Note: Your TA may not get to all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. The discussion worksheet is also a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Shortest Paths with Dynamic Programming!

In this problem we will design a dynamic programming algorithm for finding the shortest $S - E$ path in a DAG like the one above.

(a) What are the subproblems you need to solve?

(b) Can you define the optimal solution to any subproblem as a function of the solutions to other subproblems?

(c) The dependency graph of a DP is a directed graph in which each subproblem becomes a vertex, and edge $(u, v)$ denotes that subproblem $u$ requires the solution to subproblem $v$ in order to be solved. What does the dependency graph look like for this problem? What property of the dependency graph allows us to solve the problem using DP?

(d) If we proceed iteratively, what would be the best order to solve these subproblems?

(e) How many subproblems are there, and how long does it take to solve each? What is the overall running time of this algorithm?

(f) Run your algorithm on the example graph. What is the shortest path?

Solution:

(a) We need to know the minimum length paths from the start node to the nodes on possible paths from $S$ to $E$. This will allow us to evaluate which nodes we should include on our path.

(b) Yes. We know that out of all direct ancestors $A_i$ of $E$, at least one of them must be on the shortest path from $S$ to $E$. Let $dist(V)$ denote the subproblem of finding the shortest path from $S$ to some vertex $V$. For any $U$ in the graph, we can argue that shortest distance from $S$ to $U$ is $min\{dist(A_i) + l(A_i, U), ..., dist(A_N) + l(A_N, U)\}$.

(c) Let’s think about the dependency graph this gives us. We know $dist(V)$ is dependent on the $dist(A_i)$ for all direct ancestors of $V$. Thus the dependency graph will look like the reverse of the given graph! And since our given graph is a DAG, we know the dependency graph will also be a DAG. The fact that the dependency graph is a DAG is quite significant. By construction the dependency graph is directed. But more importantly, if the dependency graph is acyclic, we know that we don’t have some set of subproblems that are all mutually dependent on each other (and thus impossible to solve). When the dependency graph of a problem is a DAG, we sometimes say that this problem possesses an "optimal substructure" that makes it solvable via DP.

(d) When we get to a subproblem, we want to have computed the $dist$ value for all its direct ancestors. Thus it makes the most sense to solve subproblems in topological order, starting from $S$, since all direct ancestors of a vertex must come before it in topological order.
(e) There are $|V|$ subproblems in total. This gives us $|V|$ work to look at each subproblem once. For each subproblem, we need to iterate through all the ancestors of the given vertex (this is precisely equal to the vertex’s in-degree). In any directed graph, the sum of the indegrees of each node is precisely equal to the sum of outdegrees, which is precisely equal to $|E|$ (since each edge contributes to the indegree of exactly one node, and the outdegree of exactly one node).

(f) SCDE

2 String Shuffling

Let $x$, $y$, and $z$ be strings. We want to know if $z$ can be obtained only from $x$ and $y$ by interleaving the characters from $x$ and $y$ such that the characters in $x$ appear in order and the characters in $y$ appear in order. For example, if $x = \text{efficient}$ and $y = \text{ALGORITHM}$, then it is true for $z = \text{efficientALGiorcilenTHM}$, but false for $z = \text{efficientALGORITHMS}$ (extra characters), $z = \text{effALGiORciIenTHMt}$ (missing the final $t$), and $z = \text{effOALGRicieTTHMnt}$ (out of order). How can we answer this query efficiently? Your answer must be able to efficiently deal with strings with lots of overlap, such as $x = \text{aaaaaaaaaab}$ and $y = \text{aaaaaaca}$.

1. Design an efficient algorithm to solve the above problem and state its runtime.

2. Consider an iterative implementation of our DP algorithm in part (a). Naively if we want to keep track of every solved sub-problem, this requires $O(|x||y|)$ space (double check to see if you understand why this is the case). How can we reduce the amount of space our algorithm uses?

Solution:

1. First, we note that we must have $|z| = |x| + |y|$, so we can assume this. Let $S(i, j)$ be true if and only if the first $i + j$ characters of $x$ and the first $j$ characters of $y$ can be interleaved to make the first $i + j$ characters of $z$. Then $x$ and $y$ can be interleaved to make $z$ if and only if $S(|x|, |y|)$ is true.

For the recurrence, if $S(i, j)$ is true then either $z_{i+j} = x_i$, $z_{i+j} = y_j$, or both. In the first case it must be that the first $i − 1$ characters of $x$ and the first $j$ characters of $y$ can be interleaved to make the first $i + j − 1$ characters of $z$; that is, $S(i − 1, j)$ must be true. In the second case $S(i, j − 1)$ must be true. In the third case we can have either $S(i − 1, j)$ or $S(i, j − 1)$ or both being true. This yields the recurrence:

$$S(i, j) = (S(i − 1, j) ∧ (x_i = z_{i+j})) ∨ (S(i, j − 1) ∧ (y_j = z_{i+j}))$$

The base case is $S(0, 0) = T$; we also set $S(0, −1) = S(−1, 0) = F$ for convenience. The running time is $O(|x||y|)$.

Somewhat naively if we’d like an iterative solution, we can keep track of the solutions to all subproblems with a 2D array where the entry at row $i$, column $j$ is $S(i, j)$. If we iterate over this array row by row, going left to right, we’ll always be able to fill in the next entry using values we’ve already computed.

Notice, however, that to compute any entry, we only really need the information in the previous row, and the current row we’re filling out. Thus, rather than holding onto the entire table, we only need to store the current and previous row, reducing us from $O(m * n)$ space to $O(m)$ space.

2. We can keep track of the solutions to all subproblems with a 2D array of size $|x||y|$ where the entry at row $i$, column $j$ is $S(i, j)$. If we iterate over this array row by row, going left to right, we’ll always be able to fill in the next entry using values we’ve already computed.
Notice, however, that to compute any entry, we only really need the information in the previous row, and the current row we’re filling out. Thus, rather than holding onto the entire table, we only need to store the current and previous row, reducing from $O(|x||y|)$ space to $O(\min(|x|, |y|))$ space.

### 3 Egg Drop

You are given $k$ identical eggs and an $n$ story building. You need to figure out the maximum floor $\ell \in \{1, \ldots, n\}$ that you can drop them from without breaking them. Each egg will break if dropped from a floor greater than or equal to $\ell$, and will never break when dropped from a floor less then $\ell$. Once an egg breaks, you cannot use it any more. Let $f(n, k)$ be the minimum number of drops to find $\ell$.

(a) Find $f(1, k)$, $f(0, k)$, $f(n, 1)$ and $f(n, 0)$.

(b) Find a recurrence relation for $f(n, k)$.

**Solution:**

(a) We have that $f(1, k) = 1$, $f(0, k) = 0$, $f(n, 1) = n$ since we must do a linear search to find $\ell$ if we only have one egg, and $f(n, 0) = \infty$ for $n > 0$.

(b) The recurrence relation is

$$f(n, k) = 1 + \min_{x \in \{1\ldots n\}} \max\{f(x - 1, k - 1), f(n - x, k)\}.$$ 

We can interpret the recurrence relation as minimizing the number of drops we need in the worst case. At each step, we use one drop and choose the best floor out of the $n$ floors (hence the $1 + \min_{x \in \{1\ldots n\}} \ldots$). To determine which floor is the best, we consider that either the egg breaks, or it doesn’t, and we choose the floor whose worst case is the best.

### 4 Greedy Cards

Ning and Evan are playing a game, where there are $n$ cards in a line. The cards are all face-up (so they can both see all cards in the line) and numbered 2–9. Ning and Evan take turns. Whoever’s turn it is can take one card from either the right end or the left end of the line. The goal for each player is to maximize the sum of the cards they’ve collected.

(a) Ning decides to use a greedy strategy: “on my turn, I will take the larger of the two cards available to me”. Show a small counterexample ($n \leq 5$) where Ning will lose if he plays this greedy strategy, assuming Ning goes first and Evan plays optimally, but he could have won if he had played optimally.

(b) Evan decides to use dynamic programming to find an algorithm to maximize his score, assuming he is playing against Ning and Ning is using the greedy strategy from part (a). Help Evan develop the dynamic programming solution by providing an algorithm with its runtime and space complexity.

**Solution:**
(a) One possible arrangement is: $[2, 2, 9, 3]$. Ning first greedily takes the 3 from the right end, and then Evan snatches the 9, so Evan gets 11 and Ning gets a miserly 5. If Ning had started by craftily taking the 2 from the left end, he’d guarantee that he would get 11 and poor Evan would be stuck with 5.

There are many other counterexamples. They’re all of length at least 4.

(b) Let $A[1..n]$ denote the $n$ cards in the line. Evan defines $v(i, j)$ to be the highest score he can achieve if it’s his turn and the line contains cards $A[i..j]$.

Evan suggests you simplify your expression by expressing $v(i, j)$ as a function of $\ell(i, j)$ and $r(i, j)$, where $\ell(i, j)$ is defined as the highest score Evan can achieve if it’s his turn and the line contains cards $A[i..j]$, if he takes $A[i]$; also, $r(i, j)$ is defined to be the highest score Evan can achieve if it’s his turn and the line contains cards $A[i..j]$, if he takes $A[j]$. Then, we have,

$v(i, j) = \max(\ell(i, j), r(i, j))$

where

\[ \ell(i, j) = \begin{cases} A[i] + v(i + 1, j - 1) & \text{if } A[j] > A[i + 1] \\ A[i] + v(i + 2, j) & \text{otherwise.} \end{cases} \]

\[ r(i, j) = \begin{cases} A[j] + v(i + 1, j - 1) & \text{if } A[i] \geq A[j - 1] \\ A[j] + v(i, j - 2) & \text{otherwise.} \end{cases} \]

(The formula above assumes that if there is a tie, Ning takes the card on the left end.)

There are $n(n + 1)/2$ subproblems and each one can be solved in $\Theta(1)$ time (that’s the time to evaluate the recursive formula in part (b) for a single value of $i, j$).