FIT2004 S2/2020: Assignment 2 - Dynamic Programming

DEADLINE: Friday 18th September 2020 23:55:00 AEST

LATE SUBMISSION PENALTY: 10% penalty per day. Submissions more than 5 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 visit this page:
`https://www.monash.edu/connect/forms/modules/course/special-consideration`
and fill out the appropriate form. **Do not** contact the unit directly, as we cannot grant special consideration unless you have used the online 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 named `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 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) 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. Try to think of the **overlapping subproblems** and **optimal substructure**. Implementing your code will be much easier if you have these as a starting point.
5. 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.
6. Write down a high level description of the algorithm you will use.
7. 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 filename matches the specification.
- Make sure your functions are named correctly and take the correct inputs.

Documentation and submission (4 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!).

You are also required to submit your assignment in the format specified.

1 Robot in a corridor (12 marks)

Consider the following scenario: There is a corridor which is n floor-tiles long. Your robot starts on the first tile. Each tile has an item on it, with some **value** (this value can be negative). The robot will walk along the corridor, and at each tile, it can either pick up the item on that tile, or not. Once the robot moves off the last tile, the scenario ends.

We wish to maximize the total value of items the robot picks up, but there is a problem. Picking up items takes **energy**. The robot charges up energy when it is not picking up items, but it loses energy when it picks up an item. It can always move forward (moving does not consume energy) but sometimes it may be unable to pick up the current item.

The robot starts with 0 energy (i.e. it can never pick up the first item). It has 2 actions it can perform:

- Pick up the item on the current floor tile, move to the next tile, and lose 1 energy. This action can only be performed when the robot has at least 1 energy
- Move to the next floor tile and gain 1 energy

To solve this problem, you will write a function `optimise_single_pickup(corridor)`.

1.1 Input

`corridor` is a list of integers representing the values of the items on each tile of the corridor. This list will always contain at least 1 item. The i^{th} element of `corridor` corresponds to the value on the i^{th} tile of the corridor. The first item's value is therefore `corridor[0]`. The robot starts on tile 0.

1.2 Output

`optimise_single_pickup` returns a tuple with two elements. The first element is the maximum total value that the robot can obtain.

The second item is a list representing the choices the robot should make to obtain this value. The values in this list are the integers `1` and `0`. `0` means that the robot moved on without picking up the item. `1` means that the robot picked the item on that tile and moved on. The i^{th} item in this list corresponds to the action the robot takes on the i^{th} tile of the corridor.

1.3 Example

```
>>> optimise_single_pickup([4, 0, 4, 1, -3, 4, 3, 2])
(13, [0,0,1,0,0,1,1,1])
```

1.4 Complexity

`optimise_single_pickup` should run in $O(N^2)$ time and space, where N is the length of `corridor`

2 Robot in a corridor II (4 marks)

We have the same situation as Task 1, but now the robot is broken. When you instruct the robot to pick up an item, it gets stuck in a loop and continues picking up items until its energy reaches 0 (or until it reaches the end of the corridor). After that, it continues as normal.

The new actions the robot can take are as follows:

- If the robot has more than $e \geq 1$ energy, pick up every item on the next e tiles, ending on the tile after the last item that it picked up. In other words, it executes a series of e "pick up" actions, without a chance for you to give further instructions. At the end of this sequence of pick up actions, it will have 0 energy. Of course, if the end of the corridor is reached before e tiles have been traversed, then the scenario ends.
- Move to the next floor tile and gain 1 energy

To solve this problem you will write a function `optimise_multiple_pickup(corridor)`.

The input and output format are identical to Task 1. Note that you still record each pickup, even if it is forced.

2.1 Example

```
>>> optimise_multiple_pickup([4, 0, 4, 1, -3, 4, 3, 2])
(11, [0,0,1,1,0,1,0,1])
>>> optimise_multiple_pickup([0,0,5,-4,1,1])
(2, [0,0,0,0,1,1])
```

In the second example, the robot still has 2 energy after it steps off the last tile. The amount of energy is has left is not relevant.

2.2 Complexity

`optimise_multiple_pickup` should run in $O(N^2)$ time and space, where N is the length of `corridor`

3 Colour chooser (10 marks)

Consider the following problem: A paint warehouse is designing an interface to allow users to choose a shade of a colour. The user will have already chosen the colour they are interested in. They are then presented with a particular shade of that colour, and they can either accept that shade (in which case they are done), or they can request a lighter or darker option (if a valid option exists). If no valid lighter or darker option exists, they must choose the current shade. We will refer to this process as a decision tree.

The shade which is presented to the user always respects their prior choices, i.e. if they said "darker" to shade X, then all subsequent shades will be darker than shade X.

The paint warehouse has data on which shades are more or less popular, and they want to make the experience user friendly, so they need you to design a decision tree for the shades which will minimise the expected number of choices each user has to make.

Specifically, each shade has a **probability** which is a number between 0 and 1 (inclusive) which represents the proportion of users who have chosen that shade in the past. Each shade will also be at some **choice_depth**, which is the number of decisions the user needs to make in order to choose that shade. For the root of the decision tree, this is 1, since you at least need to choose it. For each other shade, it is the number of shades that the user sees before getting to that shade, plus one.

Let S be a set of shades. The cost of a particular decision tree for S is given by

$$\sum_{s \in S} choice\_depth(s) * probability(s)$$

We want to find the minimum cost decision tree for some given set of shades S. To solve this problem, you will write a function `optimal_shade_selector(shades, probs)`

3.1 Input

`shades` is a list of numbers between 0 and 1 inclusive, representing the darkness/lightness of each shade in S. Darker shades have lower values. This list **will be sorted** in ascending order.

`probs` is a list of numbers between 0 and 1 inclusive, representing the probabilities of each shade in S. `probs[i]` is the probability of the shade represented by `shades[i]`.

3.2 Output

The output is a single number, the cost of the optimal decision tree. You **do not** need to return any information about the structure of the decision tree.

3.3 Example

```
shades = [0.1, 0.2, 0.3, 0.4, 0.5]
probs = [0.25, 0.2, 0.05, 0.2, 0.3]
print(optimal_shade_selector(shades,probs))
>>> 2.1
```

To help you understand the example, a visualisation of the optimal decision tree is provided, along with an example of a different decision tree using the same information, which is not optimal.

The root is the first choice, and the children of a node are the options presented to the user when they select lighter/darker (since lower numbers are darker, the darker option is on the left, and the lighter is on the right).

In the diagram below, the optimal decision tree for the example is shown. Each node represents a shade, where the first value is the lightness/darkness of the shade, and the second is its probability.

Figure 1: The optimal decision tree.

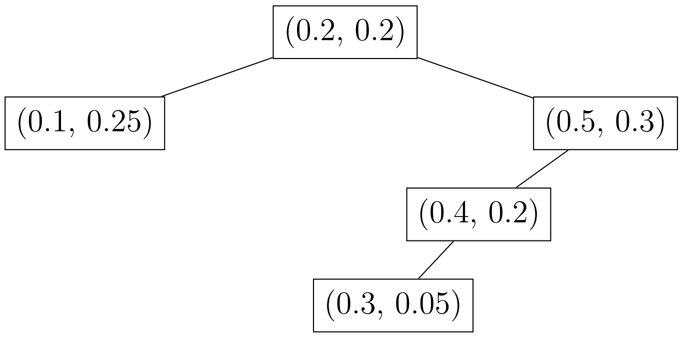
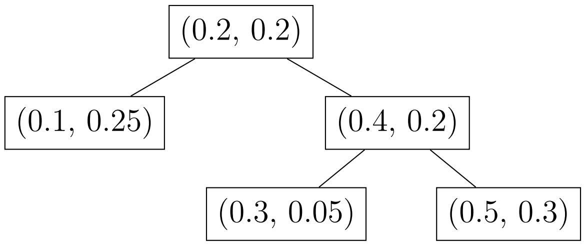


Figure 2: A suboptimal decision tree



The cost of the decision tree in Figure 1 is given by $1*0.2+2*(0.25+0.3)+3*0.2+4*0.05 = 2.1$.
The cost of the decision tree in Figure 2 is given by $1*0.2+2*(0.25+0.2)+3*(0.05+0.3) = 2.15$

Note that since this question deals with floating point values, it is possible for rounding errors to be introduced by your algorithm. You will not be penalised for computer rounding errors.

3.4 Hint

A useful way to think about the trees in this problem is that a tree is either empty, or it has a root, a left subtree and a right subtree. These subtrees are themselves trees, so the structure is recursive.

In order to get started with this task, as with any DP task, you first need to determine the overlapping subproblems. In this particular problem, we need to choose a shade to be the first choice (i.e. the root) of the decision tree. Notice that once a root is chosen, the elements which go in the left and right subtrees are determined. So would now have 2 separate subproblems to solve.

Consider the information that is needed to specify a particular subproblem, in terms of the input. How many possible subproblems are there?

3.5 Complexity

Your algorithm should run in $O(N^3)$, where N is the length of `shades`.

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!