FIT2004 S2/2021: Assignment 2 - Dynamic Programming

DEADLINE: Friday 10^{th} September 2021 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 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 **or** use the google form linked on the Moodle page for short (< 5) day extensions.

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, `assignment2.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;
- 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 Game Master (9 marks)

You and your friends are playing a table-top role playing game, and you are the game master. You want to design an encounter, and you have a certain difficulty target that you want to reach. You also have a list of monsters, and you want to select a group of monsters whose difficulty ratings sum up to the target difficulty.

You are interested in finding out how many different possible encounters there are which satisfy this requirement. To solve this problem, you will write a function `count_encounters(target_difficulty, monster_list)`.

1.1 Input

`target_difficulty` is a non-negative integer.

`monster_list` is a list of tuples. Each tuple represents a type of monster. The first value in each tuple is a string, which is the name of the type of monster. The second value is an integer, representing the difficulty of that particular type of monster.

1.2 Output

`count_encounters` returns an integer, which is the number of different sets of monsters whose difficulties sum to `target_difficulty`. A type of monster may be used more than once in an encounter.

1.3 Example

```
target_difficulty = 15
monster_list = [("bear", 5), ("imp", 2), ("kobold", 3), ("dragon", 10)]
print(count_encounters(target_difficulty, monster_list))
>>> 9
```

In the above example, the possible encounters are:

```
1 dragon, 1 bear
1 dragon, 1 kobold, 1 imp
3 bear
2 bear, 1 kobold, 1 imp
1 bear, 2 kobold, 2 imp
1 bear, 5 imp
5 kobold
3 kobold, 3 imp
1 kobold, 6 imp
```

Your answer does **not** need to compute these possible sets of monsters, this list is provided to help you understand the example answer of 9

1.4 Complexity

`count_encounters` should run in $O(DM)$ where

- D is the value of `target_difficulty`
- M is the length of `monster_list`

2 Greenhouse (18 marks)

You are a gardener in charge of a greenhouse. You are growing a variety of exotic plants as decoration for a party. You want to maximise the chance that all the plants are grown by the day of the party. In order to cause plants to grow more quickly, you can put sun lamps above the plants to give them more nutrients.

You have calculated the probability that each plant will be ready on time, based on the number of lamps you assign to it. You cannot move the lamps around once you assign them to a plant. You want to maximise the chance that all the plants are fully grown by the day of the party. To solve this problem, write a function `best_lamp_allocation(num_p, num_l, probs)`

2.1 Input

`num_p` and `num_l` are positive integers representing the number of plants and lamps respectively.

`probs` is a list of lists, where `probs[i][j]` represents the probability that plant `i` will be ready in time if it is allocated `j` lamps. Values in `probs` are floats between 0 and 1 inclusive.

Plants do not always grow faster with more light, so it is possible for the probabilities to decrease as well as increase, as the number of lamps increases. In other words, the lists within `probs` **may not** be sorted ascending.

2.2 Output

`best_lamp_allocation` returns an float, which is the highest probability of all plants being ready by the party that can be obtained by allocating lamps to plants optimally.

2.3 Example

```
probs = [[0.5, 0.5, 1],[0.25,0.1,0.75]]
best_lamp_allocation(2,2,probs)
>>> 0.375

probs = [[0.5, 0.75, 0.25],[0.75,0.25,0.8]]
print(best_lamp_allocation(2,2,probs))
>>> 0.5625
```

2.4 Explanation of example

In the first example, we have 2 lamps to use. If we assign 0 lamps to plant 0, it has a 0.5 probability of being ready. We would need to assign the full 2 lamps to it to improve its chance (to 1). We have 2 lamps left, so the best thing to do is assign both to plant 1, for a probability of 0.75. This gives an overall probability of 0.75*0.5=0.375.

We again have 2 lamps and 2 plants, but with different probabilities. This time, the most efficient thing is to not use all the lamps! Giving 1 lamp to plant 0 and 0 lamps to plant 1 give a probability of 0.5625. It would be ideal to give 1 lamp to plant 0 and 2 lamps to plant 1, but **we do not have** 3 lamps so this is impossible.

2.5 Approach

A good place to start would be answering one or both of the following questions:

- Come up with an idea for what your memo array looks like. What are its dimensions? What values will be stored in each cell (i.e. what are the **overlapping subproblems**)
- Come up with a recursive solution to this problem. If we have solved all previous (what does previous mean here?) subproblems, how can we use that information to solve the current subproblem?

To help with the above, here are some things to think about:

- Which memo cells can we fill in trivially (i.e. with no reference to smaller subproblems)?
- If we are considering how many lamps to give to plant i, and we decide to give it x lamps, what is the maximum number of lamps we can give to plants $0, 1, ...i-1$? When is the answer `num_l`$-x$? When is it not `num_l`$-x$?
- Having answered the question above, what does this tell us about the subproblems that we should be solving?
- What options do we have to try while solving each subproblem?
- In What order should we traverse the cells of the memo array?

2.6 Complexity

`best_lamp_allocation` should run in $O(PL^2)$ time and $O(PL)$ space, where P is `num_p`, and L is `num_l`

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!