FIT2004 S1/2022: Assignment 3

DEADLINE: Friday 6th May 2022 16:30:00 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 hours late means 1 day late, 27 hours late is 2 days late. For special consideration, please visit this page: `https://forms.monash.edu/special-consideration` and fill out the appropriate 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, `assignment3.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 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;
- 3) Compare and contrast various abstract data types and use them appropriately;
- 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

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 two or three sentence explanation of what this function does and the approach undertaken within the function.
- 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 Big-O 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):
    """
    High level description about the functiona and the approach you
    have undertaken.
    :Input:
        argv1:
        argv2:
    :Output, return or postcondition:
    :Time complexity:
    :Aux space complexity:
    """
    # Write your codes here.
```

1 Salesperson Revenue (4 marks + 1 mark for documentation)

You are a travelling salesperson that sells your products in n cities (numbered $0, 1, \ldots, n-1$) and you are trying to decide your schedule for the next d days (numbered $0, 1, \ldots, d-1$) in order to maximize your revenue. You need to decide when it is better to sell on the city you are located in and when it is better to move to another city.

As a super-competent salesperson, you already know how much revenue you would be able to make on each day in each city. `revenue` is a list of lists. All interior lists are length n. Each interior list represents a different day. `revenue[z][x]` is the revenue that you would make if you work in city `x` on day `z`.

You also have information about travel routes. `travel_days` is a list of lists. `travel_days[x][y]` will contain either:

- A positive integer number indicating the number of days you need to spend to travel on the direct road from city x to city y.
- -1, to indicate that there is no direct road for you to travel from city x to city y.

At the beginning of each day z, if you are located in some city x, you have the following options:

- Spend the whole day selling in that city, making a revenue of `revenue[z][x]` on day z.
- Start a travel on the direct road to some city y. In this case this travel will start in city x at the beginning of day z and will finish in city y at the end of day $z+$`travel_days[x][y]`-1 (and you will not be able to do anything other than travel on the road from city x to city y on days $z, \ldots, z+$`travel_days[x][y]`-1).

Let `start` denote the city you start in on day 0.

To solve this problem, you should write a function `best_revenue(revenue, travel_days, start)` that returns the maximum possible revenue.

1.1 Complexity

Your solution should have a worst-case time complexity of $O\left(n^2\left(d+n\right)\right)$.

2 Saving the Multiverse (4 marks + 1 mark for documentation)

You are Dr Weird, gifted with the power travel across the multiverse. Your archnemesis Master X has duplicated himself and sends copies of himself across the multiverse.

Through the power of the echo, you are able to see when Master X would attack each multiverse:

- Multiverse 1, from day 2 to day 7 with a force of 5 clones.
- Multiverse 2, from day 1 to day 4 with a force of 4 clones.
- Multiverse 3, from day 6 to day 9 with a force of 2 clones.
- ... and so on.

Through masterful planning, Master X is able to coordinate his attacks and you can only be at a single multiverse at a single moment of time. Thus, you would need to decide which multiverse to be at to stop him; leaving the rest to each multiverse's defenders. In the example above, you can either:

- Defend multiverse 1 from day 2 to day 7; defeating 5 clones; or
- Defend multiverse 2 from day 1 to day 4 and then head to multiverse 3 from day 6 to day 9. In total, you would have defeated 6 clones.

Your goal is to defeat the most Master X clones you can in order to weaken him. In order to do so, you wrote a simple Python function `hero(attacks)` to quickly determine the optimal multiverse travel before Master X rules the multiverse.

PS: You can assume that you will always defeat the Master X clones. As a hero, you would always find a way and wouldn't need any hints.

2.1 Input

`attacks` is a non-empty list of N attacks, where each attack is a list of 4 items `[m, s, e, c]`:

- `m` is the multiverse which Master X is attacking.
  - `m` is an integer in the range of 1 to N.
  - Master X will only attack each multiverse once; because he do not like setbacks.
- `s` and `e` are the starting and ending days of the attack.
  - `s` and `e` are integers in the range of 1 to D.
  - You can assume that `s <= e`.
  - You would need to be throughout the entire attack duration from day `s` to day `e` inclusive in order to defeat the clones.
- `c` is the number of Master X clones in the attack.
  - `c` is an integer in the range of 2 to C.
  - Master X will always attack with at least 2 clones because he needs friends.

2.2 Output

Your algorithm returns a list of Master X attacks that you, Dr Weird, will attend to:

- Result in the most Master X clones defeated.
- The returned list doesn't need to be sorted.
- It is possible for the return list to not be unique.

2.3 Examples

Consider the following examples below.
The first example is based on the simple example given in the question description where a total of 6 clones were defeated.

```
# Example 01
attacks = [[1, 2, 7, 5], [2, 1, 4, 4], [3, 6, 9, 2]]

>>>hero(attacks)
[[3, 6, 9, 2], [2, 1, 4, 4]]
```

For the second example below, we observe 2 possible solutions where either $[[3,5,6,2],[2,1,4,4]]$ or $[[1,2,7,6]]$ would result in 6 clones being defeated.

```
# Example 02 (not unique)
attacks = [[1, 2, 7, 6], [2, 1, 4, 4], [3, 5, 6, 2]]

>>>hero(attacks)
[[3, 5, 6, 2], [2, 1, 4, 4]]
```

In the last example below, it can be observed that upon defeating the clones in the attack $[1,2,7,6]$ you can only move on to attack $[3,8,9,5]$ because attack $[2,7,9,10]$ starts on day 7 itself when you are still occupied.

```
# Example 03 (ending condition overlap)
attacks = [[1, 2, 7, 6], [2, 7, 9, 10], [3, 8, 9, 5]]

>>>hero(attacks)
[[3, 8, 9, 5], [1, 2, 7, 6]]
```

2.4 Complexity

`hero(attacks)` should run in $O(NlogN)$ worst case time complexity and $O(N)$ auxiliary space complexity, given that N is the number of attacks in `attacks`.

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.

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