Homework 5: Dynamic Programming

Be sure to read the grading guidelines and style guidelines. Especially to see the suggested format for describing algorithms.

We sometimes describe how long are justifications or proofs are. These lengths are intended to help you estimate how much detail we're expecting, you should not take those estimates as hard length-limitations.

Our solutions for any individual problem will fit in approximately one page or less. **If you submit an answer substantially longer than 2 pages, the TAs are allowed to stop reading at the end of page 2.**

You are allowed (and encouraged!!) to collaborate with each other. Brainstorming is much easier to do in a group than alone! But you must follow the collaboration policy (which includes needing to write your submission on your own).

You will submit to gradescope; we have a different box for each problem, please give yourself time to submit.

1. Potentially Greedy Beans [10 points]

Your friends at the local bean store need your help with ensuring this year's harvest can be completed economically. Previously, they would buy a large number of planters and grow exactly 1 bean in every planter since they knew this was an acceptable arrangement and that the beans would grow well. Recently, they found out that, for some reason, you can potentially add more than 1 bean to a planter and *for some integers* the beans will all grow well (but for *other* integers, the beans to not grow well). Their goal is to grow all n beans while using as few planters as possible.

Specifically, your friends have exactly n beans to plant. They have a list M, of positive integers (of length j), M contains a list of all values, v, such that if you put exactly v beans in the planter they will grow well together. Beans are finicky and if you put some number of beans not on the list, then none of them will flourish.
You're also guaranteed that, like in the previous days, a single bean in a planter will flourish (i.e., 1 is on the list M).
Essentially you want to write the following method

`Minimum Planters`
Input: An array $M[]$ containing the possible quantities of beans that may be planted in a single planter. You may assume M contains only positive `int`s, that $M[0] = 1$, and that $M[i] < M[j]$ for all $i < j$. And a positive integer n.
Output: The minimal number of planters required to ensure all n beans are planted.

One of your friends suggests the following greedy algorithm:

```
1: function GreedyBeans(M[], n)
2:     planter ← 0
3:     for i from M.length − 1 down to 0 do
4:         while n ≥ M[i] do
5:             planter++
6:             n ← n − M[i]
7:     return planter
```

Give an example input such that the greedy algorithm above is not optimal, and justify that it is a counter-example.

2. Definitely Dynamic Beans [25 points]

Now, your friends at the local bean store ask you to produce an algorithm for minimal usage of planters for bean planting.
Specifically, you will write the following method

`Minimum Planters`
Input: An array $M[]$ containing the possible quantities of beans that may be planted in a single planter. You may assume M contains only positive `int`s, that $M[0] = 1$, and that $M[i] < M[j]$ for all $i < j$. And a positive integer n.
Output: The minimal number of planters required to ensure all n beans are planted.

In Part 1, you proved that a greedy algorithm did **not** produce the minimum number of planters.

Design a dynamic programming algorithm that *does* produce the minimum number of planters.

(a) Clearly state **in English** what your recurrence(s) calculate. Be sure to mention any parameters you use.

(b) Write a recurrence (or multiple recurrences) that you will use to solve this problem.

(c) Give a brief explanation of your recurrence; we're not looking for a formal proof, but a brief explanation of what each of the cases represent, and why that set of cases is exhaustive.

(d) Describe a memoization structure.

(e) Describe a filling-order for your memoization structure. If you wrote (iterative) pseudocode to fill the structure, what would its running time be?

(f) What is your final answer going to be? (where in the memoization structure do we look?)

3. Chess-Box-Sorrow [25 points]

After a riveting week of Chess-Boxing, a chess-boxing champion was announced. Unfortunately, everyone else was a chess-boxing loser. All the chess-boxing losers came to your chess-boxing training center to get better.
You have n wooden chairs to situate the n losers and they have a specific order in which they sit. All of these losers have some amount of chess-boxing "prowess" which you are given in c: an array of length n.
Since they are losers, they all have some amount of post-competition sorrow which is represented in the array s where $s[i]$ represents the sorrow of the i'th person who is also sitting in the i'th chair. You know that sorrow is a quantity that is distributed linearly to the right (i.e. if you sit to the right of somebody very sorrowful, you will feel proportionately sorrowful, but if you move far away enough, then you will not feel their sorrow).
In order to ensure that they perform well, you want to ensure that nobody's sorrow affects anybody else, which you will do by asking some people to leave on the first day and come back some other day. Specifically, you want to ensure that $\forall i$ you choose, that you don't choose anybody else that is within $s[i]$ to the right of this person (to ensure that this person's sorrow doesn't affect their rightmost neighbors). You want to choose the strongest (determined by the sum of their individual prowesses) combination of losers in order to restore overall happiness in the chess-boxing world quicker.
You will want to write the following function:

Maximum Sorrow
Input: An integer n representing how many losers you have (and the corresponding lengths of the following arrays). An array $c[]$ containing the prowesses of the corresponding individuals. An array $s[]$ containing the sorrows of the corresponding individuals.
Output: An integer corresponding to the maximum sum of prowesses subject to the conditions imposed by sorrow.

Specifically given arrays `c[]`, `s[]`, return the integer corresponding to the maximum sum of $c[i]'s$ you can have, while ensuring that for every index i you choose, you do not choose anybody else with indices from $'i+1'$ to i+$s[i]$.

(a) Clearly state **in English** what your recurrence(s) calculate. Be sure to mention any parameters you use.

(b) Write a recurrence (or multiple recurrences) that you will use to solve this problem.

(c) Give a brief explanation of your recurrence; we're not looking for a formal proof, but a brief explanation of what each of the cases represent, and why that set of cases is exhaustive.

(d) Describe a memoization structure.

(e) Describe a filling-order for your memoization structure. If you wrote (iterative) pseudocode to fill the structure, what would its running time be?

(f) What is your final answer going to be? (where in the memoization structure do we look?)

4. Subarray DP

Recall that a **contiguous subarray** is all of the elements in an array between indices i and j, inclusive (and if $j < i$ we define it to be the empty array).

Call a subarray is **nearly contiguous** if it is contiguous (i.e. contains all elements between indices i and j for some i, j) or if it contains all but one of the elements between i and j.

For example, in the array $[0, 1, 2, 3, 4]$,

- $[0, 2, 3]$ is nearly contiguous (from 0 to 3, skipping 1), sum is 5
- $[0, 1, 2, 3]$ is nearly contiguous (because it is contiguous from 0 to 3), sum is 6.
- $[2, 4]$ is nearly contiguous (from 2 to 4, skipping 3), sum is 6.
- $[3]$ is nearly contiguous (because it is contiguous from 3 to 3), sum is 3.
- $[]$ is nearly contiguous (because it is contiguous from 1 to 0), sum is 0.
- $[0, 2, 4]$ is **not** nearly contiguous (because you'd have to remove two elements).

The sum of a nearly contiguous subarray is the sum of the included elements.

Given `int[] A`, your task is to return the maximum sum of a nearly contiguous subarray.

For example, on input $[10, 9, -3, 4, -100, -20, 15, -5, 9]$ your algorithm should return 24 (corresponding to $i = 6, j = 8$ and skipping the -5).
In this problem, we will assume arrays to be 0-indexed, and referencing the array in the manner $A[a, b]$ means you're referencing all elements from index 'a' (inclusive) up to 'b' (exclusive). Please be explicit about whether your recurrences are start/end inclusive or exclusive. If it's not immediately obvious you **will** lose points.

(a) Clearly state **in English** what your recurrence(s) calculate. Be sure to mention any parameters you use.

(b) Write a recurrence (or multiple recurrences) that you will use to solve this problem.

(c) Give a brief explanation of your recurrence; we're not looking for a formal proof, but a brief explanation of what each of the cases represent, and why that set of cases is exhaustive.

(d) Describe a memoization structure.

(e) Describe a filling-order for your memoization structure. If you wrote (iterative) pseudocode to fill the structure, what would its running time be?

(f) What is your final answer going to be? (where in the memoization structure do we look?)