Task B: Empirical Analysis Report

Student Name	Student No.
Matthew Bentham	S3923076
John Murrowood	S3923075

B.1 Data generation and experimental setup

Data Source: https://github.com/dwyl/english-words
Original dataset source: 350,000+English words dataset
Description: text file containing a list of 370105 English words with only letters (no numbers or symbols)

In this section of the report, the three data structures and five methods implemented in each (construction, addition, search, deletion and autocompletion) with be compared using an empirical analysis of their respective time complexities. The three data structures compared in this report are:

- **Sorted array-based dictionary**: words and their respective frequencies are sorted alphabetically and stored in a python array object.
- **Linked-list-based dictionary:** words and their respective frequencies are sorted randomly in a single linked list (data structure where nodes track a value and the position of the next node in the chain)
- **Tire-based dictionary:** letters of each word are stored as nodes, with frequency being stored in the final node of each word.

For data generation, the above data source was used to generate seven random samples of varying sizes (500, 1000, 5000, 10000, 20000, 50000, and 100000) all of which are random subsamples of the larger samples (e.g., 50000 was sample from the 100000 sample etc.) to reduce overall variation in datasets so that size can be the main differentiating independent variable. Additionally, to generate the final datasets for testing a random number between zero and the size of the dataset was used as the word frequency value for each generated word. This was done to maintain zero bias and/or relationships between words and their frequencies.

Aside from size, the other independent variable used for our time efficiency analysis were the parameters used to test the data structures. To evaluate all parameter inputs individually (construction, addition, search, deletion and autocompletion), four input files were generated:

- **Testadd.in:** 50 lines all containing the letter ‘A’ to indicate addition, a random word sampled from the 370105 English words dataset and a random integer between 1 and 100000 to indicate frequency.
- **Testsearch.in:** 50 lines all containing the letter ‘S’ to indicate the search command, followed by a random word sampled from the 370105 English words dataset.
- **Testdel.in:** 50 lines all containing the letter ‘D’ to indicate the delete command, followed by a random word sampled from the 370105 English words dataset.
- **TestAC.in:** 50 lines all containing the letter ‘AC’ to indicate the search command, followed by a random prefix obtained by extracting the first half ($length\ of\ word - \frac{1}{2} length\ of\ word$) of a word sampled from the 370105 English words dataset.

To also test for the accumulative time efficiency of each structure when presented with all commands, an additional input file was generated (testfull.in). The file contains 100 lines with an equal amount of each command type (25 lines each) that was randomly generated using the above techniques.

To compare the algorithm complexity, we chose to measure the time it takes for each of the data type algorithm to run each parameter input file. To get the time, the algorithms were run on the RMIT teaching server for consistent computational capabilities and time command from the python time library was used to the difference in start and end times after each input. To reduce the effect of outliers and noise each input for each datatype was run 10 times and resultant time output was calculate by averaging the difference in start

and end times of all runs. to get the time correct to eight decimal places. Each command type (search, add word, delete, autocorrect, and combines) is run ten times and then the average time is taken for the results and graphs. The final run times were recorded to eight decimal places.

Results

Algorithm	*Data set size (words)*	*Addition*	*deletion*	*search*	*Autocorrect*	*all*
Array	500	0.02203164	0.02193327	0.02372775	0.0249809	0.02583385
Linked list	500	0.02285197	0.02882547	0.02894804	0.03151224	0.04062128
Trie	500	0.02345359	0.02130096	0.02122536	0.02295244	0.02395816
Array	1000	0.02340529	0.0234431	0.02390018	0.02526271	0.02627835
Linked list	1000	0.02548714	0.03614507	0.03594389	0.04251502	0.0497982
Trie	1000	0.02399845	0.02357018	0.02368112	0.02851083	0.0273555
Array	5000	0.02416737	0.02377417	0.02376525	0.03798699	0.03459849
Linked list	5000	0.0308516	0.08682926	0.08786736	0.10936525	0.13787234
Trie	5000	0.02210176	0.02213399	0.02218285	0.03340728	0.03170731
Array	10000	0.02569993	0.02577929	0.02361987	0.0588944	0.04678495
Linked list	10000	0.03686814	0.13883674	0.13820729	0.17948923	0.22549536
Trie	10000	0.02540679	0.02404532	0.02394917	0.04479749	0.0419383
Array	20000	0.02369866	0.02198887	0.02307191	0.11725302	0.07456689
Linked list	20000	0.05053988	0.25739565	0.24664268	0.33559561	0.41453545
Trie	20000	0.02615979	0.02607694	0.02556462	0.07573788	0.0610225
Array	50000	0.02413802	0.02441823	0.02386341	0.44699538	0.23209164
Linked list	50000	0.13016057	0.61661091	0.59440999	0.81303988	1.05502758
Trie	50000	0.02418072	0.02479043	0.0247618	0.15496087	0.10789707
Array	100000	0.02692742	0.02530375	0.02632957	1.55440056	0.69532056
Linked list	100000	0.33209376	1.20348887	1.0020308	1.58821423	2.04686997
Trie	100000	0.02471604	0.02318826	0.02324593	0.30010819	0.20730546

Table 1.

fig 1.

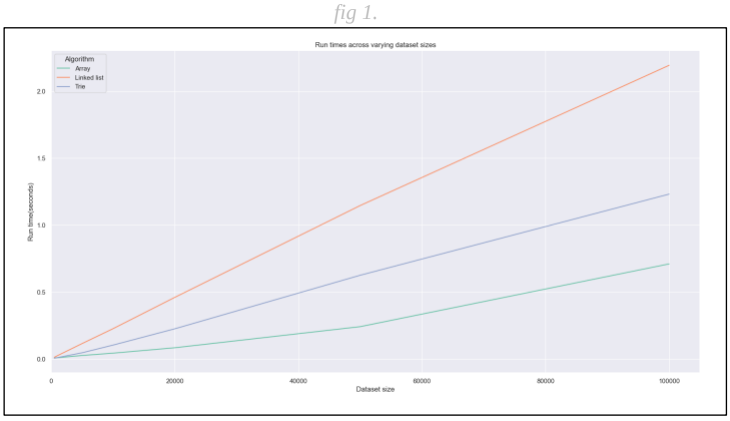
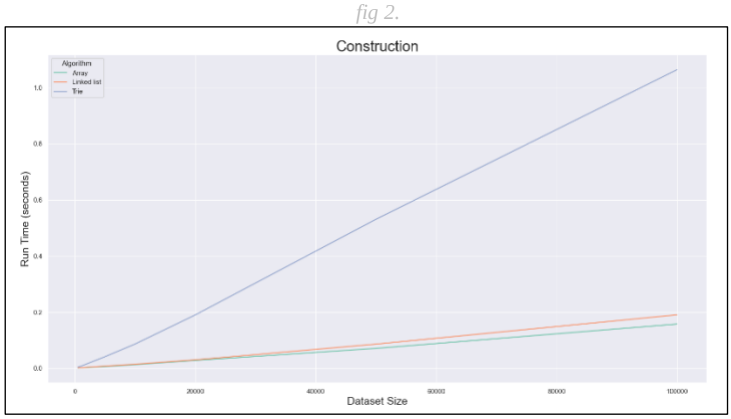


fig 2.



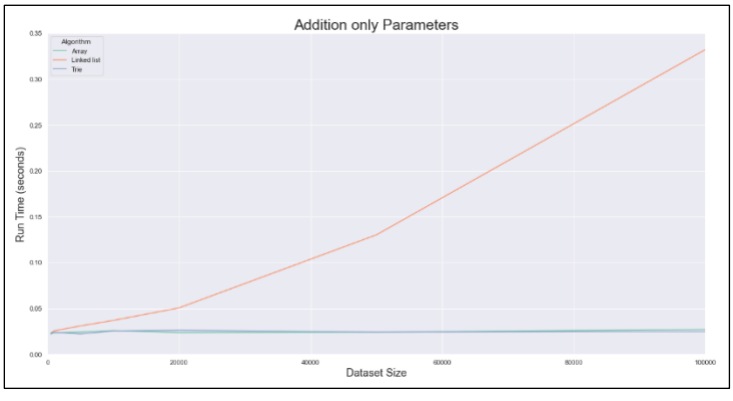


fig 3.

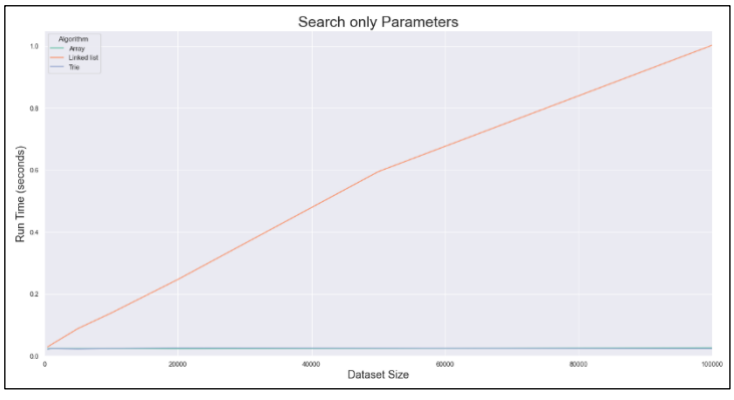


fig 4.

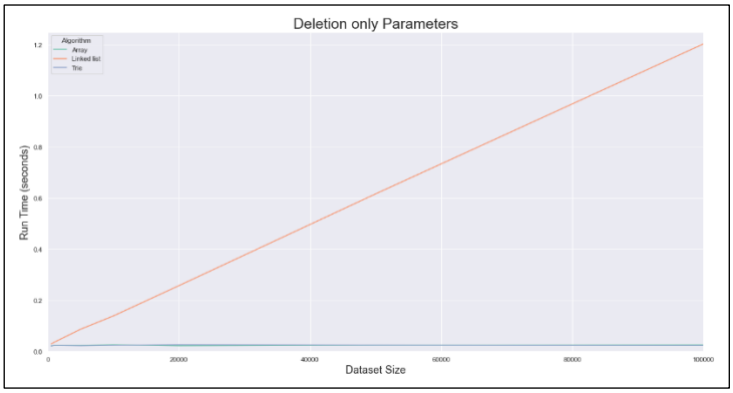


fig 5.

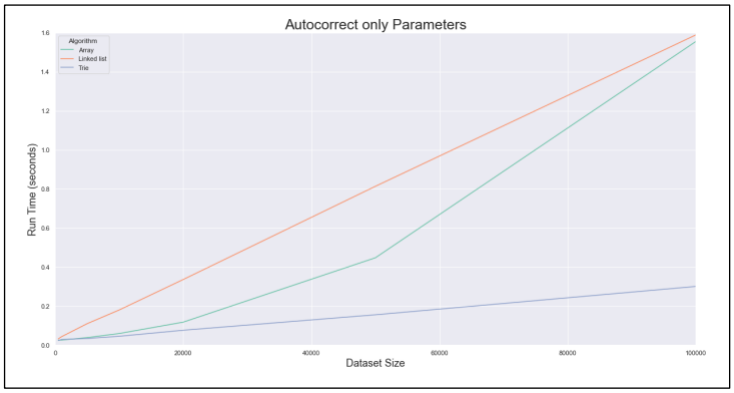


fig 6.

B.2 Evaluation and Analysis

The theoretical time complexities of every parameter setting for each data structure were calculated via identifying and determining the number of basic operations. For simplicity and to better understand each algorithms order of growth the Big-Θ notation was calculated, these values can be seen in *table 1* below:

	Theoretical average cases		
	Dictionary		
Operation	Array-based	Linked-List-based	Trie-based
Construction	$\Theta(nlog_2(n))$ - TimSort algorithm	$\Theta(n)$	$\Theta(nL)$, where L is the average length of words
Addition	$\Theta(n)$ - n complexity for insertion sort - $log_2(n)$ to check if word in already in dict	$\Theta(n)$ - $\Theta(1)$ complexity for inserting at start of linked list - n to check if word in already in dict	$\Theta(L)$, where L is the average length of words being search
Searching	$\Theta(log_2(n))$ - Binary search	$\Theta(n)$ - Linear search	$\Theta(L)$
Deletion	$\Theta(log_2(n))$ - Binary search	$\Theta(n)$ - Linear search	$\Theta(L)$
Autocompletion	$\Theta(Klog_2(n))$ Where K is the average number of words with the input prefix.	$\Theta(n)$ - Linear search	$\Theta(V)$, where V is the number of nodes after the prefix - depth-first search
Combination	$O(nlog_2(n))$	$\Theta(n)$	$\Theta(nL)$

Table 2.

B.2.1 Construction

The graph in *figure 2* above shows the run time of running the *dictionary_file_based.py* code with an empty input file, therefore demonstrating the time it takes to read the arguments and construct each dictionary. Our experimental results show the trie based dictionary has a much more time intensive construction algorithm, with the array-based dictionary being the most time efficient. These deviate from the theoretical time complexities seen in *table 2* as the array-based algorithm uses a Timsort algorithm with a time complexity of $\Theta(nlog_2(n))$ which theoretically is the most complex out of all the implementations. One possible explanation for this deviation is the use of the in-built python function. sort() when compared to the algorithms coded directly in python in the linked-list and trie-based approaches. The reason why built-in functions tend to operate faster in python is because they are already compiled after being implemented and optimized in C, meaning the code isn't interpreted at runtime like its python counterpart, it can allocate memory in larger chunks, avoid multiple validity checks etc.

B.2.2 Addition

The graph in *figure 3* above shows the run time of each data structure across the seven input sizes after having to compute 50 insertion/addition tasks. As seen in the graph, the linked list datatype defiantly had the highest time complexity and growth rate as demonstrated by the increasing linear growth of run time as input size increased. With a relatively small input size of 500 all datatypes exhibited relatively comparable run times, however as n increased the linked list's time complexity grew at a much greater rate whilst both the trie and array datatypes increased at a much smaller rate. Some of these results are further supported by the theoretical complexities generated in *table 1*, as the trie-based dictionary both achieved the smallest time complexity experimentally and theoretically due to the fact it only needs to loop through the number of letters in each inputted word, which for all tested cases is allot smaller than the input size. The array-based dictionary on the other-hand slightly deviates from the expected results, as it has the same theoretical time complexity as the linked list but exhibited a much lower experimental value. Although both addition algorithms in the linked list and array dictionaries have an average-case complexity $\in \Theta(n)$, the array-based dictionary uses in-built python functions (bisect.insort & bisect.biscet_left) to implement its insertion sort and binary search algorithms, whilst the linked list linear search is coded completely in python. The array-based implementation also exhibited a very similar time-complexity to the trie dictionary even though the theoretical complexity of the addition operation is much simpler in the trie-based dictionary.

B.2.3 Searching

The graph in *figure 4* above shows the run time of each data structure across the seven input sizes after having to compute 50 search tasks. Similarly, to *figure 2*, the linked list's time complexity has a much higher growth rate than both the array-based and trie-based dictionaries. This is observed as, like *figure 2*, the linked list run time increases at a linear growth rate whilst the array and trie dictionaries remain relatively constant with the trie being only slightly faster/less complex. These results further support the calculated theoretical complexities as the linked-list dictionary has an upper bound of $O(n)$, meaning it increases in complexity linearly with input size, and the array and trie dictionaries have theoretical complexities of $O(log_2(n))$ and $O(W)$ (where W is the number of letters in word being search) respectively. The trie dictionary therefore remains the most time effective as its complexity is only proportionate to the length of the word being searched, whilst the array-based dictionary has a slightly higher time complexity as it shares a logarithmic relationship with the input size. This logarithmic relationship is observed as a binary search algorithm (using bisect library in python) is used on the already sorted array, meaning the array can continually be halved until the desired word is found, which is much faster than a sequential search.

B.2.4 Deletion

To avoid further repetition, the trends exhibited in *figure 5* above further mimic those seen in both *figure 4* and *figure 3.* The main differentiating factor is that link-list deletion algorithm demonstrated a larger growth rate than that seen in its search and addition algorithms, which is likely due to the additional comparisons made in every iteration. The linear growth rate of the linked-list and relatively stagnant rate seen in the array and tire dictionaries is again supported by the theoretical complexities computed because the trie only must loop through the length of the word to be deleted $(O(W))$,thus has a very small- and constant-time complexity. Although the array and linked list both must search through the whole dataset, the array dictionary is able to use a binary search algorithm, due to the fact it is sorted, in its deletion implementation which is able to massively reduce search time for the word $(\log(n))$, as apposed to the linear search used in the linked list $\big(O(n)\big)$.

B.2.5 Autocompletion

The graph in *figure 6* above shows the run time of each data structure across the seven input sizes after having to compute 50 autocomplete tasks. This parameter shows a slight deviation from the previously observed results, as not only has the run time of all the data structures increased significantly, but so has the difference between the tire and array-based dictionaries. As excepted, the linked list and the autocorrect algorithm used remains to be the most time complex as input size increases, whilst the array-based

dictionary has a slightly lower growth rate, and the tire dictionary remains to be the least time complex. The results obtained from the trie dictionary further mimics the theorical results, as a recursive depth-first-search algorithm was used to traverse the trie. Although the trie-dictionary implementation was significantly less complex than the others, it still exhibited a much higher growth rate than all previous operation investigations. This can be explained by the fact that as the input size increases, generally, so does the number of nodes after each prefix that needs to be traversed. Although the array-based dictionary seems to have deviated from the theoretical complexities computed above, the average-case and best-case complexities are substantially more efficient than those seen in the linked list. This is because the array implementation only implements a binary search k times, with k being the number words with the inputted prefix. Because of the random sampling performed in this report, the likelihood of a substantial number of words containing the same prefix is low, meaning the complexity for most cases is likely to be closer to $log_2(n)$ than $nlog_2(n)$. The linked-list implementation on the other-hand, no matter the input, has a time complexity of n and therefore a much higher linear growth rate. Additionally, as described when investigating the addition parameters, because the array-based dictionary uses in-built python functions to perform its binary search algorithm, it is likely to be more efficient, as in-built functions are implemented and optimized in C and generally tend to run allot faster as a result.

B.2.6 Overall results

The graph in *figure 1* above shows the run time of each data structure across the seven input sizes after having to compute 25 additions, searches, deletions and autocomplete tasks. Overall, this graph shows the effect of the cumulative time complexities of all the algorithms investigated above. As the parameter inputted for this section was double the number of operations completed in all prior sections, all datatypes took around double the time to compete the parameters inputted. The trends exhibited in *figure 1* further consolidate the findings and assumptions made in the previous four investigations. The array-based dictionary performed the best overall as although it has a much more complex autocomplete algorithm than the trie-based dictionary, the intensive construction of the Trie outweighs this significantly. Additionally, although the trie takes allot longer to construct than the linked-list, because the linked-list is allotting more time intensive in all the tested operations, the linked-list remains the most time intensive overall.

B.3 Summary

In conclusion, for small dataset sizes around 1000 or 2000 words, as the difference between the time complexities is relatively minute, all datatypes and their respective algorithms are sufficient for the tested operations. For datasets larger than 5000 points/words for every tested case, aside from pure construction, had the greatest time complexity and is therefore not the most optimal approach in any case. Because of the drastic reduction in time complexity that arose from using the bisect() functions in the array implementation, this approach remains to be the most efficient in all cases when dictionary construction is taken into account. The only instance where I would suggest using our trie implementation would be cases where a much larger set of operations is needed to be performed on the constructed dictionary, as this would help outweigh the burden of dictionary construction. Overall, these deviations in excepted results emphasis the effect that implementation has on algorithm efficiency and therefore for future investigation in-built functions and a more optimised method should be used on the trie-dictionary to help achieve the full theoretical potential of the algorithms.