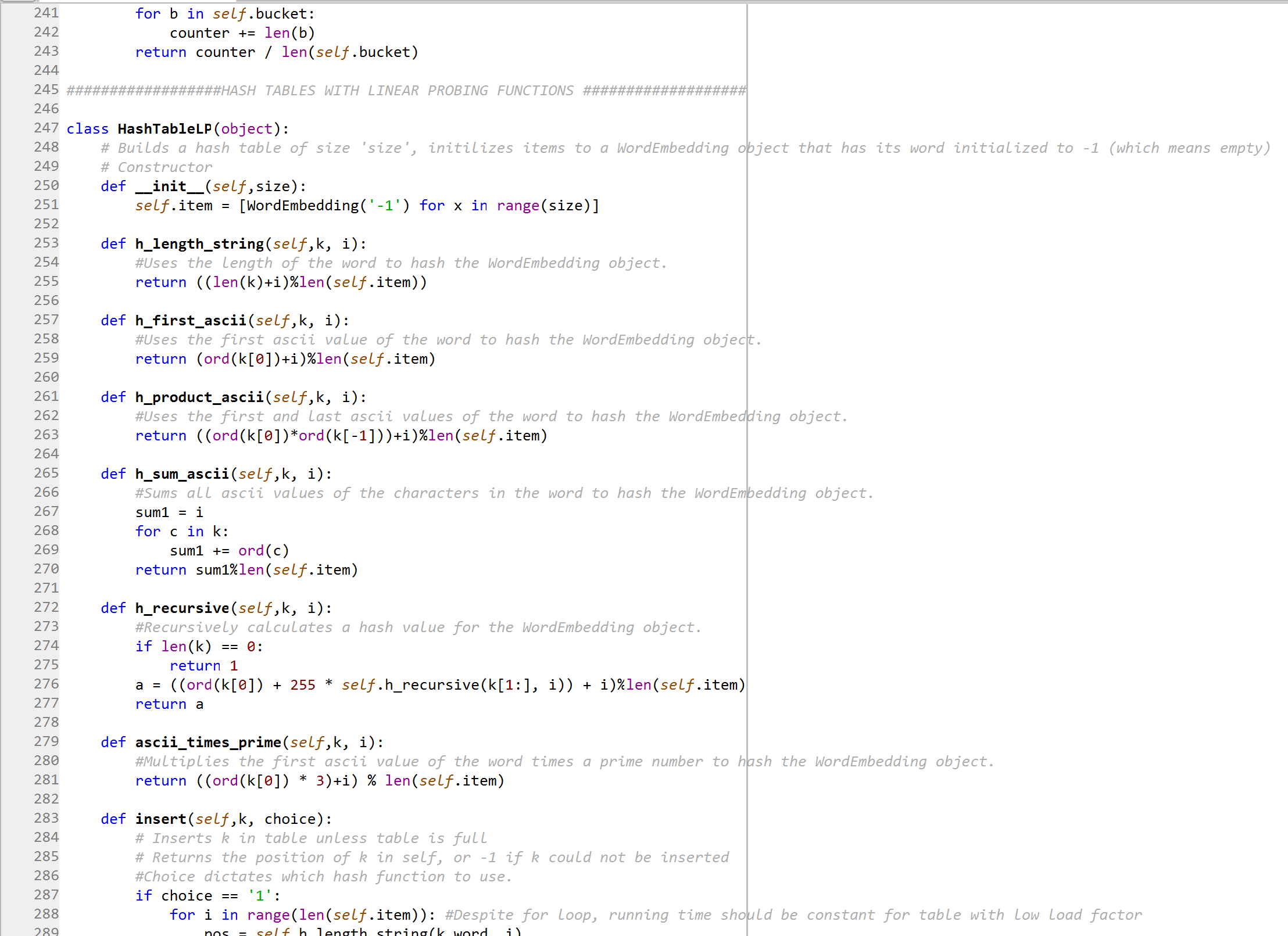
****

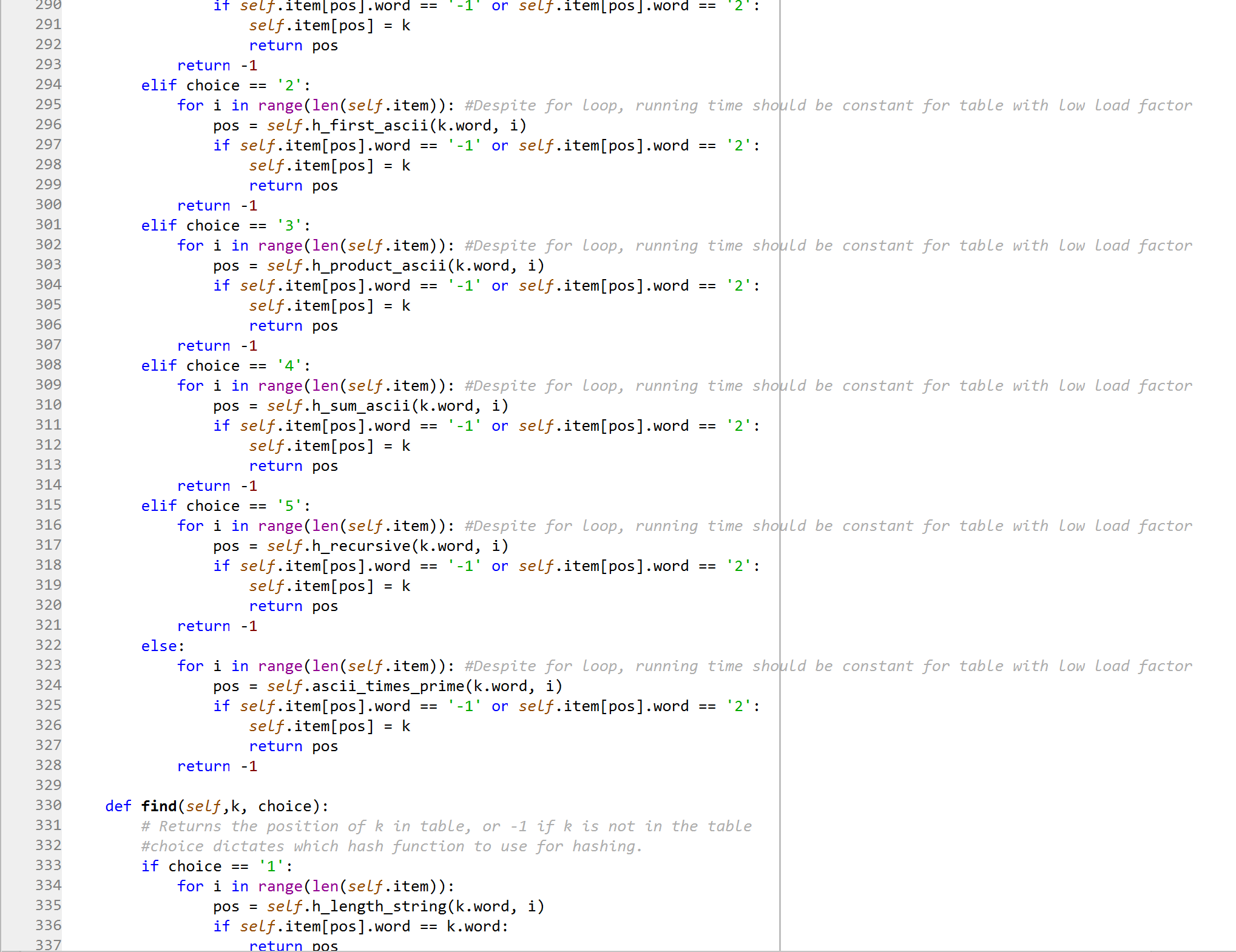
****

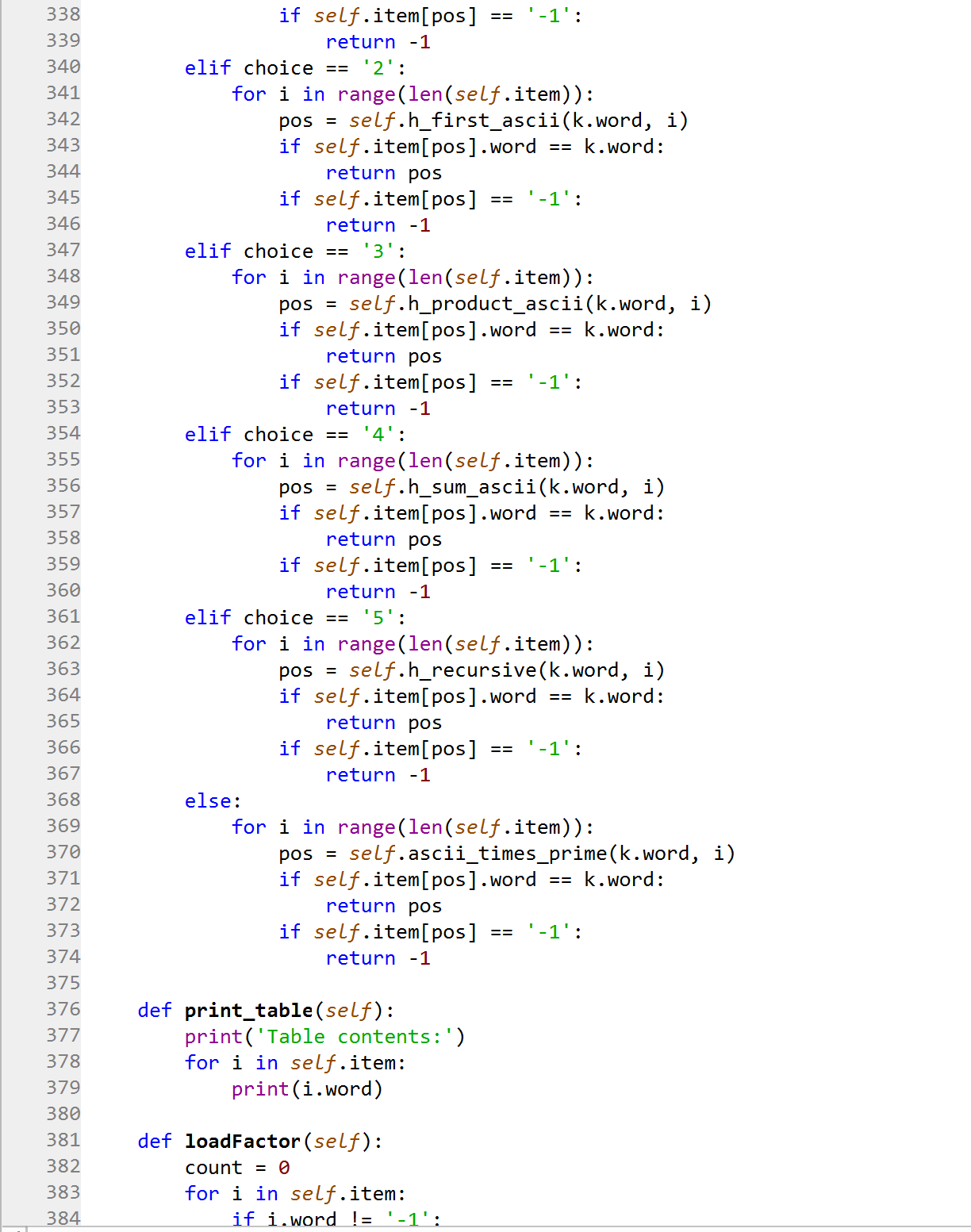
****

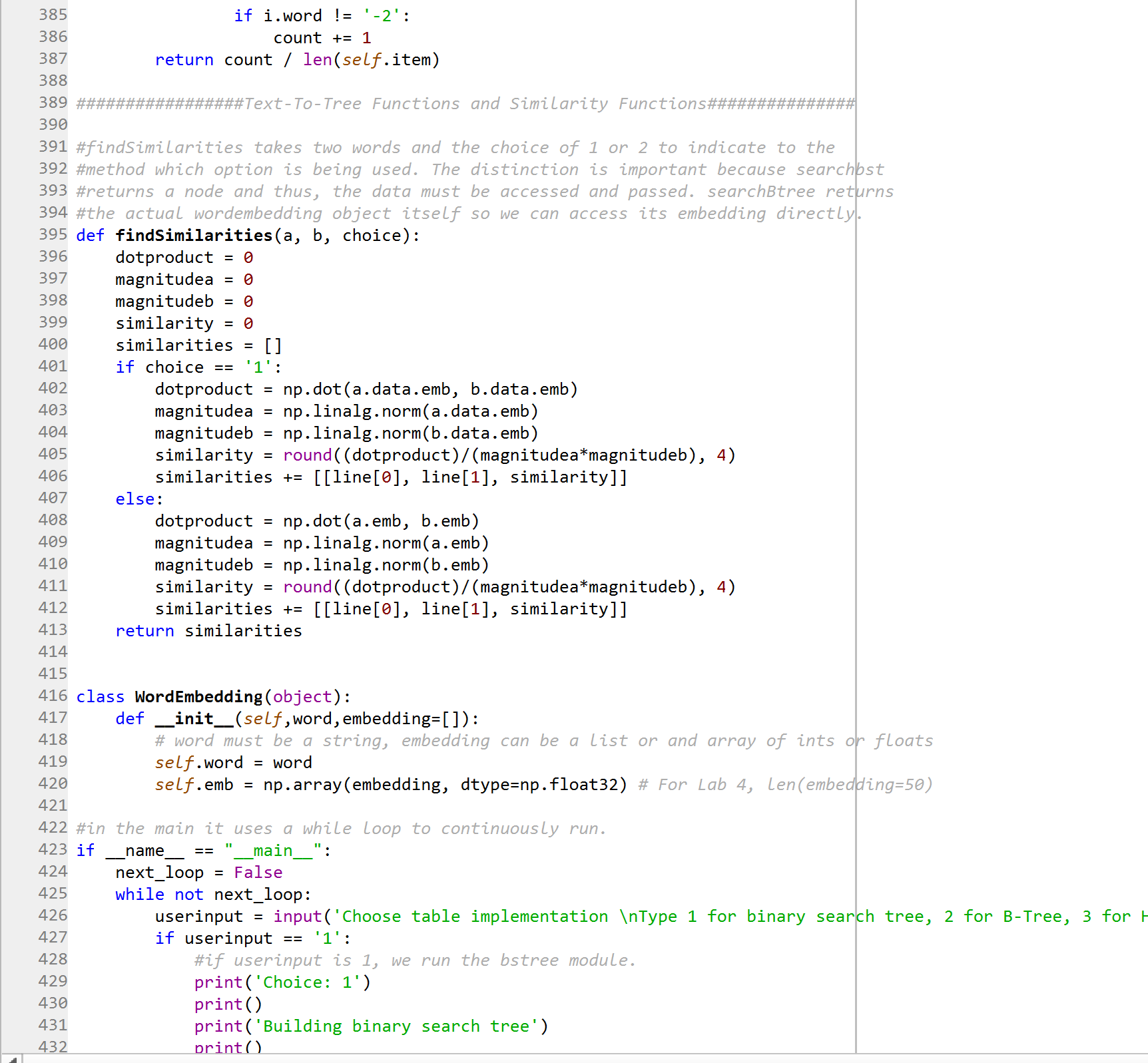
****

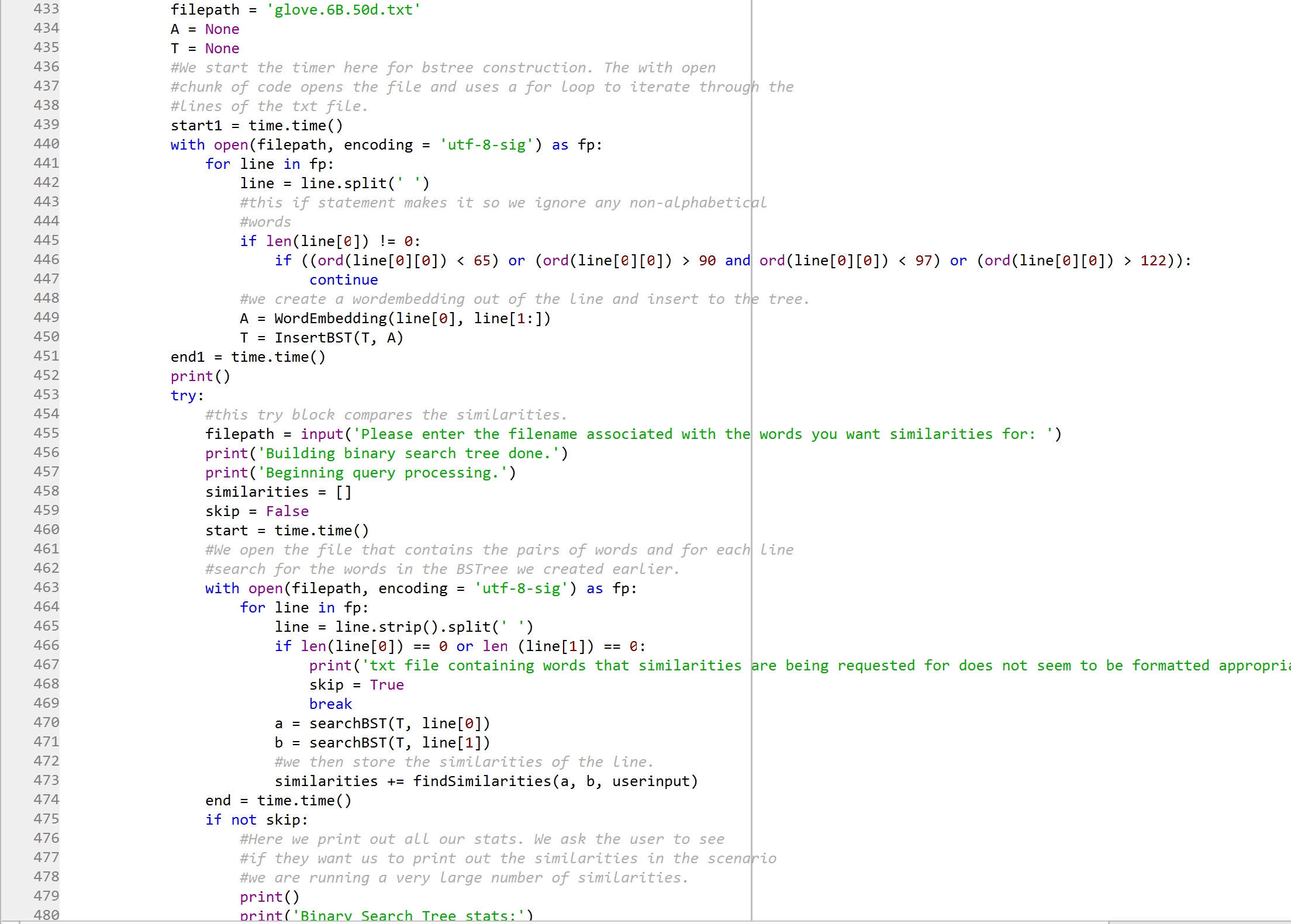
****

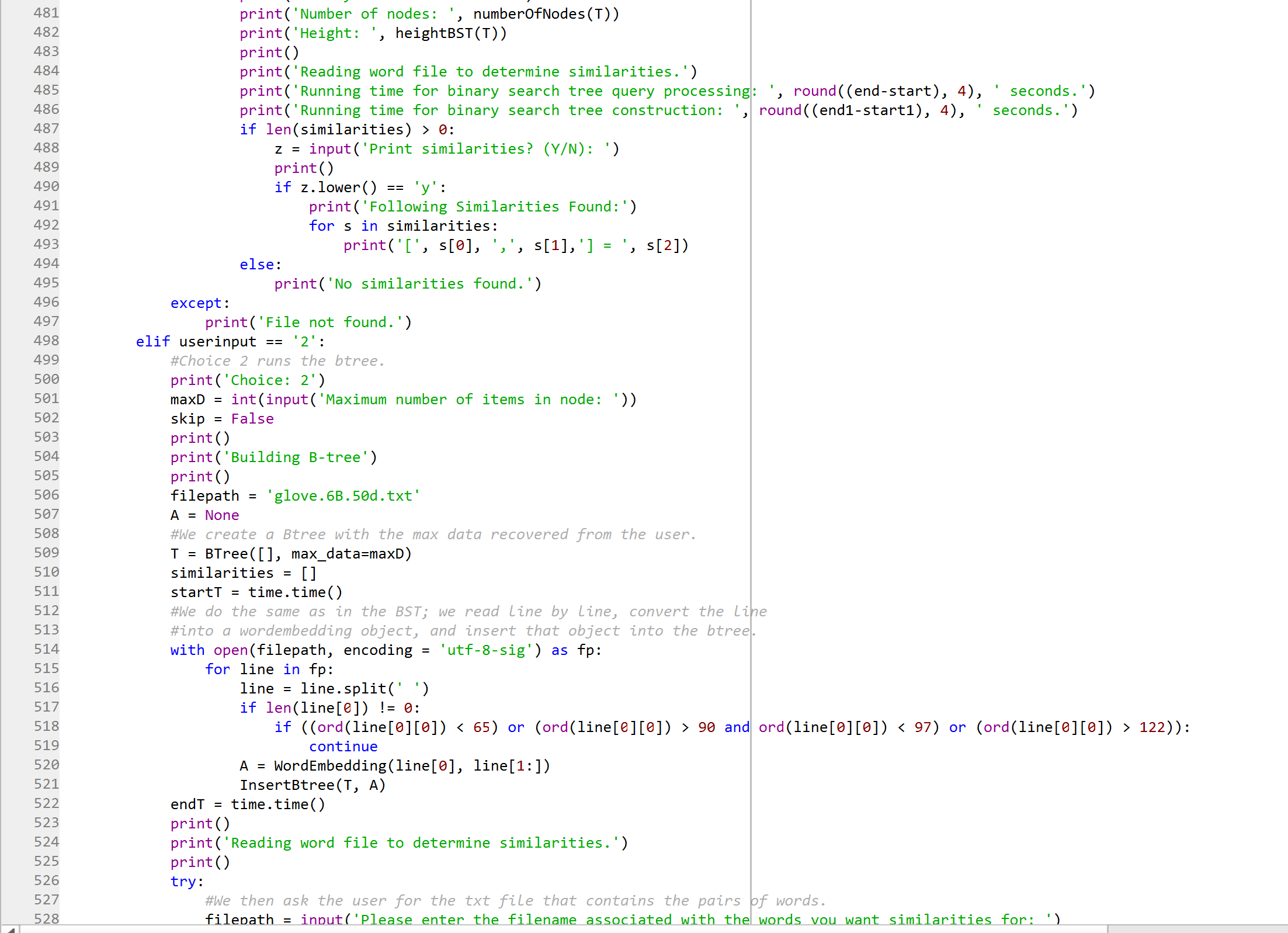
****

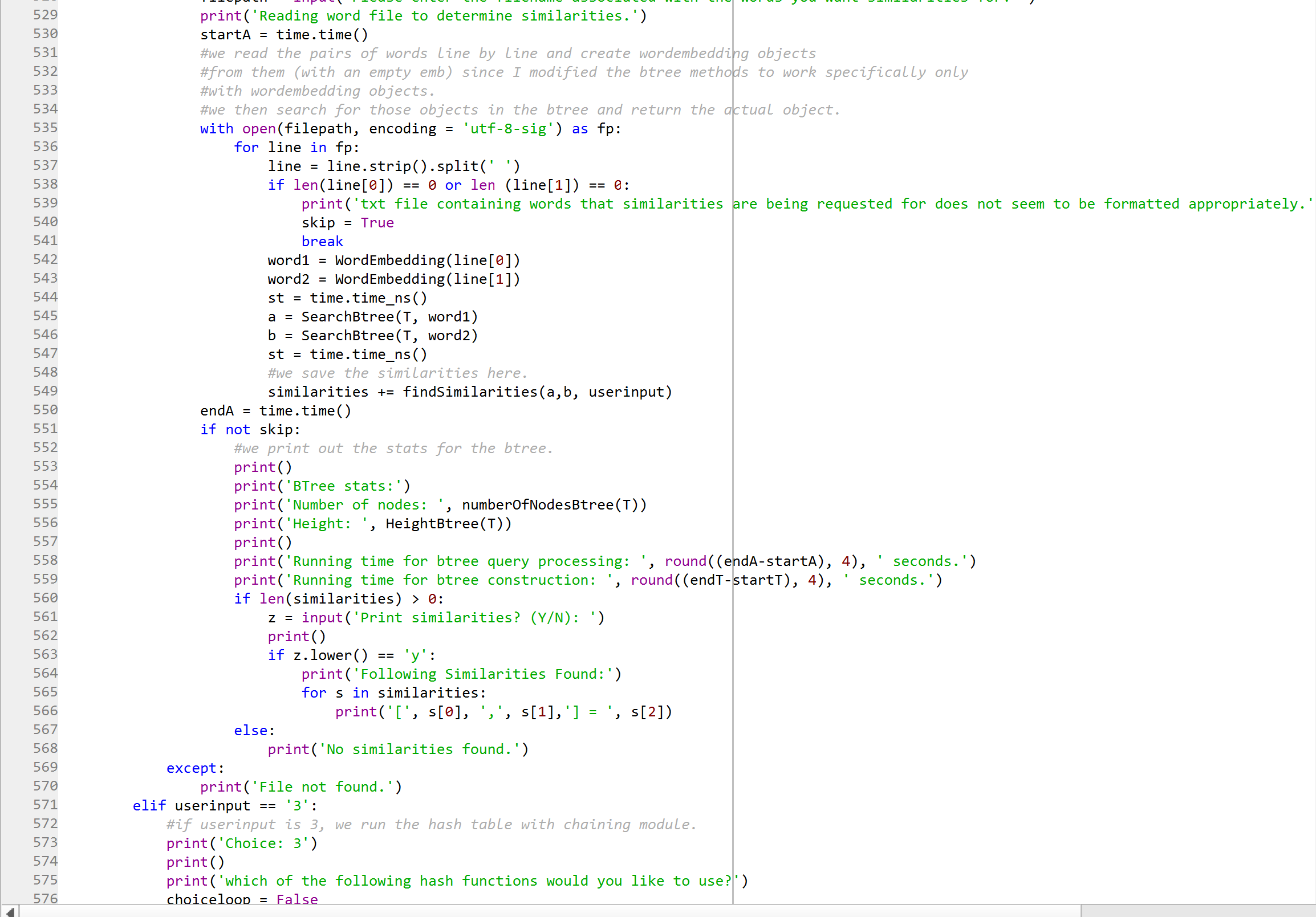
****

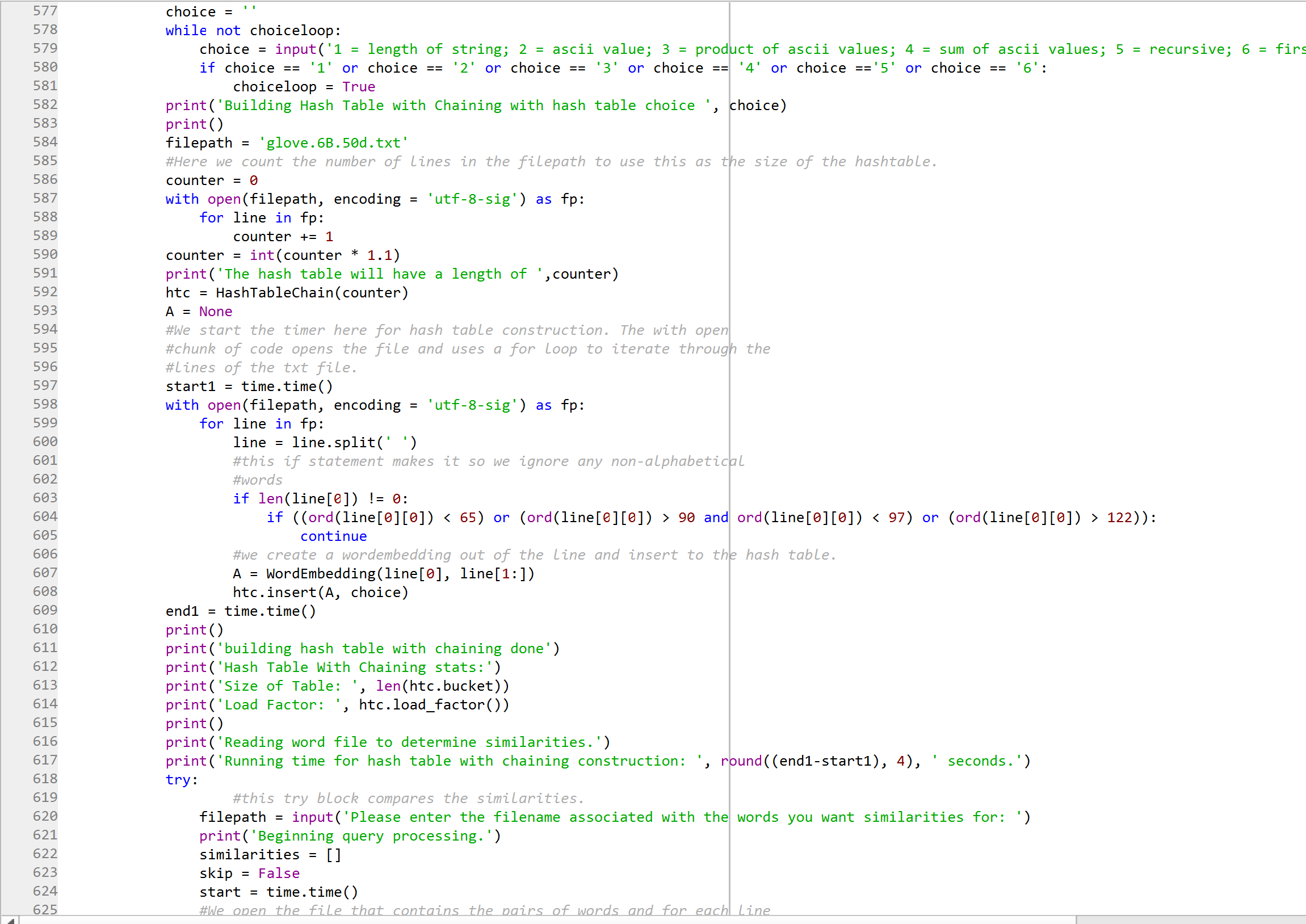
****

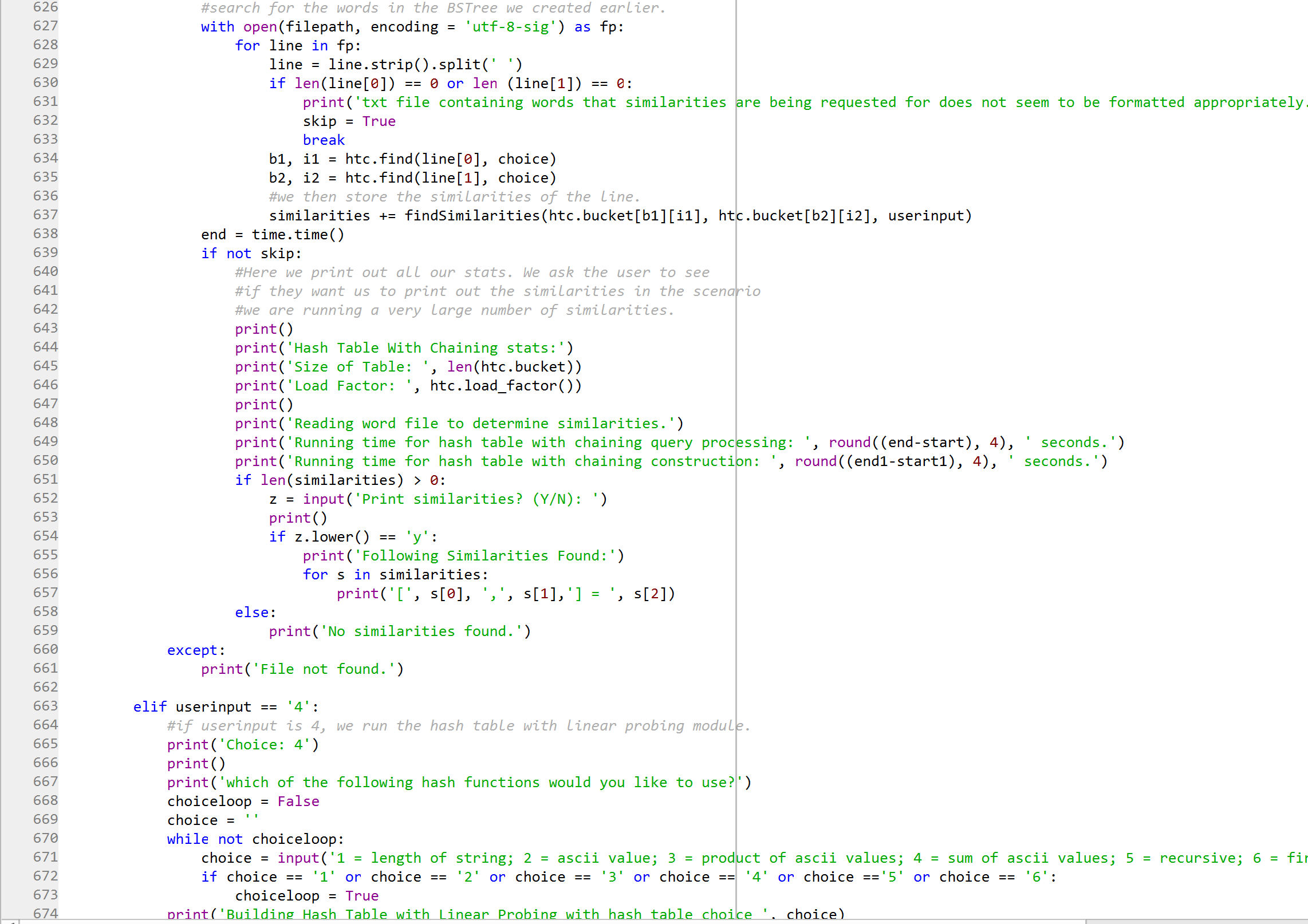
****

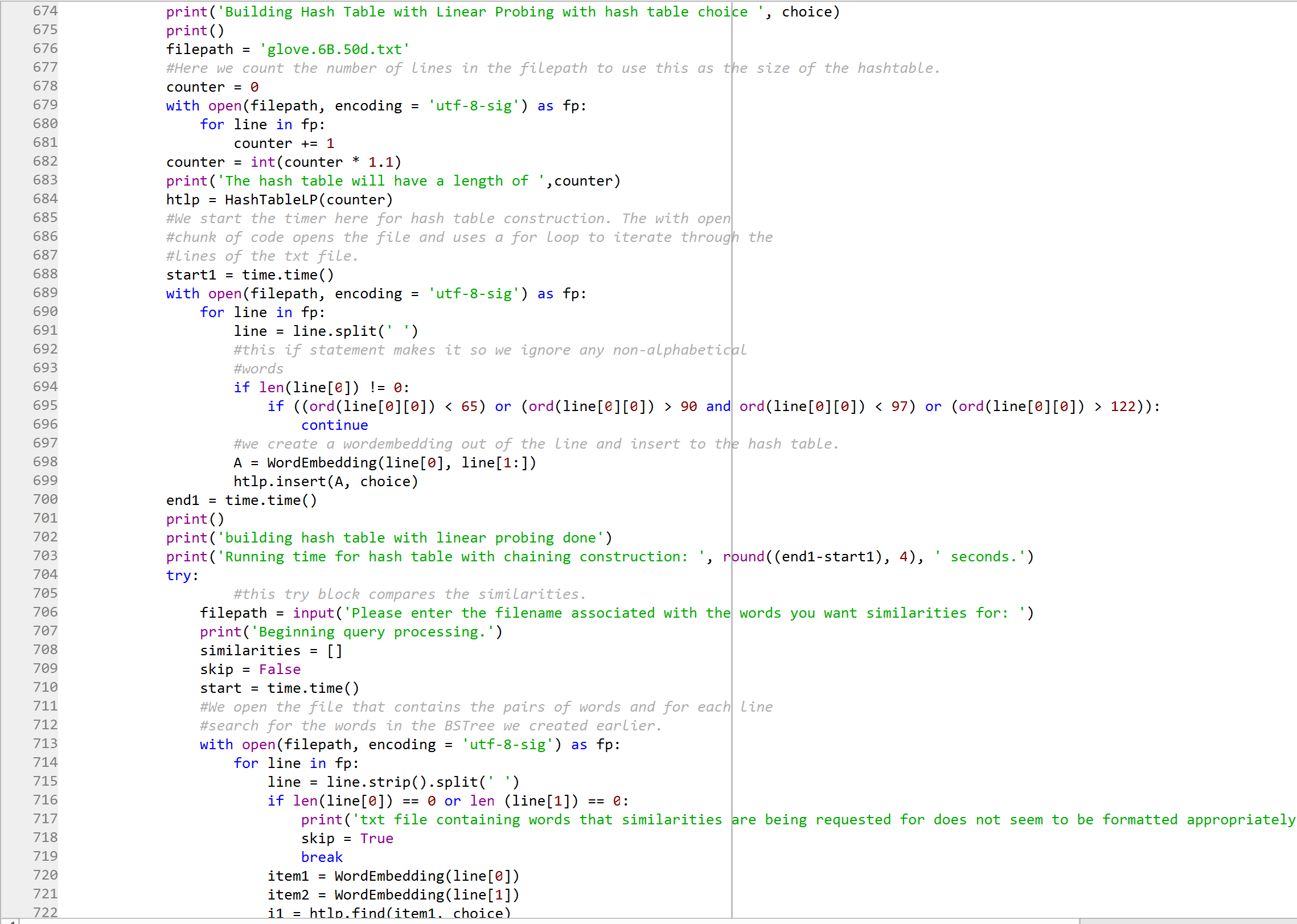
****

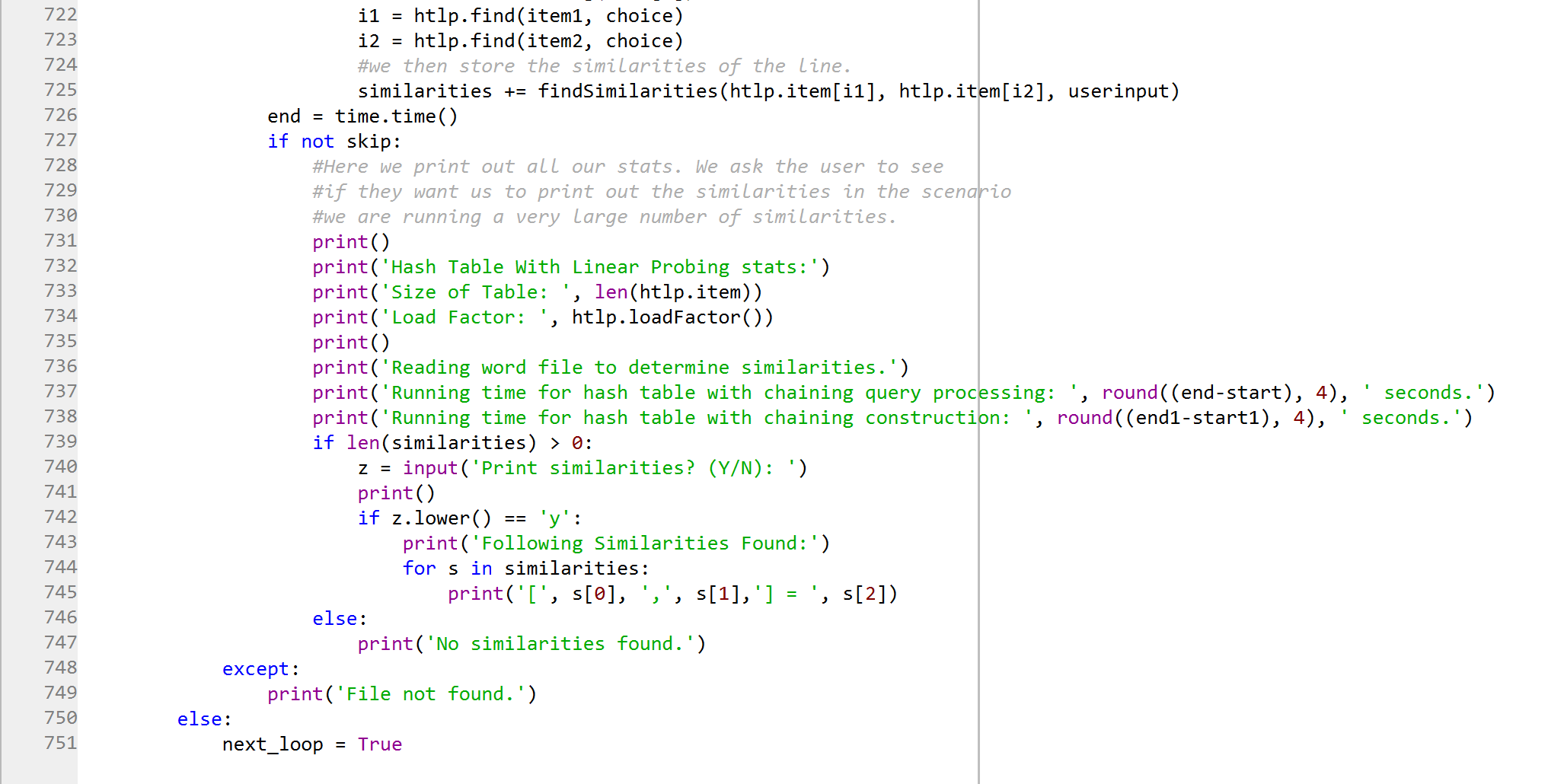
****

****

****

****

****

****

**Academic Honesty Certification**

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.

Jonatan Contreras