FIT2004 S1/2020: Assignment 2 - Dynamic Programming

DEADLINE: Friday 1st May 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 single python file, `assignment2.py`.

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 may see patterns which allow you to deduce the correct algorithm to solve the problem.
    - You will also get a feel for the kinds of edge cases you will need to handle.
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. Check the forums for test cases!
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

Special requirements

For this assignment, there are two tasks to complete. As usual, there are required complexities, but for each of these two task, there are **two different** complexities listed. One complexity we will refer to as the **optimal** complexity, and the other complexity we will refer to as the **sub-optimal** complexity.

In order to receive full marks for the assignment, you must submit an algorithm with the **optimal** complexity for **at least one** of the two tasks. If you complete both tasks within the **sub-optimal** complexity, but not the **optimal** complexity, then the maximum mark you can receive for the whole assignment is 80% (24/30).

If one or both of your tasks has a complexity worse than the **sub-optimal** complexity listed in the task description, your mark will be significantly lower (as usual).

For each of the two functions, please **clearly state** in the function documentation whether the function has been implemented **optimally** or **sub-optimally**. Note that you are allowed to submit two sub-optimal tasks! This choice is so that if you cannot see how to solve the problems optimally, you still have a chance to get most of the marks.

Summary

Submission	**Maximum Mark**
Either implementation worse than **sub-optimal**	<24/30
Both implemented **sub-optimally**	24/30
One task implemented **optimally**, one task implemented **sub-optimally**	30/30
Both tasks implemented **optimally**	30/30

Documentation (2 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 Oscillations (14 marks)

In this task, you will find the longest oscillation in a given list. To do this you will write a function `longest_oscillation(L)`.

1.1 Input

A list of integers, `L`. The list can contain duplicates or be empty.

1.2 Output

Given a list `L` of length `m`, we define an oscillation as a (possibly empty) sequence of increasing indices of `L` $a_1, a_2, ...a_n$ such that

- `L[`a_j`]` $\neq$ `L[`a_{j+1}`]`
- if `L[`a_j`] < L[`a_{j+1}`]`, then `L[`a_{j+1}`] > L[`a_{j+2}`]`
- if `L[`a_j`] > L[`a_{j+1}`]`, then `L[`a_{j+1}`] < L[`a_{j+2}`]`

Your function should return the length of the longest oscillation in `L`, and the indices in `L` at which it occurs. It should do this by returning a tuple, where the first element of the tuple is a number, which represents the length of the oscillation. The second element is a list which contains the indices of the elements in L which make up the oscillation.
The values in this list should be in ascending order (i.e. the indices should be in the same order that they are in `L`).

Example:
`longest_oscillation([1,5,7,4,6,8,6,7,1]) returns (7, [0,2,3,5,6,7,8])`.
This corresponds to the red values from `L`: ([1,5,7,4,6,8,6,7,1])

`longest_oscillation([1,1,1,1,1]) returns (1, [0])`
This corresponds to the red values from `L`: ([1,1,1,1,1])

`longest_oscillation([1,2,3]) returns (2, [0,1])`
This corresponds to the red values from `L`: ([1,2,3])

Note: Some lists may have multiple longest oscillations, you may return any one of them. Do not return more than one.
As an example, valid return values for `longest_oscillation([1,2,3])` are `(2, [0,1])`, `(2, [0,2])` and `(2, [1,2])`.

1.3 Complexity

Given an input list of length N:

1.3.1 Sub-optimal

- `longest_oscillation` must run in $O(N^2)$ time
- `longest_oscillation` must use $O(N)$ auxiliary space

1.3.2 Optimal

- `longest_oscillation` must run in $O(N)$ time
- `longest_oscillation` must use $O(N)$ auxiliary space

2 Increasing walk (14 marks)

Given a two-dimensional matrix of numbers, you will find the longest increasing walk in the matrix. To do this, you will write a function `longest_walk(M)`.

2.1 Input

The input, `M`, is a list of n lists, with each inner list being length m, and containing only integers. `M` can be thought of as an $n \times m$ matrix, with row i of the matrix being represented by `M[i]`. Thus `M[i][j]` represents the value in row i, column j of the matrix. We will refer to M as a matrix from this point onward.

`M` can contain duplicates, and can be empty.

2.2 Output

An increasing walk in a matrix M is a sequence of values of M which are

- sequentially adjacent (this can be horizontally, vertically or diagonally)
- each value in the sequence is greater than the previous value

Your function should return the length of the longest increasing walk in `M`, and the co-ordinates of the elements in that walk, in order. It should do this by returning a tuple, where the first element of the tuple is a number representing the length of the longest walk in `M`. The second element in the tuple is a list of 2-element tuples. These tuples are the (row, column) co-ordinates of the elements of `M` which make up the longest increasing walk, in the same order as they would be traversed during the walk.

Example:

```
M = [[1,2,3],
     [4,5,6],
     [7,8,9]]
longest_walk(M) = (7, [(0,0), (0,1), (0,2), (1,1), (1,2), (2,1), (2,2)])
```

```
M = [[1,2,3],
     [1,2,1],
     [2,1,3]]
longest_walk(M) = (3, [(0,0), (1,1), (2,2)])
```

```
M = [[4,6],
     [7,2]]
longest_walk(M) = (4, [(1,1), (0,0), (0,1), (1,0)])
```

Note: As in task 1, there may be multiple valid return values for a given input. You may return any one of them. Do not return more than one.

2.3 Complexity

Given an input matrix of n rows and m columns.

2.3.1 Sub-optimal

- `longest_walk` must run in $O(nm \log(nm))$ time
- `longest_walk` must use $O(nm)$ auxiliary space

2.3.2 Optimal

- `longest_walk` must run in $O(nm)$ time
- `longest_walk` must use $O(nm)$ auxiliary space

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!