FIT2004 2024 Semester 1: Assignment 2

DEADLINE: Wednesday 22^{th} May 2024 23:55:00 AEST.

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 seconds late means 1 day late, 27 hours late is 2 days late.

For special consideration, please visit the following page and fill out the appropriate form: `https://forms.monash.edu/special-consideration`.

The deadlines in this unit are strict, last minute submissions are at your own risk.

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 containing all of the questions you have answered, `assignment2.py`. Moodle will not accept submissions of other file types.

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 (or even zero marks for the unit as penalty) and, as a result, the large majority of those students failed the unit. Helping others to solve the assignment is NOT ACCEPTED. Please do not share your solutions partially or completely to others. Even after the deadline, your solutions/approaches should not be shared before the grades and feedback are released by the teaching team. Using contents from the Internet, books etc without citing is plagiarism (if you use such content as part of your solution and properly cite it, it is not plagiarism; but you wouldn't be getting any marks that are possibly assigned for that part of the task as it is not your own work).

The use of generative AI and similar tools for the completion of your assignment is not allowed in this unit!

Learning Outcomes

This assignment achieves the Learning Outcomes of:

- Analyse general problem solving strategies and algorithmic paradigms, and apply them to solving new problems;
- Prove correctness of programs, analyse their space and time complexities;
- Compare and contrast various abstract data types and use them appropriately;
- 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. Try to resolve these questions by viewing the FAQ on Ed, or by thinking through the problems over time.
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.
- Remove print statements and test code from the file you are going to submit.

Documentation

For this assignment (and all assignments in this unit) you are required to document and comment your code appropriately. Whilst part of the marks of each question are for documentation, there is a baseline level of documentation you must have in order for your code to receive marks. In other words:

Insufficient documentation might result in you getting 0 for the entire question for which it is insufficient.

This documentation/commenting must consist of (but is not limited to):

- For each function, high-level description of that function. This should be a two or three sentence explanation of what this function does.
- Your main function in the assignment should contain a generalised description of the approach your solution uses to solve the assignment task.
- For each function, specify what the input to the function is, and what output the function produces or returns (if appropriate).
- For each function, the appropriate Big-O or Big-Θ time and space complexity of that function, in terms of the input size. 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!).

A suggested function documentation layout would be as follows:

```
def my_function(argv1, argv2):
    """
    Function description:

    Approach description (if main function):

    :Input:
        argv1:
        argv2:
    :Output, return or postcondition:
    :Time complexity:
    :Time complexity analysis:
    :Space complexity:
    :Space complexity analysis:
    """
    # Write your codes here.
```

There is a documentation guide available on Moodle in the Assignment section, which contains a demonstration of how to document code to the level required in the unit.

1 Open reading frames (10 marks, including 2 marks for documentation)

In Molecular Genetics, there is a notion of an Open Reading Frame (ORF). An ORF is a portion of DNA that is used as the blueprint for a protein. All ORFs start with a particular sequence, and end with a particular sequence.

In this task, we wish to find all sections of a genome which start with a given sequence of characters, and end with a (possibly) different given sequence of characters.

To solve this problem, you will need to create a class `OrfFinder`. The constructor for this class takes a string `genome` as a parameter. Additionally, this class will need a method `find(start, end)`.

1.1 Input

`genome` is a single non-empty string consisting only of uppercase [A-D]. `genome` is passed as an arguement to the `__init__` method of `OrfFinder` (i.e. it gets used when creating an instance of the class).

`start` and `end` are each a single non-empty string consisting of only uppercase [A-D].

1.2 Output

`find` returns a list of strings. This list contains all the substrings of `genome` which have `start` as a prefix and `end` as a suffix. There is no particular requirement for the order of these strings. `start` and `end` must not overlap in the substring (see the last two cases of the example below).

1.3 Example

```
genome1 = OrfFinder("AAABBBCCC")
genome1.find("AAA","BB")
>>> ["AAABB","AAABBB"]
genome1.find("BB","A")
>>>[]
genome1.find("AA","BC")
>>>["AABBBC","AAABBBC"]
genome1.find("A","B")
>>> ["AAAB","AAABB","AAABBB","AAB","AABB","AABBB","AB","ABB","ABBB"]
genome1.find("AA","A")
>>> ["AAA"]
#note that "AA" is not valid, since start and end would need to overlap
genome1.find("AAAB","BBB")
>>> []
# note that "AAABBB" is not valid, since start and end would need to overlap
```

1.4 Complexity Requirements

- The `__init__` method of `OrfFinder` must run in time complexity $O(N^2)$, where N is the length of `genome`.
- Let T be the length of the string `start`, U be the length of the string `end`, and V be the number of characters in the output list (for a correctly generated output list according to the instructions in 1.2), then `find` must run in time complexity $(T + U + V)$.

As an example of what the complexity for `find` means, consider a string consisting of $\frac{N}{2}$ "B"s followed by $\frac{N}{2}$ "A"s. If we call `find("A","B")`, the output is empty, so V is $O(1)$. On the other hand, if we call `find("B", "A")` then V is $O(N^2)$.

2 Securing the companies (10 marks, including 2 marks for documentation)

Assume that you are the manager of a security company tasked with assigning security officers (n) to a group of companies (m) in a certain month with 30 days numbered $\texttt{D}_0, \texttt{D}_1, \ldots, \texttt{D}_{29}$. Assume that security officers are denoted by $\texttt{SO}_0, \texttt{SO}_1, \ldots \texttt{SO}_{n-1}$ and the companies are represented by $\texttt{C}_0, \texttt{C}_1, \ldots \texttt{C}_{m-1}$. There are three 8-hour shifts per day: $\texttt{S}_0$ (midnight-8am), $\texttt{S}_1$ (8am-4pm), and $\texttt{S}_2$ (4pm-midnight). For each pair company/shift $(\texttt{C}_j, \texttt{S}_k)$, the company $\texttt{C}_j$ has a different number of security officers required for the shift $\texttt{S}_k$, however, this is constant for each day of the month.

The security officers specify the shifts they are interested in working on, and they have the opportunity to indicate more than one (1) preferred shift. However, they cannot be allocated to more than one (1) shift per day. Further, the shift preference of each security officer should be the same for all the days of the month (and therefore specified only once per security officer).

Your task is to provide a plan for assigning officers to the companies for the coming month according to the requirement of the number of security officers per shift by each company. For each day of the month, and for each shift, each company should be allocated exactly the same number of security officers it requests. There are 2 integer parameters, $0 \leq$`min_shifts`≤ 30 (minimum number of shifts) and $0 \leq$`max_shifts` ≤ 30 (maximum number of shifts), and the total number of shifts allocated to each security officer in a month should be within these two integer parameters (closed interval).

As long as you comply with the constraints above, you are free to allocate the security officers any way you want.

The input to your function `allocate` consists of a list of preferred shifts provided by each security officer and the number of security officers requested by the companies for each shift: `preferences` is a list of lists, where `preferences[i][k]`, is a binary value (0 or 1), indicating whether security officer $\texttt{SO}_i$ is interested in working on shift $\texttt{S}_k$ in that month (1 indicates that the officer is interested). `officers_per_org` is a list of lists, where `officers_per_org[j][k]` is a non-negative number that specifies how many security officers company $\texttt{C}_j$ needs for shift $\texttt{S}_k$ on each day of that month.

Your task is to implement a function `allocate(preferences, officers_per_org, min_shifts, max_shifts)` which returns:

- `None` (i.e., Python NoneType), if no allocation satisfying all constraints exists.
- Otherwise, it returns a list of lists `allocation`, where `allocation[i][j][d][k]` is equal to 1 if security officer $\texttt{SO}_i$ is allocated to work for company $\texttt{C}_j$ during shift $\texttt{S}_k$ on day $\texttt{D}_d$.

Complexity requirement: The worst-case time complexity of your solution should be $O(m * n * n)$.

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 in-built 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.

Please be reasonable with your submissions and follow the coding practices you've been taught in prior units (for example, modularising functions, type hinting, appropriate spacing). While not an otherwise stated requirement, extremely inefficient or convoluted code will result in mark deductions.

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