# **COSC 2123/1285 Algorithms and Analysis**

# **Assignment 1: Word Completion**

Group 50

Student 1: s3939713 - Wing Hang Chan

Student 2: s3863706 - Richard Joseph Doherty

# Task B: Empirical Analysis Report

## Data Generation and Experiments Setup

Datasets of various sizes were generated with randomly selected data. Six datasets were generated in total ranging from small (500, 1000), to medium (5000, 10000), to large (50000, 100000).

**Sample Data Generation:**

The sample data sets are generated by a program which randomly picks the rows of the 200k sample data set from the assignment. The program will randomly select the size of N which is a parameter from calling the program and it is the size of the output. The program generates N numbers which are in the range of 1 to 200,000. The numbers generated are the line numbers. They will then be put into a set removing duplicate numbers. After that, we will get a list of numbers with lengths that may not be equal to N. Therefore, the program will then keep looping the random numbers function with the difference between distinct line number lists and N until it gets a list without duplicates numbers and equal to N. After the length of the distinct list and N are equal, it will then extract the lines from sampleData200k.txt by the list.

It is totally randomized sample data from the predefined dataset. And a random data set which means there is an equal chance of being selected. The empirical analysis then would not be biased by some extreme data.

**Empirical Data Generation:**

There are 2 python files in the folder “changed\_code\_for\_PartB”. As stated in the README.txt, copying the 2 files in the root path of the folder for replacing the original files. Then, running the program same as assignment requirement. It will then generated <data\_in\_filename>-<type>.out .

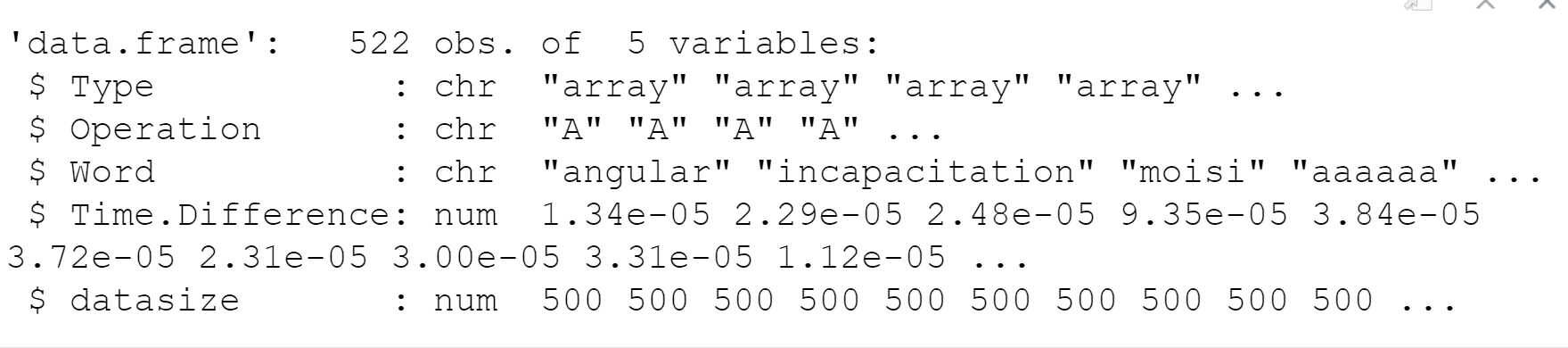
|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Factor** | **Description** |
| Type | str | {“array”, “linkedlist”, “tire”} | Type defined in assignment |
| Operation | str | {“A”, “S”, “AC”, “D”} | Operation defined in assignment  A – Add  S – Search  AC – Autocomplete  D - Delete |
| Word | str |  | Any words with the operation |
| Time Difference | float |  | Start time – End time for each operation |

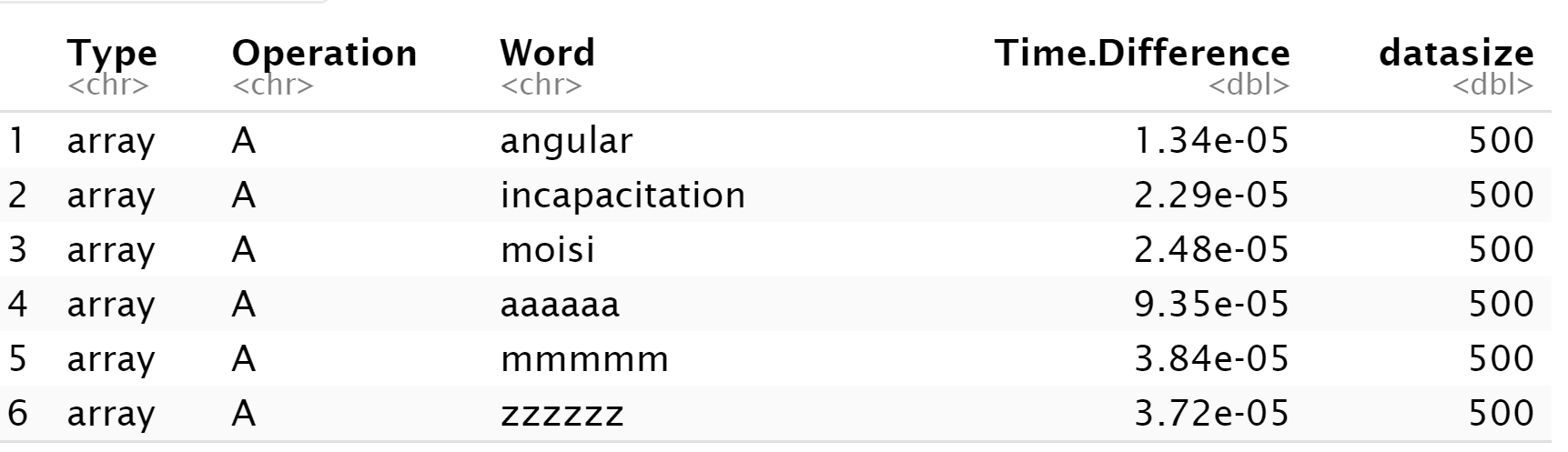
Data and files are combined with an excel file for next step.

## Evaluation/Analysis of the outcome using the generated data

After capturing the data for runtimes across the six separate datasets, data from the six separate datasets were combined into a single dataframe for further analysis.

Structure of the dataframe as follows:

****



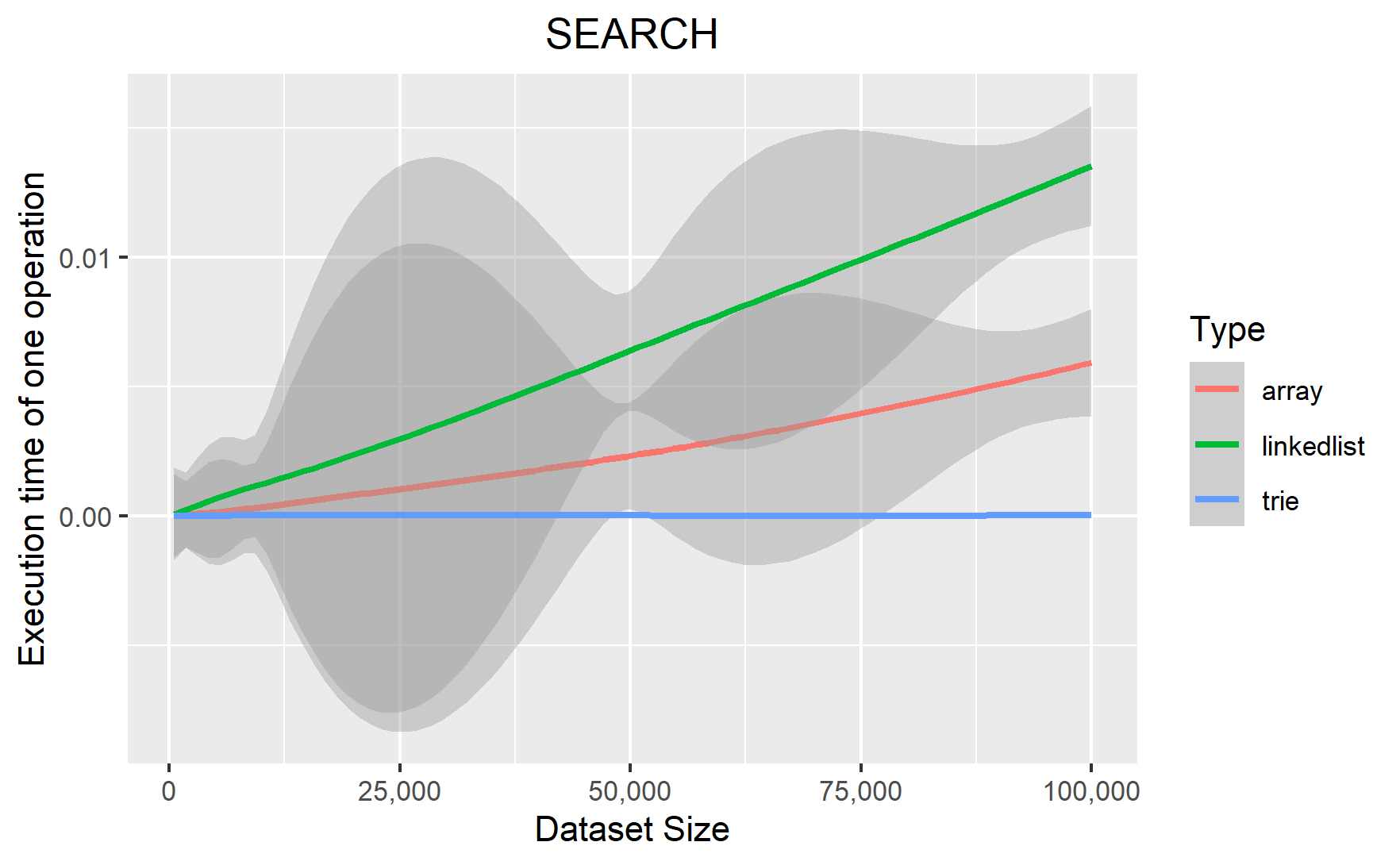
Graphs that follow compare the performance of each data structure (Array, LinkedList, Trie) across the full set of generated data and visualises the running time captured for each of the test runs as well as the overall trends across the full set of data for each operation (Search, Add, Delete, AutoComplete).

For each of the visualisations/graphs that follow:

***X-axis*** *corresponds to dataset size*

***Y-axis*** *is the running time of one single operation (example: ADD). In general, the larger the dataset, the more time to run a single operation.*

### Search operation

**

#### Array

The implementation for array is N-size of the dataset and will return the result if the program matches the result. Only one comparison for each loop of N times. So, the time complexity is O(n).

|  |
| --- |
| *for* word\_freq *in self*.arr:  *if* word\_freq.word == word:  *return* word\_freq.frequency return 0 |

#### Linkedlist

The implementation for search operation in linkedlist is like array. It has a time complexity of O(2n) as it takes two comparisons for each loop. For simplicity, its time complexity is also O(n). But it has double time compared with array.

|  |
| --- |
| cur\_node = *self*.m\_head for i *in range*(*self*.m\_length):  *if* cur\_node.word\_frequency.word == word:  *if* cur\_node.word\_frequency.frequency > 0:  *return* cur\_node.word\_frequency.frequency  *else*:  *return* 0  cur\_node = cur\_node.next return 0 |

#### Trie

It seems like there are many more comparisons than with array and linkedlist. But it has a time complexity of O(logmn) for which m is equal to the number of alpha characters in the list. If the list only accepts a-z and the list has a-z of words, then the size of m is 26. “*cur\_node.children.get(word[word\_idx])*” uses a dictionary as assignment requirement for storing the children. It stores up to 26 keys if it uses lower letter only. ***Trie runs faster than array and linkedlist as it is O(logmn).***

|  |
| --- |
| *while* cur\_node:  next\_node = cur\_node.children.get(word[word\_idx])  isLast = *len*(word) == (word\_idx + 1)   *if* next\_node:  *if* isLast:  *if* next\_node.is\_last:  *return* next\_node.frequency  *else*:  *return* 0  *else*:  cur\_node = next\_node  word\_idx += 1  *else*:  *return* 0 |

### Add Operation

*Chart

Description automatically generated*

#### Array

There is a requirement that does not allow duplicate records. Therefore, the program will call search to check if there is the same value within the array (*self.search(word\_frequency.word)*). If there is the same value, it will return directly. Otherwise, it will insert the value for looping once more. Therefore, the search algorithm of array O(2n) takes part of it. The add performance of the array is still O(n). For average, it is q (n+n/2).

|  |
| --- |
| freq = *self*.search(word\_frequency.word) if freq == 0:  i = 0  *for* word\_freq *in self*.arr:  *if* word\_frequency.word > word\_freq.word:  i += 1  *break*  *self*.arr.insert(i, word\_frequency) |

#### Linkedlist

Add operation in linkedlist has a complexity of O(n) for all non-duplicated cases. If it is duplicated case, it is hard to calculate, as linkedlist is not sorted, we do not know where the data stored. It has to loop (the while loop) all the linkedlist to confirm the data not duplicated So, it has a O(n) and the average case of add a non-duplicated is also q (n). Therefore, the graph of linkedlist is still higher than array.

|  |
| --- |
| cur\_node = *self*.m\_head last\_node = *None*  *while* cur\_node:  *if* cur\_node.word\_frequency.word == word\_frequency.word:  *return False*   *if* cur\_node.next:  cur\_node = cur\_node.next  *else*:  last\_node = cur\_node  cur\_node = *None*  *add\_node* = ListNode(word\_frequency) if last\_node:  last\_node.next = add\_node else:  *self*.m\_head = add\_node self.m\_length += 1  return True |

#### Trie

Like array, trie add operation calls its search operation which takes O(logmn). And it then takes another O(logmn) for inserting a node. Therefore, it takes O(2logmn). By implication, it takes O(logmn). The logic of add operation is like search. But it has 2 more comparisons for checking if the node has frequency or has the next node.

|  |
| --- |
| is\_exist = *self*.search(word\_frequency.word) != 0 if not is\_exist:  cur\_node = *self*.root  word\_idx = 0  *while* cur\_node:  next\_node = cur\_node.children.get(word\_frequency.word[word\_idx])  ... |

### Delete Operation

*Chart

Description automatically generated*

#### Array

It shares the same algorithm with add operation but use list.pop instead of list.insert at the end. Refer to python.org, list.pop (intermediate) takes O(n) which is the same as list.insert.

|  |
| --- |
| freq = *self*.search(word) if freq > 0:  i = 0  *for* word\_freq *in self*.arr:  *if* word == word\_freq.word:  *break* i += 1  *self*.arr.pop(i)   *return True* *else*:  *return False* |

#### Linkedlist

Implementation is the same as array. It shares the same algorithm with add operation of linkedlist but it is just pointing the node from one to another. Python will do the garbage collection for the deleted node.

|  |
| --- |
| cur\_node = *self*.m\_head prev\_node = *None*  *while* cur\_node:  *if* cur\_node.word\_frequency.word == word:  *if* prev\_node: |

#### Trie

As with the search operation, the time complexity is O(logmn). The difference between add and delete algorithm is the operation of creating a node and setting the node value frequency and is\_last or removing the node.

|  |
| --- |
| is\_exist = *self*.search(word) != 0 if is\_exist:  cur\_node = *self*.root  word\_idx = 0  *while* cur\_node:  next\_node = cur\_node.children.get(word[word\_idx])  ... |

### Auto-complete Operation

*Chart, line chart

Description automatically generated*

#### Array

The time complexity of auto complete is O(n). It also takes a sort with default python list sort (Tim Sort[1]) which has an O(m log m) for which m is equal to the size of return\_arr. For simplicity, it still takes O(n).

|  |
| --- |
| return\_arr = [] for word\_freq *in self*.arr:  prefix\_word\_len = *len*(prefix\_word)  *if* word\_freq.word[:prefix\_word\_len] == prefix\_word:  return\_arr.append(word\_freq)  return\_arr.sort(key=*lambda* x: x.frequency, reverse=*True*)  return return\_arr[:3] |

#### Linkedlist

The time complexity of auto complete is O(n). It is like an array. But it is O(2n) as there are 2 comparisons. It also takes a sort with default python list sort. It takes O(2n + m log m) For simplicity, it still takes O(n). Compared with arrays, linkedlist takes more time on average as it is not sorted.

|  |
| --- |
| return\_arr = [] cur\_node = *self*.m\_head  while cur\_node:  word\_len = *len*(word)  *if* cur\_node.word\_frequency.word[:word\_len] == word:  return\_arr.append(cur\_node.word\_frequency)   *if* cur\_node.next:  cur\_node = cur\_node.next  *else*:  cur\_node = *None*  *return\_arr*.sort(key=*lambda* x: x.frequency, reverse=*True*)  return return\_arr[:3] |

#### Trie

The time complexity of auto complete is O(logmni) where i is equal to length of word which is input for autocomplete. The method “*add\_word\_frequency\_with\_parent\_node*” will call itself for looping the i times. As the number of m is usually greater than i. For simplicity, it would be O(i logmn) and to be simplified as O(logmn).

|  |
| --- |
| return\_arr = [] cur\_node = *self*.root word\_idx = 0 last\_node = *None* *str\_prefix* = "" while cur\_node:  next\_node = cur\_node.children.get(word[word\_idx])  *if* next\_node *and* next\_node.letter:  str\_prefix += next\_node.letter   is\_end\_suggest\_word = *len*(word) == (word\_idx + 1)   *if* is\_end\_suggest\_word:  last\_node = next\_node  *if* last\_node:  *if* last\_node.is\_last:  return\_arr.append(WordFrequency(word, next\_node.frequency))  TrieDictionary.add\_word\_frequency\_with\_parent\_node(return\_arr, last\_node, str\_prefix)  cur\_node = *None*  *else*:  *if* next\_node:  cur\_node = next\_node  word\_idx += 1  *else*:  cur\_node = *None* *return\_arr*.sort(key=*lambda* x: x.frequency, reverse=*True*)  return return\_arr[:3]  @staticmethod def add\_word\_frequency\_with\_parent\_node(arr: [WordFrequency], parent\_node: TrieNode, parent\_word: *str*):  word = parent\_word  *for* child\_node *in* parent\_node.children.values():  *# print("add\_word\_frequency\_with\_parent\_node2: {}".format(arr))*  *if* child\_node:  *if* child\_node.is\_last:  arr.append(WordFrequency((parent\_word + child\_node.letter), child\_node.frequency))   *if len*(child\_node.children) > 0:  w = word + child\_node.letter  TrieDictionary.add\_word\_frequency\_with\_parent\_node(arr, child\_node, w) |

## Summary & Recommendations

**Search Operation:** Trie runs faster than array and linkedlist as it is O(logmn).

**Add Operation:** The logic of add operation is like search. But it has 2 more comparisons for checking if the node has frequency or has the next node. Similar to array, **Trie** add operation calls its search operation which takes O(logmn). And it then takes another O(logmn) for inserting a node.

**Delete Operation:** As with the search operation, Trie gives the best performance with time complexity of O(logmn).

**Auto-Complete Operation:** The time complexity of auto complete for **Trie** is again preferred with O(logmni) where i is equal to length of word which is input for autocomplete. The method “add\_word\_frequency\_with\_parent\_node” will call itself for looping the i times. As the number of m is usually greater than i. For simplicity, it would be O(i logmn) and can be simplified as O(logmn)

## Reference:

[1] TimSort: Algorithm and Implementation in Python

- https://www.pythonpool.com/python-timsort/