COSC 2123/1285 Algorithms and Analysis
Assignment 1: Word Completion

	Assessment Type	Pair (Group of 2) Assignment. Submit online via Canvas → Assignments → Assignment 1. Clarifications/updates/FAQ can be found in Ed Discussion Forum → Assignment 1 General Discussion.
	Due Date	Week 7, 11:59pm, Friday, April 22, 2022
	Marks	30

1 Objectives

There are a number of key objectives for this assignment:

- Understand how a real-world problem can be implemented by different data structures and/or algorithms.
- Evaluate and contrast the performance of the data structures and/or algorithms with respect to different usage scenarios and input data.

In this assignment, we focus on the word completion problem.

2 Background

Word/sentence (auto-)completion is a very handy feature of nowadays text editors and email browsers (you must have used it in your Outlook). While sentence completion is a much more challenging task and perhaps requires advanced learning methods, word completion is much easier to do as long as you have a dictionary available in the memory. In this assignment, we will focus on implementing a dictionary comprising of words and their frequencies that allows word completion. We will try several data structures and compare their performances. One of these data structures is the *Ternary Search Tree*, which is described below.

Ternary Search Trees

Ternary search trees (TST) is a data structure that allows memory-efficient storage of strings and fast operations such as spell checking and auto-completion.

Each node of the a TST contains the following fields:

- a lower-case letter from the English alphabet (‘a’ to ‘z’),
- a positive integer indicating the word’s *frequency* (according to some dataset) if the letter is the last letter of a word in the dictionary,

- a *boolean* variable that is True if this letter is the last letter of a word in the dictionary and False otherwise,
- the *left* pointer points to the left child-node whose letter is smaller (in alphabetical order, i.e. 'a' < 'b'),
- the *right* pointer points to the right child-node whose letter is larger,
- the *middle* pointer points to a node that stores the next letter in the word.

As an example, consider Figure 1.

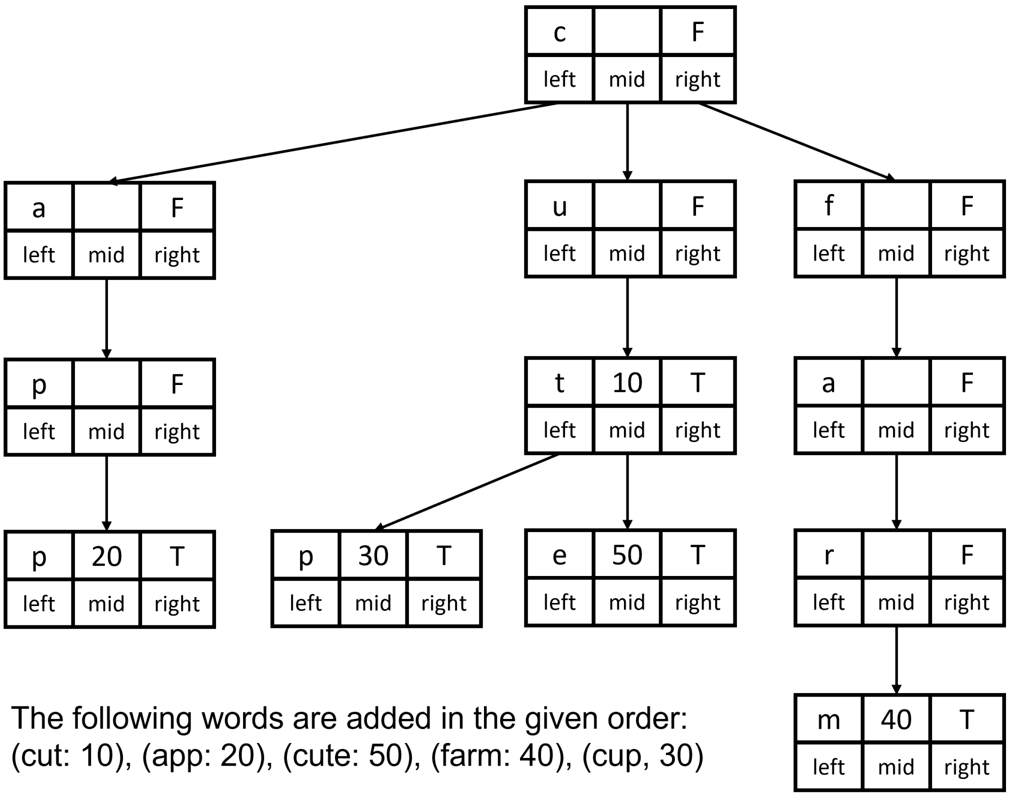


Figure 1: An example of a ternary search tree storing five words and their frequencies. The boolean value (T)rue indicates that the letter is the end of a word. In that case, a frequency (an integer) is shown, e.g., 10 for 'cut'. Note that a word can be a prefix of another, e.g., 'cut' is a prefix of 'cute'.

Construction

A TST can be built by adding words to the tree one by one, noting that different orders of words can lead to different trees. One can start from the root node (or create one when adding the first word) and follow the left, middle, or right pointer depending on whether the current letter is smaller, equal to, or larger than the letter of the node (in alphabetical order).

If the new word is a prefix of an existing word in the tree then we can simply change the boolean field of the node storing the last letter to True. For instance, in the example in Figure 1, if (cut, 10) is added after (cute: 50), then one can simply change the boolean field of the node containing the letter 't' to True and set the frequency to 10, which signifies that now (cut, 10) is part of the tree. In another case, when (cup, 30) is added, then one new node has to be constructed to store 'p', which has a True in the boolean field and 20 as its frequency. In this case, 'cup' can reuse the two nodes storing ‘c’ and ‘u’ from the previous words. However, when (app, 20) and (farm, 40) are added, three and four new nodes have to be created, respectively.

Searching

To search for a word (and to get its frequency) in a TST, we use its letters to navigate the tree. For each letter of the word (from the first to the last), we follow either the left, middle, or right pointer to the corresponding child node depending on whether this letter is smaller than, equal to, or larger than the letter stored at the current node. Once we hit a node storing the same letter, we will follow the middle pointer and start using the next letter of the word on the middle child node and repeat the process.

The search fails, that is, the word is not in the tree, if either a) the search algorithm hits a Null node while there is still at least one letter in the word that hasn’t been matched, or b) it finds that the searched word is a prefix of an existing word in the tree but the boolean field of the node corresponding to the last letter is False.

Deletion

The deletion succeeds if the word is already included in the tree. If the word is a prefix of another word then it can be deleted by simply setting the boolean field of the node storing the last letter to False. For example, if (cut, 10) is to be deleted from the tree in Figure 1, then we only need to change the boolean field of the node storing ‘t’ to False. Otherwise, if the word has some unique suffix, then we need to delete nodes corresponding to the suffix only. For example, if (cup, 30) is to be removed, then only the node storing the last letter ‘p’ needs to be deleted from the tree.

Auto-completion

One simple way to achieve auto-completion from a partial word is to return three words in the dictionary (tree) of *highest frequencies* that have the given partial word as a prefix. For example, in the tree given in Figure 1,

- the auto-completion list for ‘cu’ is: [(cute, 50), (cup, 30), (cut, 10)],
- the auto-completion list for ‘f’ is: [(farm, 40)].

Suppose we add one more word (curiosity, 60) into this tree, then the auto-completion list of ‘cu’ will be changed to [[(curiosity, 60), (cute, 50), (cup, 30)]. In this example, although ‘cut’ contains ‘cu’ as a prefix, it is not in the top three of the most common words having ‘cu’ as a prefix. In general, the auto-completion list contains either three, two, one, or no words. They must be sorted in a *decreasing* frequencies, that is, the first word has the highest frequency and the last has the lowest frequency.

3 Tasks

The assignment is broken up into a number of tasks, to help you progressively complete the assignment.

3.1 Task A: Implement the Nearest Neighbour Data Structures and their Operations (12 marks)

In this task, you will implement a dictionary of English words that allows auto-completion, using three different data structures: Python’s list (array), Python’s dictionary (hash table), and ternary search tree. Each implementation should support the following operations:

- Build a dictionary from a list of words and frequencies: create a dictionary that stores words and frequencies taken from a given list (using Add below).
- (A)dd a word and its frequency to the dictionary.
- (S)earch for a word in a dictionary and return its frequency (return 0 if not found).
- (D)elete a word from the dictionary.
- (AC)Auto-complete a given string and return a list of three most frequent words (if any) in the dictionary that have the string as a prefix.

3.1.1 Implementation Details

List-Based Dictionary In this subtask, you will implement the dictionary using Python's lists, which are equivalent to arrays in other programming languages. In this implementation, all operations on lists are allowed. Other data structures should NOT be used directly in the main operations of the list-based dictionary.

Hashtable-Based Dictionary In this subtask, you will use the native Python's dictionary. In this implementation, all standard Python operations on dictionaries are allowed.

Ternary-Search-Tree-Based Dictionary In this subtask, you will implement the ternary search tree and use it to support auto-completion. Both iterative or recursive implementations are acceptable.

3.1.2 Operations Details

Operations to perform on the implementations are specified on the command file. They are in the following format:

```
<operation> [arguments]
```

where `operation` is one of {S, A, D, AC} and `arguments` is for optional arguments of some of the operations. The operations take the following form:

- `S word` – searches for `word` in the dictionary and returns its frequency (returns 0 if not found).
- `A word frequency` – adds a new word and its frequency to the dictionary, returns True if succeeded and False if the word already exists in the dictionary.
- `D word` – deletes `word` from the dictionary. If fails to delete (`word` is not in the dictionary), returns False.
- `AC partial_word` – returns a list of three words of highest frequencies in the dictionary that has `partial_word` as a prefix. These words should be listed in a decreasing order of frequencies. Maximum three words and minimum zero word will be returned.

As an example of the operations, consider the input and output from the provided testing files, e.g., `sampleDataToy.txt`, `testToy.in`, and the expected output, `testToy.exp` (Table 1).

Note, you do NOT have to do the input and output reading yourself. The provided Python files will handle the necessary input and output formats. Your task is to implement the missing methods in the provided classes.

sampleDataToy.txt	testToy.in (commands)	testToy.exp (expected output)
cute 10	S cute	Found ‘cute’ with frequency 10
ant 20	D cute	Delete ‘cute’ succeeded
cut 30	S cute	NOT Found ‘cute’
cuts 50	S book	NOT Found ‘book’
apple 300	A book 10000	Add ’book’ succeeded
cub 15	S book	Found ‘book’ with frequency 10000
courage 1000	S apple	Found ‘apple’ with frequency 300
annotation 5	D apple	Delete ‘apple’ succeeded
further 40	S apple	NOT Found ‘apple’
furniture 500	D apple	Delete ‘apple’ failed
find 400	AC c	Autocomplete for ‘c’: [calm: 1000 cuts: 50 cut: 30]
farm 5000	AC cut	Autocomplete for ‘cut’: [cuts: 50 cut: 30]
farming 1000	D cut	Delete ‘cut’ succeeded
farmer 300	AC cut	Autocomplete for ‘cut’: [cuts: 50]
appendix 10	AC farms	Autocomplete for ‘farms’: [ ]
apology 600		
apologetic 1000		
fur 10		
fathom 40		
apps 60		

Table 1: The file `sampleDataToy.txt` provides the list of words and frequencies for the dictionary, while `testToy.in` and `testToy.exp` have the list of input commands and expected output. For instance, as ‘cute’ belongs to the dictionary, the expected outcome of “`S cute`” should be “`Found ‘cute’ with frequency 10`” and “`D cute`” should be successful. After deleting ‘cute’, “`S cute`” should return “`NOT Found ‘cute’`”. Note that although there are more than three words having ‘c’ as a prefix, “`AC c`” only returns the three words of highest frequencies. Also, “`AC cut`” returns “`[ cuts: 50 cut: 30`”, but after deleting ‘cut’, it must return “`[ cuts: 50`” only.

3.1.3 Testing Framework

We provide Python skeleton codes (see Table 2) to help you get started and automate the correctness testing. You may add your own Python files to your final submission, but please ensure that they work with the supplied files and the Python testing script (to be released later). Below we describe code structure in Python.

To compile, from the directory where `dictionary_file_based.py` is, execute:

```
> python dictionary_file_based.py [approach] [data filename] [command filename]
                                                            [output filename]
```

where

- `approach` is one of the following: `list`, `hashtable`, `tst`,
- `data filename` is the name of the file containing the initial set of words and frequencies,
- `command filename` is the name of the file with the commands/operations,
- `output filename` is where to store the output of program.

For example, to run the test with sampleDataToy.txt, type (in Linux or Pycharm’s terminal, **replacing ‘python’ by ‘python3’ in Linux**):

file	description
`dictionary_file_based.py`	Code that reads in operation commands from file then executes those on the specified nearest neighbour data structure. *Do NOT modify this file.*
`node.py`	Class defining the nodes in the ternary search tree. *Do NOT modify this file.*
`based_dictionary.py`	The base class for the dictionary. *Do NOT modify this file.*
`list_dictionary.py`	Skeleton code that implements a list-based dictionary. Complete all the methods in the class.
`hashtable_dictionary.py`	Skeleton code that implements a hash-table-based dictionary. Complete all the methods in the class.
`ternarysearchtree_dictionary.py`	Skeleton code that implements a ternary-search-tree-based dictionary. Complete all the methods in the class.

Table 2: Table of Python files. The file `dictionary_file_based.py` is the main module and should NOT be changed.

```
> python dictionary_file_based.py list sampleDataToy.txt testToy.in testToy.out
```

Then compare `testToy.out` with the provided `testToy.exp`. In Linux, we can use the `diff` command:

```
> diff testToy.out testToy.exp
```

If nothing is returned then the test is successful. If something is returned after running `diff` then the two files are not the same and you will need to fix your implementation. Similarly, you can try the larger data file `sampleData.txt` together with `test1.in` and `test1.exp`.

In addition, we will provide a Python script (released later) that automates testing, based on input files of operations (such as example above). These are fed into the script which then calls your implementations. The outputs resulting from the operations are stored, as well as error messages. The outputs are then compared with the expected output. We have provided two sample input and expected files for your testing and examination.

For our evaluation of the correctness of your implementation, we will use the same Python script and a set of **different** input/expected files that are in the same format as the provided examples. **To avoid unexpected failures, please do not change the Python script nor `dictionary_file_based.py`.** If you wish to use the script for your timing evaluation, make a copy and use the unaltered script to test the correctness of your implementations, and modify the copy for your timing evaluation. Same suggestion applies for `dictionary_file_based.py`.

3.1.4 Notes

- If you correctly implement the "To be implemented" parts, you in fact do not need to do anything else to get the correct output formatting. `dictionary_file_based.py` will handle this.
- We will run the supplied test script on your implementation on the university's core teaching servers, e.g., `titan.csit.rmit.edu.au`, `jupiter.csit.rmit.edu.au`, `saturn.csit.rmit.edu.au`. If you develop codes on your own machines, please ensure your code compiles and runs on these servers. If your code doesn't compile or run on the core teaching servers, we unfortunately do not have the resources to debug each one and cannot award marks for testing.

- All submissions should compile with no warnings on **Python 3.6** or any newer version of Python on the core teaching servers.

3.1.5 Test Data

We provide two sample datasets. The small one, sampleDataToy.txt, is for debugging and the larger one, sampleData.txt, which contains 5000 words from English-Corpora.org, is for testing and automarking. Each line represents a word and its frequency. For each line, the format is as follows:

```
word frequency
```

This is also the expected input file format when passing data information to `dictionary_file_based.py`. When creating your own files for the the analysis part, use this format.

3.2 Task B: Evaluate your Data Structures for Different Operations (18 marks)

In this second task, you will evaluate your implementions in terms of their time complexities for the different operations and different use case scenarios.

3.2.1 Use Case Scenarios

Perform the empirical analysis and report the process and the outcome in the following scenarios.

- **Scenario 1 - Growing dictionary**: The dictionary is growing in size and you are to evaluate and compare the running times of the Add operations across different data structures.
- **Scenario 2 - Shrinking dictionary**: The dictionary is shrinking in size and you are to evaluate and compare the running times of the Delete operations across different data structures.
- **Scenario 3 - Static dictionary**: You are to evaluate and compare the running times of the Search and Auto-completion operations on fixed size dictionaries.

3.2.2 Empirical Analysis

In your analysis, you should evaluate each of your implementations in terms of the different scenarios outlined above. Write a report on your analysis and evaluation of the different implementations. Consider and recommend in which scenarios each approach would be most appropriate. The report should be no more than **6 pages**, in font size 12 and A4 pages. See the assessment rubric (Appendix A) for the criteria we are seeking in the report.

Data Generation and Experiment Setup

Apart from `sampleData.txt` (from iWeb), which contains 5,000 words, we also provide another large dataset `sampleData200k.txt` (from Kaggle) with 200,000 unique words and frequencies. You may either

- use these datasets to create a collection of datasets to run your implementations on for Task B, or
- write a separate program to generate datasets.

Either way, in the report you should explain *in detail* how the datasets are generated and why they supports a robust empirical analysis.

For Scenario 1 (growing dictionary), we suggest you grow your dictionary from small sizes to large sizes, e.g., 50, 500, 1000, 2000, 5000, 10000, etc., and evaluate the running times of the Add operation. For Scenario 2 (shrinking dictionary), you can reverse that, starting from a large dictionary and delete words repeatedly to get down to smaller dictionaries, e.g., 10000, 5000, 2000, 1000, 500, 50, etc. Numbers here are for illustration only, please make your own decision and justify it in the report. For Scenario 3, you should perform Search/Auto-completion on dictionaries of various sizes from small, medium, to large. Words/strings used for operations must be randomly selected/generated.

To summarize, data generation and experiments have to be done in a way that guarantees reliable analysis and avoids bias caused by special datasets or special input parameters chosen, e.g., words, for evaluated operations.

4 Report Structure

As a guide, the report could contain the following sections:

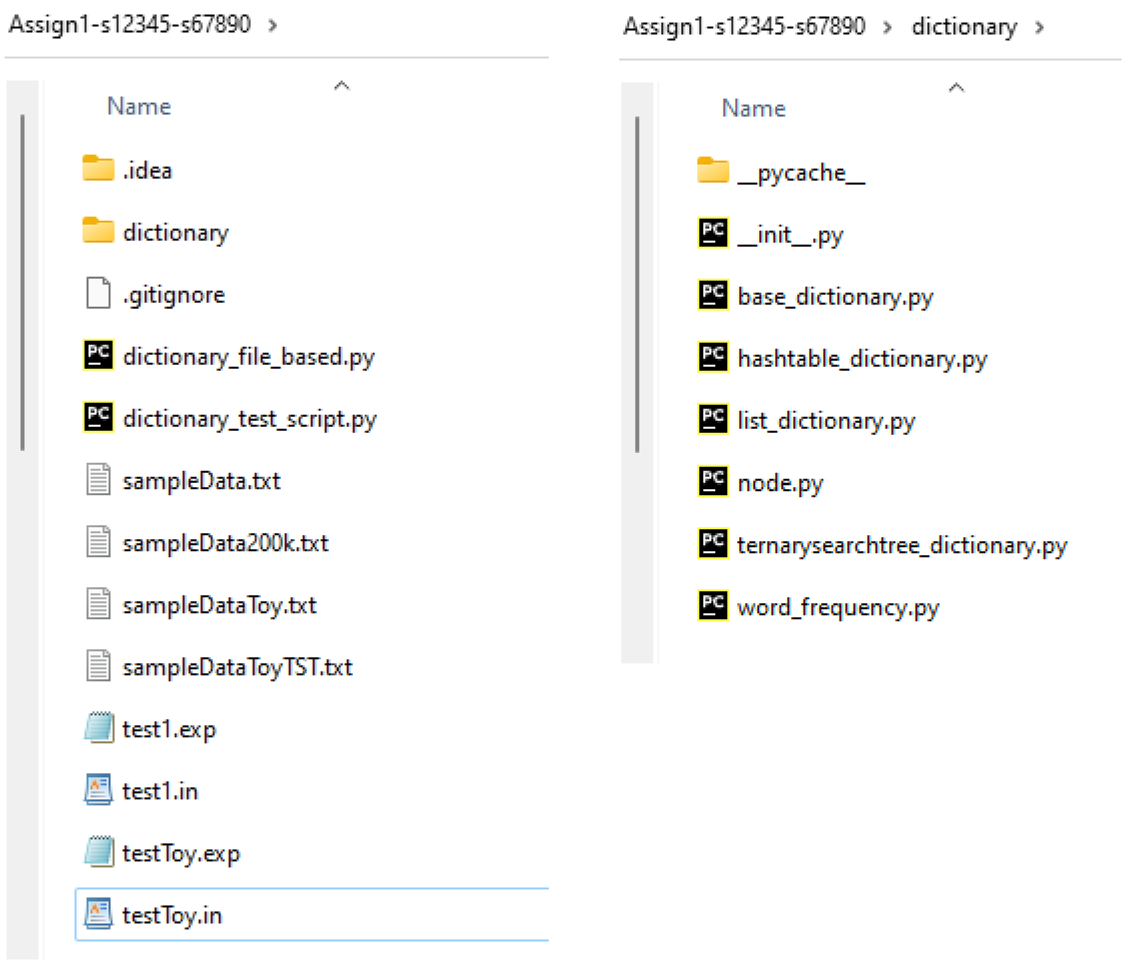
- Explain your data generation and experimental setup. Things to include are explanations of the generated data you decide to evaluate on, the parameter settings you tested on, describe how the scenarios were generated, which method you decide to use for measuring the running times and how the running times in the report are collected from the experiments. We recommend to use tables and graphs to better illustrate your observations.
- Evaluation of the approaches using the generated data. Analyse, compare and discuss your results across different parameter settings, data structures/algorithms and scenarios. Provide your explanation on why you think the results are as you observed. Are the observed running times supported by the theoretical time complexities of the operations of each approach?
- Summarise your analysis as recommendations, e.g., for this certain data scenario of this parameter setting, I recommend to use this approach because such an such.

5 Submission

The final submission (**in one single .zip file**) will consist of the codes and the report, and the contribution sheet.

- Your **Python source code** of your implementations. Your source code should be placed into in the same code hierarchy as provided. The root directory/folder should be named as `Assign1-<student_number_1>-<student_number_2>`. More specifically, if you are a team of two and your student numbers are s12345 and s67890, then the Python source code files should be under the folder `Assign1-s12345-s67890` as follows (see also Figure 2):
  - `Assign1-s12345-s67890/dictionary_file_based.py`.
  - `Assign1-s12345-s67890/dictionary/*.py` (all other Python files must be in the `/dictionary` sub-directory.
  - Any files you added, make sure they are in the appropriate directories/folders such that we can run script with the following command:
    `python dictionary_test_script.py <path to your folder>/Assign1-s12345-s67890...`
  - `Assign1-s12345-s67890/generation` (generation files, see below).

  When we unzip your submission, then everything should be in the folder `Assign1-s12345-s67890`.



Assign1-s12345-s67890 >

Name

.idea
dictionary
.gitignore
dictionary_file_based.py
dictionary_test_script.py
sampleData.txt
sampleData200k.txt
sampleDataToy.txt
sampleDataToyTST.txt
test1.exp
test1.in
testToy.exp
testToy.in

Assign1-s12345-s67890 > dictionary >

Name

__pycache__
__init__.py
base_dictionary.py
hashtable_dictionary.py
list_dictionary.py
node.py
ternarysearchtree_dictionary.py
word_frequency.py

Figure 2: Please keep the above folder/files structure to ensure our auto-test to run properly.

- Similarly, that folder also contains your **written report for part B** in PDF format, called "`assign1-s12345-s67890.pdf`". We have `Assign1-s12345-s67890/assign1-s12345-s67890.pdf`.
- Your **data generation code** should be in `Assign1-s12345-s67890/generation`. We will not run the code, but will examine their contents.
- Your group's **contribution sheet** in docx or PDF. See the following 'Team Structure' section for more details. This sheet should also be placed in `Assign1-s12345-s67890/`.

Note: **submission of the report and code will be done via Canvas**. More detailed instructions will be provided closer to submission date.

5.1 Clarification to Specifications & Submissions

Please periodically check the assignment's FAQ page for important aspects of the assignment including clarifications of concepts and requirements, typos and errors, as well as submission.

`https://edstem.org/au/courses/8416/discussion/754137`

6 Assessment

The assignment will be marked out of 30. Late submissions will incur a deduction of 3 marks per day, and no submissions will be accepted 7 days beyond the due date (i.e., last acceptable time is on 23:59, April 29, 2021).

The assessment in this assignment will be broken down into two parts. The following criteria will be considered when allocating marks.

Task A: Implementation (12/30):

- You implementation will be assessed on whether they implement the correct data structures (list, dictionary (hash table), and ternary search tree), and on the number of tests it passes in the automated testing.
- While the emphasis of this assignment is not programming, we would like you to maintain decent coding design, readability and commenting, hence these factors may contribute towards your marks.

Task B: Empirical Analysis Report (18/30):

The marking sheet in Appendix A outlines the criteria that will be used to guide the marking of your evaluation report. Use the criteria and the suggested report structure (Section 4) to inform you of how to write the report.

7 Team Structure

This assignment could be done in pairs (group of two) or individually. If you have difficulty in finding a partner, post on the discussion forum. If you opt for a team of two, **it is still your sole responsibility to deliver the implementation and the report** by the deadline, even when the team breaks down for some reason, e.g., your partner doesn't show up for meetings, leaves the team, or, in some rare case, withdraws from the course. Effective/frequent communication and early planning are key.

In addition, please submit what percentage each partner made to the assignment (a contribution sheet will be made available for you to fill in), and submit this sheet in your submission. The contributions of your group should add up to 100%. If the contribution percentages are not 50-50, the partner with less than 50% will have their marks reduced. Let student A has contribution X%, and student B has contribution Y%, and $X > Y$. The group is given a group mark of M. Student A will get M for assignment 1, but student B will get $\frac{M}{\frac{X}{Y}}$.

8 Plagiarism Policy

University Policy on Academic Honesty and Plagiarism: You are reminded that all submitted assignment work in this subject is to be the work of you and your partner. It should not be shared with other groups. **Multiple automated similarity checking software will be used to compare submissions**. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the students concerned. Plagiarism of any form may result in zero marks being given for this assessment and result in disciplinary action.

For more details, please see the policy at `http://www1.rmit.edu.au/students/academic-integrity`.

9 Getting Help

There are multiple venues to get help. There are weekly lectorial Q&A sessions (see Canvas Collaborate Ultra for time and location details). We will also be posting common questions on the Assignment 1 Q&A section on Ed Discussion Forum and we encourage you to check and participate in the discussions. However, please **refrain from posting solutions**.

A Marking Guide for the Report

Design of Evaluation (Maximum = 6 marks)	**Analysis of Results** (Maximum = 10 marks)	**Clarity, Structure, Comprehensiveness** (Maximum = 2 marks)
5-6 marks Data generation is well designed, systematic and well explained. All suggested scenarios, data structures/approaches and a reasonable range of parameters were evaluated. Each type of test was run over a number of runs and results were averaged.	9-10 marks Analysis is thorough and demonstrates understanding and critical analysis. Well-reasoned explanations and comparisons are provided for all the data structures/approaches, scenarios and parameters. All analysis, comparisons and conclusions are supported by empirical evidence and theoretical complexities. Well reasoned recommendations are given.	2 marks Very clear, well structured and accessible report, an undergraduate student can pick up the report and understand it with no difficulty. All required parts of the report are included with sufficient writing and supporting data. Discussions are comprehensive and at great depth.
3-4 marks Data generation is reasonably designed, systematic and explained. There are at least one obvious missing suggested scenarios, data structures/approach or reasonable parameter setting. Each type of test was run over a number of runs and results were averaged.	6-8 marks Analysis is reasonable and demonstrates good understanding and critical analysis. Adequate comparisons and explanations are made and illustrated with most of the suggested scenarios, data structures/approaches and parameters. Most analysis and comparisons are supported by empirical evidence and theoretical analysis. Reasonable recommendations are given.	1 mark Clear and structured for the most part, with a few unclear minor sections. Most required parts of the report are covered and discussed with reasonable depth.
2 marks Data generation is somewhat adequately designed, systematic and explained. There are several obvious missing suggested scenarios, data structures/approaches or reasonable densities. Each type of test may only have been run once.	3-5 marks Analysis is adequate and demonstrates some understanding and critical analysis. Some explanations and comparisons are given and illustrated with one or two scenarios, data structures/approaches and parameter settings. A portion of analysis and comparisons are supported by empirical evidence and theoretical analysis. Adequate recommendations are given.	0 mark The report is unclear on the whole and the reader has to work hard to understand. Missing important parts required for the report or with very shallow discussions.
0-1 marks Data generation is poorly designed, systematic and explained. There are many obvious missing suggested scenarios, data structures/approaches or reasonable parameters. Each type of test has only have been run once.	0-2 marks Analysis is poor and demonstrates minimal understanding and critical analysis. Few explanations or comparisons are made and illustrated with one scenario, data structure/approach and parameter setting. Little analysis and comparisons are supported by empirical evidence and possibly theoretical analysis. Poor or no recommendations are given.	