




COSC 2123/1285 Algorithms and Analysis

Assignment 1: Word Completion

	Assessment Type	(Group of 1 or 2) Assignment. Submit online via Canvas → Assignments → Assignment 1. Clarifications/Updates/FAQ: check Ed Discussion Forum → Assignment 1: General Discussion.
	Due Date	Week 7, 11:59pm, Friday, September 9, 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, which are array (that is, Python list), linked list, and trie, which are described one by one below. Please read them very carefully. Latest updates and answers for questions regarding this assignment will be posted on the Discussion Forum. It is your responsibility to check the post frequently for important updates.

Array-Based Dictionary

Python's built-in 'list' is equivalent to 'array' in other language. In fact, it is a dynamic array in the sense that its resizing operation (when more elements are inserted into the array than the original size) is managed automatically by Python. You can initialize an empty array in Python, add elements at the end of the array, remove the first instance of a given value by typing the following commands (e.g., on Python's IDLE Shell).

```

>>> array = []
>>> array.append(5)
>>> array.append(10)
>>> array.append(5)
>>> array
[5, 10, 5]
>>> array.remove(5)
>>> array
[10, 5]

```

In the array-based dictionary implementation, we use the Python list (a data structure) to implement common operations for a dictionary (an abstract data type). We treat each element of the array as a pair (**word**, **frequency**) (defined as an object of the simple class `WordFrequency`), where **word** is an English word (a string), e.g., ‘ant’, and **frequency** is its usage frequency (a non-negative integer), e.g., 1000, in a certain context, e.g., in some forums or social networks.

The array must be **sorted** in the alphabetical order, i.e., ‘ant’ appears before ‘ape’, to facilitate search. A new word, when added to the dictionary, should be inserted at a correct location that preserves the alphabetical order (using the module `bisect` is allowed - but you need to know what it does). An example of a valid array is [(‘ant’, 1000), (‘ape’, 200), (‘auxiliary’, 2000)]. Adding (‘app’, 500) to the array, we will have [(‘ant’, 1000), (‘ape’, 200), (‘app’, 500), (‘auxiliary’, 2000)]. Note that the pair (‘ant’, 1000) in our actual implementation should be an object of the class `WordFrequency` and not a tuple.

A **Search** for ‘ape’ from the array above should return its frequency 200. If the word doesn’t exist, 0 is returned.

A **Delete** for a word in the dictionary should return True and remove the word from the dictionary if it exists, and return False otherwise.

An **Autocompletion** for a string should return a sorted list (most frequent words appear first) of the three words (if any) of highest frequencies in the dictionary that have the given string as a prefix. For the array above, an autocompletion for ‘ap’ should return the list [(‘app’, 500), (‘ape’, 200)]. Notice that while both ‘app’ and ‘ape’ have ‘ap’ as a prefix, ‘app’ has a larger frequency and appears first in the returned list of autocompletion.

Linked-List-Based Dictionary

A linked list is precisely what it is called: a list of nodes linked together by references. In a singly linked list, each *node* consists of a data item, e.g., a string or a number, and a reference that holds the memory location of the next node in the list (the reference in the last node is set to Null). Each linked list has a **head**, which is the reference holding memory location of the first node in the list. Once we know the **head** of the list, we can access all nodes sequentially by going from one node to the next using references until reaching the last node.

In the linked-list-based implementation of dictionary, we use an **unsorted** singly linked list. You can use the implementation of the linked list in the workshop as a reference for your implementation. Each node stores as data a pair of (**word**, **frequency**) (an object of the class `WordFrequency`) and a reference to the next node. A word and its frequency are added as a new node at the front of the list by updating the **head** reference. Apart from the fact that they are carried out in the linked list, **Search**, **Delete**, and **Autocomplete** work similarly as in the array-based implementation. Note that unlike the array-based dictionary, the words in the linked list are **not** sorted.

Trie-Based Dictionary

Trie (pronounced as either ‘tree’ or ‘try’) is a data structure storing (key, value) pairs where keys are strings that allows fast operations such as spell checking and auto-completion. Introduced in the context of computer decades ago, it is no longer the most efficient data structure around. However, our purpose is to focus more on the understanding of what data structures mean, how they can be used to implement an abstract data type, and to empirically evaluate their performance. Thus, we stick to the original/simple idea of ‘trie’. You are strongly encouraged to read about more advanced data structures evolving from trie.

Each node of a trie contains the following fields:

- a lower-case letter from the English alphabet (‘a’ to ‘z’), or Null if it is the root,
- a *boolean* variable that is True if this letter is the last letter of a word in the dictionary and False otherwise,
- 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,
- an array of $A = 26$ elements (could be Null) storing references pointing to the children nodes. In our implementation, for convenience, we use a hashtable/Python’s dictionary to store the children.

As an example, consider Figure 1.

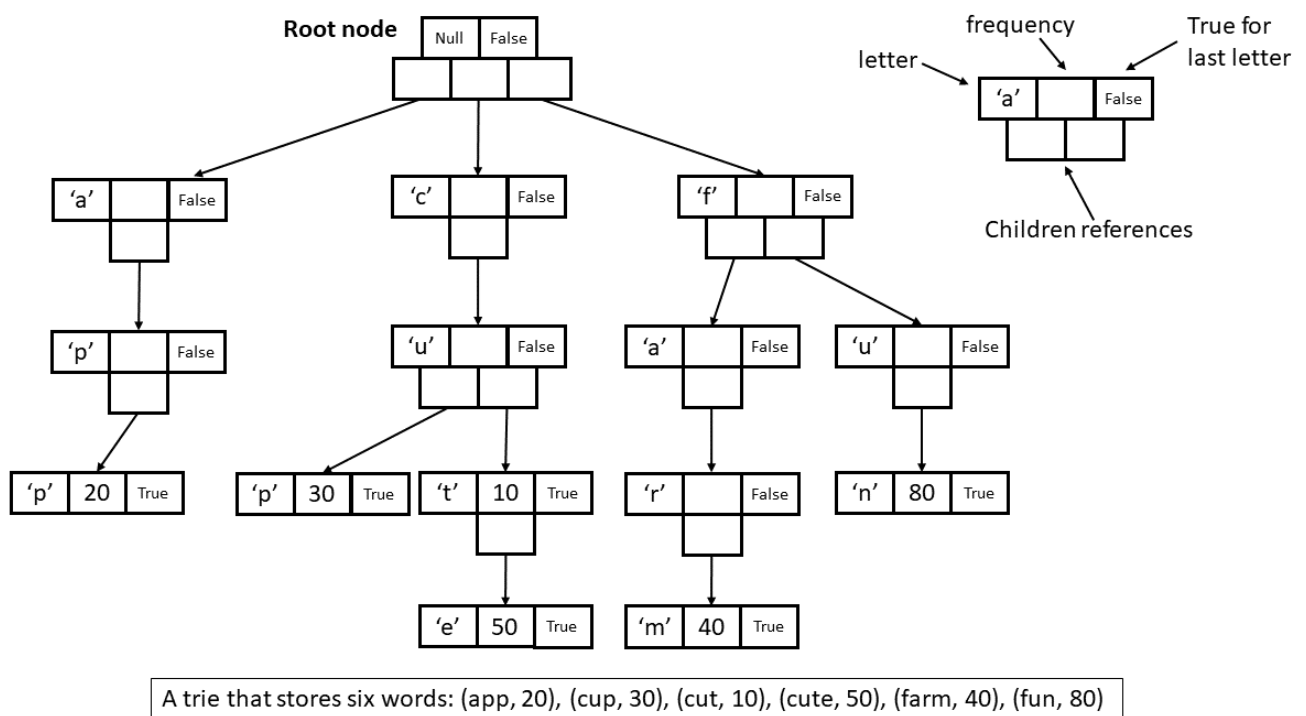


Figure 1: An example of a trie storing six words and their frequencies. The boolean value True indicates that the letter is the end of a word. In that case, a frequency (an integer) is shown, e.g., 10 for ‘cut’. Note that a word can be a prefix of another, e.g., ‘cut’ is a prefix of ‘cute’.

Construction. A trie can be built by simply adding words to the tree one by one (order of words being added is not important). If a new word is the prefix of an existing word in the trie then we can

simply change the boolean field of the node storing the last letter to True and update its frequency. 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, signifying that now ('cut', 10) is part of the tree. In another case, when ('cup', 30) is added, a new node has to be constructed to store 'p', which has a True in the boolean field and 30 as its frequency. In this case, 'cup' can reuse the two nodes storing 'c' and 'u' from the previous words.

Searching. To search for a word (and to get its frequency) in a trie, we use its letters to navigate the tree by following the corresponding child node. The search is successful if we can reach a node storing the last letter in the word and has the boolean field True. In that case, the frequency stored in this node is returned. The search fails, that is, the word is not in the tree, if either a) the search algorithm couldn't find a child node that matches a letter in the word, or b) it finds all the nodes matching all the letters of the words 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 trie in Figure 1, then we only need to change the boolean field of the node storing 't' to False. Otherwise, if the word has a unique suffix, then (only) nodes corresponding to the suffix are to be deleted. For example, if (cup, 30) is to be removed, then the node storing the last letter 'p' must be deleted from the trie to save space but not 'c' and 'u' because these still form part of other words.

Auto-completion. Auto-completion returns a list of three words (if any) in the dictionary (trie) of *highest frequencies* that have the given string as a prefix. For example, in the trie given in Figure 1,

- the auto-completion list for 'cu' is: [(cute, 50), (cup, 30), (cut, 10)],
- the auto-completion list for 'far' 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 *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 Dictionary and Its Operations Using Array, Linked List, and Trie (12 marks)

In this task, you will implement a dictionary of English words that allows **Add**, **Search**, **Delete**, and **Auto-completion**, using three different data structures: Array (Python's list), Linked List, and Trie. 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. This operation is not tested.
- (A)dd a word and its frequency to the dictionary. Return True if successful or False if it already exists in the dictionary.
- (S)earch for a word in a dictionary and return its frequency. Return 0 if not found.

- (D)delete a word from the dictionary. Returns True if successful and False if it doesn't exist in 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. The list can be empty.

3.1.1 Implementation Details

Array-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 standard operations on lists are allowed. Other data structures should NOT be used directly in the *main* operations of the array-based dictionary. The usage of the module 'bisect' is allowed. Other data structures should NOT be used directly in the *main* operations of the array-based dictionary. See the Background Section for more details and an example.

Linked-List-Based Dictionary. In this subtask, you will implement the dictionary by using a singly linked list. Other data structures should NOT be used directly in the *main* operations of the array-based dictionary (but Python's list can be used to store intermediate data or the input/output). See the Background Section for more details and an example.

Trie-Based Dictionary. In this subtask, you will implement the dictionary using the trie data structure. Both iterative or recursive implementations are acceptable. See the Background Section for more details and an example.

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. Unneeded nodes (after deletion) must be removed.
- **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 modules to your final submission, but please ensure that they work with the supplied modules and the Python test script.

Debugging. To run the code, from the directory where `dictionary_file_based.py` is, execute (use 'python3' on Linux, 'python' on Pycharm):

```
> python3 dictionary_file_based.py <approach> <data filename> <command filename>
                                     <output filename>
```

where

- `approach` is one of the following: `array`, `linkedlist`, `trie`,
- `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, use 'python3', on Pycharm's terminal, use 'python'):

file	description
dictionary_file_based.py	Code that reads in operation commands from file then executes those on the specified nearest neighbour data structure. For debugging your code. DO NOT MODIFY.
dictionary_test_script.py	Code that executes dictionary_file_based.py and tests your code on Linux servers. Make sure your implementation passes the tests before submission . DO NOT MODIFY.
base_dictionary.py	The base class for the dictionary. DO NOT MODIFY.
array_dictionary.py	Skeleton code that implements an array-based dictionary. COMPLETE all the methods in the class.
linkedlist_dictionary.py	Skeleton code that implements a linked-list-based dictionary. COMPLETE all the methods in the class.
trie_dictionary.py	Skeleton code that implements a trie-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.

```
> python3 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`, `test1.exp`, and `test2.in` and `test2.exp`.

Automark script. We use the Python script `dictionary_test_script.py` on our Linux servers to automate testing based on input files of operations (such as the example in Table 2). 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. The script can be run on a Linux server as follows (assuming you are in the folder containing the script and the test files).

```
> python3 dictionary_test_script.py $PWD <approach> <data filename> <command filename1>
<command filename2> ...
```

You can test multiple files (containing commands to be tested) simultaneously. For example,

```
> python3 dictionary_test_script.py $PWD array sampleData.txt test1.in test2.in
```

To mark 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 because `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 write codes on your own computer, **make sure they run without errors/warnings on these servers before submission**. If your codes do not run on the core teaching servers, we unfortunately won’t have the resources to debug each one and cannot award marks for testing.
- Please **avoid including non-standard Python modules not available on the servers**, as that will cause the test script to fail on your submission.
- All submissions should run with no warnings on **Python 3.10** on the core teaching servers. We will discuss how to install Python 3.10 on our server on a later post on the Ed Discussion Forum.

3.1.5 Test Data

We provide three sample datasets. The smallest one, `sampleDataToy.txt`, is for debugging, the medium one, `sampleData.txt`, which contains 5000 words from English-Corpora.org, is for testing and automarking, while the largest one, `sampleData200k.txt`, which contains 200k words from Kaggle, is for running experiments and report (you could use other data sources or generate yourself for the report as well). Each line in these files 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 implementations in terms of their time complexities for different operations. You will perform the empirical analysis and report the process and the outcome, and provide comparisons, comments, interpretations and explanations of the outcome, and your overall recommendations. The report should be no more than **5 pages**, in font size 12 and A4 pages (210 × 297 mm or 8.3 × 11.7 inches). 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 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 support a robust empirical analysis.

We suggest you to use datasets of various sizes (at least six sizes) ranging from small (e.g., 500, 1000), medium (e.g., 5000, 10000), to large (20000, 50000, 100000). You should explain how the words to be tested are generated in detail as well.

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 for evaluated operations, and must be reported in detail.

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, which method you decide to use for measuring the running times and how the running times in the report are collected from the experiments.
- Evaluation/Analysis of the outcome using the generated data. Analyse, compare and discuss your results across different parameter settings and data structures/algorithms. 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? If not, why? Please **use tables and graphs** to better illustrate your observations.
- Summarise your analysis as recommendations.

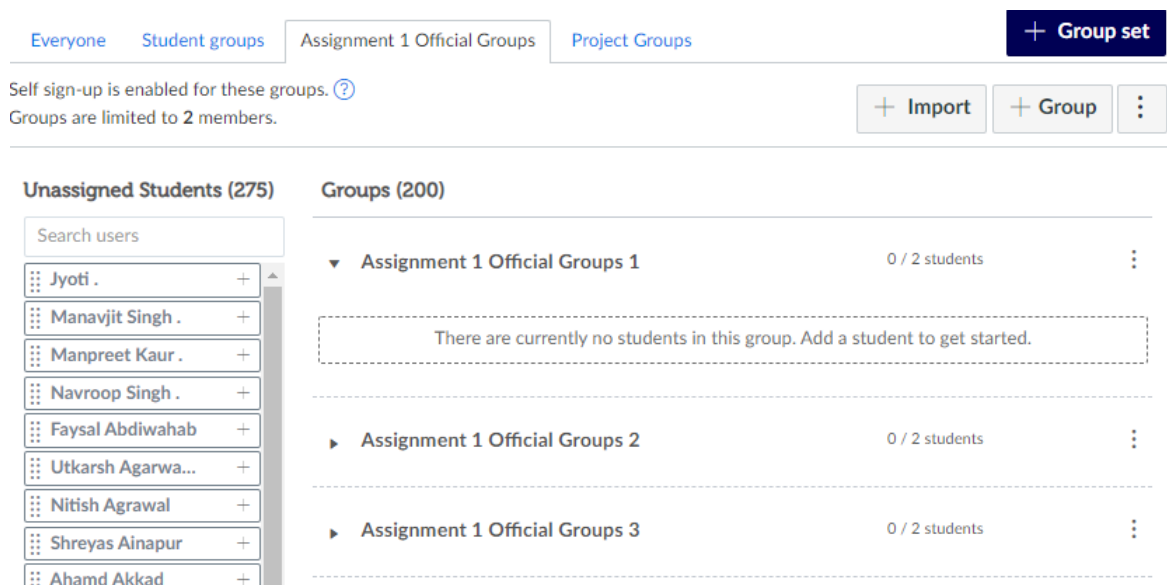


Figure 2: Add yourself (+ a team mate if any) to any available group. Do not create your own group.

5 Submission

We follow a **2-Step Submission Process** to facilitate marking in groups/individuals.

1. **Step 1** (Group registration): Go to Canvas → People → **Assignment 1 Official Group** and add yourself (with a team mate if any - see Fig. 2) to one group. Even when you work alone, please still choose a group and add yourself in to make the logistic of the marking/feedback process easier (faster to sort/search using group numbers than student numbers). **DO NOT** submit in the group nor create your own group!

2. Step 2 (Submission):

- Compress everything (code + report) into a single zip file named `Assign1-s12345-s67890.zip` (REPLACING with your student number so that when we batch decompress all submissions, your submission has a distinct name and won't be erased by others when decompressed) and submit in Assignment 1 page on Canvas.
- Follow SECTION 5 - Submission in Assignment Description to have the correct file/folder structure for your code.
- Submit: Make sure you submit the LATEST/CORRECT version of your code/report well before the deadline to avoid congestion, and VERIFY that the uploading has been done successfully. We will mark the version submitted last before the deadline. Any replacement after the deadline has passed will be marked with the penalty applied (3 marks per day).

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 3):
 - `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 the test script still runs.
 - `Assign1-s12345-s67890/generation` (generation files, see below).

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

- 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/`.

5.1 Clarification to Specifications & Submissions

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

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, September 16, 2022).

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

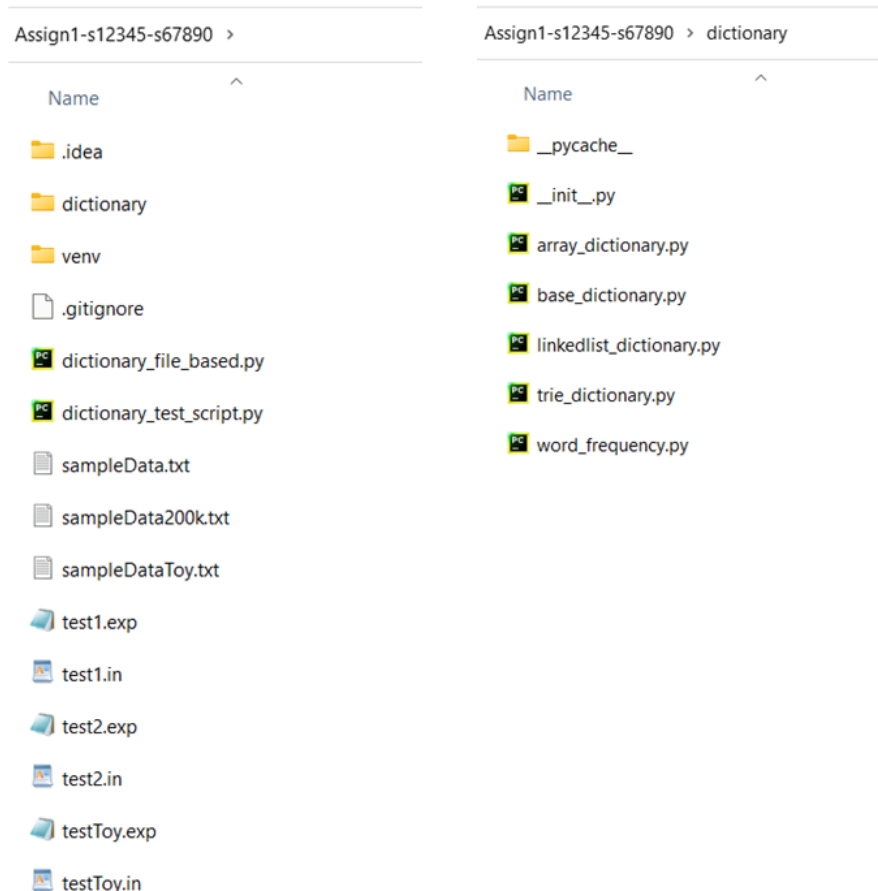


Figure 3: Please keep the above folder/files structure for the auto-test to run properly.

Task A: Implementation (12/30):

- Your implementation will be assessed on whether they implement the correct data structures and on the number of tests it passes in the automated tests.
- 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. Either case, you (and your partner, if any) must add yourself into an Official Group for Assignment 1 (see Section 5). 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 $\frac{M}{X}$.

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 as well as consultation sessions. We will also be posting common questions on the Assignment 1 Q&A section on Ed Discussion Forum and we encourage you to check regularly 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, Comprehensiveness (Maximum = 2 marks)
<p>4.5-6 marks</p> <p>Data generation is well designed, systematic and well explained with sufficient details. All suggested 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. The method used to measure the running times is clearly explained/justified.</p>	<p>8-10 marks</p> <p>Analysis is thorough and demonstrates an excellent level of understanding and critical analysis. Well-reasoned explanations and comparisons are provided for all the data structures/approaches and parameter settings (illustrated with high-quality graphs). All analysis, comparisons and conclusions are supported by empirical evidence and theoretical complexities. Well-reasoned recommendations are given.</p>	<p>1.5-2 marks</p> <p>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.</p>
<p>3-4 marks</p> <p>Data generation is reasonably designed, systematic and explained. There could be a missing data structure/approach or parameter setting. Each type of test was run over a number of runs and results were averaged. The method used to measure the running times is mentioned and explained/justified.</p>	<p>5-7.5 marks</p> <p>Analysis is reasonable and demonstrates good understanding and critical analysis. Adequate comparisons and explanations are made and illustrated with most of the data structures/approaches and parameter settings (illustrated with good-quality graphs). Most analysis and comparisons are supported by empirical evidence and theoretical analysis. Reasonable recommendations are given.</p>	<p>1 mark</p> <p>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.</p>
<p>1.5-2.5 marks</p> <p>Data generation is somewhat adequately designed, systematic and explained. There could be several obvious missing data structures/approaches or parameter settings. Each type of test may only have been run once. The method used to measure the running times is not or barely mentioned/justified.</p>	<p>3-4.5 marks</p> <p>Analysis is adequate and demonstrates some understanding and critical analysis. Some explanations and comparisons are given and illustrated with one or two data structures/approaches and parameter settings. A portion of analysis and comparisons are supported by empirical evidence and theoretical analysis. Adequate recommendations are given.</p>	<p>0-0.5 mark</p> <p>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.</p>
<p>0-1 mark</p> <p>Data generation is poorly designed, systematic and explained. There are many obvious missing data structures/approaches or parameter settings. Each type of test has only have been run once. The method used to measure the running times is not or barely mentioned.</p>	<p>0-2.5 marks</p> <p>Analysis is poor and demonstrates minimal understanding and critical analysis. Few explanations or comparisons are made and illustrated with one 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.</p>	