*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 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 $i$th spot, it will produce exactly $x_i$ apples. However, each tree needs space to grow, so if you place a tree in the $i$th 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 $i$th 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 Complementary Connected Components

Given a graph $G$ on vertex set $\{1, \ldots, n\}$ and $m$ edges, give an $O((m+n)\log n)$ time algorithm to output the number of connected components in the *complement* of $G$.

**Solution:**

The complement of a graph edges + the edges of original graph = the edges of a complete graph.

Let us take a step back and look at a vertex which is connected to all other vertices in G. We know that this specific vertex v has to be in its own connected component in the complement because all edges that could connect that vertex to something else would be in G. If another vertex z was not connected to v in the original graph, then that z would be in the same connected component as v in the complement.

Trying to generalize this, let us assume we know a connected component of a subpart of the graph lets say $x$. Now what we are trying to figure out is whether the vertex $v$ is connected to $x$. This ends up being if we could count all the edges connecting v to $x$ and it ends up being $|x|$, the number of vertices in the component containing $x$, we know that there cant be a direct connection from $x$ to v (ie it might be connected through some other vertex and then connected to x but theres no edge which has v as an endpoint and also another endpoint in x).

We have kind of seen something like this before with Bellman Ford, where we know the shortest path of length k, and we are trying to find the shortest path of length k + 1. So we need to figure out what to iterate on, the paragraph above suggests that we need to iterate on vertices specifically in order, and check connected components of the smaller section.

We will maintain 2 disjoint set data structures, H1 and H2. Our data structure will be augmented with two counters: one for the size of the set (which represents connected components), and the other for the number of edges in iteration i that connect the set to vertex i in the original graph. H1 contains the connected components of the previous iteration i - 1 and H2 contains the connected components of the current iteration i that cant be connected to vertex i. At the end of the iteration we say H1 is assigned to H2, and H2 becomes empty.

Every iteration we iterate through all edges connecting i to vertices less than i which must be in some set in H1. If there is an edge, we increment the counter for that set by one. If we ever get to the case that the counter of the edges from i to everything in the set is equal to the number of vertices in that set then we know that vertex i cant be directly connected to that set, so we remove the set from H1 and move it to H2. At the end of the iteration, if there are sets left in the H1, we know there must be some missing edge in G that doesnt connect H1 sets to vertex i, and so we union them all with vertex i and add this connected component to H2.

Via induction, you get all the connected components.

# 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 $i, j$).