CS 161 **Problem Set 6**
Fall 2025 **Due:** Friday November 21, at 11:59pm on Gradescope.

Style guide and expectations: Please see the top of the "Homework" page on the course webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards.

Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Collaboration policy: You may do the HW in groups of size up to three. Please submit one HW for your whole group on Gradescope. (Note that there is an option to submit as a group). See the "Policies" section of the course website for more on the collaboration policy.

LLM policy: Check out the course webpage for best practices on how to productively use LLMs on homework, if you use them at all.

Exercises

We recommend you do the exercises on your own before collaborating with your group.

---

1. **(6 pt.)** In this exercise, we'll practice identifying recursive relationships between subproblems.

   For each of the parts below, refer to the following possible answers:

   (A) $D[i][a] = \begin{cases} D[i-1][a] + D[i-1][a-c_i] & a \geq c_i \\ D[i-1][a] & a < c_i \end{cases}$

   (B) $D[i][a] = \begin{cases} D[i-1][a] + D[i][a-c_i] & a \geq c_i \\ D[i-1][a] & a < c_i \end{cases}$

   (C) $D[i][a] = \begin{cases} \max\{D[i-1][a], D[i-1][a-c_i]+1\} & a \geq c_i \\ D[i-1][a] & a < c_i \end{cases}$

   (D) $D[i][a] = \begin{cases} \max\{D[i-1][a], D[i][a-c_i]+1\} & a \geq c_i \\ D[i-1][a] & a < c_i \end{cases}$

   (a) **(2 pt.)** Suppose you have a pile of m coins, with values $c_1, c_2, \ldots, c_m$ (cents). Let $D[i][a]$ be the number of ways that you can make a cents using the coins $\{c_1, \ldots, c_i\}$. Note that you can only use each coin once.

   For example, suppose that $c_1, \ldots, c_m$ was $1, 1, 5, 5, 10, 25$ (so you have two pennies, two nickels, a dime and a quarter). Then $D[3][6] = 2$, since you can make 6 cents using the coins $\{1, 1, 5\}$ in two different ways: the nickel and the first penny, or the nickel and the second penny.

   Which of the recurrence relations above does $D[i][a]$ satisfy? (Ignoring base cases).

   [**We are expecting:** *Just your answer, A, B, C or D. No justification is required.*]

(b) **(2 pt.)** Now suppose that you have a list $c_1, \ldots, c_m$ of m coin denominations in a particular currency (say, cents). You have as many copies of each coin as you like. Let $D[i][a]$ be the number of ways to make a cents using the first i *types* of coins.

For example, in US currency, we'd have $m = 4$ and $c_1, c_2, c_3, c_4 = 1, 5, 10, 25$ (corresponding to pennies, nickels, dimes and quarters). Then $D[1][6] = 1$, since there is one way to make 6 cents out of only pennies (use six pennies); and $D[2][6] = 2$, since there are two ways to make 6 cents out of pennies and nickels (either six pennies; or one penny and one nickel).

Which of the recurrence relations above does $D[i][a]$ satisfy? (Ignoring base cases).

[**We are expecting:** *Just your answer, A, B, C or D. No justification is required.*]

(c) **(2 pt.)** As in the previous part, let $c_1, c_2, \ldots, c_m$ denote different coin denominations. Let $D[i][a]$ be the *maximum* number of coins that can be used to make a, using as many coins as you want with denominations $c_1, c_2, \ldots, c_i$. (If there is no way to make a out of those denominations, then $D[i][a] = -\infty$).

For example, suppose you are in a different country with no pennies, but two-cent coins (tuppennies) instead, so $c_1, c_2, c_3, c_4 = 2, 5, 10, 25$. Then $D[3][10] = 5$, since you can make 10 cents either as five tuppennies, two nickels, or one dime; so "five" is the biggest number of coins that you can use.

Which of the recurrence relations above does $D[i][a]$ satisfy? (Ignoring base cases).

[**We are expecting:** *Just your answer, A, B, C or D. No justification is required.*]

Recommended exercise (not required and not to be turned in): For each of the parts above, write down base cases, and turn these relationships into DP algorithms for all three problems. (That is, each of the problems would take as input a target value A and a set of m values, and you are asked to find the number of ways to make (or the maximum number of coins that can make) A out of the m values, in the three different settings).

Problems

2. **(7 pt.)** (**Improving on Vanilla Recursion**) Consider the following problem, MinElementSum.

   > MinElementSum(n, S): Let S be a set of positive integers, and let n be a non-negative integer. Find the minimal number of elements of S needed to write n as a sum of elements of S. (Note: it's okay to use an element of S more than once, but this counts towards the number of elements). If there is no way to write n as a sum of elements of S, return `None`.

   For example, if $S = \{1, 4, 7\}$ and $n = 10$, then we can write $n = 1 + 1 + 1 + 7$ and that uses four elements of S. The solution to the problem would be "4." On the other hand if $S = \{4, 7\}$ and $n = 10$, then the solution to the problem would be "`None`," because there is no way to make 10 out of 4 and 7.

   Your friend has devised a recursive algorithm to solve MinElementSum. Their pseudocode is below.

```
def minElementSum(n, S):
     if n == 0:
         return 0
     if n < min(S):
         return None
     candidates = []
     for s in S:
         cand = minElementSum( n-s, S )
         if cand is not None:
              candidates.append( cand + 1 )
     if len(candidates) == 0:
         return None
     return min(candidates)
```

   Your friend's algorithm correctly solves MinElementSum. Before you start doing the problems on the next page, it would be a good idea to walk through the algorithm and to understand what this algorithm is doing and why it works.

[Questions on next page]

(a) **(1 pt.)** Argue that for $S = \{1, 2\}$, your friend's algorithm has exponential running time. (That is, running time of the form $2^{\Omega(n)}$).

[**Hint:** *You may use the fact (stated in class) that the Fibonacci numbers $F(n)$ satisfy $F(n) = 2^{\Omega(n)}$.*]

[**We are expecting:**

- *A recurrence relation that the running time of your friend's algorithm satisfies when $S = \{1, 2\}$.*
- *A convincing argument that the closed form for this expression is $2^{\Omega(n)}$. You do not need to write a formal proof.*

]

(b) **(3 pt.)** Turn your friend's algorithm into a top-down dynamic programming algorithm. Your algorithm should take time $O(n|S|)$.

[**Hint:** *Add an array to the pseudocode above to prevent it from solving the same subproblem repeatedly.*]

[**We are expecting:**

- *Pseudocode **AND** a short English description of the idea of your algorithm.*
- *An informal justification of the running time.*

]

(c) **(3 pt.)** Turn your friend's algorithm into a bottom-up dynamic programming algorithm. Your algorithm should take time $O(n|S|)$.

[**Hint:** *Fill in the array you used in part (b) iteratively, from the bottom up.*]

[**We are expecting:**

- *Pseudocode **AND** a short English description of the idea of your algorithm.*
- *An informal justification of the running time.*

]

3. **(12 pt.)** [**Corn Maze!**] You are getting into the autumnal spirit and have decided to visit your local pumpkin patch and corn maze.[1] The paths in the maze are directed (to prevent traffic jams), and there are no cycles (to prevent visitors from getting lost/stuck forever). At various points throughout the maze there are fun fall-themed activities (pumpkin toss, scarecrow selfie station, leaf pile for jumping, etc). Thus, you can view the maze as a directed acyclic graph (DAG), where each vertex is an activity, and an edge from activity a to activity b means that you can walk directly from a to b in the corn maze. There are also special vertices s and t, denoting the entry and exit point of the maze, respectively; you may assume that s is the unique vertex with no incoming edges in G, and t is the unique vertex with no outgoing edges in G.

   You are given a map of the maze (the DAG $G = (V, E)$), and you have your own estimate $f(a)$ of how fun each activity a would be for you. (That is, $f : V \to \mathbb{R}^+$ is a function that assigns a positive real number to each vertex of G). Your goal is to maximize the amount of fun you can have with one pass through this corn maze. Suppose that G has n vertices and m edges.

   Design a dynamic programming algorithm that runs in time $O(n + m)$, takes as input the DAG G and the "fun function" f, and returns the most fun path you can take through the maze. You may assume that for all $a \in V$, you can evaluate $f(a)$ in time $O(1)$. Do this by answering the two parts below.

   (a) **(3 pt.)** What sub-problems will you use in your dynamic programming algorithm? What is the recursive relationship satisfied between the sub-problems?

   [**Hint:** *As part of the above question, you also need to impose some sort of order on the sub-problems...*]

   [**We are expecting:**

   - *A clear description of your sub-problems, including how they are indexed*
   - *A recursive relationship that they satisfy, along with any necessary base case(s)*

   ]

   (b) **(3 pt.)** Write pseudocode for your algorithm. It should take in G and f, and return a single number that is the maximum amount of fun you can have in the corn maze. Your algorithm does not need to return the optimal path through the maze. You may use any algorithm we have seen in class as a black box.

   [**We are expecting:** *Clear pseudocode, and a short justification that the running time is $O(n + m)$. You do not need to justify that your algorithm is correct, but it should follow from your answer to part (a).*]

   (c) **(0 pt.) [OPTIONAL. This problem is not required and won't be graded, but it's good practice if you want to do it on your own.]** Write pseudocode for an algorithm that returns the optimal path through the maze, in addition to the optimal amount of fun.

   [**We are expecting:** *Nothing. This problem is not required.*]

[1] It's too late this year, but if you are in the Bay Area next fall and want to experience a pumpkin patch extravaganza, check out Spina Farms Pumpkin Patch in Morgan Hill (there are several others in the area as well). It may be a bit kitschy, but they have a corn maze featuring *animatronic dinosaurs*! [Note: we are not sponsored by Spina Farms, we're just really into animatronic dinosaurs.]

(d) **(6 pt.)** Shucks, after all that work on parts (a) and (b) (and optionally (c)), you realized that there's a complication you forgot! At the beginning of the corn maze you are given C tickets to spend on the activities. The activity at vertex v has a cost of $c(v)$ tickets, where $c(v)$ is a non-negative integer. (As with the function f, you may assume that you can evaluate $c(v)$ in time $O(1)$.) At each activity along your path through the maze, you can either choose to pay $c(v)$ tickets and get $f(v)$ fun, *or* you can decide to skip that activity and pay 0 tickets and get 0 fun. Design a DP algorithm that takes as input the DAG G, the functions f and c, and the number of tickets C that you start with; and outputs the maximum amount of fun you can have while finishing the maze, without going over your budget of C tickets. (You must still walk along paths in the DAG, as with the previous parts). If there is no way for you to finish the maze within budget, you just don't enter to begin with, so you get fun zero.

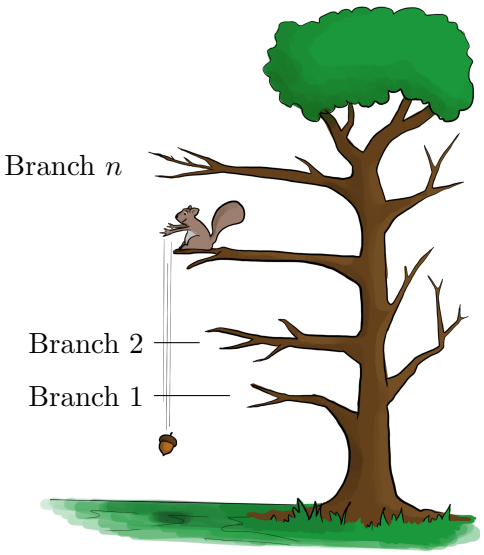
Your algorithm should run in time $O(C(n+m))$.

[**Hint:** *Consider filling in a two-dimensional array.*]

[**We are expecting:** *A description of your subproblems, the recursive relationship they satisfy, any base cases, and pseudocode that implements your algorithm. You do not need to justify the correctness or the running time.*]

4. **(8 pt.) [Nuts! (part 2)]**

Socrates the Scientific Squirrel (from HW1) is back! Recall that Socrates lives in a very tall tree with n branches, and she wants to find out what is the lowest branch $i \in \{1, \ldots, n\}$ so that an acorn will break open when dropped from branch i. If an acorn breaks open when dropped from branch i, then an acorn will also break open when dropped from branch j for any $j \geq i$. (If no branch will break an acorn, Socrates should return $n + 1$).



The catch is that, once an acorn is broken open, Socrates will eat it immediately and it can't be dropped again.

In HW1, you designed a strategy for Socrates to use very few drops, given that she had k acorns. She was pretty pleased with that algorithm, but now she wants to compute *exactly* the number of drops she needs, in the worst case.

For $n \geq 0$ and $k \geq 1$, let $D(n, k)$ be the *optimal worst-case number of drops* that Socrates needs to determine the correct branch out of n branches using k acorns. That is, $D(n, k)$ is the number of drops that the best algorithm would use in the worst-case. Write a dynamic program to compute $D(n, k)$, given n and k, in time $O(n^2 k)$.

[**Hint:** *Suppose that the first drop of the optimal algorithm is from branch x. Can you write $D(n, k)$ in terms of $D(x - 1, something)$ and $D(n - x, something)$?*]

[**We are expecting:** *The following things:*

- *A clear description of your sub-problems, the relationship they satisfy, and any base cases*
- *Clear pseudocode for your algorithm, which follows the sub-problem relationship you wrote down.*
- *A short justification of the running time.*

]

5. **(4 pt.) [EthiCS: Vaccine Distribution]**

In all of the problems above (and in general in the sorts of DP problems we solve on pedagogical CS161 HW assignments), we have a very tidy set of options and constraints. In the corn maze problem, we want to maximize "fun" subject to a ticket budget and some graph constraints; in Socrates's acorn-dropping problem, we want to know the best way to drop acorns, subject to a budget of k acorns. However, the real world is not so tidy. For example, imagine it is early 2021, and you are deciding how to allocate a limited number of COVID-19 vaccines. At a first pass, it might look a bit like our optimization problems with options and constraints. We'd like to optimize public health, by vaccinating some people, given a limited amount of vaccine and with the additional constraint of being equitable.

We can see that our goals and constraints are not as easy to mathematically formulate as with our tidy story problems above, but in order to come up with a good policy, we need to try! How might you define the goal of "maximizing population health" and the constraint of "equitable access" for vaccine distribution? What other factors do you need to consider that aren't necessarily represented in the problem formulation of optimization and constraints?

Of course, there is not a right answer to this question! What we are looking for is a serious attempt to think about how you might model this, and what the issues might be.

[**We are expecting:** *(1) A paragraph or so describing what might contribute to formal definitions of "population health" and "equitable access," and (2) 2-3 sentences describing at least two other factors you might need to consider about vaccine distribution when designing a policy.*]

(Optional Reading: Nature and NIH have articles about research that tested different vaccine distribution strategies)

This problem set is long enough, but more practice with dynamic programming is always good. If you'd like another practice problem, here's one you can try!

6. **(0 pt.) [OPTIONAL: this problem is for extra practice and will not be graded] [Fish fish eat eat fish.]** Plucky the Pedantic Penguin enjoys fish, and wants to catch as many fish as possible from two nearby Lakes named A and B. They have discovered that on some days the fish supply is better in Lake A, and some days the fish supply is better in Lake B. Plucky has access to two tables A and B, where $A[i]$ is the number of fish they can catch in Lake A on day i, and $B[i]$ is the number of fish they can catch in Lake B on day i, for $i = 0, \ldots, n-1$.

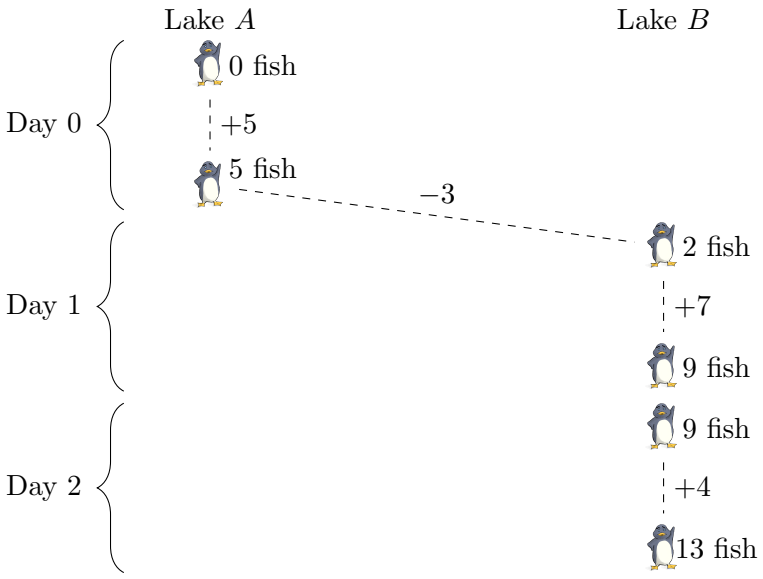
   If Plucky is at Lake A on day i and wants to be at Lake B on day $i+1$, they may pay L fish to a polar bear who can take them from Lake A to Lake B overnight; the same is true if they want to go from Lake B back to Lake A. The polar bear does not accept credit, so **Plucky must pay *before* they travel**. (And if they cannot pay, they cannot travel).

   Assume that when day 0 begins, Plucky is at Lake A, and they have zero fish. Also assume that $A[i]$ and $B[i]$ are positive integers for $i = 0, 1, \ldots, n-1$ and that L is also a positive integer.

   For example, suppose that $n = 3$, $L = 3$, and that A and B are given by

   $$A = [5, 2, 3] \qquad B = [2, 7, 4].$$

   Then Plucky might do:

   So Plucky's total fish at the end of day $n - 1 = 2$ is 13.

   In this question, you will design an $O(n)$-time dynamic programming algorithm that finds the maximum number of fish that Plucky can have at the end of day $n-1$. Do this by answering the two parts below.

(a) **(0 pt.)** What sub-problems will you use in your dynamic programming algorithm? What is the recursive relationship which is satisfied between the sub-problems?

[**We are expecting:** *Nothing, this part isn't required. But for food practice, write down:*

- *A clear description of your sub-problems.*
- *A recursive relationship that they satisfy, along with a base case.*
- *An informal justification that the recursive relationship is correct.*

]

(b) **(0 pt.)** Design a dynamic programming algorithm that takes as input A, B, L and n, and in time $O(n)$ returns the maximum number of fish that Plucky can have at the end of day $n-1$.

[**We are expecting:** *Nothing, this part is not required. But for good practice, write down clear pseudocode, and a justification of why it runs in time $O(n)$. You do not need to justify the correctness of your pseudocode, but it should follow from your reasoning in part (a).*]