Experimentation Report

Introduction:

Purpose of the project:

The aim of Assignment 2 is to implement an autocomplete dictionary using two different algorithms - sorted array and radix tree. This report analyses the difference in complexity and efficiency in using the two different approaches and further conducts experiments with diverse data size, data characteristics and search algorithms.

Data structures:

1. Struct for place data: place_t stores the attributes for each place data. The trading name is used as the key.

```c
struct place {
    int censusYear;                 // The year the information was recorded for. (integer)
    int blockId;                    // The city block ID. (integer)
    int propertyId;                 // The ID of the property. (integer)
    int basePropertyId;             // The ID of the building the business is in. (integer)
    char *buildingAddress;          // The address of the building. (string)
    char *clueSmallArea;            // The CLUE area of Melbourne that the building is in. (string)
    char *businessAddress;          // The address of the business itself. (string)
    char *tradingName;              // The name of the business. (string)
    int industryCode;               // The ID for the category of the business. (integer)
    char *industryDescription;      // The description of the category of the business. (string)
    char *seatingType;              // The type of seating the record describes. (string)
    int numberOfSeats;              // The number of seats provided of this type. (integer)
    double longitude;               // The longitude (x) of the seating location. (double)
    double latitude;                // The latitude (y) of the seating location. (double)
};
typedef struct place place_t;
```

2. Struct for array algorithm: array_t consists of the array that stores the pointer to place data and keeps track of the number of place data inserted.

```c
struct array {
    place_t **A;        // Array of data for places
    int size;           // Current array size
    int n;              // Current number of elements used
};
typedef struct array array_t;
```

3. Struct for radix tree algorithm: rt_t stores the attributes for each tree node (common prefix bits, prefix itself) while also containing the pointer to its leaf nodes and a list of corresponding place data.

```c
typedef struct rt rt_t;
struct rt {
    int prefixBits;             // Number of common bits
    char *prefix;               // Binary representation of common bits
    rt_t *branchA;              // Pointer to leaf node that begins with 0
    rt_t *branchB;              // Pointer to leaf node that begins with 1
    list_t *dataList;           // Linked list of data for places
};
```

Inputs:
Adopting the csv file data processing implementation used in Stage 1, both Stage 2 and 3 takes place data stored in an unsorted linked list as the starting point for inserting data into an array or a radix tree. For searching, the partial query strings are compared to the trading name of each place data.

Experimentation approach:
To observe and compare the efficiency of utilising sorted array and radix tree for autocomplete dictionaries, the number of bits and strings compared will be examined in particular as they effectively depict the characteristic of the two different algorithms and clearly shows the efficiency.

Stage 2 - Sorted Array

Bit, Char and String Comparison Counting Logic:

- Bit: To calculate the total char comparisons, increment the counter by 1 for every character comparison between two strings, either until the end of the string (null terminator "\0" inclusive) or until the very first mismatch.
- Char: In stage 2, only characters were compared. Hence bits comparison is essentially just the number of bits for the characters compared (charCounter * 8, 1 byte = 8 bits).
- String: String comparison counter increments by 1 for every call for the function "strcmp", where two strings (trading names) are compared.

Search Complexity Expectations:

The required implementation details of stage 2 involves a sorted array with possible duplicates of trading names. To analyse the appropriateness of sorted array as an autocomplete dictionary, two other test cases were made - sorted array without duplicates of trading names and unsorted array with possible duplicates.

1. Sorted array with duplicates: According to the theory, the expected worst case complexity for inserting data into an array with alphabetically ascending order is O(n log n) and binary searching in a sorted array requires O(log n). However, due to possible duplicates of trading names, the performance of linear search involves the upper-bound complexity of O(n). Hence, unfortunately, the dominating complexity for dict2 becomes O(n) - less efficient.

2. Sorted array without duplicates: The worst case complexity is expected to be O(n log n) for inserting and O(log n) for searching. Without duplicates, linear searches can be neglected. Hence the overall search behaviour should be O(log n).

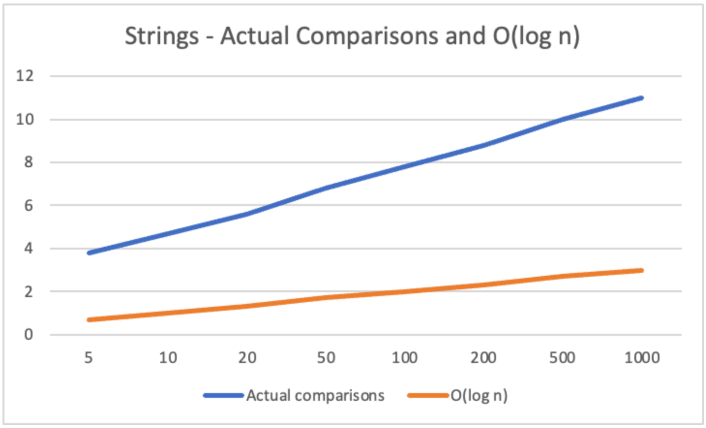
3. Unsorted array with duplicates: Based on theory, the expected worst case complexity for both inserting and searching in an unsorted array is O(n). Without sorting, binary search is not applicable and hence linear search has to be used throughout with the complexity of O(n) for searching.

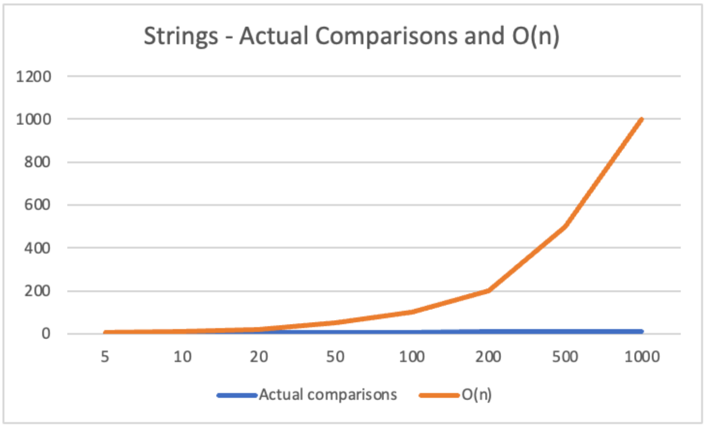
Results:

Test cases were generated for csv files with 5, 10, 20, 50, 100, 200, 500 and 1000 rows of place data. The average comparison values for bits and strings was recorded.

	1. Sorted, duplicates		**2. Sorted, no duplicates**		**3. Unsorted, duplicates**	
n	**Bits**	**Strings**	**Bits**	**Strings**	**Bits**	**Strings**
5	163.2	3.8	144.0	3.8	198.6	3.0
10	192.8	4.7	165.6	4.7	240.5	11.0
20	194.0	5.6	162.8	5.6	272.6	24.0
50	217.1	6.8	183.5	6.8	311.1	52.3
100	248.3	7.8	206.1	7.1	345.7	127.6
200	253.9	8.8	207.1	7.6	373.5	199.7
500	300.8	10.0	234.1	8.2	400.3	526.8
1000	351.6	11.0	250.3	8.8	449.6	1139.7

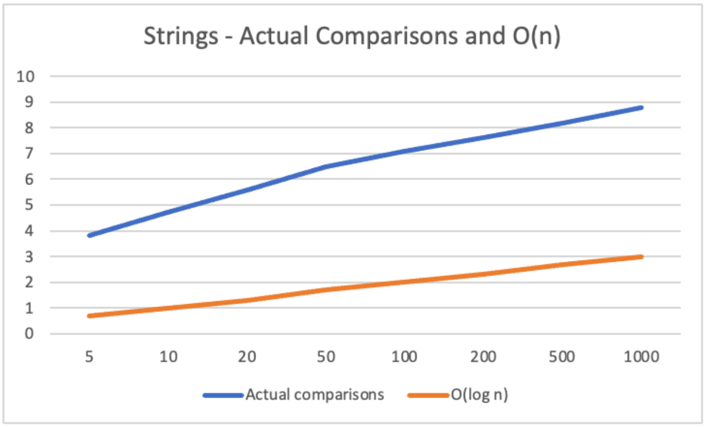
1. Sorted array with duplicates:





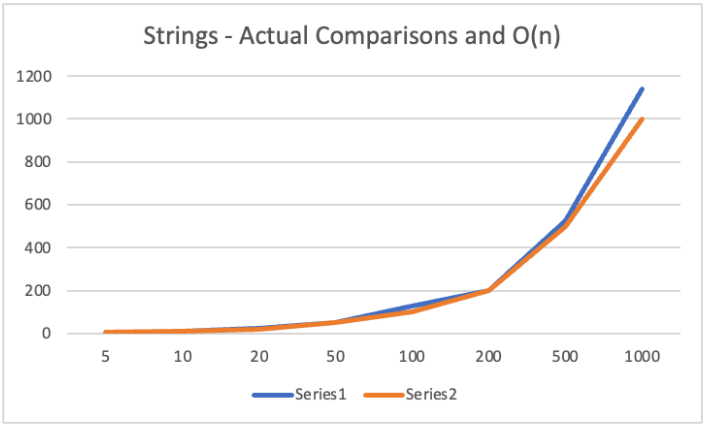
Unlike the theory expected, the graph depicts that the sorted array with binary search in practice follows more of order (log n) than log(n). The non-aligning trend between the actual data collected and expected O(n) complexity is due to the small amount of linear search required, as there were only few (>7) duplicates. For O(log n) on the other hand, the overall growth trend looks similar but the actual number of comparisons are larger in the collected data. This is also due to some linear searches done for duplicates.

2. <u>Sorted array without duplicates:</u>



Based on the graph, the sorted array is of the order O(log n), when the additional linear search is neglected.

3. <u>Unsorted array with duplicates:</u>



The graph shows that an unsorted array with linear search has the worst case complexity of O(n) as the graph for the data collected aligns well with the expected growth.

Overall, it is clear that sorting the array compared to unsorting and using binary search rather than linear search provides the most efficient algorithm to implement an autocomplete dictionary using arrays.

Stage 3 - Radix Tree:

Bit, Char and String Comparison Counting Logic:

- Bit: Bit comparison counter increments by 1 for every comparison of a single bit of 8-bit binary representation of a character. In stage 3, the null terminators were not considered in the comparisons.
- Char: The char comparison counters are the sums of the number of times of moving down to leaf nodes, the length of query string length and the number of common characters found during the search function.
- String: In stage 3, the string comparison counter is 1 for all query inputs, since the traverseAndFind function only operates until the finding of a single node that contains the query input.

Search Complexity Expectations:

By theory, the expected worst case complexity for both inserting and searching in a radix tree is O(m), where ‘m’ is the length of a key. The randomness (sorted or unsorted) of input data is insignificant in radix trees, as they do not rely on any specific order of insertion or arrangement of keys. Moreover, since the duplicates are stored within one list struct, no further search algorithm is required.

Results:

Test cases were generated for csv files with 5, 10, 20, 50, 100, 200, 500 and 1000 rows of place data. The average comparison values for bits and strings was recorded.

n	Bits	Strings
5	123.2	1
10	135.2	1
20	130.8	1
50	133.2	1
100	143.9	1
200	125.2	1
500	125.7	1
1000	138.1	1

Based on the collected data, the overall average bits comparison is approximately 121.9 and stringer comparison of 1. It is notable that radix tree bit comparisons between each test case are independent of the test case size as it does not increase proportionally to the size of the test case like in the sorted array observed in Stage 2. Rather, these numerical values imply that the complexity of the radix tree aligns with the theory with order of O(m).

Discussion:

Theory vs Actual data collected:
While some findings aligned well with the theoretical complexity for searching data in array and radix tree autocomplete dictionaries, some data collected showed slightly different, yet interesting characteristics of each algorithm and their complexity.

For Stage 2 sorted array implementation, analysing the significance of sorting and effect of different search algorithms was the key in understanding the overall complexity. According to the experiment results, randomness of data was important and sorted insertion increased the efficiency for searching as it enabled the use of binary search. Yet, involvement of linear search (O(n)) due to the need for handling duplicates slightly lagged the efficiency of the overall sorted array algorithm. However, despite the use of linear search, the overall growth trend for the main test case 'sorted array with duplicates' was more similar to O(log n) instead of O(n), implying that linear search was not highly significant in our test cases. This outcome could have resulted as there were not many duplicates in test cases and the data was sorted in insertion. Hence it can be assumed that the program is completed with complexity of O(n log n) for inserting and O(log n) for searching.

For Stage 3 radix tree implementation, observing the amount of bit comparison showed its advanced capability to compare data with least amount of mismatches. Comparing every bit instead of the whole characters enabled minimised comparisons. Most importantly, making each tree node to only hold a common prefix stem significantly increased the efficiency of the overall algorithm. The experiment showed that a radix tree performed with complexity of O(m), where 'm' is the length of a key, as expected by the theory.

Radix tree vs Sorted array:
It was observed that both the amount of bit and string comparisons for radix tree were less than for the sorted array, suggesting that radix tree is a more efficient and hence the suitable algorithm to be utilised for autocomplete look-up dictionary.

Through experimentations and inspections, the few notable benefits for using radix tree for autocomplete dictionary seems to be:

- Enhanced handling of unsorted data.
- The fact that it is possible to guess which node to traverse to (eg. move to the left branch if the next bit is 0 and to the right branch if it is 1).
- Minimised mismatched comparisons by comparing each bit.
- Separate storing of all the place data with identical trading names, eliminating the need for linear search.