

1. (a) Describe an algorithm that given two arrays $A[1..m]$ and $B[1..n]$ of integers computes the length of the longest common subsequence of A and B that is *decreasing*.

Solution: Add a sentinel value $A[0] = \infty$. For any $i, j \in \{0, \dots, m\}$ and $k \in \{0, \dots, n\}$, let $LCDS(i, j, k)$ be the length of the longest common decreasing subsequence of $A[j..m]$ and $B[k..n]$ where every element is strictly less than $A[i]$. We need to compute $LCDS(0, 1, 1)$. The function obeys the following recurrence, some of whose cases may overlap:

$$LCDS(i, j, k) = \begin{cases} 0 & \text{if } j > m \text{ or } k > n \\ LCDS(i, j+1, k) & \text{if } j \leq m \text{ and } A[j] \geq A[i] \\ LCDS(i, j, k+1) & \text{if } k \leq n \text{ and } B[k] \geq A[i] \\ \max \begin{cases} LCDS(i, j+1, k), \\ LCDS(i, j, k+1) \end{cases} & \text{if } j \leq m, k \leq n, \text{ and } A[j] \neq B[k] \\ \max \begin{cases} LCDS(i, j+1, k), \\ LCDS(i, j, k+1), \\ 1 + LCDS(j, j+1, k+1) \end{cases} & \text{otherwise} \end{cases}$$

We can memoize this function into a 3D array $LCDS[0..m, 1..m, 1..n]$. We can fill this array in constant time per entry using a triply nested loop which goes over decreasing j , k , and i as the outermost to innermost loops. **The resulting algorithm runs in $O(n^3)$ time.** ■

Rubric: 5 points: standard dynamic programming rubric. A correct $O(n^4)$ time algorithm is worth 4 points.

Solution: To speed up the previous solution, we make the following observation: if the longest common decreasing subsequence of $A[j..m]$ and $B[k..n]$ takes $B[k]$ as its first element from B , its first element from A should be at the first index in $A[j..m]$ that has the same value as $B[k]$. Since this is a greedy optimization, we will have to formally prove this.

Formally, for each $1 \leq j \leq m$ and $1 \leq k \leq n$, let $\text{Next}(j, k)$ denote the smallest value of ℓ such that $\ell \geq j$ and $A[\ell] = B[k]$, or $\perp$ if no such ℓ exists. We claim that for any common decreasing subsequence of $A[j..m]$ and $B[k..n]$ that contains $B[k]$, there exists an equal-length common decreasing subsequence that also contains $A[\text{Next}(j, k)]$.^a

Indeed, by the definition of a common decreasing subsequence, we know that the first element from A such a subsequence takes must have value equal to $B[k]$. Letting ℓ be the index of this element, we know from the definition of Next that $\ell \geq \text{Next}(j, k)$. If we swap the first element of our common decreasing subsequence from $A[\ell]$ to $A[\text{Next}(j, k)]$, we still have a subsequence, because all other elements have index greater than ℓ and therefore greater than $\text{Next}(j, k)$, it's still decreasing, because the *values* of the subsequence did not change, and it's the same length, because we swapped exactly one element for another.

With this optimization, we can now collapse the roles of the indices i and j from the previous solution into a single index. Formally, let $\text{LCDS}(j, k)$ be the length of the longest common decreasing subsequence of $A[j..m]$ and $B[k..n]$ where every element is strictly less than $A[j-1]$, where as before we add a sentinel value $A[0] = \infty$. We need to compute $\text{LCDS}(1, 1)$. The function obeys the following recurrence:

$$\text{LCDS}(j, k) = \begin{cases} 0 & \text{if } j > m \text{ or } k > n \\ \text{LCDS}(j, k+1) & \text{if } \text{Next}(j, k) = \perp \\ & \text{or } B[k] \geq A[j-1] \\ \max \left\{ \begin{array}{l} \text{LCDS}(\text{Next}(j, k) + 1, k + 1), \\ \text{LCDS}(j, k + 1) \end{array} \right\} & \text{otherwise} \end{cases}$$

where in the final case we used our greedy optimization.

However, we aren't quite done. In particular, if we compute $\text{Next}(j, k)$ on the fly, we may spend $O(n)$ time per subproblem, and so won't improve over the $O(n^3)$ runtime of the previous (much simpler) solution! To fix this, we can precompute the values of $\text{Next}(j, k)$ using the recurrence

$$\text{Next}(j, k) = \begin{cases} \perp & \text{if } j > m \\ j & \text{if } A[j] = B[k] \\ \text{Next}(j+1, k) & \text{otherwise} \end{cases}$$

We can memoize our two functions into two 2D arrays $\text{LCDS}[1..m, 1..n]$ and $\text{Next}[1..m+1, 1..n]$. We can fill Next in constant time per entry using a doubly nested loop which goes over j in decreasing order in the outer loop and k in

arbitrary order in the inner loop. Once we have done this, we can fill in *LCDS* in constant time per entry using a doubly nested loop which goes over k in decreasing order in the outer loop and j in arbitrary order in the inner loop. **The resulting algorithm runs in $O(n^2)$ time.** ■

^aIn particular, this means that if a longest common decreasing subsequence contains $B[k]$, there is a longest common decreasing subsequence that contains $B[k]$ and $A[\text{Next}(j, k)]$.

Rubric: 7 points: standard dynamic programming rubric. *Proving that the greedy optimization is correct is necessary.*

- (b) Describe an algorithm that given two arrays $A[1..m]$ and $B[1..n]$ of characters computes the length of the longest common subsequence of A and B that is a *palindrome*.

Solution: For any $i \in \{1, \dots, m+1\}$, $j \in \{0, \dots, m\}$, $k \in \{1, \dots, n+1\}$ and $\ell \in \{0, \dots, n\}$, let $LCPS(i, j, k, \ell)$ be the length of the longest common palindromic subsequence of $A[i..j]$ and $B[k..\ell]$. We need to compute $LCPS(1, m, 1, n)$. The function obeys the following recurrence, some of whose cases may overlap:

$$LCPS(i, j, k) = \begin{cases} 0 & \text{if } i > j \text{ or } k > \ell \\ 1 & \text{if } i = j, k = \ell, \text{ and } A[i] = B[k] \\ \max \begin{cases} LCPS(i+1, j, k, \ell), \\ LCPS(i, j-1, k, \ell) \end{cases} & \text{if } i \leq j, k \leq \ell, \text{ and } A[i] \neq A[j] \\ \max \begin{cases} LCPS(i, j, k+1, \ell), \\ LCPS(i, j, k, \ell-1) \end{cases} & \text{if } i \leq j, k \leq \ell, \text{ and } B[k] \neq B[\ell] \\ \max \begin{cases} LCPS(i+1, j, k, \ell), \\ LCPS(i, j, k+1, \ell) \end{cases} & \text{if } i \leq j, k \leq \ell, \text{ and } A[i] \neq B[k] \\ \max \begin{cases} LCPS(i, j, k+1, \ell), \\ LCPS(i, j, k, \ell-1), \\ LCPS(i+1, j, k, \ell), \\ LCPS(i, j-1, k, \ell), \\ 2 + LCPS(i+1, j-1, k+1, \ell-1) \end{cases} & \text{otherwise} \end{cases}$$

We can memoize this function into a 4D array $LCPS[1..m+1, 0..m, 1..n+1, 0..n]$. We can fill this array in constant time per entry using a quadruply nested loop which goes over decreasing i , increasing j , decreasing k , and increasing ℓ as the outermost to innermost loops. **The resulting algorithm runs in $O(n^4)$ time.** ■

Rubric: 5 points: standard dynamic programming rubric.

2. Describe an algorithm that given a set S of n (possibly overlapping) axis-aligned rectangles in 2D computes the maximum size of any orthogonally separable subset of S .

Solution: Observe that in the definition of an orthogonally separable set, we may assume the separating line is placed just to the left or just below a rectangle. We're going to define recursive subproblems based on subsets of S lying left, right, above, and below given lower left corners of rectangles in S . Below is one way we can do so.

Let $\sigma : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ be the permutation describing how to sort the rectangles in S by their lower left corners from left to right. Specifically, $A[\sigma(i)] < A[\sigma(j)]$ for all $i, j \in \{1, \dots, n\}$. Similarly, let $\tau : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ be the permutation describing how to sort the rectangles in S by their lower left corners from low to high. Both permutations can be computed in $O(n \log n)$ time by doing a mergesort.

For all $h, i, j \in \{1, \dots, n\}$ and $k, \ell \in \{1, \dots, n+1\}$, let $InBox(h, i, j, k, \ell)$ be TRUE if and only if all of the following are TRUE:

- $A[h] \geq A[\sigma(i)]$
- $B[h] \geq B[\tau(i)]$
- $k < n+1$ implies $C[h] < A[\sigma(k)]$
- $\ell < n+1$ implies $D[h] < B[\tau(\ell)]$

Also, for all $i, j \in \{1, \dots, n\}$ and $k, \ell \in \{1, \dots, n+1\}$, let $OSS(i, j, k, \ell)$ be the maximum size of any orthogonally separable subset X of S such that if the h th rectangle of S is in X , then $InBox(h, i, j, k, \ell)$ is True. We need to compute $OSS(1, 1, n+1, n+1)$. The function obeys the following recurrence:

$$OSS(i, j, k, \ell) = \begin{cases} 0 & \text{if } |\{h \mid InBox(h, i, j, k, \ell)\}| = 0 \\ 1 & \text{if } |\{h \mid InBox(h, i, j, k, \ell)\}| = 1 \\ \max \left\{ \begin{array}{l} \max_{i < h < k} OSS(i, j, h, \ell) + OSS(h, j, k, \ell), \\ \max_{j < h < \ell} OSS(i, j, k, h) + OSS(i, h, k, \ell) \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into an array $OSS[1..n, 1..n, 1..n+1, 1..n+1]$. We can fill this array in $O(n)$ time per entry by decreasing i , decreasing j , increasing k , increasing ℓ , and increasing h (to check conditions or do the max) as the outermost to innermost loop. **The resulting algorithm runs in $O(n^5)$ time.** ■

Rubric: 10 points: standard dynamic programming rubric.

3. Practice only. Do not submit solutions.

Suppose we need to broadcast a message to all the nodes in a rooted binary tree. Initially, only the root node knows the message. In a single round, any node that knows the message can forward it to at most one of its children. See the figure below for an example.

Design an algorithm to compute the minimum number of rounds required to broadcast the message to every node.

Solution: Let $MinR(v)$ denote the minimum number of rounds need to spread a message from v to all its descendants. We need to compute $MinR(root)$. This function obeys the following recurrence:

$$MinR(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ 1 + MinR(w) & \text{if } w \text{ is only child of } v \\ \min \left\{ \begin{array}{l} \max \{1 + MinR(v.left), 2 + MinR(v.right)\} \\ \max \{2 + MinR(v.left), 1 + MinR(v.right)\} \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into the tree itself. Since the value at every node depends only on the values at its children, we can evaluate this function at all nodes by a postorder traversal of the tree. The resulting algorithm runs in $O(n)$ time. ■

Rubric: Standard dynamic programming rubric.