FIT2004 S1/2021: Assignment 1

DEADLINE: Friday 20^{th} August 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 zip file, `assignment1.zip`. It will contain two files. A python file, `assignment1.py`, and a pdf, `explanation.pdf`.
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 to solve the assignment 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 Integer Radix Sort (9 marks)

Consider the Radix sort algorithm presented in lectures. As discussed in the second part of lecture 2, it is possible to vary the base used by this algorithm.

In this task you need to use radix sort to sort a given list of integers into ascending numerical order, **using a given base**. To do this, you will write a function `num_rad_sort(nums, b)`.

There will be a **video on moodle** which discusses some concepts related to representing numbers in different bases. If you are confused about any aspect of this task, please watch the video!

1.1 Input

- `nums` is a unsorted list of non-negative integers
- `b` is an integer, with value ≥ 2

1.1.1 FAQ

Q: In what base will the input be given?
A: This question does not really make sense. If the input list were a list of strings such as "14", then you would need to know what base was being used to interpret them. For example, "14" in base ten is different to "14" in base 5. Since the input list is composed of integers, not strings, each value in the input list is a numerical value, which is independent of a base.

Q: When I write a list like `[1,45,173]` is this not in base 10?
A: Python by default assumes that any numerical values written by the programmer are in base 10. This means that if we are writing a list of values into our code, we must write in base 10. We have to use a particular base because we are **representing** the numbers, and when numbers are being represented, they need a base. Once python has read the numbers in, there is no longer a need to represent them to the programmer, so their representation, and therefore their base, is no longer relevant to us.

Another way to think about this is that if Python decided to convert he numbers to a different base once it had read in the contents of the list, it would not affect any of the operations we might ask Python to perform on those numbers.

Q: How can a number not have a base?
A: A base is only relevant when we are talking about the **representation** of a number. The concept of **three**, for example, exists independently of its representation as a word, the digit "3", or the digits "11" (which is three in base 2), or indeed, ". . ." (which is three dots representing the quantity). All of these are just different ways of writing down the same numerical value.

Q: In what base should we give out output?
A: Since the output does not need to be printed, and is not a list of strings, the same answer applies as regarding the input. The output does not have a base.

Q: How does the parameter `b` affect the output?
A: It does not. Varying `b` while keeping `nums` unchanged will result in the same output every time. It does, however, change how the algorithm operates. If you do not correctly use the parameter `b`, you will not receive full marks for this task.

1.2 Output

`num_rad_sort` returns a list of integers. This list will contain exactly the same elements as `nums`, but sorted into ascending numerical order.

Example:

```
nums = [43, 101, 22, 27, 5, 50, 15]
>>>num_rad_sort(nums, 4)
[5, 15, 22, 27, 43, 50, 101]
```

1.3 Complexity

`num_rad_sort` should run in $O((n + b) * log_b M)$ time where

- n is the length of `nums`
- b is the value of `b`
- M is the numerical value of the maximum element of `nums`

2 Timing bases (9 marks))

In this task, you should **re-use** `num_rad_sort`.

We saw that varying the base in Task 1 did not change the output. However, it does have an effect on the **runtime** of radix sort. In this task we will investigate the relationship between the base and the runtime.

You will write a function `base_timer(num_list, base_list)` which you will use to investigate the relationship between the base used and the runtime.

2.1 Input

- `num_list` is a list of non-negative integers
- `base_list` is a list of integers, all with values ≥ 2, sorted ascending

2.2 Output

`base_timer` a list of numbers. Element i in this list is the time taken to run your radix sort from Task 1 on `num_list` using element i from `base_list` as the base.

Since the actual runtimes will vary between students due to differences in implementation and hardware, there are no marks for the exact values of the times you obtain as output. In other words, just because two students obtain different outputs for the same input does **not** mean that one student has an error.

The marks for this task come from the nature of the output, and your explanation of it (details in section 2.3)

2.3 Explanation

For this task, you will run `base_timer` on various inputs, and explain the results.

Insert the following code snippet into your assignment in order to create four lists of data and produce output for them:

```
random.seed("FIT2004S22021")

data1 = [random.randint(0,2**25) for _ in range(2**15)]
data2 = [random.randint(0,2**25) for _ in range(2**16)]

bases1 = [2**i for i in range(1,23)]
bases2 = [2*10**6 + (5*10**5)*i for i in range(1,10)]

y1 = base_timer(data1, bases1)
y2 = base_timer(data2, bases1)
y3 = base_timer(data1, bases2)
y4 = base_timer(data2, bases2)
```

In `explanation.pdf` document, include 2 graphs. One comparing `y1` and `y2`, with `bases1` as the horizontal axis. The other comparing `y3` and `y4`, with `bases2` as the horizontal axis.

For the first graph, you should use a **logarithmic** scale on your horizontal axis, but for the second graph you should use a **linear** scale.

The vertical axis corresponds to the **runtimes**. This axis should **not** use a logarithmic scale.

In the same pdf, answer the following questions:

1. Why do the base/time curves for the first two graphs show a U shape? In other words, why are the times high when the base is low and when the base is high, but low when the base is in between? Justify your answer using the complexity of radix sort.
2. Why are the times for `y2` about twice as long as for `y1` when the base is low? Include a mathematical arguement based on the complexity of radix sort in your answer.
3. Why are the times for `y2` **not** twice as long as for `y1` (and in fact are very close) when the base is high? Include a mathematical arguement based on the complexity of radix sort in your answer.
4. Why are the times for `y3` and `y4` almost the same, despite `data2` having twice as many elements as `data1`? Include a mathematical arguement based on the complexity of radix sort in your answer.
5. Why do the graphs for for `y3` and `y4` show an almost linear shape? Include a mathematical arguement based on the complexity of radix sort in your answer.

2.4 Complexity

The overall complexity for this task is not important.

3 Interest Groups (9 marks)

Consider a database consisting of people, each of whom like a set of things. We want to create groups of people with identical interests.

To do this, you will write a function `interest_groups(data)`.

3.1 Input

`data` is a list, where each element is a 2-element tuple representing a person. The first element is their name, which is a nonempty string of lowercase a-z with no spaces or punctuation. Every name in the list is unique.

The second element is a nonempty list of nonempty strings, which represents the things this person likes. The strings consist of lowercase a-z and also spaces (i.e. they can be multiple words) but no other characters. This list is in no particular order.

3.2 Output

`interest_groups` returns a list of lists. For each distinct set of liked things, there is a list which contains all the names of the people who like **exactly those things**. Within each list, the names should appear in ascending alphabetical order. **The lists may appear in any order.**

Example:

```
data = [("nuka", ["birds", "napping"]),
        ("hadley", ["napping birds", "nash equilibria"]),
        ("yaffe", ["rainy evenings", "the colour red", "birds"]),
        ("laurie", ["napping", "birds"]),
        ("kamalani", ["birds", "rainy evenings", "the colour red"])]

>>> interest_groups(data)
[["laurie", "nuka"], ["hadley"], ["kamalani", "yaffe"]]
```

3.3 Complexity

Input constraint: If we call the number of strings in any list of liked things X and the length of the longest string in that list Y, then XY is O(M)

`interest_groups` must run in $O(NM)$

- N is the number of elements in `data`
- M is the maximum number of characters among all sets of liked things. You may assume that all names are also shorted than M characters.

In the example above, $N = 5$ (there are 5 people), and $M = 33$, since there are 33 characters in the sets belonging to yaffe and kamalani, and they are the longest sets.

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!