Homework 5: More Dynamic Programming

Due Date: The written parts of this assignment (problems 1-3) will be due at 11:59 PM on **Monday** November 15th (note the unusual day). As always, that means that the TAs will grade **every** written problem you submit to gradescope before that deadline. You will also be able to resubmit two written problems with every homework; the resubmission for this week is due on **Wednesday** November 17th (note the unusual day).

The coding problems may be submitted at any time through the last day of classes (Friday Dec. 9), but we strongly recommend you finish any you intend to do by the due date for the written assignment so you have time to do next week's problems next week.

Collaboration: You are allowed (and encouraged!) to discuss these problems at a high level with others. But, please read the full collaboration policy to ensure you are keeping your discussions at the right level, and remember to cite any collaborators.

Problems to Submit: We will count your 1 best mechanical question and your 3 best long-form questions.
We strongly recommend you skim all the problems in a section before attempting them. While we try to make problems in a given section of approximately equal difficulty, you may find some to be easier than others.

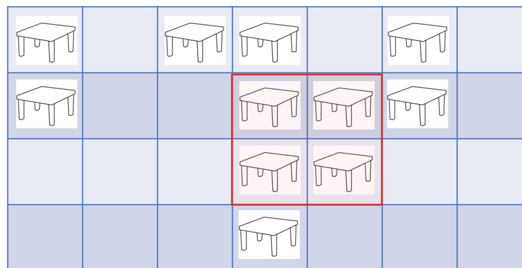
Directions Unless otherwise noted, you are allowed to use algorithms described in class (or prerequisite courses) as though you had library implementations of them. For example, you can say "run the BFS-based 2-coloring algorithm from class on the graph G" or "run the bipartite checking algorithm from lecture 5 on G." We also have a list of data structures and algorithms you can use from 373 here.

Since this is a course about efficient algorithms, algorithms that are slower than necessary are unlikely to get "E" scores, but (unless otherwise noted) we do not care about constant factors. In general, an algorithm that is correct but (mildly) slower than optimal will get an equal or higher score to an algorithm that is fundamentally incorrect, but fast.

Mechanical Problems

1. Sauerkraut Table Size

You want to spend your 23rd birthday at Robbie's Sauerkraut Restaurant and to accommodate all the guests, Robbie wants to put a large square table in the restaurant. The restaurant has many tables, and we would like to find the area of the largest **square** table. In the example below, the largest square table has a size of 4 and side lengths of 2.



The restaurant is huge and might have many tables, so we decided to use dynamic programming to help us solve the problem. Luckily, the building managers gave us the following recurrence to solve this problem.

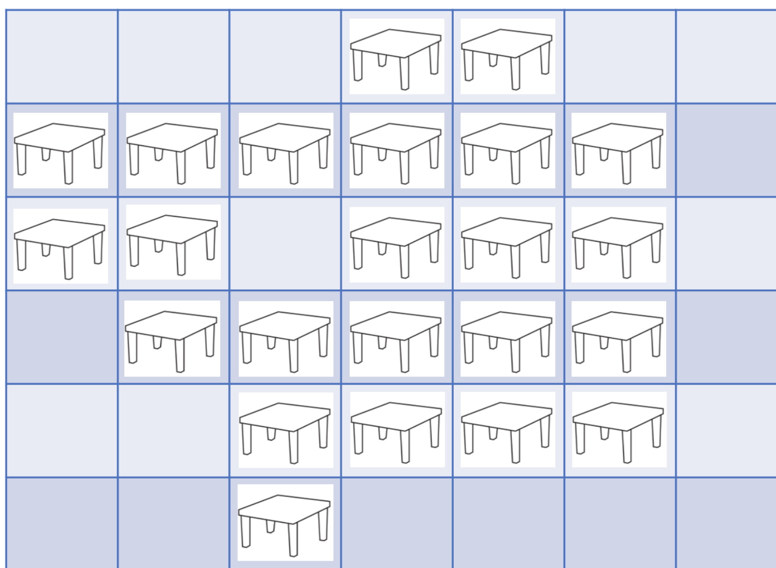
$$\text{AREA}(i,j) = \begin{cases} \infty & \text{if } i < 0, j < 0, \\ \mathbb{1}\left[(i,j) \text{ has a table}\right] & \text{if } i = 0 \text{ or } j = 0, \\ 0 & \text{if } \mathbb{1}[(i,j) \text{ has table}] = 0, \\ \min\{\text{AREA}(i-1,j),\ \text{AREA}(i,j-1),\ \text{AREA}(i-1,j-1)\} + 1 & \text{otherwise.} \end{cases}$$

Here $\mathbb{1}$ denotes the *indicator* function, which "casts" bool to int – if the expression is true, the function outputs 1, otherwise it outputs 0.

The example given above would have the following AREA values.

1	0	1	1	0	1	0
1	0	0	1	1	1	0
0	0	0	1	2	0	0
0	0	0	1	0	0	0

Note that the AREA value does not give you the actual area of a combined table. Rather AREA(i, j) gives the length of the largest combined-square table whose bottom-right corner is a table located at (i, j) For the instance of the problem pictured below, compute the AREA value for all (i, j).



Long-Form Problems

2. 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, your goal is to return the maximum sum of happiness values subject to the week-skipping penalties above.

Example 1: Suppose `happy =` $\{10, -3, -21, -8, 5\}$. Then:

- Ordering meals weeks $0, 4$ would result in a score of -5 ($10 + 5$ happiness, but with a -20 penalty for skipping three weeks in the middle).
- 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).

Example 2: Suppose `happy =` $\{-1, -2\}$. Then:

- Skipping both weeks would result in a score of -20.
- Ordering meals in week 0 only would result in a score of -1.

Your algorithm should return -1.

Example 3: Suppose `happy =` $\{-100, -100, 40, -100, -100\}$. Then:

- Ordering meals in week 2 only would result in a score of 0 (40 from the array, but with two penalties).
- Ordering meals in weeks $1, 3$ would result in a score of -200. (no penalties, but -200 from the array).
- Ordering no meals would result in a score of -20 (going two weeks without meals gives just the penalty).

Your algorithm should return 0.

(a) Write a recurrence (or multiple recurrences) to describe what you are going to calculate. In your response, you should give us:

- The recurrence(s) itself
- An English description of any recurrence parameters
- An English description of what value your recurrence is intended to calculate.

(Take a look at Lecture 16, slides 6 and 7 to get a sense of roughly what we're looking for.)

Hint: This is somewhat similar to the contiguous subarray sum problem, but instead of requiring the elements to be exactly contiguous (i.e., taking only elements right next to each other), we're allowing for some skipping,

but with penalties. You'll want to think about all the potential places where the "most recent" week of ordering could be. We have two recurrences (similar to subarray sum) but you may have more or less.

(b) Explain in English how your recurrence works and what it's doing. Your response should include:

- Why you chose the expressions that the recurrence returns in each branch that you did
- Why you chose the cases that you did
- What your base case(s) is
- An explanation for what recursive calls are made and why they are combined in the way that they are. (By "combine", we mean things like adding together, taking a min/max over, or if the return values are boolean, taking `AND` or `OR`.)

(For reference, take a look at the bottom paragraph of Lecture 16, Slide 26, it should be in light purple text.)

Make sure that your recurrence actually carries out the computations to get what you claimed it should represent in the third bullet point of part (a).

(c) To get your final answer, which recurrence would you call, and what values of your parameters would you call it with?

(d) If you were to convert your recurrence into an iterative algorithm:

- What would your memoization structure be?
- What order would you fill in the values?
- Assuming memoization (all values from would-be recursive calls that you need to make have been pre-calculated and are ready), what is the computational cost of filling out a single entry?
  (In most DP problems, computing a single entry can usually be done in constant-time, but if you wanted to, say, sum over a number of pre-computed values where the number of terms in the sum scales with some parameter of the problem (not the recurrence, the problem), it could be non-constant unless you do something clever.)

(You do **not** have to write the code itself.)

(e) Based on your response to (c), what would the runtime of an algorithm that carries out your memoization be?

3. Buy Low, Sell High

You've decided to translate your expert knowledge of art into profit. You have (extremely accurate) predictions of how valuable various pieces of art will be at the end of each calendar year. Conveniently enough, the art that you don't own goes on the market consistently at the end of each calendar year (you may decide to sell at the end of the year or hold the piece for another year)! Inconveniently, you have to pay an auction fee every time you **sell** a piece of art, and you only have enough room in your climate-controlled vault to hold one piece of art at a time.

(a) We'll start with a simpler case. Our goal is to calculate the maximum profit we could get over y years buying and selling a single piece of art, where we are charged $\$f$ when we sell the piece. You have as input an array that contains how much the piece is worth at the end of that year. Write a recurrence (or multiple recurrences) which would allow you to calculate your profit. Briefly explain what your recurrence(s) are.

For this part, do not write code or an algorithm.

Hint: Consider what states you can end up in at the end of each year and think about how to transition between these states. Our approach would take $\mathcal{O}(y)$ if we wrote iterative code and ran that algorithm for y years.

Example:

Year:	1	2	3	4	5	6	7	8
Price:	4	8	10	3	7	5	12	9

Transaction fee: 3

The optimal transaction sequence is: buy on year 1, sell on year 3, buy on year 4, sell on year 7
Profit: $(-4) + (10 - 3) + (-3) + (12 - 3) = 9$

A correct algorithm would return 9.

(b) Using the insights from part (a), you will design an algorithm that calculates the maximum profit we could get over y years buying and selling some combination of a pieces of art where we are charged $\$f$ when we sell a piece. For this version, you are allowed to buy and sell any of the pieces of art, but you are only allowed to own one piece of art at a time (you **are** however allowed to buy one piece and sell another "simultaneously"). Your final algorithm should run in time $\mathcal{O}(ay)$.

Example:

Year:	1	2	3	4	5	6
Artwork 1 Price:	3	2	6	3	8	3
Artwork 2 Price:	6	1	7	2	3	9
Artwork 3 Price:	5	6	2	4	8	1

Transaction fee: 2
Optimal transaction sequence: Hold nothing on year 1, buy piece 2 on year 2, sell piece 2 and buy piece 3 on year 3, hold piece 3 on year 4, sell piece 4 and buy piece 2 on year 5, sell piece 2 on year 6
Your algorithm should return: 12

(i) Give a recurrence (or multiple recurrences) for solving this problem. Your response should include:

- The recurrence(s) itself
- An English description of any recurrence parameters
- An English description of what value your recurrence is intended to calculate.

(Take a look at Lecture 16, slides 6 and 7 to get a sense of roughly what we're looking for.)

(ii) To get your final answer, which recurrence would you call, and what values of your parameters would you call it with?

(iii) Give **iterative** pseudocode to solve this problem.

4. Programming: make a change and get the choices

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).

5. Programming: Unit Expression

A "unit expression" is an expression involving only:

- parentheses

- $+$ (that is, addition)
- $*$ (that is, multiplication)
- The number 1

For example $(1+1)*(1+1+1)$ is a unit expression that evaluates to 6. This expression has 5 copies of the number 1.
$1+1+1+1+1+1$ also is a unit expression that evaluates to 6. This expression has 6 copies of the number 1.

Your task is to write a program that given the number n, returns the fewest number of copies of 1 that can appear in a unit expression evaluating to n. You do not have to return the expression itself.

For example, the correct output for $n = 6$ would be 5 (corresponding to the number of 1's in the first expression above).