Homework 4: More Dynamic Programming

Version 2: We corrected a typo in the first example of problem 3 (4 is the correct return value).

Due Date: This assignment is due at 11:59 PM Friday February 12 (Seattle time, i.e. GMT-8).
except for the programming problem, which will be due one week later with HW5.

You will submit the written problems as a PDF to gradescope. Please put each numbered problem on its own page of the pdf (this will make selecting pages easier when you submit).

Remember to assign pages in your submission!

Collaboration: Please read the full collaboration policy. If you work with others (and you should!), you must still write up your solution independently and name all of your collaborators somewhere on your assignment.

Style and Assumptions: You should check the Style Guide for general principles on how to format your answers. You may use any algorithm described in this list without further explanation. If you wish to use an algorithm you learned in 373 (or 142/143) that isn't on the list, please ask on Ed so we can add it to the list for everyone.

1. Wrap up HW3

Remember to complete the programming question from HW3!

2. Change is Coming

In this problem we'll design a dynamic programming algorithm for making change.

Suppose you are given:

- a list of the values of coins `int[] coins` – sorted so `coins[i] < coints[i+1]`, and `coins[0] = 1` (i.e. there is a one-cent coin, so we can always successfully make chage).
- `int n` a number of cents of change to make.

(a) Write a recurrence that you can use to calculate the minimum **number** of coins required to make n cents of change (do not write code for this part, just the recurrence. In this part, you only need the number, not the type of coins).
Along with the recurrence, write an **English** description of what the recurrence is calculating (especially what the parameter(s) in your recurrence represent) and state the settings of the parameter(s) to make change for n coins.

You should look at Lecture 14, slides 25 and 28 for examples of what we're looking for.

(b) Give pseudocode (or English) for a Dynamic Programming algorithm to find the optimal change (i.e. the change with the fewest number of coins). Your algorithm must be iterative (not recursive).
For this problem, you should return an `int[] change`, where `change[i]` is the number of coins of value `coins[i]` that are in an optimal set of coinage. For example, if your input were: `coins=[1,5,15]` and $n = 20$ you should output $[0, 1, 1]$.
If your input were: `coins=[1,20,30]` and $n = 45$ you should output $[5, 2, 0]$

(c) What is the running time of your code? What is the memory usage of your code? Briefly (1-2 sentences) justify each. Let c be the length of `coins` and n be the number of cents of change required.

(d) We talked about a greedy algorithm for this problem in homework 3. Give a reason why you might prefer the greedy algorithm over the DP algorithm. Give a reason why you might prefer the DP algorithm over the greedy algorithm.

3. Goodbye Stale

Robbie hates grocery shopping, so he decides to try a meal delivery service. The service lists its meals for the next n weeks, and lets you choose for each week whether to get meals delivered or get nothing delivered.

Robbie can predict for each week how happy the meals will make him, with a numerical score. But he's a very picky eater, so sometimes it's a negative number. Since his goal is to avoid the store, there are penalties for skipping weeks:

- If he skips meals for one week, there's no additional penalty.
- Every time he skips meals for exactly two or three **consecutive** weeks, he needs to go to the store, which causes a penalty of -20 points to his total happiness.
- If he skips meals for four or more **consecutive** weeks, he has to do a big stockup grovery trip, which causes a penalty of $-\infty$ (i.e., you cannot skip 4 consecutive weeks)

Your goal is to calculate the maximum sum of happiness scores.

More formally, given: `int[] happy` which gives the happiness of getting meals delivered in that week. return the maximum sum of happiness values subject to the week-skipping penalties above.

For example,

if `happy` = $\{10, -3, -21, -8, 5\}$:
Ordering meals weeks $0, 4$ would result in a score of -5
Ordering meals weeks $0, 1, 4$ would result in a score of -8
Ordering meals weeks $0, 1, 3, 4$ would result in a score of 4
Your algorithm should return 4 (which is the maximum possible).

if `happy` = $\{-1, -2\}$:
skipping both weeks would result in a score of -20.
Ordering meals in week 1 only would result in a score of -1.
Your algorithm should return -1.

if `happy` = $\{-100, -100, 40, -100, -100\}$
Ordering meals in week 2 only would result in a score of 0
Ordering meals in weeks $1, 3$ would result in a score of -200.
Your algorithm should return 0.

(a) Write a recurrence to describe what you are going to calculate. In defining your recurrence include an English description of what number you're calculating, and English descriptions of any parameters. You should look at Lecture 14, slides 25 and 28 for examples of what we're looking for
Hint: To figure out whether there's a penalty, you'll need to somehow keep track of how many consecutive weeks you've skipped.

(b) Briefly (2-4 sentences) describe your intution for your recurrence. You should look at the example in lecture 14 slide 25.

(c) If you converted your recurrence into an iterative algorithm, what would your memoization structure be, what order would you fill it in, and what value would be your final answer (you do **not** have to write the code itself).

(d) What would the running time of your iterative algorithm be? (you still do **not** have to write the code itself).

4. Programming: make a change and get the choices

This question will be **due with HW5**. We still strongly suggest aiming to complete it this week, but want to make sure you have at least a full week of work time.

We've given you a function `LIS`, which returns the **length** of the longest increasing subsequence of an array. Your task is to write code for a *variant* of the longest increasing subsequence problem.

Specifically, given an array, you should return the longest **geometrically** increasing subsequence. A subsequence is **geometrically** increasing if for chosen elements where $i < j$, we have $3A[i] < A[j]$. I.e. each successive element must be more than three times larger than the previous. For simplicity, you may assume there are no indices i, j such that $3A[i] = A[j]$ and that there are no repeated elements.

Your code will return an ArrayList containing **in order** the **elements** of a longest geometrically increasing subsequence. You therefore will need to modify the code given both to handle the geometric requirement AND to build the subsequence itself. (You're also free to start from scratch, but we think you'll find the starter code helpful).