Homework 4: Dynamic Programming

Due Date: The written parts of this assignment (problems 1-3) will be due at 11:59 PM on Friday February 9th. You will also be able to resubmit two written problems with every homework; the resubmission for this week is due on Monday February 12th.

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.

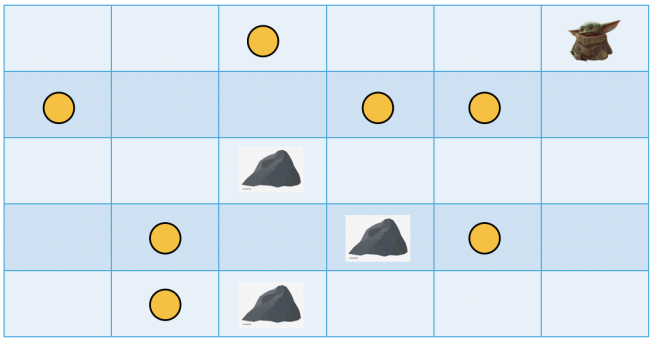
Since this is a course about efficient algorithms, algorithms that are slower than necessary are unlikely to get "E" scores, but (unless otherwise noted) we do not care about constant factors. In general, an algorithm that is correct but (mildly) slower than optimal will get an equal or higher score 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 recently had an interview to become a TA for CSE417 although you could only partially solve the problem.[1] Thankfully, you remembered the problem and have decided to re-examine it so you can improve your skills. The goal was to create a recurrence that would calculate the number of arrangements of integers that satisfied the following:

- The number of integers is $n \geq 3$, i.e. the length of the arrangement
- In the arrangement, we can use integers $1 \ldots k$ where $k \geq 2$
- No two of the same integer may be adjacent
- The first slot must contain the first integer and the last slot must be an integer x where $1 \leq x \leq k$

What you remembered from your recurrence, where ??? is the part at which you got stuck:

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

Lastly, you recall that your attempted recurrence used 1-indexing and the interviewer gave you the below example, where $n = 4$, $k = 3$, and $x = 3$, i.e. each arrangement is length 4, can only include the integers 1, 2, and 3, and the 4th position must contain a 3.

We cannot put a 1 in the 2nd slot (since we have to put it in the first slot), so the 2nd slot has to be either a 2 or 3. If it is a 2, the 3rd slot can't be a 2 or 3, so it must be a 1. Otherwise, the 3rd slot can be a 1 or 2. To represent these arrangements as arrays:

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

There are multiple parts to this problem, but you shouldn't take this to mean that they necessarily need to be done in order. In particular, **parts (a), (c), and (d) are especially closely related, so you should work on them in parallel.**

(a) In this part, you'll explain what the recurrence calculates and how it should be interpreted in the context of the scenario.

(i) Give an intuitive interpretation in plain English of each of the recurrence inputs (i and y).

Hint: Remember that the first index must always be a 1 and that in our recursive case we're doing OPT(i + 1).

(ii) Give an explanation (again, in plain English) of what the value OPT$(i,\, y)$ intuitively represents.

Caution: be careful when describing how the problem that the recurrence solves is constrained compared to the original problem.

(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.)

[1] Those dang DP questions!

(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!

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

(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 $y \neq x$.

Think about how you would solve a problem instance with $n = 3$, some k, and $x = 1$ from first principles, without a recurrence – this should help motivate the intuition behind the 2nd and 3rd branches.

(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 integer arrangements).

Hint: In general, the recurrence solves a more restrictive version of the original problem.

Think about what values of i and y would make the additional restrictions "redundant", in the sense that restrictions imposed by the recurrence would be the same as restrictions already present as part of the problem.

(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"). You can modify the recurrence to 0-indexing for the purposes of writing your pseudocode if you'd like or leave it as 1-indexing, so long as you make clear which one you're using.

(ii) State *and* explain the asymptotic runtime of your pseudocode in terms of n, k, and x.

3. Cookies

Congrats! You passed your interview and became a CSE 417 TA. As a new TA, Robbie has asked you to pick-up cookies for the staff meeting. However, the bakery only sells cookies in specific batch sizes, one batch per box. Since you have limited room, you want to find the least number of boxes to pick-up.

You are given:

- a list of the batch sizes `int[] S` – sorted so that `S[i] < S[i+1]`, and `S[0] = 1` (i.e. you can always buy a single cookie, so any order can be satisfied).
- `int n`, the total number of cookies Robbie wants.

For example, a set of batch sizes could be `S = [1,6,12,24,32]`.

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

```
1: function CountBoxes(S[ ], n)
2:     numBoxes ← 0
3:     for i from S.length − 1 down to 0 do
4:         while n ≥ S[i] do
5:             numBoxes++
```

```
6:            n ← n − S[i]
7:    return numBoxes
```

This algorithm does **not** always work.

Come up with a list of batch sizes and total cookies where this algorithm fails, i.e. it uses more boxes than necessary. Describe the batch sizes found by the algorithm and a way to get a smaller total number of boxes.

(b) Write a recurrence that you can use to calculate the minimum **number** of boxes required to store n total cookies. (Do not write code, just the recurrence. You only need the *total* number of boxes used, *not* the number per batch size.)

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 calculate for n cookies.

You should look at Lecture 11, slide 5 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 b be the length of `S` and n be the number of cookies 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:

- `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[2], 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).

Resubmission

[2]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.

You may resubmit two problems from any combination of past homeworks. When you do, remember to fill out the form on the assignments page so we know which problem you submitted.