LAB REPORT #5

Sergio Ortiz

November 1, 2019

Professor – Olac Fuentes

TA – Anindita Nath

CS 2302 Data Structures – Fall 2019

Introduction

For this lab we were, again, asked to create an algorithm that would be able to determine the similarities between two words by comparing their embeddings. This time however, we were asked to do so with hash tables and compare their runtimes to those of our previous lab. We were also asked to implement several different hash functions to see which one could be better for the problem.

Proposed Solution Design and Implementation

Hash Table with Chaining:

For a while I was pondering on how I would hash a WordEmbedding object to a hash table since the only implementations of hashing I had seen were with numbers. However, after some thinking, I decided I would use the ASCII value of the first character to do this. Around this time, I decided to look at the instructions for the lab and found out that we were already given 5 hashing methods. Nonetheless, I decided to proceed with the idea I had as the main hashing function and then I would implement the other ones as secondary test functions. From here, there were not many changes I had to do to the class implementation of a hash with chaining other than making the insert function take in a WordEmbedding object. Also, I commented out the delete function because it was giving me errors that I did not feel like debugging since we would not really need that function for this lab. Finally, I created a function that would take in a word, look for it in the hash table, and return the embedding.

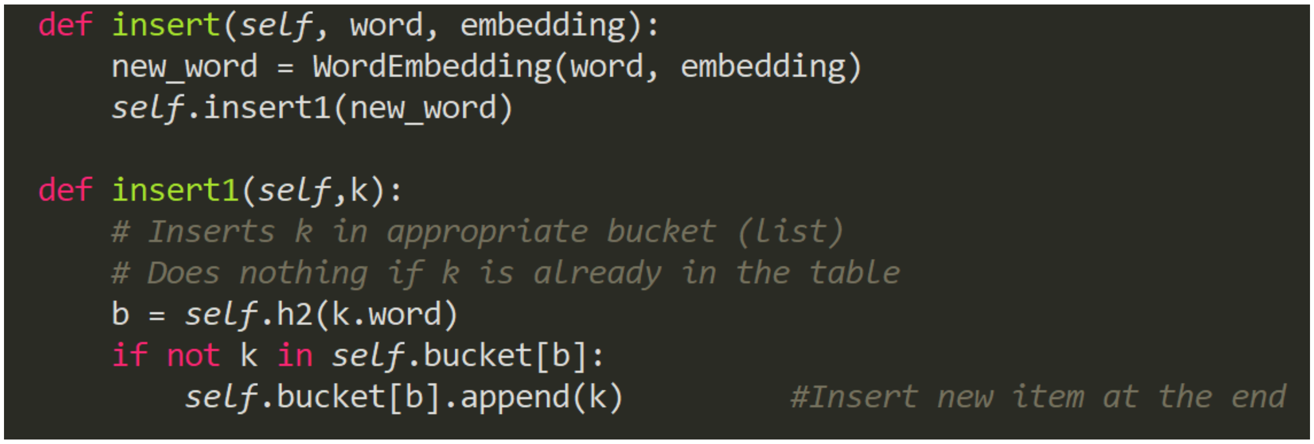
Hash Table with Linear Probing:

I proceeded with the same hash function as the hash table with chaining. For this hash table, I had a harder time implementing it. I do not quite remember why, but my implementation was treating the numpy array as simply a list or something because every time I added a WordEmbedding object, the size of the array would increase by one. This was making the has function essentially useless because the size kept changing and so the number by which I was diving to get the position kept changing. I would not be able to say exactly what I did because I do not know, but I played around with the numpy function until I got an initial array filled with all -1’s and these negative ones would be replaced by WordEmbedding objects each time we added one. Then, I created a function that would return the embedding of a given word if it could be found in the hash table.

Experimental Results

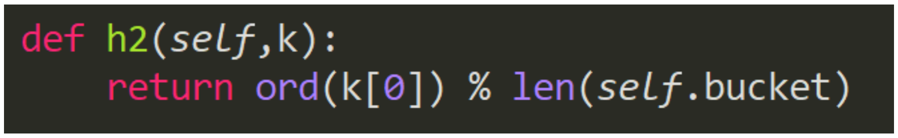
Hash Table with Chaining:

The only real change I did to the implementation we already had on the class website was the insert function by making it take in a WordEmbedding object. Due to the fact that the hash table with chaining uses python lists at every bucket, it lent itself really easily to simply append a WordEmbedding object here.



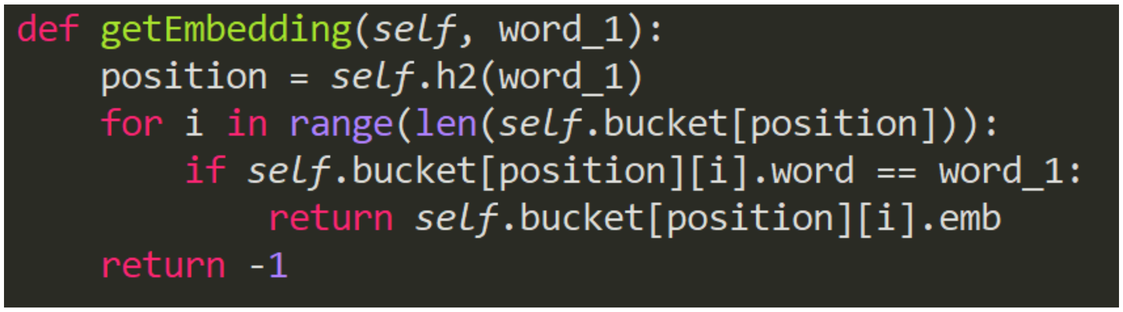
```python
def insert(self, word, embedding):
    new_word = WordEmbedding(word, embedding)
    self.insert1(new_word)

def insert1(self,k):
    # Inserts k in appropriate bucket (list)
    # Does nothing if k is already in the table
    b = self.h2(k.word)
    if not k in self.bucket[b]:
        self.bucket[b].append(k)         #Insert new item at the end
```

Also, it is worth mentioning that the primary hashing function I used checked the ASCII value of the first character and used that to hash to the appropriate location.



```python
def h2(self,k):
    return ord(k[0]) % len(self.bucket)
```

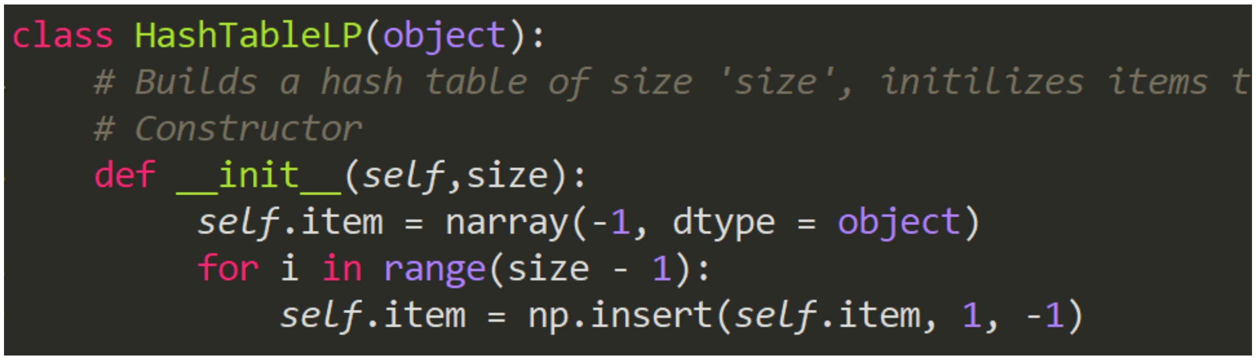
Next, I created a function that would return the embedding of a given word. First, it would find the bucket that the word had to be in. Then, it would traverse the list in this bucket until it either found the word or reached the end, if it found the word, it would return its embedding. If not, it would return negative one.



```python
def getEmbedding(self, word_1):
    position = self.h2(word_1)
    for i in range(len(self.bucket[position])):
        if self.bucket[position][i].word == word_1:
            return self.bucket[position][i].emb
    return -1
```

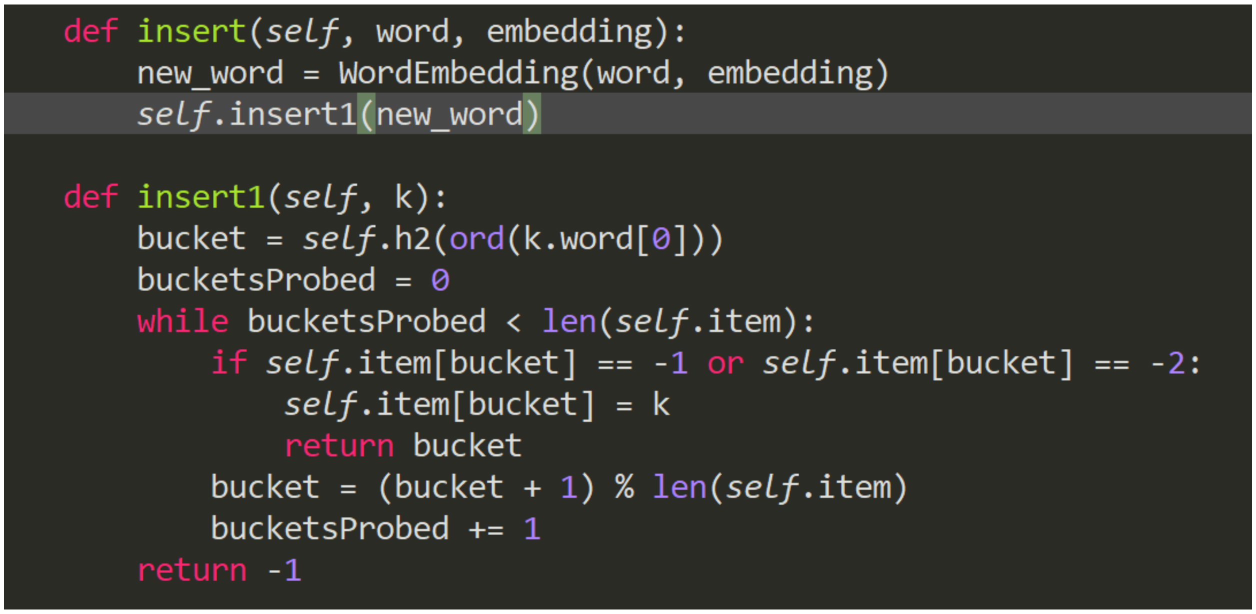
Hash Table with Linear Probing:

The first thing I did to implement the hash table with linear probing was make sure that it would have an array of a fixed size filled with negative ones. Here is how that looked.



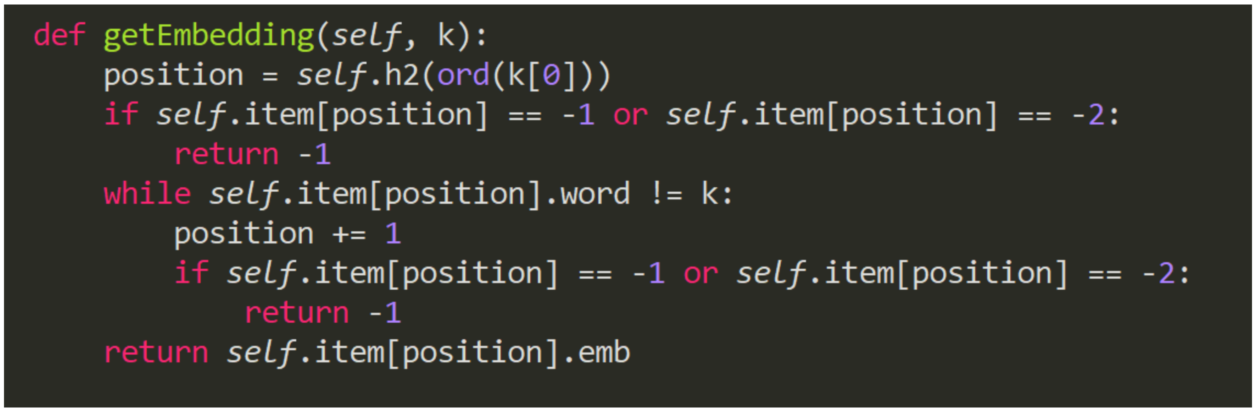
```
class HashTableLP(object):
    # Builds a hash table of size 'size', initilizes items t
    # Constructor
    def __init__(self,size):
        self.item = narray(-1, dtype = object)
        for i in range(size - 1):
            self.item = np.insert(self.item, 1, -1)
```

Something notable here is that the ‘dtype’ is object so that it can take the WordEmbedding object. Next, I began working on an implementation for the insert function. This implementation is identical to the zybooks one, checking where the object should go, and the advancing until it finds an empty slot. The reason I did not simply leave the class website implementation is because it was a bit confusing and hard for me to understand.



```
    def insert(self, word, embedding):
        new_word = WordEmbedding(word, embedding)
        self.insert1(new_word)

    def insert1(self, k):
        bucket = self.h2(ord(k.word[0]))
        bucketsProbed = 0
        while bucketsProbed < len(self.item):
            if self.item[bucket] == -1 or self.item[bucket] == -2:
                self.item[bucket] = k
                return bucket
            bucket = (bucket + 1) % len(self.item)
            bucketsProbed += 1
        return -1
```

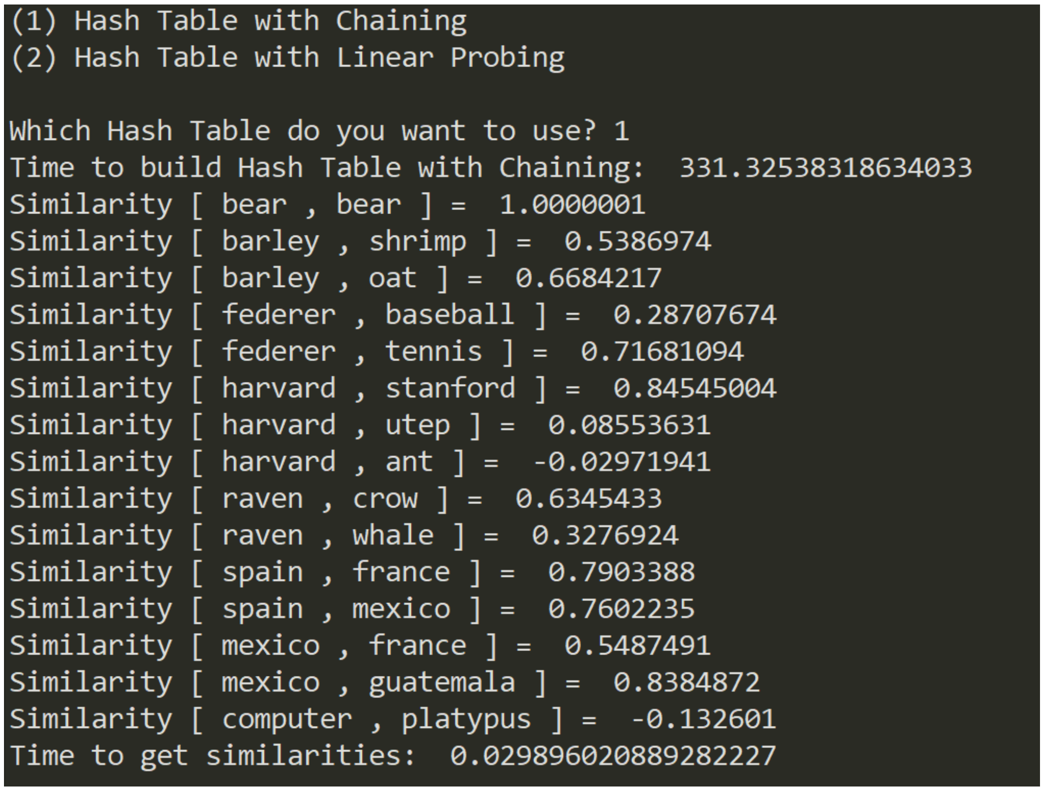
Finally, I created a method that would return the embedding of a word, similar to the one of the Hash Table with Chaining.



```
def getEmbedding(self, k):
    position = self.h2(ord(k[0]))
    if self.item[position] == -1 or self.item[position] == -2:
        return -1
    while self.item[position].word != k:
        position += 1
        if self.item[position] == -1 or self.item[position] == -2:
            return -1
    return self.item[position].emb
```

Analysis:

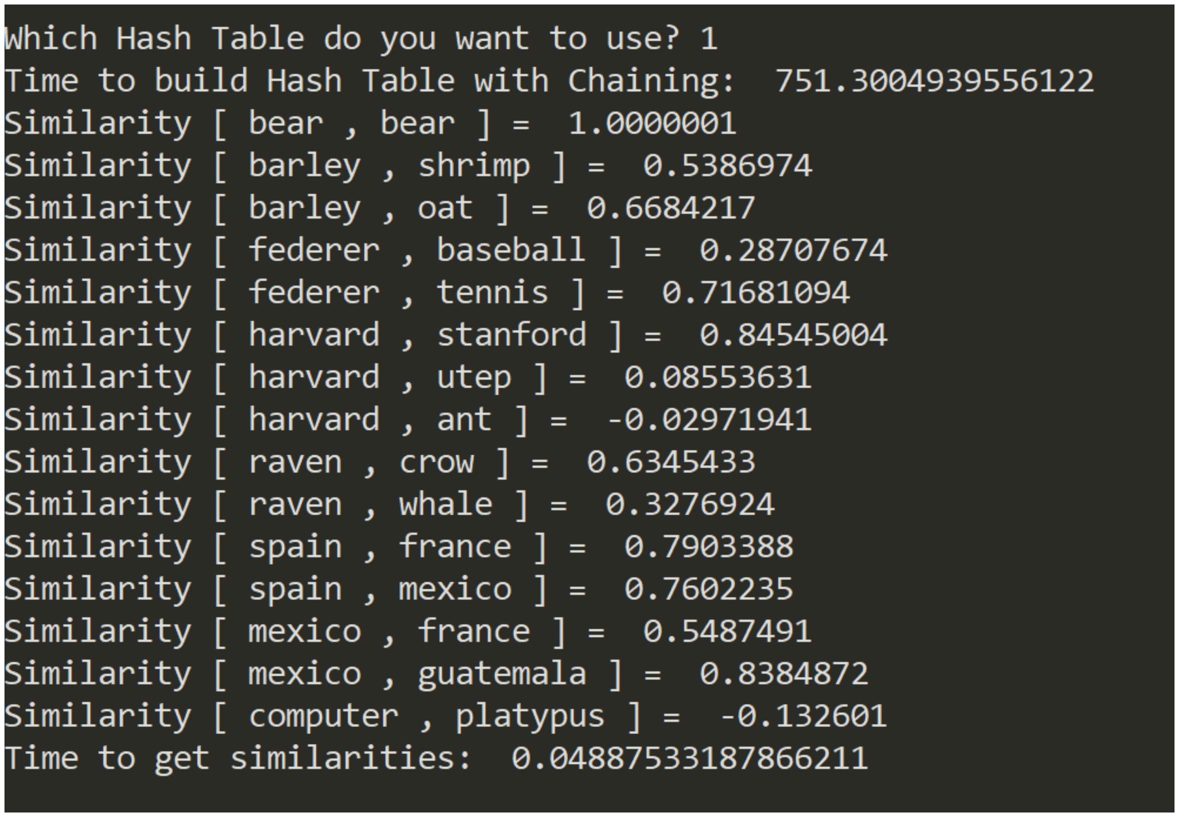
To begin the analysis, I want to throw out a few discrepancies. First off, my implementation of The Hash Table with Linear Probing was not working properly. In order for it to finish running at all, I had to limit the number of items it read off the file to somewhere around 5,000, but at this point it was not reading a lot of the words, so I was not getting accurate comparisons. The ones I did get were good though. Also, for some reason, it seems like the hash tables are taking much longer to get to get this problem done when compared to the trees. Therefore, I will be basing my results for this section off the hash table with chaining.



```
(1) Hash Table with Chaining
(2) Hash Table with Linear Probing

Which Hash Table do you want to use? 1
Time to build Hash Table with Chaining:  331.32538318634033
Similarity [ bear , bear ] =  1.0000001
Similarity [ barley , shrimp ] =  0.5386974
Similarity [ barley , oat ] =  0.6684217
Similarity [ federer , baseball ] =  0.28707674
Similarity [ federer , tennis ] =  0.71681094
Similarity [ harvard , stanford ] =  0.84545004
Similarity [ harvard , utep ] =  0.08553631
Similarity [ harvard , ant ] =  -0.02971941
Similarity [ raven , crow ] =  0.6345433
Similarity [ raven , whale ] =  0.3276924
Similarity [ spain , france ] =  0.7903388
Similarity [ spain , mexico ] =  0.7602235
Similarity [ mexico , france ] =  0.5487491
Similarity [ mexico , guatemala ] =  0.8384872
Similarity [ computer , platypus ] =  -0.132601
Time to get similarities:  0.029896020889282227
```

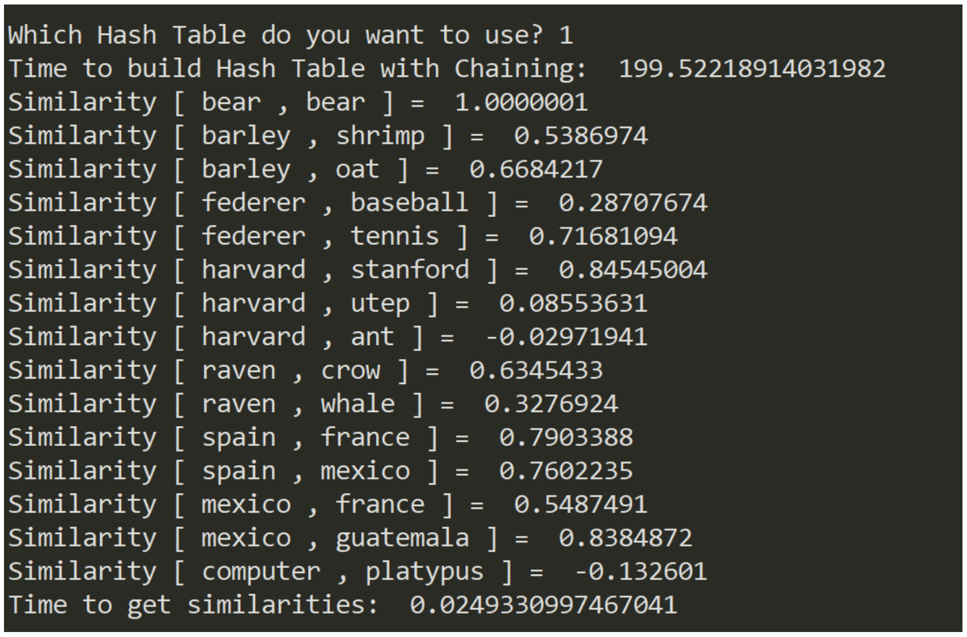
Here is a sample run of the program with the second hash function. As we can see if takes almost six minutes to build the hash table, and it is also taking longer than the trees to get the similarities, with the best ones for the trees taking around .001 seconds.

Next, I saw how it would react when I tried the first hashing function, the length of the word.



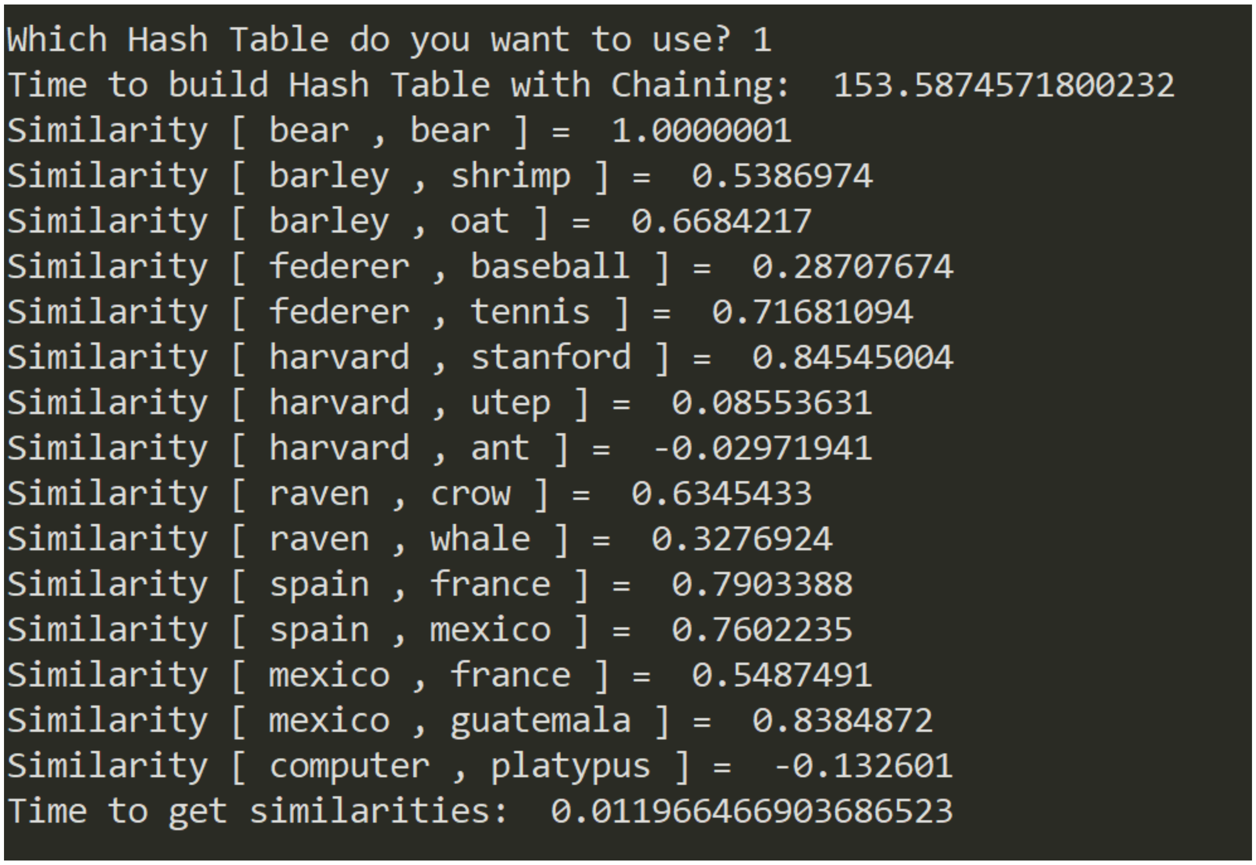
```
Which Hash Table do you want to use? 1
Time to build Hash Table with Chaining:  751.3004939556122
Similarity [ bear , bear ] =  1.0000001
Similarity [ barley , shrimp ] =  0.5386974
Similarity [ barley , oat ] =  0.6684217
Similarity [ federer , baseball ] =  0.28707674
Similarity [ federer , tennis ] =  0.71681094
Similarity [ harvard , stanford ] =  0.84545004
Similarity [ harvard , utep ] =  0.08553631
Similarity [ harvard , ant ] =  -0.02971941
Similarity [ raven , crow ] =  0.6345433
Similarity [ raven , whale ] =  0.3276924
Similarity [ spain , france ] =  0.7903388
Similarity [ spain , mexico ] =  0.7602235
Similarity [ mexico , france ] =  0.5487491
Similarity [ mexico , guatemala ] =  0.8384872
Similarity [ computer , platypus ] =  -0.132601
Time to get similarities:  0.04887533187866211
```

As we can see, this one took even longer, taking longer than ten minutes! The query processing was also slightly longer. Next, I proceeded to test out the third hashing method the product of the first and last ASCII values.



```
Which Hash Table do you want to use? 1
Time to build Hash Table with Chaining:  199.52218914031982
Similarity [ bear , bear ] =  1.0000001
Similarity [ barley , shrimp ] =  0.5386974
Similarity [ barley , oat ] =  0.6684217
Similarity [ federer , baseball ] =  0.28707674
Similarity [ federer , tennis ] =  0.71681094
Similarity [ harvard , stanford ] =  0.84545004
Similarity [ harvard , utep ] =  0.08553631
Similarity [ harvard , ant ] =  -0.02971941
Similarity [ raven , crow ] =  0.6345433
Similarity [ raven , whale ] =  0.3276924
Similarity [ spain , france ] =  0.7903388
Similarity [ spain , mexico ] =  0.7602235
Similarity [ mexico , france ] =  0.5487491
Similarity [ mexico , guatemala ] =  0.8384872
Similarity [ computer , platypus ] =  -0.132601
Time to get similarities:  0.0249330997467041
```

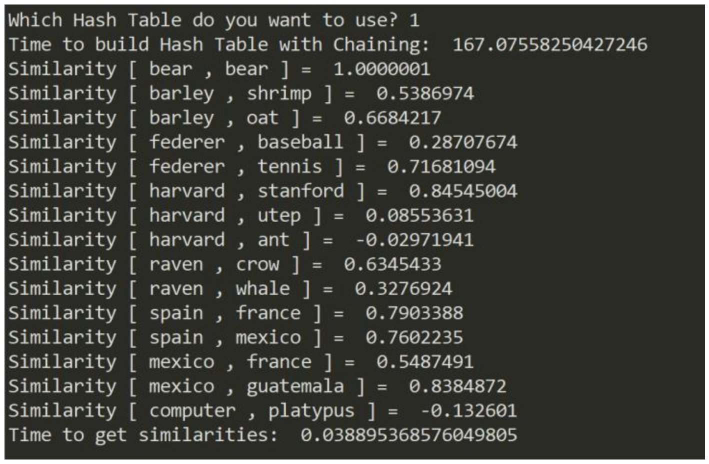
Wow! Amazingly this one took only three minutes to build, but the query processing was still not the greatest. Next, I proceeded to try the next hashing method, the sum of the ASCII values.



```
Which Hash Table do you want to use? 1
Time to build Hash Table with Chaining:  153.5874571800232
Similarity [ bear , bear ] =  1.0000001
Similarity [ barley , shrimp ] =  0.5386974
Similarity [ barley , oat ] =  0.6684217
Similarity [ federer , baseball ] =  0.28707674
Similarity [ federer , tennis ] =  0.71681094
Similarity [ harvard , stanford ] =  0.84545004
Similarity [ harvard , utep ] =  0.08553631
Similarity [ harvard , ant ] =  -0.02971941
Similarity [ raven , crow ] =  0.6345433
Similarity [ raven , whale ] =  0.3276924
Similarity [ spain , france ] =  0.7903388
Similarity [ spain , mexico ] =  0.7602235
Similarity [ mexico , france ] =  0.5487491
Similarity [ mexico , guatemala ] =  0.8384872
Similarity [ computer , platypus ] =  -0.132601
Time to get similarities:  0.011966466903686523
```

This one was even more amazing, taking a little over 2 minutes to build. The query processing however, was still aligned with our previous results. Next, I proceeded to test out the recursive hash method. For some reason, I could not get that one to work, even though I knew it was bound to be the best hashing method.

Next, I decided to test out my own selected hash function, where I added all the ASCII values and then divided them by two.



```
Which Hash Table do you want to use? 1
Time to build Hash Table with Chaining:  167.07558250427246
Similarity [ bear , bear ] =  1.0000001
Similarity [ barley , shrimp ] =  0.5386974
Similarity [ barley , oat ] =  0.6684217
Similarity [ federer , baseball ] =  0.28707674
Similarity [ federer , tennis ] =  0.71681094
Similarity [ harvard , stanford ] =  0.84545004
Similarity [ harvard , utep ] =  0.08553631
Similarity [ harvard , ant ] =  -0.02971941
Similarity [ raven , crow ] =  0.6345433
Similarity [ raven , whale ] =  0.3276924
Similarity [ spain , france ] =  0.7903388
Similarity [ spain , mexico ] =  0.7602235
Similarity [ mexico , france ] =  0.5487491
Similarity [ mexico , guatemala ] =  0.8384872
Similarity [ computer , platypus ] =  -0.132601
Time to get similarities:  0.038895368576049805
```

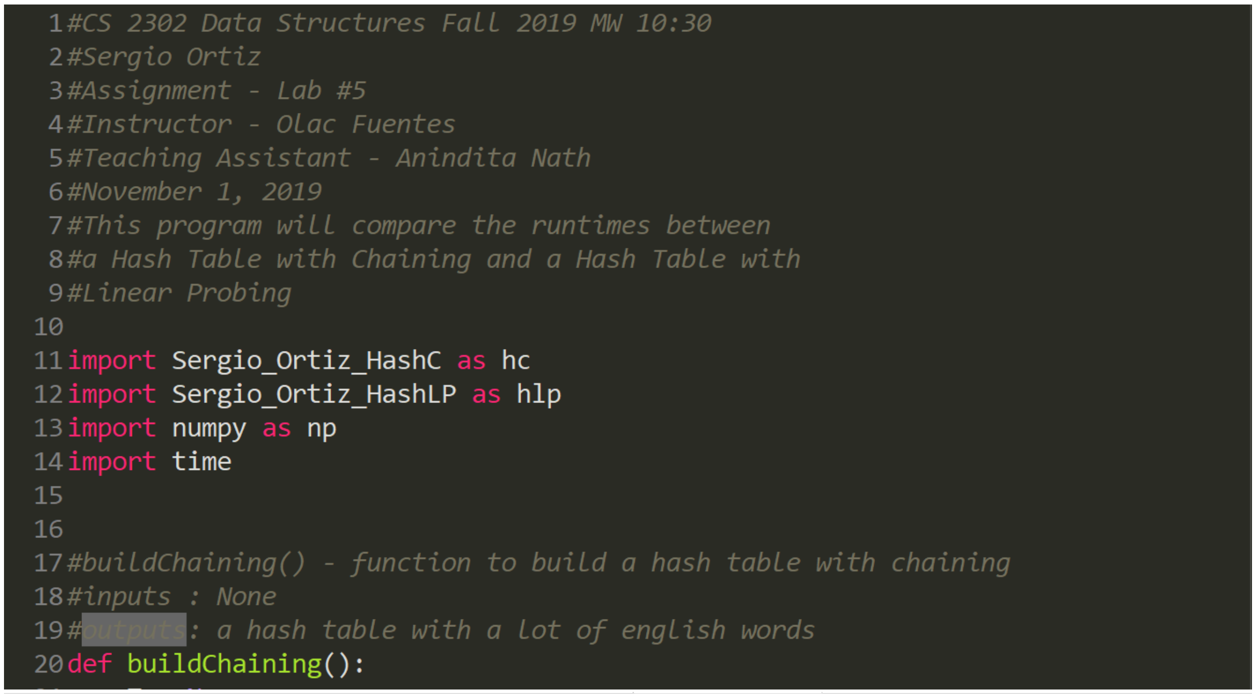
Surprisingly, this one did much better than the first hashing function, diving the time by around 5. Strangely, the query processing seems to be taking longer, but I am sure this is simply an outlier in the data.

Insert Hash Table with Chaining	O(1)
Insert Hash Table with Linear Probing	O(n)
Get Embedding (Chaining)	O(1) <O(n), where n is the length of the sub list>
Get Embedding (Linear Probing)	O(n)
Query Processing (Chaining)	O(n)
Query Processing(Linear)	O(n)

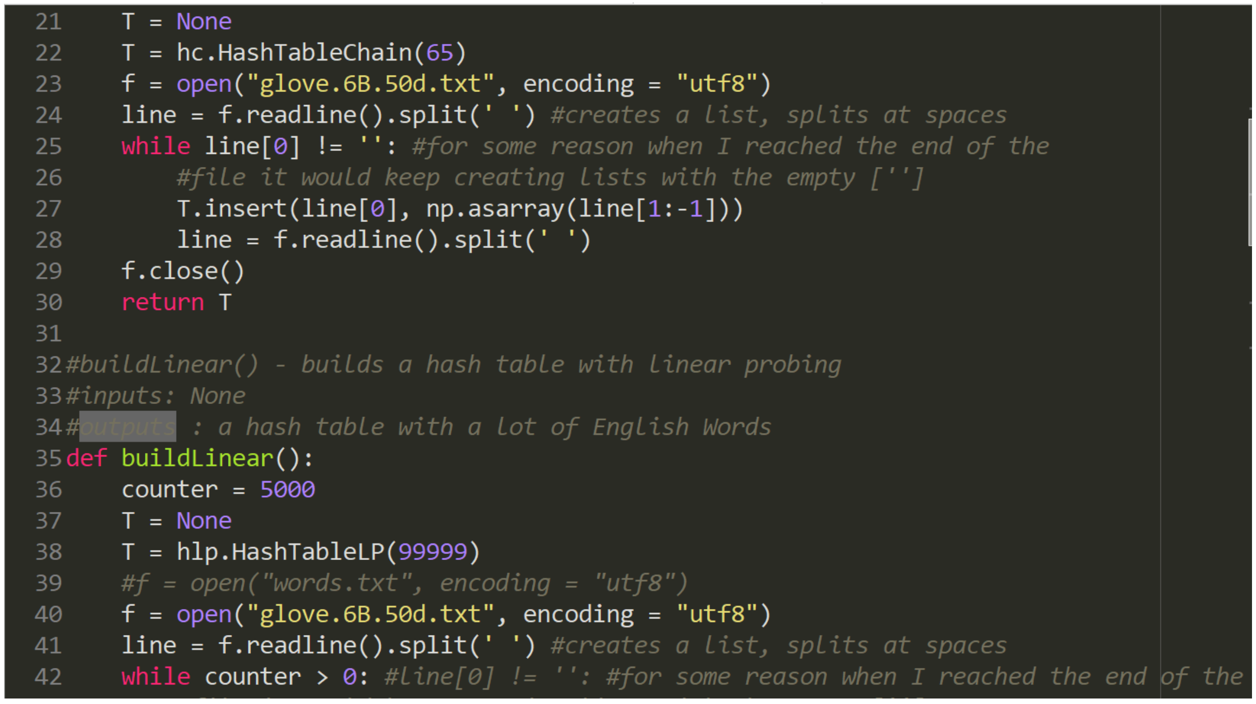
Conclusion

This lab was honestly very frustrating to me because the methods were taking very long to execute. I am not sure if that was intended or if my implementations were wrong, but it was quite tough to run all these experiments knowing that they would take a while to give me an answer, barring any compilation errors that might have occurred. I will say that figuring out how to hash other things besides integers to a table was quite intriguing. Also, it was fun to see how these programs would return the same results as the previous lab even though we were now using completely different data structures.

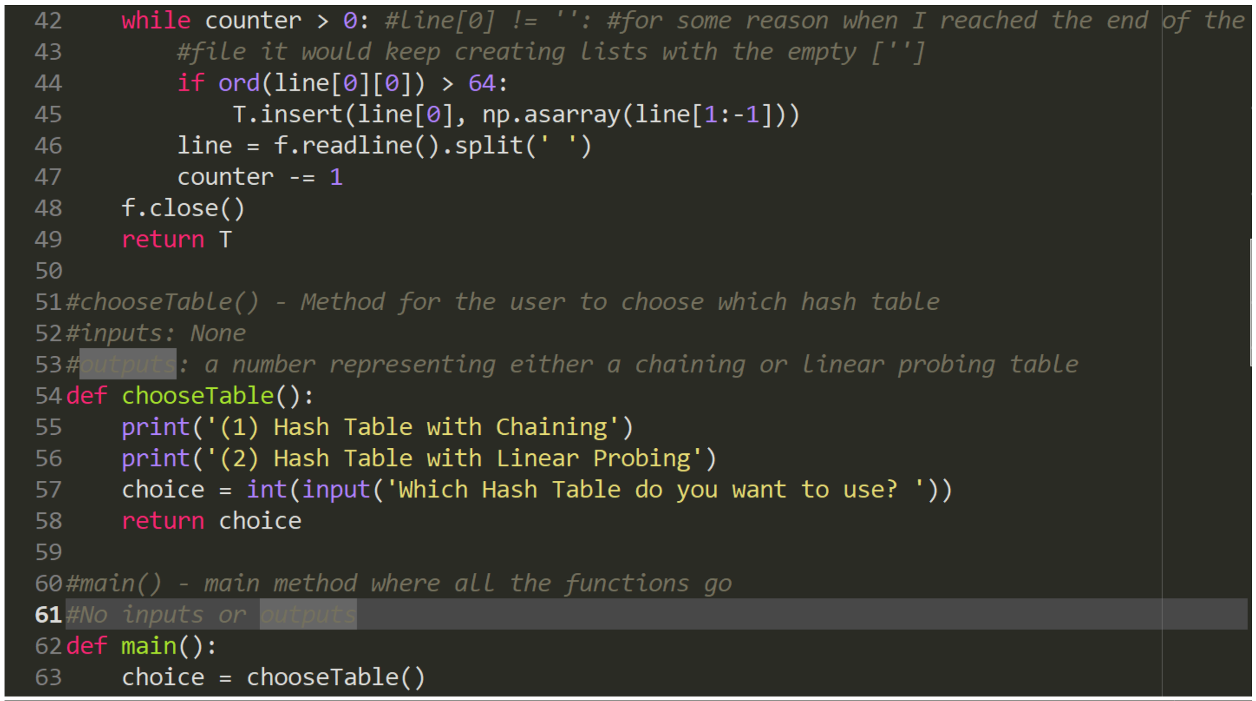
Appendix



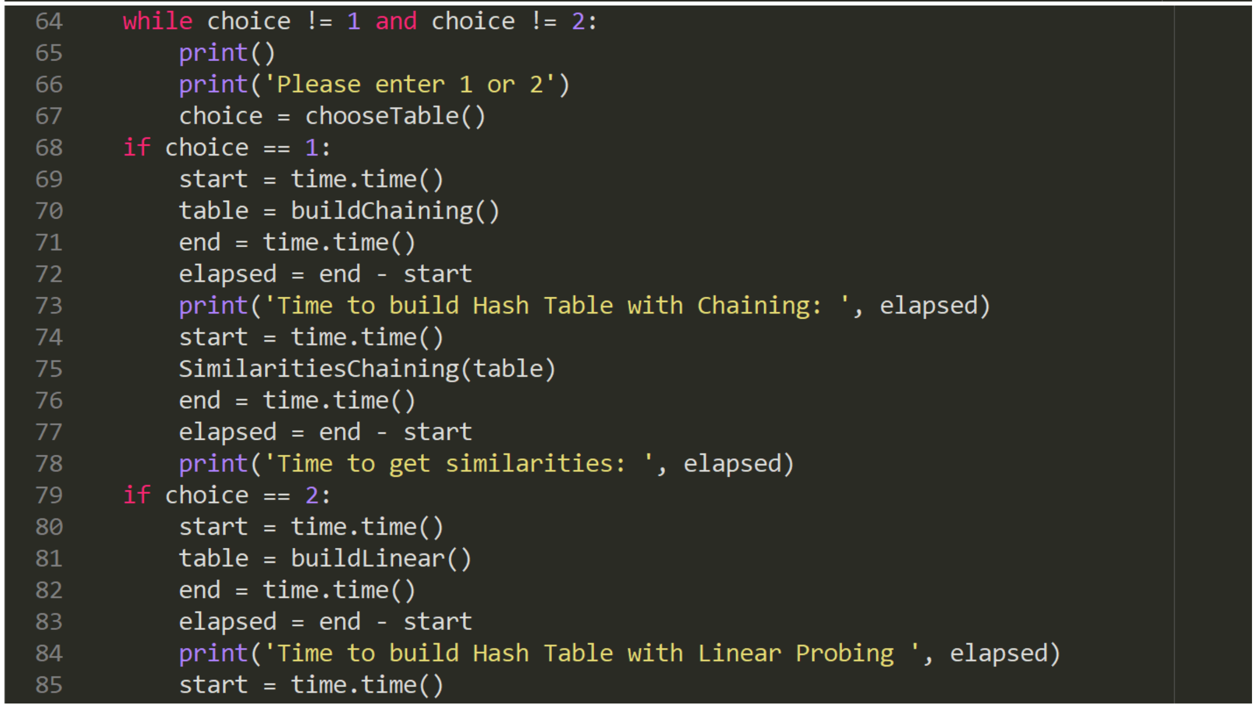
```python
#CS 2302 Data Structures Fall 2019 MW 10:30
#Sergio Ortiz
#Assignment - Lab #5
#Instructor - Olac Fuentes
#Teaching Assistant - Anindita Nath
#November 1, 2019
#This program will compare the runtimes between
#a Hash Table with Chaining and a Hash Table with
#Linear Probing

import Sergio_Ortiz_HashC as hc
import Sergio_Ortiz_HashLP as hlp
import numpy as np
import time


#buildChaining() - function to build a hash table with chaining
#inputs : None
#outputs: a hash table with a lot of english words
def buildChaining():
```



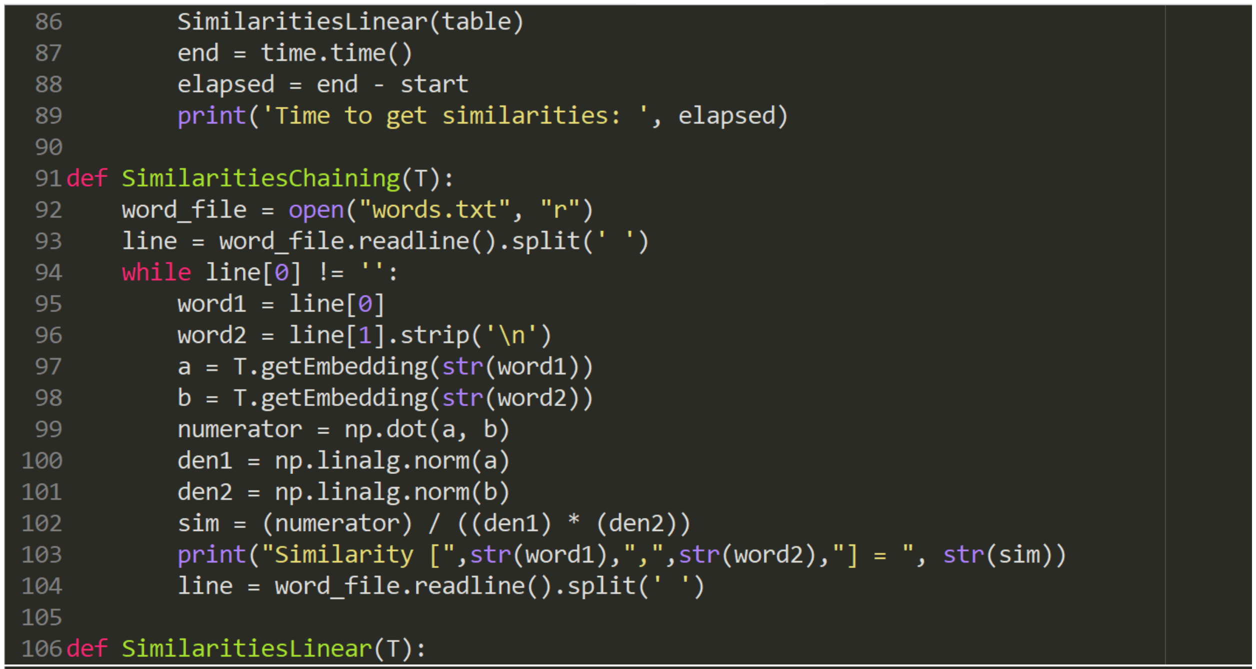
```python
    T = None
    T = hc.HashTableChain(65)
    f = open("glove.6B.50d.txt", encoding = "utf8")
    line = f.readline().split(' ') #creates a list, splits at spaces
    while line[0] != '': #for some reason when I reached the end of the
        #file it would keep creating lists with the empty ['']
        T.insert(line[0], np.asarray(line[1:-1]))
        line = f.readline().split(' ')
    f.close()
    return T

#buildLinear() - builds a hash table with linear probing
#inputs: None
#outputs : a hash table with a lot of English Words
def buildLinear():
    counter = 5000
    T = None
    T = hlp.HashTableLP(99999)
    #f = open("words.txt", encoding = "utf8")
    f = open("glove.6B.50d.txt", encoding = "utf8")
    line = f.readline().split(' ') #creates a list, splits at spaces
    while counter > 0: #line[0] != '': #for some reason when I reached the end of the
```



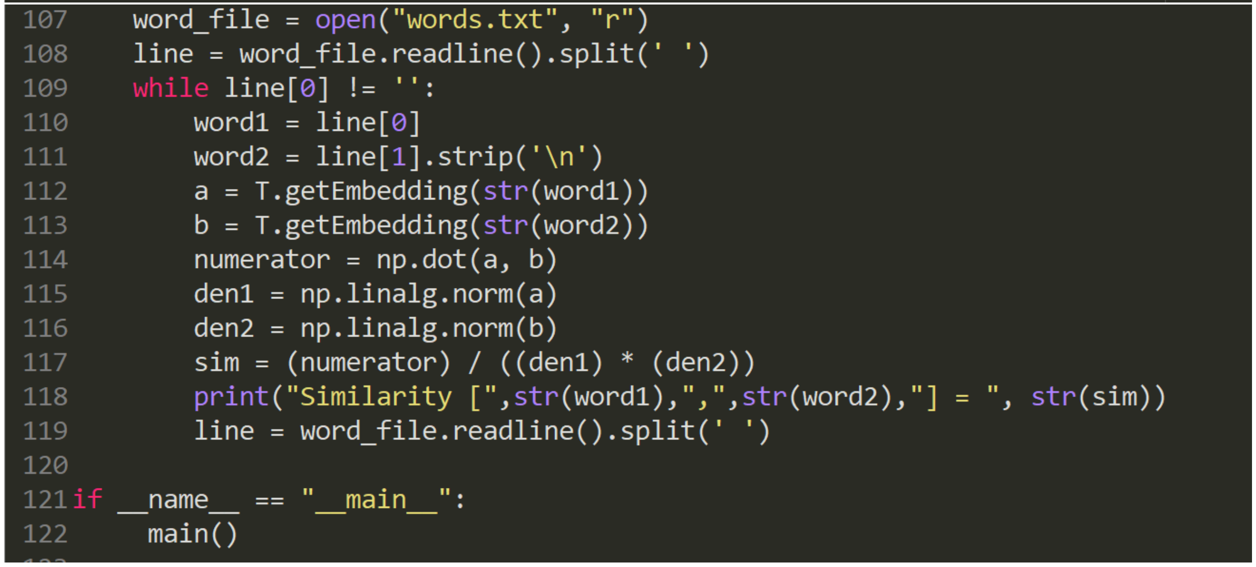
```python
    while counter > 0: #line[0] != '': #for some reason when I reached the end of the
        #file it would keep creating lists with the empty ['']
        if ord(line[0][0]) > 64:
            T.insert(line[0], np.asarray(line[1:-1]))
        line = f.readline().split(' ')
        counter -= 1
    f.close()
    return T

#chooseTable() - Method for the user to choose which hash table
#inputs: None
#outputs: a number representing either a chaining or linear probing table
def chooseTable():
    print('(1) Hash Table with Chaining')
    print('(2) Hash Table with Linear Probing')
    choice = int(input('Which Hash Table do you want to use? '))
    return choice

#main() - main method where all the functions go
#No inputs or outputs
def main():
    choice = chooseTable()
```



```python
    while choice != 1 and choice != 2:
        print()
        print('Please enter 1 or 2')
        choice = chooseTable()
    if choice == 1:
        start = time.time()
        table = buildChaining()
        end = time.time()
        elapsed = end - start
        print('Time to build Hash Table with Chaining: ', elapsed)
        start = time.time()
        SimilaritiesChaining(table)
        end = time.time()
        elapsed = end - start
        print('Time to get similarities: ', elapsed)
    if choice == 2:
        start = time.time()
        table = buildLinear()
        end = time.time()
        elapsed = end - start
        print('Time to build Hash Table with Linear Probing ', elapsed)
        start = time.time()
```



```
        SimilaritiesLinear(table)
        end = time.time()
        elapsed = end - start
        print('Time to get similarities: ', elapsed)

def SimilaritiesChaining(T):
    word_file = open("words.txt", "r")
    line = word_file.readline().split(' ')
    while line[0] != '':
        word1 = line[0]
        word2 = line[1].strip('\n')
        a = T.getEmbedding(str(word1))
        b = T.getEmbedding(str(word2))
        numerator = np.dot(a, b)
        den1 = np.linalg.norm(a)
        den2 = np.linalg.norm(b)
        sim = (numerator) / ((den1) * (den2))
        print("Similarity [",str(word1),",",str(word2),"] = ", str(sim))
        line = word_file.readline().split(' ')

def SimilaritiesLinear(T):
```



```
    word_file = open("words.txt", "r")
    line = word_file.readline().split(' ')
    while line[0] != '':
        word1 = line[0]
        word2 = line[1].strip('\n')
        a = T.getEmbedding(str(word1))
        b = T.getEmbedding(str(word2))
        numerator = np.dot(a, b)
        den1 = np.linalg.norm(a)
        den2 = np.linalg.norm(b)
        sim = (numerator) / ((den1) * (den2))
        print("Similarity [",str(word1),",",str(word2),"] = ", str(sim))
        line = word_file.readline().split(' ')

if __name__ == "__main__":
    main()
```

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.