**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #5**

Due: November 5, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab, I implemented different functions using hash tables with chaining and hash tables with linear-probing. These functions were used in order to find the similarity between a pair of words. The purpose of this lab was to implement the two different types of hash tables in a way that compared two words to each other by their word-embeddings which consists of 50 floating point numbers. The similarity was then measured from very different to identical. Search algorithms, and Insert algorithms were used for both types of hash tables to aid in finding these similarities. I then compared the hash tables to the trees to see which implementation is the most efficient.

**Proposed Solution Design and Implementation**

**Hash Table with Chaining Implementation:** For the hash table implementation I began by creating a hash table with a fixed size. I then continued by reading the glove file that contains all the words and embeddings. While the file was read, I added each word and embedding to the hash table. I used an insert function that inserts items into a hash table into the correct place by getting the remainder of a division that divides the length of the word by the length of the table. After the hash table was created, I then read the file that contained the many pairs of words. Each word was searched for in the hash table. The similarity between the pairs were then computed by using the cosine distance which ranges from -1 to 1, -1 being very different, and 1 being very similar. While all of this was being done, I had a timer running to calculate the times required to build the hash table with chaining and to compute the similarities. After this I implemented the extra functions that work with ASCII codes, I prompted the user to insert a string and the different functions were performed on that one string.

**Hash Table with Linear Probing Implementation:** For the hash table with linear probing, I first created a hash table with a size that is the same as the amount of words in the word file. I did this since, in linear probing every single item has to be inserted into a separate index. Word embedding objects were then created while reading the text file that contained words along with their embeddings. I inserted these objects, then had the program read the file that contains pairs of words. Each word from each pair was then searched for within the hash table using a search algorithm that searches based on the word in each object. Once it was found, the embeddings were then used to compute the similarity between the two words. The similarity was calculated using the cosine distances between the two words. Meanwhile, a timer was running for the construction of the hash table, and while the word was being searched and the cosine distance was being calculated. After this I implemented the extra functions that work with ASCII codes, I prompted the user to insert a string and the different functions were performed on that one string.

**Experimental Results**

To test the program, I tested each function with different text files. The first text file contained 2 pairs of different words, the second one contained 10 pairs, and the last one contained 15 pairs of words. The efficiency of the two methods was found by timing the construction of the hash tables, and the query processing where the searches for the words were made. I compared the running times of insertion for both types of hash tables, and the search method for both.

**Test #1:**

The first test consisted of a text file with 2 different pairs of words. The size of the hash table with chaining was 3. The hash table

**Test #2:**

The second test consisted of a text file with 10 different pairs of words. The size of the hash table with chaining was changed to 6.

**Test #3:**

The third test consisted of a text file with 15 different pairs of words. The size of the hash table with chaining remained at 6.

**Running Times for creation of Hash Table:**

|  |  |  |
| --- | --- | --- |
| **Test** | **Chaining** | **Linear Probing** |
| 2 pairs of words | 959.04813 | 2 hours+ |
| 10 pairs of words | 15 mins+ | 2 hours+ |
| 15 pairs of words | 15 mins+ | 2 hours+ |

**Running Times for query processing:**

|  |  |  |
| --- | --- | --- |
| **Test** | **Chaining** | **Linear-Probing** |
| 2 pairs of words | 0.0654433 | 0.019942 |
| 10 pairs of words | 0.0167689 | 0.032287 |
| 15 pairs of words | 0.0550267 | 0.057820 |

**Running Times for construction of Trees:**

|  |  |  |
| --- | --- | --- |
| **Test** | **Binary Tree** | **B-Tree** |
| 15 pairs of words | 17.191797 | 47.33265 |
| 50 pairs of words | 34.645288 | 46.84082 |
| 100 pairs of words | 17.216610 | 20.61792 |



**Running Times for query processing:**

|  |  |  |
| --- | --- | --- |
| **Test** | **Binary Tree** | **B-Tree** |
| 15 pairs of words | 0.0151245 | 0.019942 |
| 50 pairs of words | 0.0167689 | 0.032287 |
| 100 pairs of words | 0.0550267 | 0.057820 |



Overall, for construction of the data structures, the trees were the fastest. The hash tables took much longer to construct. The linear probing especially took long to construct, it took over 2 hours to run completely. When it came to searching for the specific words, the hash tables were faster since there is more direct access to a word. The binary tree had the quickest running times compared to all the other implementations when it came to construction. The implementation that was the slowest during insertion was the hash table with linear probing. However, when it came to searching and computing similarities, the running times were very fast. The searching overall was very efficient since the word is found immediately in constant time in the hash tables.

**Chaining:**

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Running time with word “the” | Running time with word “computer” | Running time with word “standard” |
| Length of string % n | 0.0 | 0.0 | 0.0 |
| 1st character % n | 0.0 | 0.0 | 0.0 |
| Product | 0.0 | 0.0 | 0.0 |
| Sum | 0.0 | 0.0 | 0.0 |
| Load Factor | 0.0 | 0.0 | 0.0 |

**Linear Probing:**

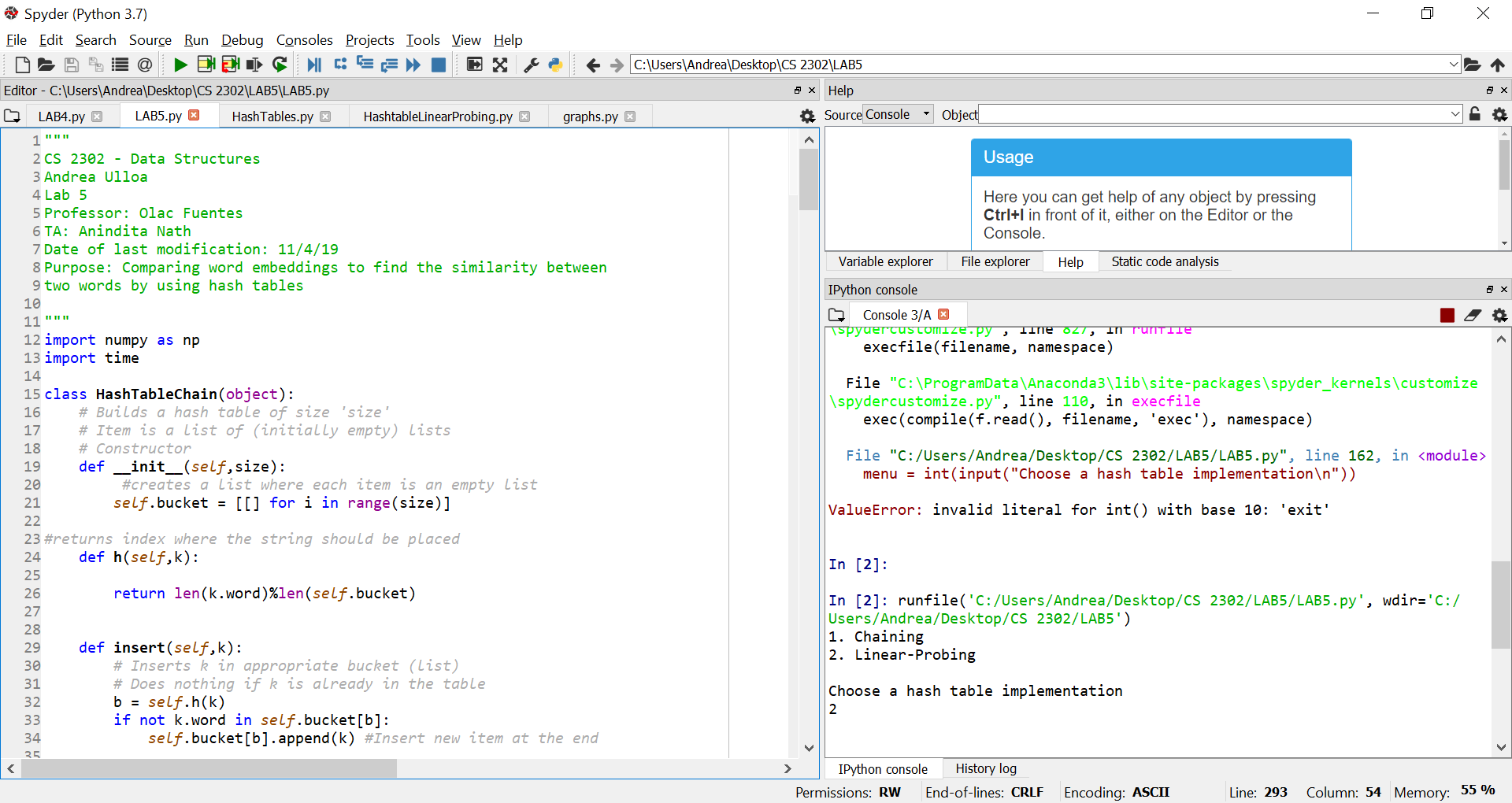
|  |  |  |  |
| --- | --- | --- | --- |
| Function | Running time with word “the” | Running time with word “computer” | Running time with word “standard” |
| Length of string % n | 0.0 | 0.0 | 0.0 |
| 1st character % n | 0.0 | 0.0 | 0.0 |
| Product | 0.0 | 0.0 | 0.0 |
| Sum | 0.0 | 0.0 | 0.0 |
| Load Factor | 0.0 | 0.0 | 0.0 |

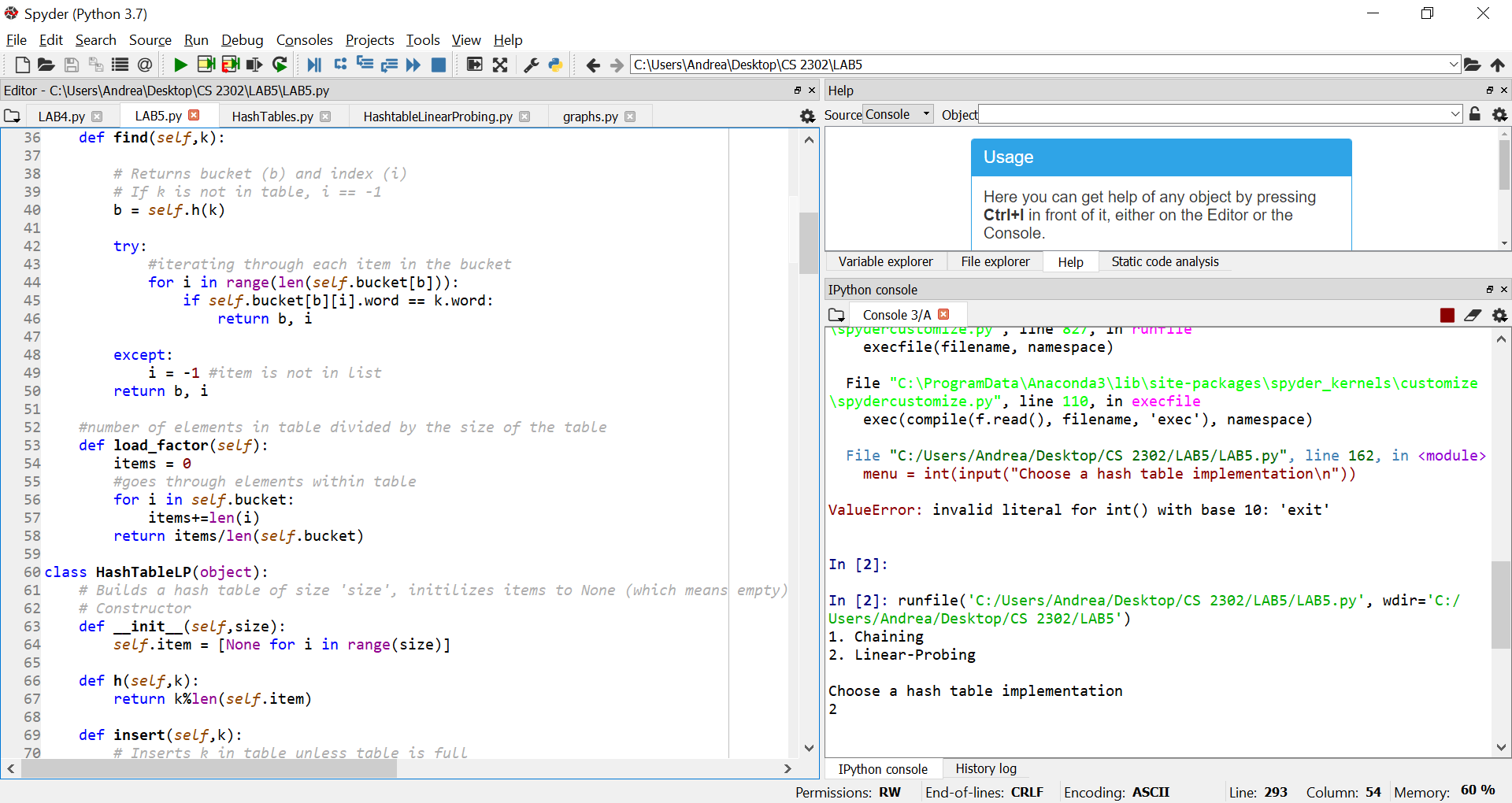
When testing the extra functions for the hash tables, they all took the same amount of time to run which was zero seconds. This shows that the efficiency of both hash table types is very good since it performs operations in constant time. This is due to the way that hash table items can be accessed directly without having to iterate.

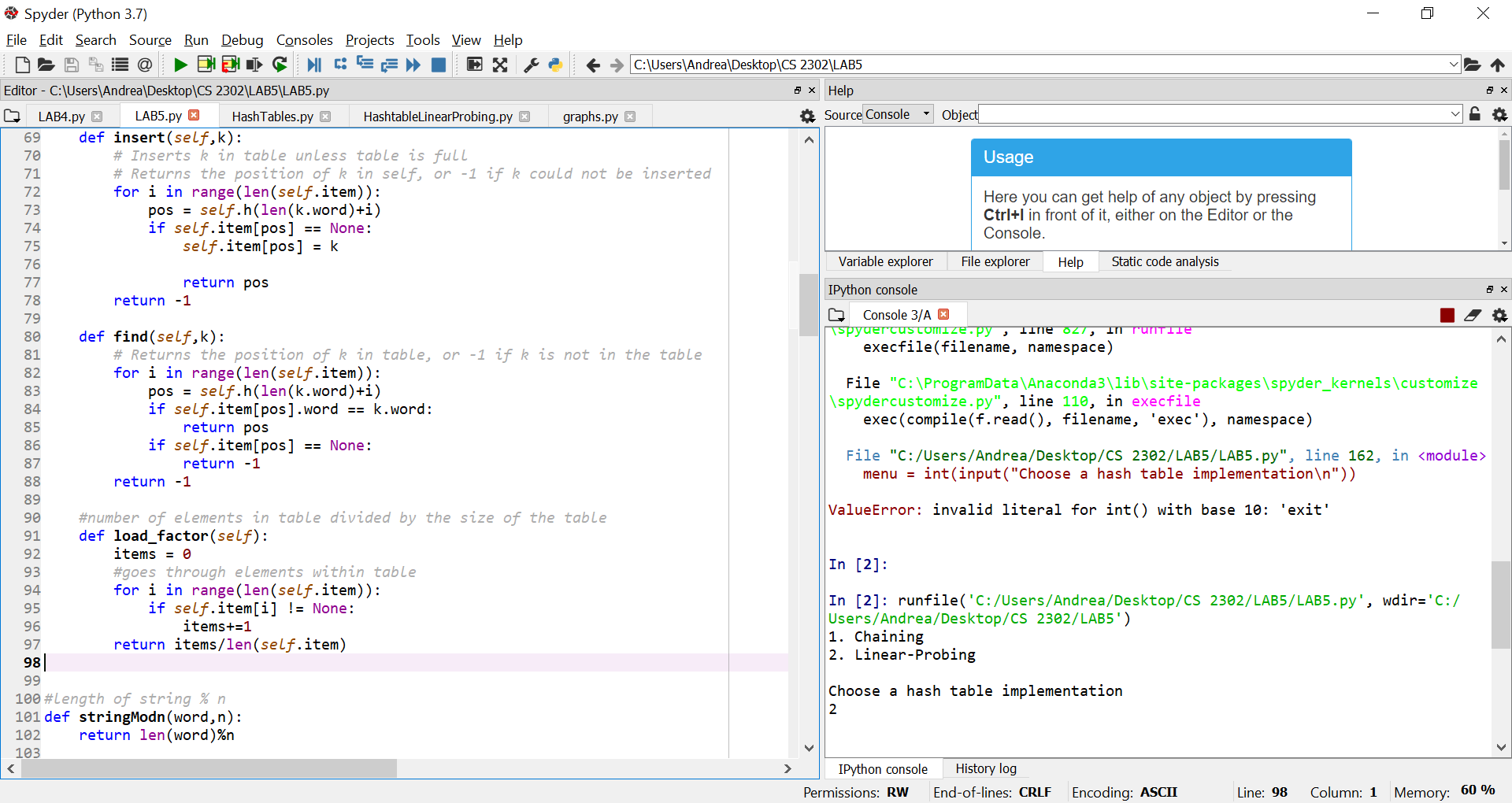
**Conclusion**

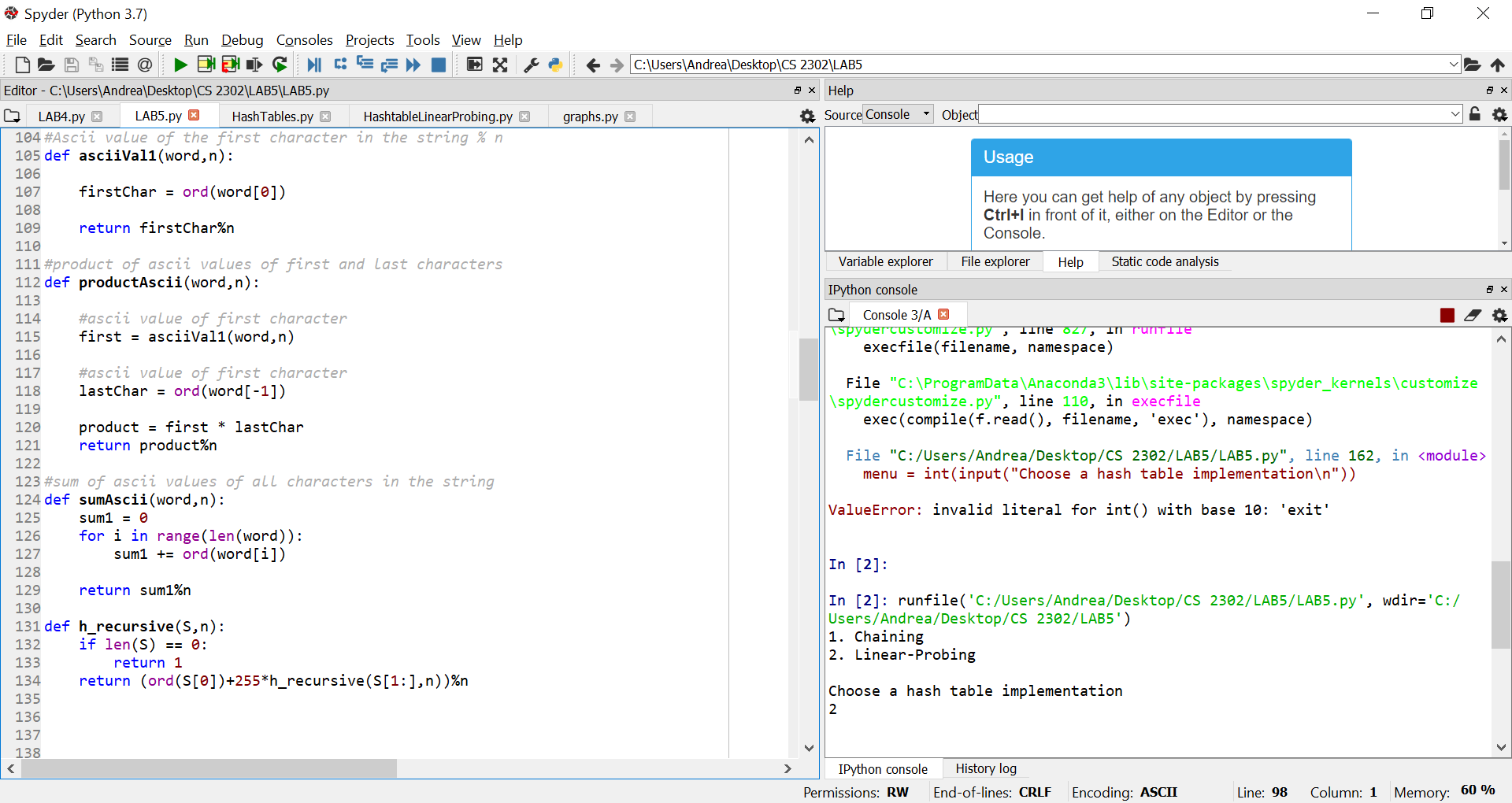
In this lab I learned more about how to implement the two different types of hash tables. I also learned about the efficiency of each type of hash table while performing different operations. Both types of hash tables perform similarly when searching for words. Increasing inputs determine the overall efficiency of the insertion and search functions for both types of hash tables, however they still take a longer time for construction. Overall, the most efficient implementation for insertion was the Hash table with chaining. When it came to searching, both hash table types were efficient in most cases. Another aspect of this lab that I learned more about were the different word embeddings that every word has. I learned that making comparisons of words is done by comparing the word embeddings by using the cosine distance. In addition to this, I also learned how to find ASCII codes for certain words and letters and perform different functions with ASCII numbers.

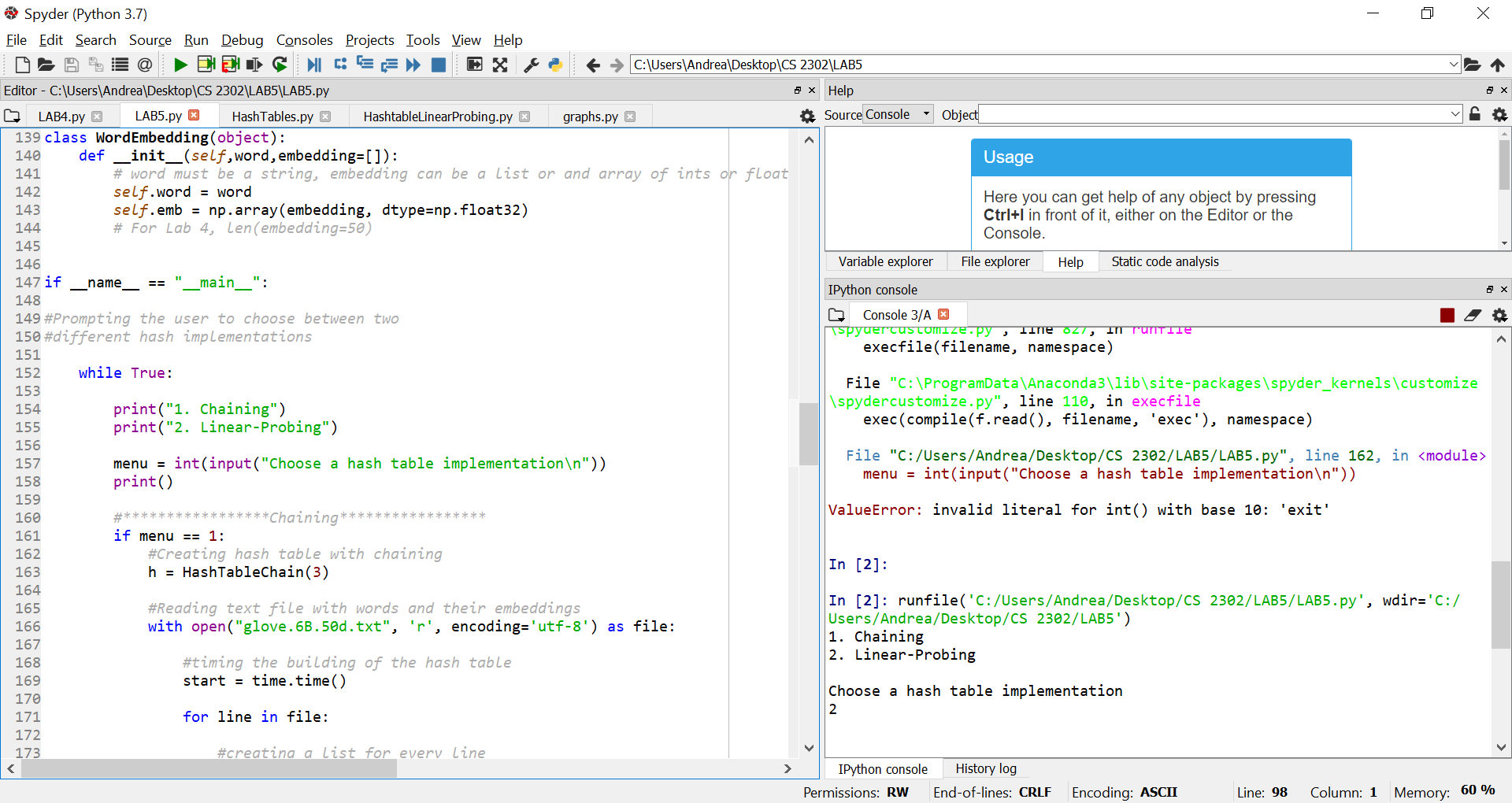
**Appendix**

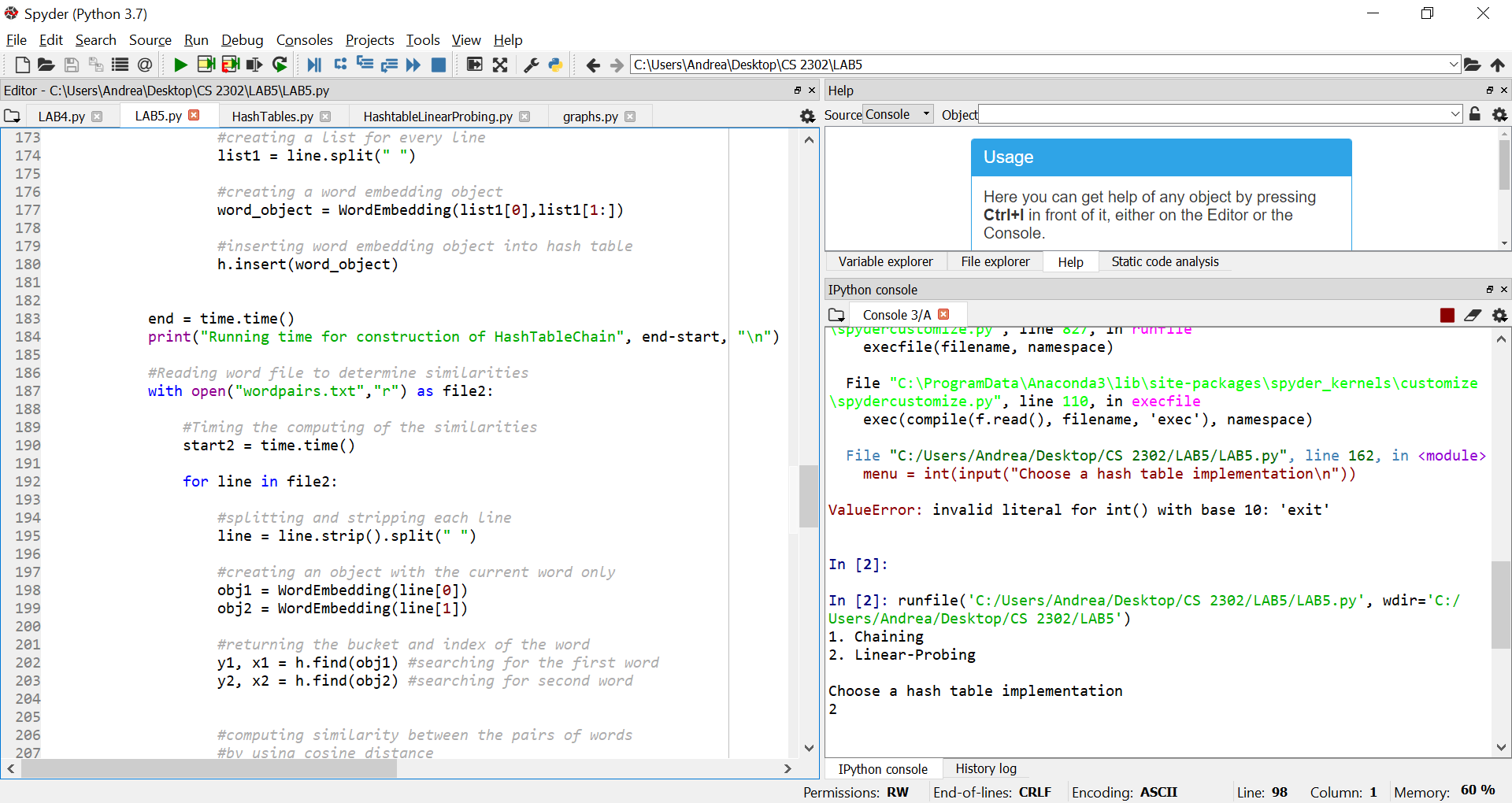


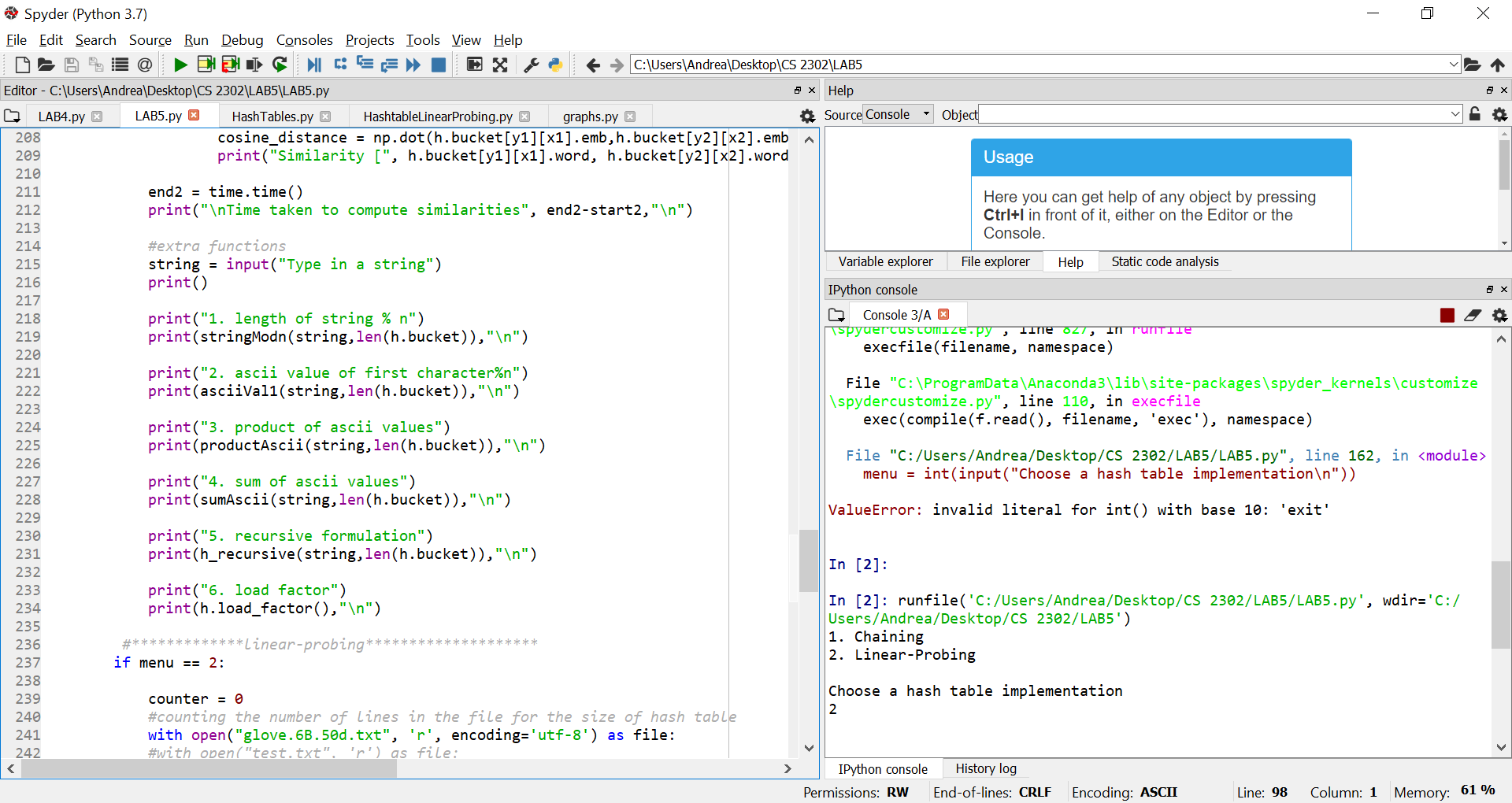


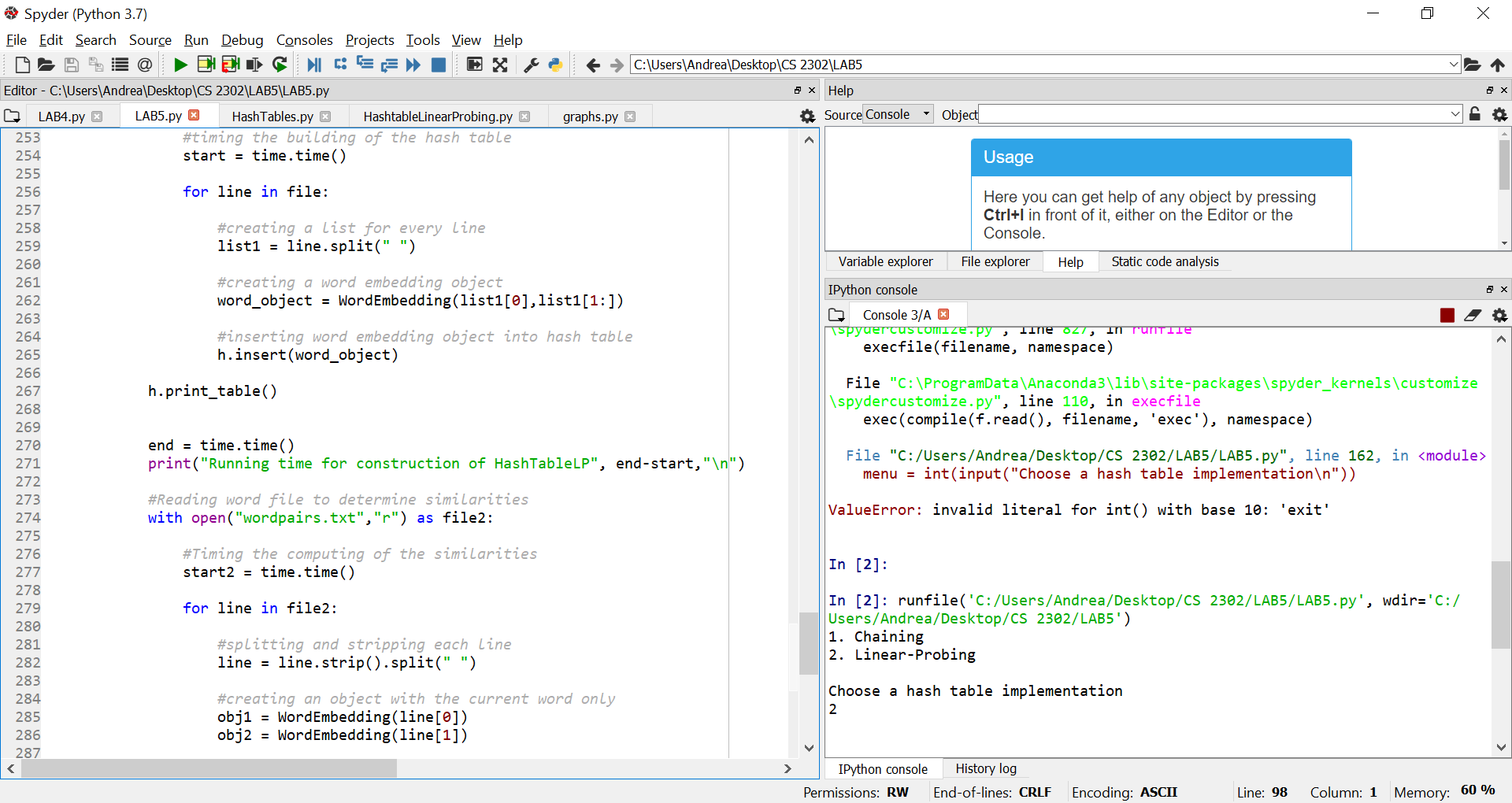


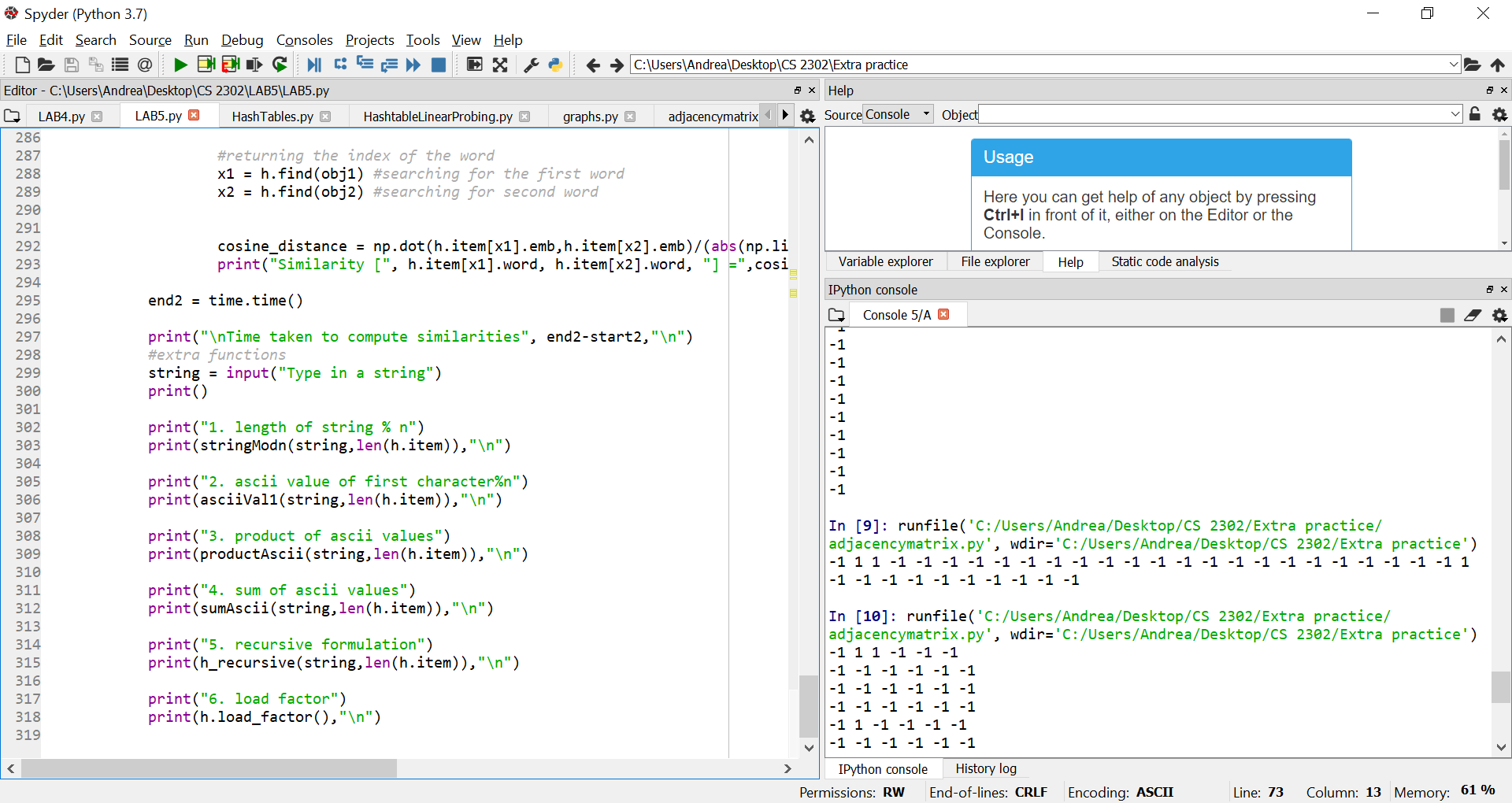












I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class