FIT2004 S1/2021: Assignment 1

DEADLINE: Friday 26th March 2021 23:55:00 AEDT

LATE SUBMISSION PENALTY: 10% penalty per day. Submissions more than 7 calendar days late will receive 0. The number of days late is rounded up, e.g. 5 hours late means 1 day late, 27 hours late is 2 days late. For special consideration, please visit this page:
`https://www.monash.edu/connect/forms/modules/course/special-consideration`
and fill out the appropriate form. **Do not** contact the unit directly, as we cannot grant special consideration unless you have used the online form.

PROGRAMMING CRITERIA: It is required that you implement this exercise strictly using the **Python programming language** (version should not be earlier than 3.5). This practical work will be marked on the time complexity, space complexity and functionality of your program, and your documentation.
Your program will be tested using automated test scripts. It is therefore critically important that you name your files and functions as specified in this document. If you do not, it will make your submission difficult to mark, and you will be penalised.

SUBMISSION REQUIREMENT: You will submit a single python file, `assignment1.py`

PLAGIARISM: The assignments will be checked for plagiarism using an advanced plagiarism detector. In previous semesters, many students were detected by the plagiarism detector and almost all got zero mark for the assignment and, as a result, many failed the unit. "Helping" others is NOT ACCEPTED. Please do not share your solutions partially or completely to others. If someone asks you for help, ask them to visit a consultation for help.

Learning Outcomes

This assignment achieves the Learning Outcomes of:

- 1) Analyse general problem solving strategies and algorithmic paradigms, and apply them to solving new problems;
- 2) Prove correctness of programs, analyse their space and time complexities;
- 4) Develop and implement algorithms to solve computational problems.

In addition, you will develop the following employability skills:

- Text comprehension
- Designing test cases
- Ability to follow specifications precisely

Assignment timeline

In order to be successful in this assessment, the following steps are provided as a **suggestion**. This is an approach which will be useful to you both in future units, and in industry.

Planning

1. Read the assignment specification as soon as possible and write out a list of questions you have about it.
2. Clarify these questions. You can go to a consultation, talk to your tutor, discuss the tasks with friends or ask in the forums.
3. As soon as possible, start thinking about the problems in the assignment.
    - It is strongly recommended that you **do not** write code until you have a solid feeling for how the problem works and how you will solve it.
4. Writing down small examples and solving them by hand is an excellent tool for coming to a better understanding of the problem.
    - As you are doing this, you will also get a feel for the kinds of edge cases your code will have to deal with.
5. Write down a high level description of the algorithm you will use.
6. Determine the complexity of your algorithm idea, ensuring it meets the requirements.

Implementing

1. Think of test cases that you can use to check if your algorithm works.
    - Use the edge cases you found during the previous phase to inspire your test cases.
    - It is also a good idea to generate large random test cases.
    - Sharing test cases **is** allowed, as it is not helping solve the assignment.
2. Code up your algorithm, (remember decomposition and comments) and test it on the tests you have thought of.
3. Try to break your code. Think of what kinds of inputs you could be presented with which your code might not be able to handle.
    - Large inputs
    - Small inputs
    - Inputs with strange properties
    - What if everything is the same?
    - What if everything is different?
    - etc...

Before submission

- Make sure that the input/output format of your code matches the specification.
- Make sure your filenames match the specification.
- Make sure your functions are named correctly and take the correct inputs.
- Make sure you zip your files correctly (if required)

Documentation (3 marks)

For this assignment (and all assignments in this unit) you are required to document and comment your code appropriately. This documentation/commenting must consist of (but is not limited to)

- For each function, high level description of that function. This should be a one or two sentence explanation of what this function does. One good way of presenting this information is by specifying what the input to the function is, and what output the function produces (if appropriate)
- For each function, the Big-O complexity of that function, in terms of the input. Make sure you specify what the variables involved in your complexity refer to. Remember that the complexity of a function includes the complexity of any function calls it makes.
- Within functions, comments where appropriate. Generally speaking, you would comment complicated lines of code (which you should try to minimise) or a large block of code which performs a clear and distinct task (often blocks like this are good candidates to be their own functions!).

1 Transaction Interval (8 marks)

Consider a large dataset of transaction records. Given a number `t`, we wish to determine which interval of length `t` contains the most transactions. All times in this question are measured in whole seconds after midnight 1/1/1970, i.e. they are non-negative integers.

To solve this problem, you will write a function `best_interval(transactions, t)`,

1.1 Input

`transactions` is a unsorted list of non-negative integers. Each integer in `transactions` represents the time that some transaction occurred.

`t` is a non-negative integer, representing a length of time in seconds.

1.2 Output

An **interval** is a set of numbers that contains all numbers lying between some starting number and some larger ending number (inclusive). These two numbers are called the endpoints of the interval. The length of an interval is the absolute difference between the two endpoints.

Consequently, and interval starting at **a** and ending at **b** has length **b-a**. Conversley, an interval starting at **a** of length **d** will end at **a**+**d**.

`best_interval` returns a two element tuple, `(best_t, count)`. `best_t` is the time such that the interval starting at `best_t` and ending at `best_t + t` contains more elements from `transactions` than any other interval of length `t`.

If there are multiple such intervals, return the interval with minimal start time. Note that this may mean the interval begins at a time which is not in `transactions` (see the example).

`count` is the number of elements in the interval of length `t` starting at `best_t`.

Example:

```
t = 5
transactions = [11, 1, 3, 1, 4, 10, 5, 7, 10]
>>> best_interval(transaction, t)
(0, 5)
```

1.3 Complexity

`best_interval` should run in $O(nk)$ time where

- n is the number of elements in `transactions`
- k is the greatest number of digits in any element in `transactions`

2 Anagrams (9 marks)

Consider the following problem: Given two lists of words, we wish to find all words in the first list which have an anagram in the second list.

Strings **r** and **s** are anagrams of one another if the characters in **s** can be permuted to form **r**. Trivially, if **r** = **s**, then they are anagrams of one another.

To solve this problem, you will write a function `words_with_anagrams(list1, list2)`.

2.1 Input

Both arguments, `list1` and `list2`, are lists of strings. All characters are lowercase a-z. Neither list contains duplicate strings, but there may be strings which appear in both lists.
Note: The strings in these lists need not be actual English words.

2.2 Output

A list of strings (in no particular order) from `list1` which have at least one anagram appearing in `list2`. Note that it **is** possible for two different strings in `list1` to share an anagram in `list2`.

2.3 Example

```
list1 = [spot, tops, dad, simple, dine, cats]
list2 = [pots, add, simple, dined, acts, cast]
>>> words_with_anagrams(list1, list2)
[cats, dad, simple, spot, tops]
```

Note: "spot" and "tops" share an anagram in `list2`, "pots". This means they should both be included in the output.
"cats" has two different anagrams in `list2`, "acts" and "cast". "cats" should still only appear once in the output.

2.4 Complexity

Your algorithm should run in $O(L_1M_1 + L_2M_2)$ where

- L_1 is the number of elements in `list1`
- L_2 is the number of elements in `list2`
- M_1 is the number of characters in the longest string in `list1`
- M_2 is the number of characters in the longest string in `list2`

Careful! The longest string in the input may be very long. Do not assume that operations that involve the length of a single string can be ignored/assumed to be O(1).

Warning

For all assignments in this unit, you may **not** use python **dictionaries** or **sets**. This is because the complexity requirements for the assignment are all deterministic worst case requirements, and dictionaries/sets are based on hash tables, for which it is difficult to determine the deterministic worst case behaviour.

Please ensure that you carefully check the complexity of each inbuilt python function and data structure that you use, as many of them make the complexities of your algorithms worse. Common examples which cause students to lose marks are **list slicing**, inserting or deleting elements **in the middle or front of a list** (linear time), using the `in` keyword to **check for membership** of an iterable (linear time), or building a string using **repeated concatenation** of characters. Note that use of these functions/techniques is **not forbidden**, however you should exercise care when using them.

These are just a few examples, so be careful. Remember, you are responsible for the complexity of every line of code you write!