Homework 4: Dynamic Programming

Due Date: The written parts of this assignment (problems 1-3) will be due at 11:59 PM on Friday November 4th. 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 Monday November 7th.

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.

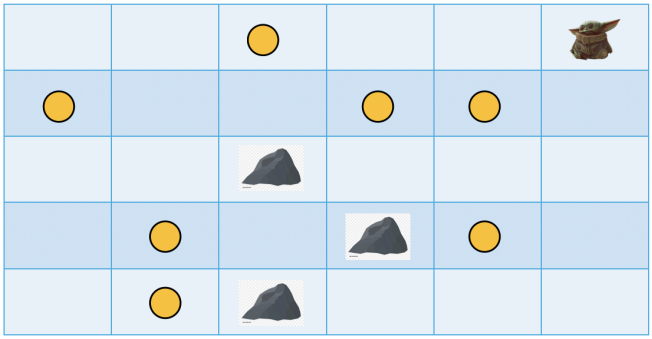
Since this is a course about efficient algorithms, algorithms that are slower than necessary are unlikely to get "E" scores, but (unless otherwise noted) we do not care about constant factors. In general, an algorithm that is correct but (mildly) slower than optimal will get an equal or higher score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

Mechanical Problems

1. Mechanical Baby Yoda

Baby Yoda has to get from the upper-right $(5, 4)$ to the lower-left $(0, 0)$ by only moving left and down. Baby Yoda is also hungry, so we will need to figure out the maximum number of eggs that can be collected, moving around the rocks, and still reaching the destination. Recall the recurrence from class where $\text{OPT}(i, j)$ is the maximum number of eggs Baby Yoda can get on a legal path from (i, j) to $(0, 0)$. Compute OPT for all (i, j) in the grid. Write your solution in a 5×6 grid.

$$\text{OPT}(i,j) = \begin{cases} -\infty & \text{if } \texttt{rocks}(i,j) = \texttt{true} \\ -\infty & \text{if } i < 0 \text{ or } j < 0 \\ \texttt{eggs}(0,0) & \text{if } i = 0 \text{ and } j = 0 \\ \max\{\text{OPT}(i-1,j), \text{OPT}(i,j-1)\} + \texttt{eggs}(i,j) & \text{otherwise} \end{cases}$$



Long-Form Problems

2. Count the Number of Arrangements

You are part of a committee organizing a large cultural fair that will take place on a long street. You are responsible for setting up food stalls. There are many food vendors, and many of these vendors offer similar foods. Unfortunately, having two adjacent competing food stalls with the same kind of food is not great for business, so it's preferable to spread out food stalls that sell the same kind of food and not have them next to each other.

Let $n \geq 3$ be the number of food stall slots available, and suppose that there are $k \geq 2$ possible kinds of food (e.g., hamburgers, hot dogs, corn on the cob, kebabs, etc.). The two food stall slots on the far ends of the street are highly coveted. Because of sponsorship agreements, the very first slot must be utilized by a food stall selling the 1st kind of food, and the very last slot must be utilized by a food stall selling the x-th kind of food, where $1 \leq x \leq k$ is given to you as part of the input.

The committee head is interested in the number of valid arrangements possible and started writing a recurrence to calculate, but since they're too busy (or lazy?) to finish it, you've begrudgingly agreed to finish the recurrence for them.

For a reasonably small example, suppose there are $n = 4$ slots, $k = 3$ kinds of food, and the last stall must be the 3rd kind of food ($x = 3$). Then there are 3 possible arrangements. To see this, observe that we cannot put a food stall of the 1st kind in the 2nd slot, so it must either be of the 2nd kind or 3rd kind. If it is of the 2nd kind, the 3rd slot can neither be of the 2nd kind nor the 3rd kind, so it must be the 1st kind of food. Otherwise, the 3rd slot can be a food stall of the 1st kind or the 2nd kind. To represent these arrangements as arrays:

$$L = [1, 2, 1, 3]$$
$$L = [1, 3, 1, 3]$$
$$L = [1, 3, 2, 3]$$

Consider the following partially filled out recurrence:

$$\text{OPT}(i, y) = \begin{cases} \displaystyle\sum_{1 \leq z \leq k \text{ and } z \neq y} \text{OPT}(i+1, z) & \text{if } i \neq n-1, \\ k-1 & \text{if } i = n-1 \text{ and } y = x, \\ ??? & \text{otherwise.} \end{cases}$$

(a) Determine what the recurrence calculates, then give an explanation in plain English of what the value $\text{OPT}(i, y)$ intuitively represents in the context of the scenario.

(b) In this part, we make a small digression to think about the cases in a recurrence – both how to interpret and read them in a recurrence that's already been written, as well as pitfalls to avoid while writing/designing them.

(i) The condition for OPT to enter its third branch is described as "otherwise". Give a more explicit description for when the recurrence should enter its third branch. (Your response is expected to be short.)

(ii) Below are two incorrectly defined functions/recurrences (unrelated to the scenario in this problem). Explain why these function definitions don't actually properly define functions (our responses are 1-2 sentences for each). Once you've got your explanations, look back at your response in (b)(i), and make sure you didn't make either of those mistakes!

- $f(n) = \begin{cases} 2 + f(n-2) & \text{if } n \geq 1, \\ 1 & \text{if } n = 0. \end{cases}$
- $g(x) = \begin{cases} 4 & \text{if } n \geq 4, \\ 5 & \text{if } n \leq 5. \end{cases}$

(c) Fill in the missing expression (the "???") in the third branch of the recurrence, then briefly explain your reasoning for how you came up with your answer and why it makes sense.

(*Hint*: figure out what the intuition behind the 2nd branch is, then think about why the possibilities for the second-to-last slot becomes more constrained when $y \neq x$.)

(d) Determine what values of i and y the recurrence should be called with in order to obtain the final answer (total number of valid food stall arrangements).

(e) In this part, you'll translate the recurrence into an iterative program.

(i) Write iterative pseudocode that computes the same thing as the recurrence. Be sure that you have a return statement in your code that gives back the overall final answer to "how many arrangements are there?" (in addition to "building the table").

(ii) State *and* explain the asymptotic runtime of your pseudocode in terms of n, k, and x.
(You should not be alarmed if the runtime ends up being worse than you expected as long as it's still polynomial time; this particular algorithm is comparatively less efficient than the ones in most other problems.)

3. Be the change you want to see in the world

You have just been hired for a company that makes cash registers to be used in a variety of countries. Your clients really like efficiency – they want their cashiers to give exact change using the minimum number of coins possible. But since the clients have stores in multiple countries, you're going to need to handle many different money systems (that have coins of different values).

You are given:

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

For example, the set of (commonly used) coins in the U.S. would be `coins = [1,5,10,25,100]`.

(a) The first thing that comes to mind is an algorithm like this:

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

This algorithm **sometimes** works (in fact, it works for U.S. coins!) But it does **not** always work. Find a set of coins and target change value where this algorithm fails, i.e., it uses more coins than necessary. Describe the set of coins found by the algorithm and a way to get a smaller number of coins.

(b) 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, just the recurrence. You only need the *total* number coins used; you do *not* need to give the number of coins taken from each denomination/coin of a given value.)

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 13, slide 21 for examples of what we mean by English descriptions.

(c) Evaluate your recurrence on the example you wrote in part (a). Make sure you get the correct answer! It's easy to accidentally write a recurrence that is actually doing the code from part (a).

(d) What memoization structure would you use to evaluate your recurrence?

(e) Give a filling order for your memoization structure. (you do not have to give the pseudocode to fill it, just an order you could fill it in).

(f) What would the running time of your code be? Let c be the length of `coins` and n be the number of cents of change required.

4. Coding: Fibonacci

The goal of this problem is to see just how inefficient recursive solutions can be when they have to recalculate the same value repeatedly.

You might have seen the Fibonacci numbers in another course. They follow this rule:

$$F(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F(n-1) + F(n-2) & \text{for other positive integers } n \end{cases}$$

(a) Start by writing a "naive" recursive algorithm to calculate the n^{th} Fibonacci number. By "naive" we mean "pretend you have never seen dynamic programming; don't store any calculated values."

Experiment with some values of `n`. How large can you make it before your code noticeably slows down? How large of n can your code handle in a minute? If you increase n by 3 or 4 from that value, how long does it take?

We won't grade anything from this part, but it should help to explain why Dynamic Programming is so important.

(b) Now write a dynamic programming version of the code (either iterative or recursive — the key is to be able to look up values). Our tests on gradescope will require finding values of n much larger than the ones where your naive version slowed down.

For this problem, we will only grade the gradescope submission associated with this part.

Some tips:

- There is a version of the code that needs only a few variables (no array), though you should not feel obligated to optimize for memory.
- Remember not to change the header of the method stub we give you (e.g., don't change the name or return type or add extra parameters). If you find yourself tempted to make such a change, you probably want to call a private helper method.
- Remember to upload a file of the same name to gradescope.
- We will verify that you're really doing dynamic programming at the end of the quarter.

5. Coding: Force Dynamics

In this problem, you'll implement Java code that corresponds to the high-level ideas we developed in Lecture 11.

The heart of the problem is the same as the one in lecture, but a few details have changed. In particular the coordinate system was changed to match how arrays are usually drawn on paper (this should make it easier for you to design your own test cases if you need to)

Baby Yoda needs to get from location $(r-1, c-1)$ to location $(0, 0)$. He can only move up (decreasing the first coordinate) or to the left (decreasing the second coordinate). He cannot pass through a spot with rocks (unless he first uses the Force to knock over the rocks). Finally, he can collect an egg to eat by passing through that location.

Write a method, that given

- `bool[][] rocks`. `rock[i][j]` is true if and only if there is a rock in location (i, j).
- `bool[][] eggs`. `eggs[i][j]` is true if there is one egg in location (i, j) (and it's false if there are no eggs in location (i, j)).
- `int force`. `force` is a non-negative integer, with the number of times Baby Yoda can use the Force to knock over rocks.

returns the maximum number eggs Baby Yoda can collect on his way to reach $(0, 0)$.
If Baby Yoda cannot reach $(0, 0)$, return -1.

You may make the following simplifying assumptions:

- `rocks[i][j]` and `eggs[i][j]` will not both be 1 for any location.
- `force` will be 0, 1, or 2.
- If the starting point of Baby Yoda has a rock, you do not need to use the force in order to move out of it.
- There is no need to validate input (e.g. you do not need to explicitly check that the dimensions of `rocks` and `eggs`) match, nor that `force` is not 3. We will never test invalid input.

Make sure you're following our updated indexing conventions – the row comes first, then the column. This is consistent with how arrays are usually drawn (the origin is in the upper left), but not consistent with the drawings from lecture. You may wish to look at the provided test case to help you visualize.

Your final response must be **iterative** not recursive[1], and it should run in time $\Theta(frc)$ where f is `force` and the map is $r \times c$.

You do not need to optimize memory (though you may if you wish).

[1] You can still use helper methods, but the main structure of your code needs to be loops, not using recursion. We will verify this requirement by hand at the end of the quarter.