Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Planting Trees

This problem will guide you through the process of writing a dynamic programming algorithm.

You have a garden and want to plant some apple trees in your garden, so that they produce as many apples as possible. There are n adjacent spots numbered 1 to n in your garden where you can place a tree. Based on the quality of the soil in each spot, you know that if you plant a tree in the ith spot, it will produce exactly x_i apples. However, each tree needs space to grow, so if you place a tree in the ith spot, you can't place a tree in spots $i-1$ or $i+1$. What is the maximum number of apples you can produce in your garden?

(a) Give an example of an input for which:

- Starting from either the first or second spot and then picking every other spot (e.g. either planting the trees in spots $1, 3, 5\ldots$ or in spots $2, 4, 6\ldots$) does not produce an optimal solution.
- The following algorithm does not produce an optimal solution: While it is possible to plant another tree, plant a tree in the spot where we are allowed to plant a tree with the largest x_i value.

(b) To solve this problem, we'll thinking about solving the following, more general problem: "What is the maximum number of apples that can be produced using only spots 1 to i?". Let $f(i)$ denote the answer to this question for any i. Define $f(0) = 0$, as when we have no spots, we can't plant any trees. What is $f(1)$? What is $f(2)$?

(c) Suppose you know that the best way to plant trees using only spots 1 to i does not place a tree in spot i. In this case, express $f(i)$ in terms of x_i and $f(j)$ for $j < i$. (Hint: What spots are we left with? What is the best way to plant trees in these spots?)

(d) Suppose you know that the best way to plant trees using only spots 1 to i places a tree in spot i. In this case, express $f(i)$ in terms of x_i and $f(j)$ for $j < i$.

(e) Describe a linear-time algorithm to compute the maximum number of apples you can produce. (Hint: Compute $f(i)$ for every i. You should be able to combine your results from the previous two parts to perform each computation in $O(1)$ time).

Solution:

(a) For the first algorithm, a simple input where this fails is $[2, 1, 1, 2]$. Here, the best solution is to plant trees in spots 1 and 4. For the second algorithm, a simple input where this fails is $[2, 3, 2]$. Here, the greedy algorithm plants a tree in spot 2, but the best solution is to plant a tree in spots 1 and 3.

(b) $f(1) = x_1$, $f(2) = \max\{x_1, x_2\}$

(c) If we don't plant a tree in spot i, then the best way to plant trees in spots 1 to i is the same as the best way to plant trees in spots 1 to $i-1$. Then, $f(i) = f(i-1)$.

(d) If we plant a tree in spot i, then we get x_i apples from it. However, we cannot plant a tree in spot $i-1$, so we are only allowed to place trees in spots 1 to $i-2$. In turn, in this case we can pick the best way to plant trees in spots 1 to $i-2$ and then add a tree at i to this solution to get the best way to plant trees in spots 1 to i. So we get $f(i) = f(i-2) + x_i$.

(e) Initialize a length n array, where the ith entry of the array will store $f(i)$. Fill in $f(1)$, and then use the formula $f(i) = \max\{f(i-1), x_i + f(i-2)\}$ to fill out the rest of the table in order. Then, return $f(n)$ from the table.

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 $x =$ **efficient** and $y =$ **ALGORITHM**, then it is true for $z =$ **effALGiORciIenTHMt**, but false for $z =$ **efficientALGORITHMS** (extra characters), $z =$ **effALGORITHMicien** (missing the final t), and $z =$ **effOALGRicieITHMnt** (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 $x =$ **aaaaaaaaaab** and $y =$ **aaaaaaaac**.

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 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:

   $$S(i, j) = (S(i - 1, j) \wedge (x_i = z_{i+j})) \vee (S(i, j - 1) \wedge (y_j = z_{i+j}))$$

   The base case is $S(0, 0) = T$; we also set $\forall i \in [0, |x|], S(i, -1) = F$ and $\forall j \in [0, |y|] = S(-1, j) = F$. 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 infomration 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 infomration 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 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).

4 Horn Formulas

Describe an algorithm which finds a satisfying assignment for a Horn formula, or reports that none exists, in time linear in the size of the formula. Prove that your algorithm is correct and runs in linear time.

Solution:

Main Idea

We first build a graph of the implications of the Horn formula. We have a node for each implication, and a node for each variable. If a variable x appears on the LHS of the implication I, then there is a

directed edge $(x, I) \in E$. If x appears on the RHS of I, there is a directed edge (I, x). When we set a variable x to true, we delete it from the graph. If this causes an implication I to have indegree 0, we delete it and its consequent from the graph. Once there are no implications with indegree 0, we check whether the negative clauses are satisfied by setting the remaining variables to false. Initially we have to search the graph for implications with indegree 0 and add them to a queue. Then we add every implication whose indegree is newly 0 after deleting some vertex to that queue.

Proof of Correctness The truth assignment corresponding to a graph G is given by setting every variable to false if the corresponding vertex is in G, and true otherwise. When the algorithm terminates, every implication in G has nonzero indegree. This means that for every implication in G, at least one of its antecedents is false, and so the consequent can be set to false. We only delete a vertex corresponding to a variable if it *must* be set to true. The Horn formula is then satisifiable if and only if the assignment corresponding to G satisfies the negative clauses.

Running Time If we use the appropriate data structure for the graph, building it takes time linear in the size of the formula. Every step deletes some vertex from the graph, and so we can only do linearly many. The total cost of deleting all the vertices and edges is linear in the size of the graph, and so the total running time is linear.

5 Longest Huffman Tree

Under a Huffman encoding of n symbols with frequencies $f_1, f_2, \ldots, f_n$, what is the longest a codeword could possibly be? Give an example set of frequencies that would produce this case, and argue that it is the longest possible.

Solution: The longest codeword can be of length $n - 1$. An encoding of n symbols with $n - 2$ of them having probabilities $1/2, 1/4, \ldots, 1/2^{n-2}$ and two of them having probability $1/2^{n-1}$ achieves this value. No codeword can ever by longer than length $n-1$. To see why, we consider a prefix tree of the code. If a codeword has length n or greater, then the prefix tree would have height n or greater, so it would have at least $n+1$ leaves. Our alphabet is of size n, so the prefix tree has exactly n leaves.