ALGORITHM AND ANALYSIS ASSIGNMENT 1

Samuele Andre Reyes (S3910311)
Fernaldy Winata (S3912660)

PART A:
For running our runtime experiments, we imported the time module from python and used the time() method to get the time at the beginning and end of each of our experiments. The difference between these gives us each of the runtimes for each of the datasets according to the specific functions.We take each of the runtimes and we graph them to see the runtime of array/linked list/trie's implementation of search, add, delete and autocomplete. We do the runtime for each 10 times and take the average to increase the reliability of our values. We also used our own inputs to test out the runtime of each code.

Our hypothesis is derived from analyzing our code that we implemented in the data structure files. We also hypothesized that for all the functions and for each type of data structure, the runtime should generally increase in nature, as the size of the data sets increases. As for which data structure is the fastest, based on our code we hypothesized that Trie would be the fastest followed by Array and then LinkedList.

Below is our hypothesis.

Time Complexity	Array	Linked List	Trie
A	O(n)	O(n)	O(g)
D	O(n)	O(n)	O(g)
S	O(n)	O(n)	O(g)
AC	O(n^2)	O(n)	O(n^2)

Array data structure -
In the array data structure, the Add, Search and Delete functions all have the same linear time complexity of O(n) where n is the size of the input/dictionary of words. This means that the running time only increases with the size of the input data. The 3 functions first iterates through the whole array to search whether the word exists in a single loop, which then returns True or False depending on whether the word exists in the array. The add and Delete functions will then add or delete the input word depending on the results while the search function returns the requested word if it exists.

The auto-complete function however, has a time complexity of O(n^2), a quadratic time complexity. After searching for the words that contain the prefixes of the input word with the python class of startswith and appending the results into a new array, we then sort the prefix array with a nested loop based on the frequency in descending order. Therefore, we hypothesized that the auto-complete function has a time complexity of O(n^2).

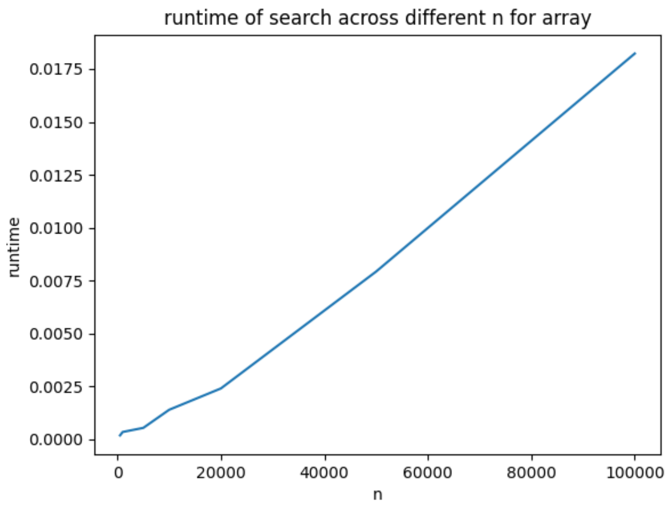
Linked List data structure -
The Add, Search, delete and auto-complete function will have a linear time complexity of O(n) where n is the size of the input/dictionary of words. This is because for all the cases, they will have to iterate

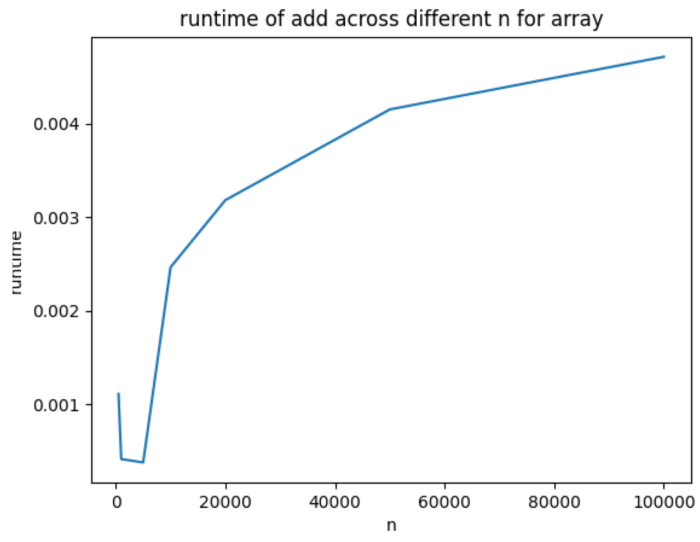
through the entire linked list to search for the input word or prefix, and if the word or prefix exists, they will either add a new word, return the word or delete the word respectively. In the case of the auto-complete function, after searching for the prefix with python's startswith(), it will proceed to append it to a new array which contains the list of the nodes/objects of matching words. After that, it will sort the words in the array based on the highest frequency and return the top 3 words. The time complexity stays as O(n) as the sorting method time complexity O(nlogn) is still smaller than the time complexity of O(n)

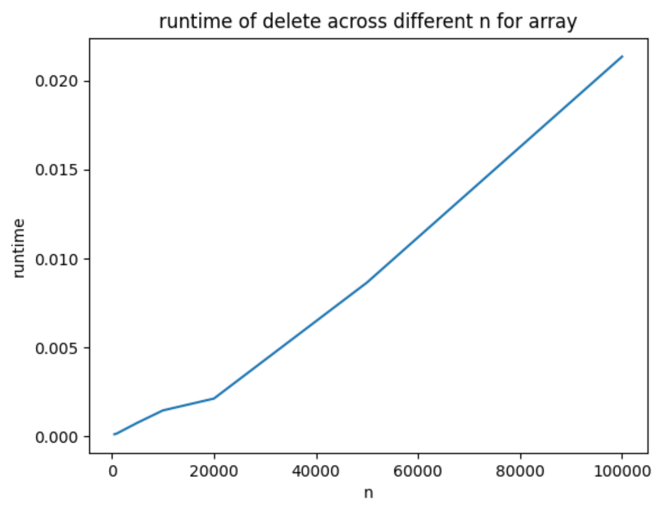
Trie data structure -
The Trie data structure works differently than the other 2. Similarly to before, the Add,Search and Delete functions will have the same time complexity as they are first required to search whether the words exist or not. Hence and because of how the search function works in a trie data structure, the time complexity of the Add, Search and Delete functions would be O(g) rather than O(n), where g represents the amount of letters in the input.
Therefore in theory, the time complexity should not be affected by n because the search function traverses down the nodes rather than iterating through the entire dictionary of words.

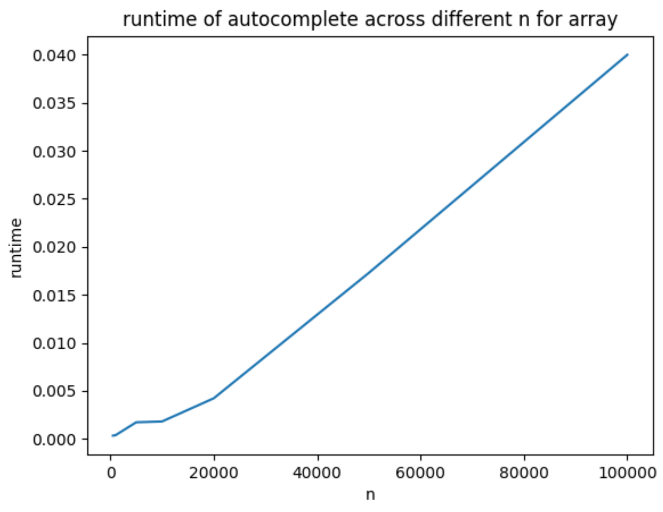
As for the auto-complete function, it will have a time complexity of O(n^2) as we used 2 loops in the implementation. The complexity of O(n^2) comes from sorting through the words in descending order based on frequency after searching and getting the matching words. The words are then appended into a new and empty array where n will represent the total number of words in this array.

Part B:



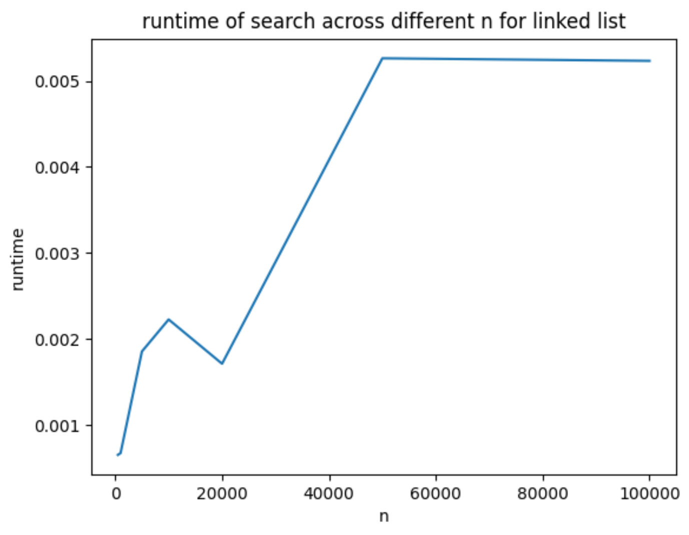


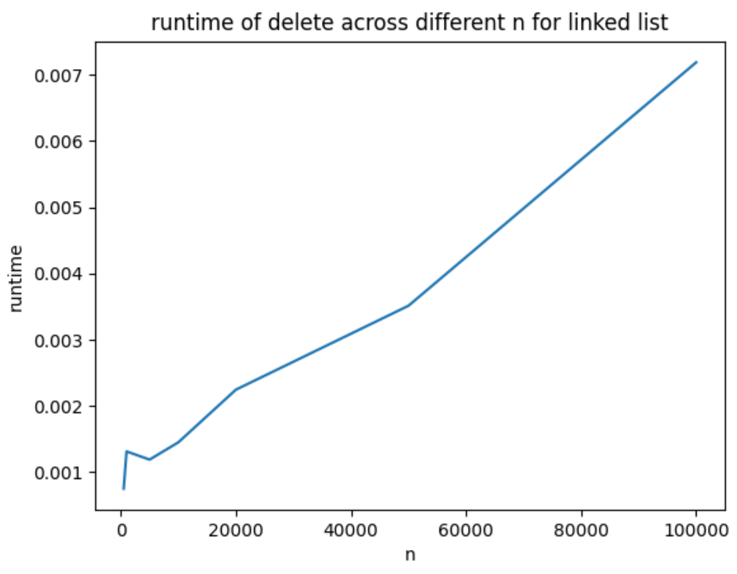


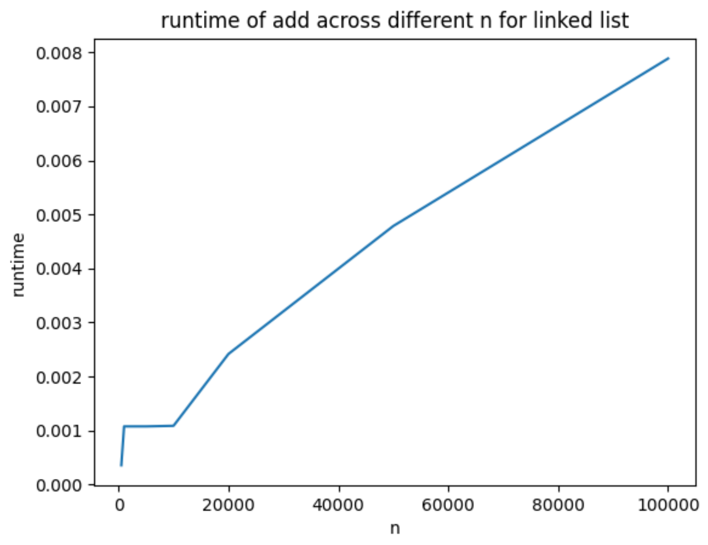


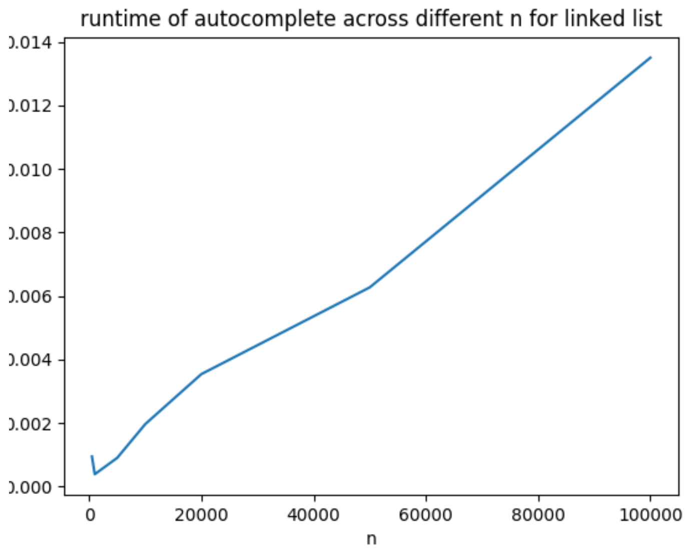
Array conclusion -

After observing the graphs, we can conclude that our hypothesis has been supported except for a few anomalies. Firstly, the add function has a dip at the start of the test and also starts to look similar to a logarithmic graph even though towards the end it seems to increase in trend generally. Secondly the auto-complete looks more like a linear graph rather than a quadratic graph. The 2 things we could derive from this was that we did not do enough tests hence the amount of data is too small to determine whether it is supposed to be quadratic and that because we used the same inputs,



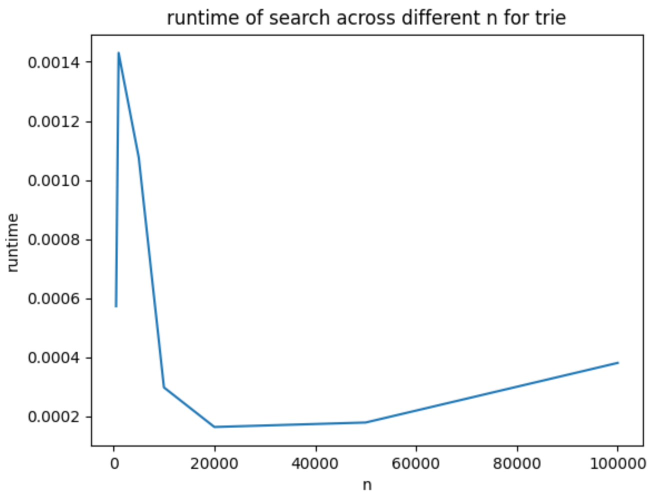


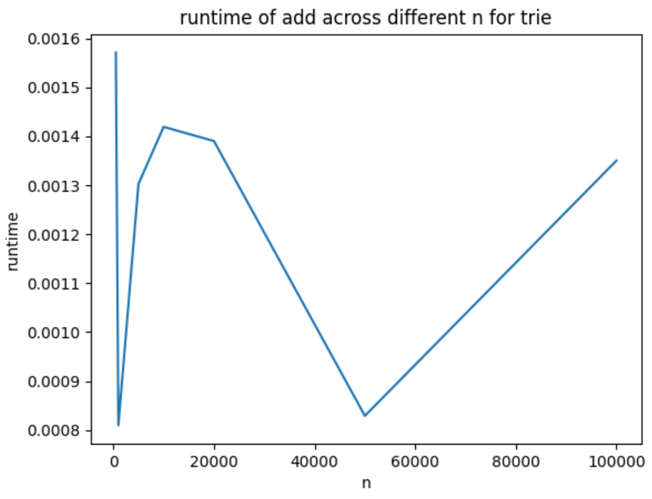


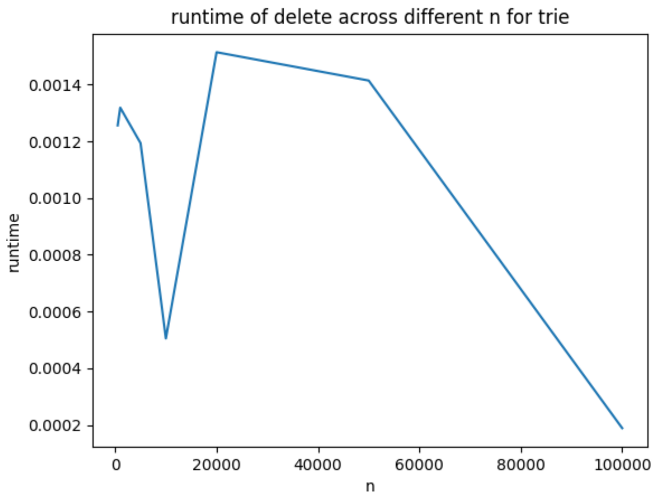


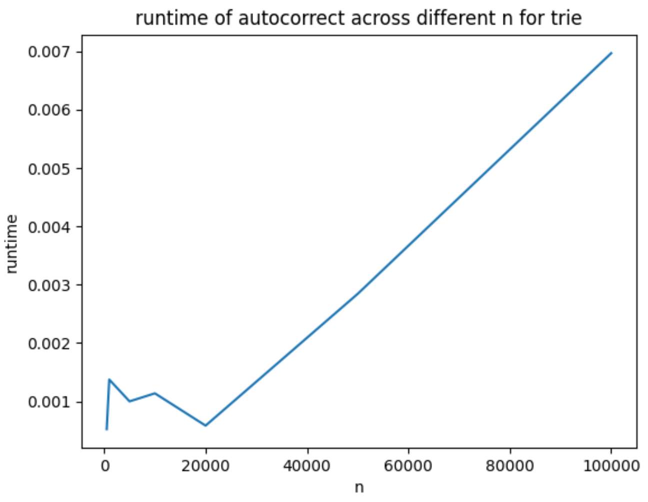
Linked List conclusion -

As per our hypothesis, the graphs have all increased in general trend as a linear graph of O(n). However, there are anomalies at the start of the graph and we assume that it is due to fluctuations of the CPU. As for the search function, we think that it makes sense about it following the general trend of a linear graph, but we are unable to make sense of why it is a straight line at the top.



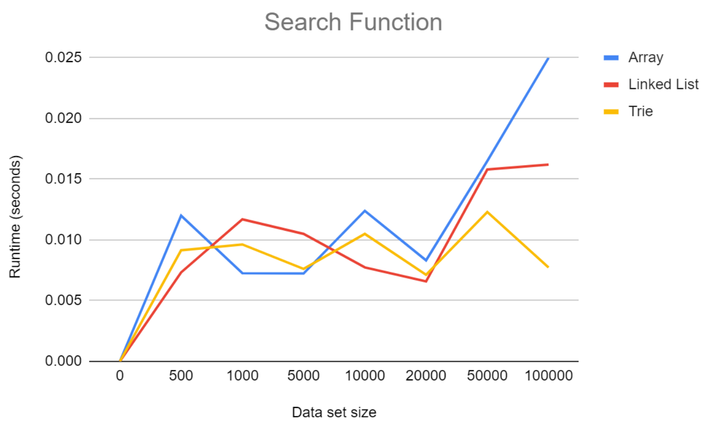


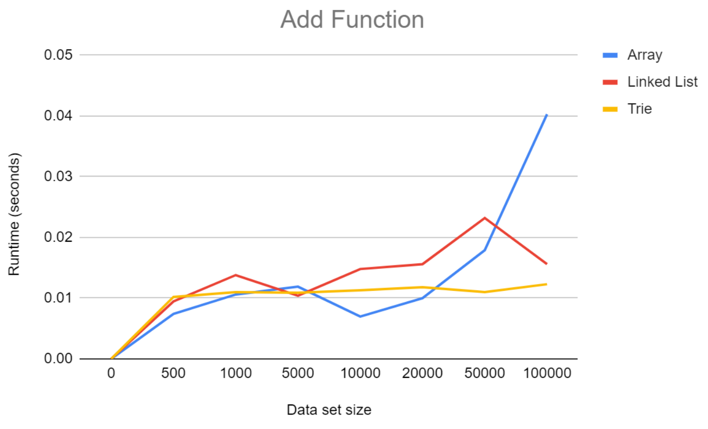


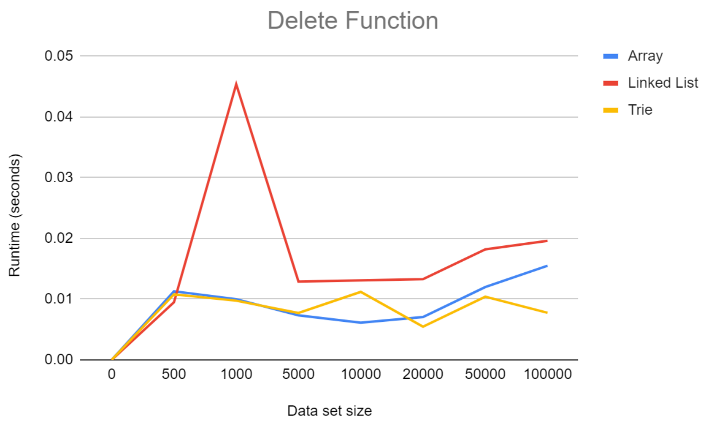


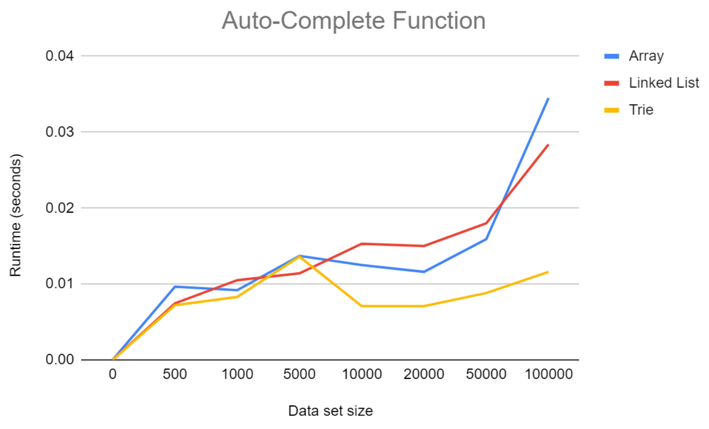
Trie conclusion -

For the trie graph results we have, it matches with our hypothesis that the runtime is not affected by n (where n is the dictionary of words) but g (where g is the length of the input) instead. The auto-correct graph has the issue of CPU fluctuation at the start and then turns into a linear graph, we also deduced here that we did not run the test enough times to properly show it as a quadratic graph. However, the runtime does generally increase which is what we are looking for.









By comparing all the runtime of each function of the 3 data structures above, we can safely deduce that trie is indeed the fastest, followed by linked list and array being the slowest. Although there is an anomaly in the delete function where it spiked, we believe that it is due to a fluctuation in CPU processing

In conclusion, we believe that we are on the right track, however, we could have done this experiment with better variables and consistency. To make it a better experiment, we can run more tests and get better average results to make sure that the quadratic graphs or theoretical quadratic runtime complexities such as the auto-complete for Trie and Array appear properly. Since we are using the same inputs for our current tests, we could also use the random module to generate a random list of words for the inputs for our test to make sure that it is not biased.