FIT2004 S1/2020: Assignment 1

DEADLINE: Friday 10th April 2020 23:55:00 AEST

LATE SUBMISSION PENALTY: 10% penalty per day. Submissions more than 7 days late are generally not accepted. 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 complete and send the *in-semester special consideration* form with appropriate supporting document **before the deadline** to `fit2004.allcampuses-x@monash.edu`.

PROGRAMMING CRITERIA: It is required that you implement this exercise strictly using **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.
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 zipped file (named `studentId_A1.zip`, e.g. if your student id is XXXX, the name of zipped file must be `XXXX_A1.zip`). It should contain a single python file, `assignment1.py`, and a single pdf file, `task2.pdf`

PLAGIARISM: The assignments will be checked for plagiarism using an advanced plagiarism detector. Last year, 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/and completely to others. If someone asks you for help, ask them to visit us during consultation hours 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

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 Sorting with radix sort (9 marks)

In this task you will be implementing radix sort. You will write a function `radix_sort(num_list, b)` which takes as input a list of numbers to be sorted, and a base. You need to sort those numbers using radix sort in the base specified by the second parameter.

1.1 Input

The first input to this task is a list containing positive integers. They will be in the range $[1, 2^{64} - 1]$. The second input is an integer in the range $[2, \text{inf})$, which is the base you need to use in your radix sort.

Example:
`b` = 10, `num_list` contains the following:

```
[18446744073709551615,
18446744073709551614,
1,
1111111111111111111,
2111111111111111111,
3111111111111111111]
```

1.2 Output

`radix_sort` will return a list of integers, sorted in ascending order. This list will contain the same integers as the input list. `radix_sort` **should not** modify the input list.

Example:
Calling `radix_sort` on the above example would return the list

```
[1,
3111111111111111111,
2111111111111111111,
1111111111111111111,
18446744073709551614,
18446744073709551615]
```

Note that the output would be the same regardless of the base used, 10 is used here as an example.

1.3 Complexity

`radix_sort` must run in $O((N + b)M)$ time, where

- N is the total number of integers in the input list
- b is the base
- M is the number of digits in the largest number in the input list, when represented in base `b`

2 Analysis (6 marks)

Since your function from task 1 allows you to vary the base used for radix sort, you will now test the effect that varying the base has on the run time of your algorithm. You will write a function `time_radix_sort()` which will create a list of numbers, and then call `radix_sort` on this list with different bases and record the times. Use this line of code (and the random module) to create your test data:

```
test_data = [random.randint(1,(2**64)-1) for _ in range(100000)]
```

You should test your function for an appropriate range of bases. (it is up to you to think about what range of bases might be appropriate, but you should test a large range of values, and at least 5 different values)

In task2.pdf, discuss the results of this test. This documents must include

- An explanation of why you chose the bases that you did
- A graph showing the times obtained, along with the bases
- A written explanation of why you obtained the times that you did. Justify the times using your understanding of radix sort.

Remember to not include the creation of the test data in your timing. You should only be timing the running of your function from task 1.

2.1 Input

The function `time_radix_sort()` has no input.

2.2 Output

`time_radix_sort()` will produce a list of tuples as output. Each tuple will correspond to the time for one base, and will have two elements. The first element will be the base used, and the second will be the time in seconds.

Example:
Suppose that the bases tested were 2, 3 and 4 (note, this is **not** a good set of bases to test). The return value of `time_radix_sort()` might (times may vary) be

```
[(2, 5.323850631713867),
(3, 3.4008474349975586),
(4, 2.6056511402130127)]
```

2.3 Complexity

There is no complexity requirement for this function.

3 Finding rotations (12 marks)

Background

This task is about rotations of strings. A left rotation of a string is when we take the leftmost character and place it on the right end of the string. For example, left rotating `abcd` gives us `bcda`. We can also rotate more than once. For example, left rotating `abcd` twice gives us `cdab`. We call this a "2-rotation", and in general, when we rotate a number p places to the left, we call that an "p-rotation".

-1-rotation	`dabc`
original	`abcd`
1-rotation	`bcda`
2-rotation	`cdab`
3-rotation	`dabc`
4-rotation	`abcd`
5-rotation	`bcda`

In this task, you will be given a list of strings and a rotation size p. You need to find all the strings in the list whose p-rotations also appear in the list. To do this you will write a function `find_rotations(string_list, p)`

3.1 Input

The first input to this task, `string_list` is a list of strings. Each string in this list is unique. Strings in this list will only contain lowercase alphabet characters (a-z). `p` is the number of left-rotations. Negative p values correspond to right rotations.

Example:
`p` = 1, `string_list` contains

```
["aaa",
"abc",
"cab",
"acb",
"wxyz",
"yzwx"]
```

3.2 Output

`find_rotations` will return a list of all string in `string_list` whose `p`-rotations also exist in `string_list`

Example:
Calling `find_rotations` on the above example would return

```
["aaa", "cab"]
```

To see why, first lets rotate all the strings one place left

```
aaa
bca
abc
cba
xyzw
zwxy
```

The two rotated strings which appear in the original list are `"aaa"` and `"cab"`, so these are the two elements in the solution.

3.3 Complexity

`find_rotations` must run in $O(NM)$, where

- N is the number of strings in the input list
- M is the maximum number of letters in a word, among all words in the input list

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!