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).

SOLUTION:

(a) The answer is (A). To see why, suppose we are trying to make a cents out of the first i coins. If $a \geq c_i$, then we have the option to either include the i'th coin or not. If we don't include it, the number of ways are $D[i-1][a]$ (the ways to make a using the first $i-1$ coins); and if we do, the number ways is $D[i-1][a-c_i]$ (the ways to make $a - c_i$ using the first $i-1$ coins). Adding them together gives the total number of ways. (On the other hand, if $a < c_i$, then we just drop the second term, since it's not possible to use c_i to make a).

(b) The answer is (B). The logic is similar to part (A), except that now, once we've used c_i, we're allowed to use it again. So the number of ways to use c_i (at least) once when $a \geq c_i$ is $D[i][a-c_i]$, not $D[i-1][a-c_i]$.

(c) The answer is (D). Again, the logic is similar to the previous part, except that we take the max instead of summing, since we are trying to maximize the number of coins we can possibly use.

For the "Recommended Exercise:"

(a) Base cases are $D[0][0] = 1$, $D[0][a] = 0$ for $a > 0$, and $D[i][0] = 1$ for all i. The DP algorithm (for finding the number of ways to make an input value A) is along the lines of:

```
Initialize an m-times-A array D as above
For i = 1, ..., m:
     For a = 1, ..., A:
          D[i][a] = D[i-1][a]
          if c_i <= a:
               D[i][a] += D[i-1][a-c_i]
return D[m][A]
```

(b) Base cases are $D[i][0] = 1$ for all i and $D[0][a] = 0$ for all $a > 0$. The DP algorithm (for finding the number of ways to make an input value A) is along the lines of:

```
Initialize an m-times-A array D as above
For i = 1, ..., m:
     For a = 1, ..., A:
          D[i][a] = D[i-1][a]
          if c_i <= a:
               D[i][a] += D[i][a-c_i]
return D[m][A]
```

(c) Base cases are $D[i][0] = 0$ for all i and $D[0][a] = -\infty$ for all $a > 0$. The DP algorithm (for finding the max number of coins to make an input value A) is along the lines of:

```
Initialize an m-times-A array D as above
For i = 1, ..., m:
     For a = 1, ..., A:
          D[i][a] = D[i-1][a]
          if c_i <= a:
               D[i][a] = max{ D[i][a], D[i][a-c_i] + 1}
return D[m][A]
```

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.*

]

SOLUTION: Solution 1: When $S = \{1, 2\}$, then the algorithm running on n makes two recursive calls, one to $n - 1$ and one to $n - 2$. The running time of this algorithm satisfies the recurrence

$$T(n) = T(n-1) + T(n-2) + O(1) \geq T(n-1) + T(n-2),$$

since the $O(1)$ term is non-negative. Thus, the running time of this algorithm grows at least as fast as the Fibonacci numbers. We saw in class that these grow exponentially quickly.

Solution 2: To see this from first principles (rather than appealing to the statement from class), we can also write

$$T(n) \geq T(n-1) + T(n-2) + O(1) \geq 2T(n-2),$$

using the fact that T is increasing and the $O(1)$ term is positive. Let

$$C = 2^{-1/2} \cdot \min\{T(1), T(2)\},$$

so $C > 0$ is some constant. Then we use the substitution method with the inductive hypothesis that $T(n) \geq C \cdot 2^{n/4}$. For the base case, we have by our choice of C that $T(1) \geq C \cdot 2^{1/2} \geq C \cdot 2^{1/4}$ and $T(2) \geq C \cdot 2^{1/2}$. Then for the inductive step, assume that $n > 2$ and that the inductive hypothesis holds for all positive integers less than n. Then we have

$$T(n) \geq 2T(n-2) \geq 2 \cdot C \cdot 2^{(n-2)/4} \geq C \cdot 2^{n/4 - 1/2 + 1} \geq C \cdot 2^{n/4},$$

so this establishes the inductive hypothesis for n. We conclude that $T(n) \geq C \cdot 2^{n/4} = 2^{n/4 + \log(C)}$ for all sufficiently large n, which means that $T(n) = 2^{\Omega(n)}$, as desired.

(Note, a formal proof by induction is not required for credit on this problem).

(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.*

]

SOLUTION:

To make a top-down DP algorithm, we add an array D to keep track of previous solutions. We set $D[k]$ to be the minimal number of elements of S needed to make k.

```
Initialize a global array D of length n+1 to all -1's.
D[0] = 0
Set D[k] = None for all 0 < k < min(S).
def minimumElements(n, S):
     if n < 0:
         return None
     if D[n] != -1:
         return D[n]
     candidates = []
     for s in S:
         cand = minimumElements( n-s, S )
         if cand is not None:
              candidates.append( cand + 1 )
     if len(candidates) == 0:
          D[n] = None
     else:
          D[n] = min(candidates)
     return D[n]
```

The running time of this algorithm is $O(n|S|)$. For each $k \leq n$, we run `minimumElements`(k, S) at most once. For each such call, we do work $O(|S|)$ by performing $O(1)$ operations for each $s \in S$ while calling the recursive calls and aggregating the results.

(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.*

]

SOLUTION: To make a bottom-up DP algorithm, we'll use the same interpretation of the array as before, but fill it in in bottom-up order.

```
Initialize a global array D of length n+1 to all -1's.
D[0] = 0
Set D[k] = None for all 0 < k < min(S).

for i = min(S), ..., n:
     candidates = []
     for s in S:
          if i < s:
              continue
          cand = D[i-s]
          if cand is not None:
                candidates.append( cand + 1 )
     D[i] = min(candidates)

return D[n]
```

Again the running time is $O(n|S|)$, since the outer for loop has n iterations, and the inner for loop has $|S|$ iterations. Inside the inner loop we do $O(1)$ work.

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)*

   ]

   SOLUTION: In order to put an order on the vertices, we will topologically sort them. Let $s = v_0, v_1, \ldots, v_{n-1} = t$ be any legitimate topological sorting of V. Notice that we must have $s = v_0$ and $t = v_{n-1}$, since s is the unique vertex with no incoming edges and t is the unique vertex with no outgoing edges.
   Given this ordering, our problems will be indexed by $i = 0, \ldots, n-1$, and will be:

   $A[i] =$ the maximum amount of fun you can get from a path ending at v_i (including at v_i)

   Our base case is $A[0] = f(s)$, since the maximum amount of fun we can have at the starting vertex is, by definition, $f(s)$. Then the sub-problems satisfy the recursive relationship:

$$A[i] = \max\{A[j] + f(v_i) \,:\, (v_j, v_i) \in E\}.$$

[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.]

That is, $A[i]$ the the maximum, over all j so that there is an edge from v_j to v_i in E, of the amount of fun you can have from a path ending at v_j, plus the amount of fun you can have at v_i.

Note that the reason that we topologically sorted the vertices is so that the recursive relationship is actually ordered (that is, so that $A[i]$ only depends on $A[j]$ for $j < i$, and not on any $A[\ell]$ for $\ell > i$). Indeed, if $(v_j, v_i) \in E$, then we must have $j < i$ by the definition of a topological ordering. (This discussion was not required for credit, we are just including it in the solutions for clarity).

(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).*]

SOLUTION: The algorithm is as follows:

```
def funnestCornMaze(G, f):
     Run TopoSort on G to obtain an ordering on the vertices.
     Arrange the vertices so that  V[j] is the j'th vertex in that order.
     Initialize an array A of length n
     Set A[0] = f(V[0]), and A[i] = 0 for all i > 0
     for i = 1, ..., n-1:
          for all j in V[i].in_neighbors:
               A[i] = max( A[i], A[j] + f(V[i]) )
     return A[n-1]
```

For the running time, running TopoSort takes time $O(n+m)$, as we saw in class. Then we loop over all n vertices, and for each vertex v we do $O(1) + O(\text{in-degree}(v))$ work. So the total running time for this part is

$$\sum_{v \in V} (O(1) + O(\text{in-degree}(v))) = O(n + m).$$

Altogether, the running time is $O(n + m)$, as desired.

(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.*]

SOLUTION: We use similar code as before, except when we take the maximum over in-neighbors, we keep track of which in-neighbor attained that maximum.

```
def funnestCornMaze(G, f):
     Run TopoSort on G to obtain an ordering on the vertices.
     Say that V[j] is the j'th vertex in this ordering.
     Initialize an array A of length n.
     Set A[0] = (f(V[0]), None), and A[i] = (0, None) for all i > 0
     for i = 1, ..., n-1:
          for all j in V[i].in_neighbors:
               jVal = A[j][0] + f(V[i])
               if jVal > A[i][0]:
                    A[i] = ( jVal, j )
     retVal = A[n-1][0]
     retPath = []
     current = n-1
     while current != None:
         insert current at the beginning of retPath
            // suppose retPath is implemented as a linked list so the above is O(1)
         current = A[current][1]
     return retVal, retPath
```

(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.*]

SOLUTION: Our sub-problems will be $A[i, c]$ for $i = 0, \ldots, n-1$ and $c = 0, \ldots, C$. As before, we sort the vertices of G in topological order, so the order is $s = v_0, v_1, \ldots, v_{n-1} = t$, and we interpret $A[i, c]$ as "the most amount of fun you can have ending at v_i (including

v_i), with a budget of exactly c tickets."

Note: It would also be fine to define it with "a budget of at most c tickets," it's just that the base cases and what you read out would change slightly.

The recursive relationship that our problems obey are

$$A[i,c] = \max\left\{\max\{A[j, c - c(v_i)] + f(v_i) \,:\, (v_j, v_i) \in E\}, \max\{A[j,c] \,:\, (v_j, v_i) \in E\}\right\}.$$

That is, at vertex v_i, we look at each incoming edge (v_j, v_i). Coming from vertex v_j, we can either pay cost $c(v_i)$ and get fun $f(v_i)$ (this is the first term in the big max above), or we can choose to pay zero and get fun zero (this is the second term in the big max above). Above, we have the convention that $A[j, \text{any negative number}] = -\infty$.

As our base case, we have $A[0,0] = 0$, $A[0, c(v_0)] = f(v_0)$, and $A[0,x] = -\infty$ for all $x \notin \{0, c(v_0)\}$. (Note that if $c(v_0) = 0$, the second over-writes the first).

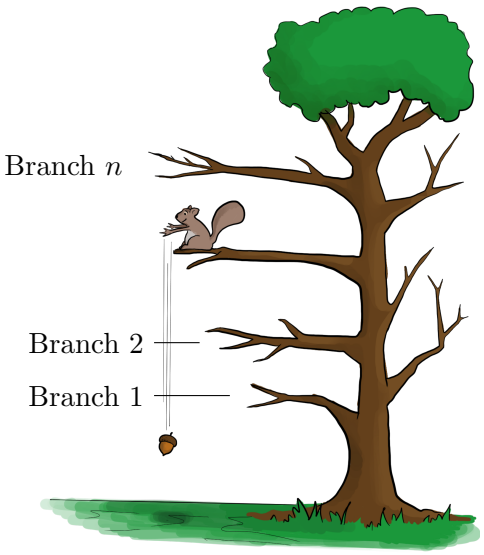
The algorithm is as follows:

```
def funnestCornMaze(G, f):
     Run TopoSort on G to obtain an ordering on the vertices.
     Arrange the vertices so that  V[j] is the j'th vertex in that order.
     Initialize a two-dimensional array A of size n by (C+1)
     Set A[i,c] = -Infinity for all i,c
     Set A[0,0] = 0.
     Set A[0,c(V[0])] = f(V[0]).
     for i = 1, ..., n-1:
          for all j in V[i].in_neighbors:
               for b = 0, 1, ..., C:
                    # consider the option to come from j and spend 0 tickets:
                    A[i,b] = max( A[i,b], A[j,b] )
                    if b >= c(V[i]):
                          # consider the option to come from j and spend c(V[i]):
                          A[i,b] = max( A[i,b], A[j, b - c(V[i])] + f(V[i]))
     ret = max( A[n-1, b] for b = 0,1,...,C )
     if ret = -Infinity, return 0 # in this case we just don't enter the maze
     return ret
```

The running time is a factor of C larger than before (since we basically repeat part (b) for each of the $C+1$ rows of the table), so the running time is $O(C(n+m))$, as desired.

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.*

]

SOLUTION: As suggested by the problem statement, our sub-problems are $D[m, j] = D(m, j)$ for $m \leq n$ and $j \leq k$. As base cases, we have:

$D[0, i] = 0$ because if there are zero branches then Socrates needs zero drops.

$D[1, i] = 1$ because if there is only one branch then Socrates should drop an acorn from that branch to see if it breaks or not.

$D[m, 1] = m$ because with one acorn Socrates needs m drops in the worst case, since she must start from the bottom of the tree and work up one branch at a time.

Now, suppose that the optimal algorithm (whatever it is) drops its first acorn from branch x. In that case, we claim that

$$D[n, k] = 1 + \max\{D[x - 1, k - 1], D[n - x, k]\}.$$

This is because if the acorn breaks from branch x, then Socrates has $k - 1$ acorns left, and she knows that there are only $x - 1$ possible branches, so this is the same as trying to solve the original problem with $k \leftarrow k - 1$ and $n \leftarrow x - 1$, so the number of drops is $D[k - 1, x - 1]$. On the other hand, if the acorn does not break, then Socrates still has k acorns left, and she knows that there are only $n - x$ possible branches (the ones above branch x), so this is the same as trying to solve the original problem with $k \leftarrow k$ and $n \leftarrow n - x$. Thus the number of drops in this case is $D[k, n - x]$. Since we are considering worst-case outcomes, the number of drops Socrates needs is the maximum of the two.

This implies that

$$D[n, k] = 1 + \min\left\{\max\{D[x - 1, k - 1], D[n - x, k]\}, x \in \{1, \ldots, n\}\right\}.$$

The reason is because the best algorithm must drop its first acorn from some branch x. By part (b), if the best algorithm drops its first acorn from x, then the number of drops remaining is $\max\{D[x - 1, k - 1], D[n - x, k]\}$; so then after we take the min we must have the number of drops of the best algorithm.

Now we can formulate our algorithm by implementing the above recursive relationship. We start by filling in the first two rows of the table, starting with $n = 0$ and $n = 1$. Then we use the recurrence relation above to fill in the table from bottom to top, going across each row from left to right. The pseudocode is below.

Algorithm 1: Find $D[n, k]$

```
Input: n,k
Initialize an (n + 1) × (k + 1) array D (zero-indexed).
Set D[0, i] = 0 for all i = 0, 1, ..., k
Set D[1, i] = 1 for all i = 0, 1, ..., k
Set D[i, 0] = ∞ for all i = 1, ..., n
Set D[i, 1] = i for i = 0, ..., n
for m = 2, ..., n do
    for j = 2, ..., k do
        D[m, j] = ∞
        for x = 1, ..., m do
            D[m, j] = min{D[m, j], max{D[x − 1, j − 1], D[m − x, j]}}
            /* Notice that D[m − x, j] and D[x − 1, j − 1] have already been filled
               out.                                                              */
        D[m, j] = D[m, j] + 1
return D[n, k]
```

The running time is $O(n^2 k)$ because there are three for-loops, one over n, one over k and one

over $m \leq n$, and the work done inside the three for-loops is $O(1)$ (just looking up items in a table and taking mins/maxes).

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)

SOLUTION: One way to quantify "maximizing population health" might be by trying to minimize the number of infected people; another might be by trying to minimize the number of extreme negative health outcomes, for example hospitalizations and deaths. This second approach could mean prioritizing individuals who are more likely to be more severely impacted such as elderly and immunocompromised individuals, as well as people whose occupation or living situation put them at higher risk of contracting and further spreading the disease, such as healthcare workers and apartment or communal living communities residents. Formally, we could use an epidemiological model (like the SIR model) that takes into account risk factors, and try to minimize deaths in that model.

The constraint of "equitable access" ensures that the distribution of the vaccine is accessible and fair for those who need the vaccine, regardless of their demographics or socioeconomic status. This can mean ensuring the availability of vaccination sites in underserved neighborhoods or accessible by public transport, providing language support for non-English speakers, or providing alternative vaccination appointment registration methods to ensure access for those without internet access or digital experience. Additionally, we have to consider areas that have resource constraints such as financial capability, human capital to operate vaccine centers, or cold storage equipment to store and transport the vaccines. Formally, this constraint could be modeled by requiring a vaccine center in every city/neighborhood.

Of course, even if we came up with the perfect way of formalizing the objective function and the constraints, and even if we came up with an efficient algorithm that tells us exactly whom we should vaccinate, there are plenty of other things we'd need to take into consideration. For example, how do we effectively get vaccines to exactly those people? This might require way more infrastructure than a less targeted approach. Another issue is personal choice: what if some of the identified people don't want to be vaccinated?

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 $i = 0, \ldots, n-1$.

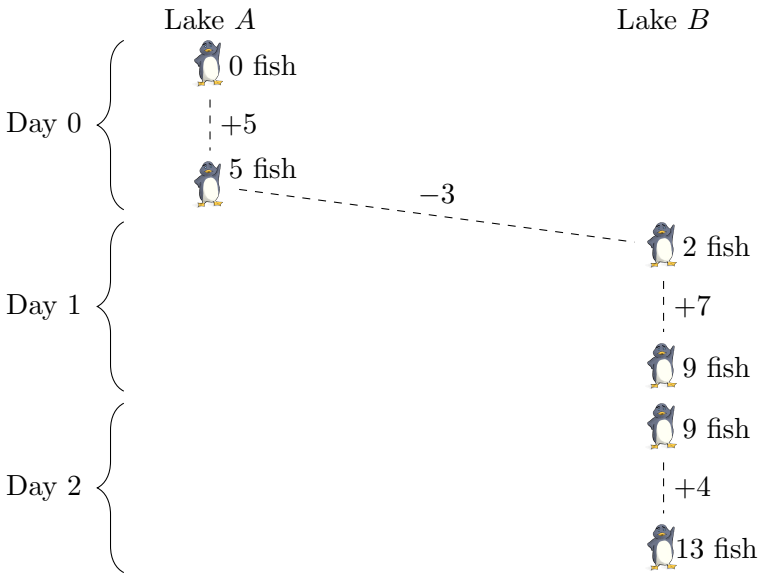
   If Plucky is at Lake A on day i and wants to be at Lake B on day $i+1$, they may pay L fish to a polar bear who can take them from Lake A to Lake B overnight; the same is true if they want to go from Lake B back to Lake A. The polar bear does not accept credit, so **Plucky must pay *before* they travel**.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

]

SOLUTION:

Our sub-problems will be indexed by a flag ℓ (either A or B), an integer $i \in \{0, \ldots, n-1\}$. Let $K[\ell, i]$ denote the number of fish that Plucky owns on day i, assuming they are in location ℓ on that day. The recursive relationship is:

$$K[A,i] = \begin{cases} \max\{K[A,i-1] + A[i], K[B,i-1] - L + A[i]\} & \text{if } K[B,i-1] \geq L \\ K[A,i-1] + A[i] & \text{else} \end{cases}$$

and

$$K[B,i] = \begin{cases} \max\{K[B,i-1] + B[i], K[A,i-1] - L + B[i]\} & \text{if } K[A,i-1] \geq L \\ K[B,i-1] + B[i] & \text{else} \end{cases}$$

The base case is that $K[A,0] = A[0]$ and $K[B,0] = -\infty$.

To see that the recursive relationship is correct, consider just the first one, $K[A,i]$. The maximum amount of fish that Plucky can eat if they are in location A on day i is the maximum of two possibilities: either they were in location A on day $i-1$, stayed in location A, and gained $A[i]$ fish; or else they were in location B on day $i-1$, paid L to the polar bear to move from B to A, and then ate $A[i]$ fish. Plucky only has the possibility of traveling if their current amount of fish, $K[B,i-1]$, exceeds L. This explains the expressions in the first equation. The second equation is similar.

To see why the base case is correct, observe that $K[A,0]$ should be the number of fish that Plucky has at the end of day 0, assuming they were in location A; this is just $A[0]$. $K[B,0]$ should be the number of fish that Plucky has at the end of day 0 assuming they were in location B; Plucky is not allowed to be in location B on day 0, so we'll set $K[B,0]$ to negative infinity and we observe that this works with our recursive relationship because the maximum will never choose $-\infty$ (and it will never deviate from $-\infty$, incorrectly counting $B[i]$ for days it would be impossible to be on lake B).

(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).]

SOLUTION:

```
fishFishEatEatFish(A, B, L, n):
    initialize a 2-by-n array K
           // the first coordinate of K is indexed by {A,B}
           // the second coordinate of K is indexed by {0, ..., n-1}
    K[A,0] = A[0]
    K[B,0] = -Infinity
    for i = 1, ..., n-1:
        K[A,i] = K[A,i-1] + A[i]
        if K[B,i-1] >= L:
              K[A,i] = max{ K[A,i], K[B,i-1] - L + A[i] }
        K[B,i] = K[B,i-1] + B[i]
        if K[A,i-1] >= L:
              K[B,i] = max{ K[B,i], K[A,i-1] - L + B[i] }
   return max{ K[B,n-1], K[A,n-1] }
```

This works because it is implementing the recursive relationship from part (a). The base case is that on day 1, we have $K[A, 0] = A[0]$ (Plucky can stay and fish) and $K[B, 0] = -\infty$ (Plucky is not allowed to start Day 0 at Lake B). In the end, we correctly return the maximum number of fish obtainable because Plucky must end on one lake, so the maximum number of fish is either the maximum number of fish obtainable given Plucky ends up on lake A or the maximum number of fish obtainable where Plucky ends up on lake B.

The running time is $O(n)$, because the loop is over $i = 1, ..., n - 1$, and the amount of work inside each iteration is $O(1)$.