

Describe and analyze **dynamic programming** algorithms for the following longest-subsequence problems. Use the recursive backtracking algorithms you developed on Wednesday.

1. Given an array $A[1..n]$ of integers, compute the length of a longest **decreasing** subsequence of A .

Solution (two parameters, design): Add a sentinel value $A[0] = \infty$. Let $LDS(i, j)$ denote the length of the longest decreasing subsequence of $A[j..n]$ where every element is larger than $A[i]$. This function obeys the following recurrence:

$$LDS(i, j) = \begin{cases} 0 & \text{if } j > n \\ LDS(i, j+1) & \text{if } j \leq n \text{ and } A[i] \leq A[j] \\ \max\{LDS(i, j+1), 1 + LDS(j, j+1)\} & \text{otherwise} \end{cases}$$

We need to compute $LDS(0, 1)$.

We can memoize the function LDS into an array $LDS[0..n, 1..n+1]$. Each entry $LDS[i, j]$ depends only on entries in the next column $LDS[:, j+1]$, so we can fill the array in reverse column-major order, scanning right to left (decreasing j) in the outer loop, and bottom to top (decreasing i) in the inner loop.

The resulting algorithm runs in $O(n^2)$ time. ■

Solution (two parameters, pseudocode): The following algorithm runs in $O(n^2)$ time:

```

LDS( $A[1..n]$ ):
   $A[0] \leftarrow \infty$            «Add a sentinel»
  for  $i \leftarrow 0$  to  $n$        «Base cases»
     $LDS[i, n+1] \leftarrow 0$ 
  for  $j \leftarrow n$  down to 1
    for  $i \leftarrow j-1$  down to 0
      if  $A[i] \leq A[j]$ 
         $LDS[i, j] \leftarrow LDS[i, j+1]$ 
      else
         $LDS[i, j] \leftarrow \max\{LDS[i, j+1], 1 + LDS[j, j+1]\}$ 
  return  $LDS[0, 1]$ 

```

Here, $LDS[i, j]$ stores the length of the longest decreasing subsequence of $A[j..n]$ where every element is strictly less than $A[i]$. ■

Solution (one parameter, design): Add a sentinel value $A[0] = \infty$. Let $LDS(i)$ denote the length of the longest decreasing subsequence of $A[i..n]$ that begins with $A[i]$. This function obeys the following recurrence:

$$LDS(i) = 1 + \max \{LDS(j) \mid j > i \text{ and } A[j] < A[i]\}$$

(Here we define $\max \emptyset = 0$ so that the base cases are correct.) We need to compute $LDS(0) - 1$.

We can memoize the function LDS into a one-dimensional array, which we can fill in reverse order. For each index i , computing $LDS[i]$ requires a for-loop over all $j > i$, so the overall algorithm runs in $O(n^2)$ time. ■

Solution (one parameter, pseudocode): The following algorithm runs in $O(n^2)$ time.

```

LDS( $A[1..n]$ ):
   $A[0] = \infty$                                 ⟨⟨Add a sentinel⟩⟩
  for  $i \leftarrow n$  down to 0
     $LDS[i] \leftarrow 1$ 
    for  $j \leftarrow i + 1$  to  $n$ 
      if  $A[j] < A[i]$  and  $1 + LDS[j] > LDS[i]$ 
         $LDS[i] \leftarrow 1 + LDS[j]$ 
  return  $LDS[0] - 1$                             ⟨⟨Don't count the sentinel⟩⟩

```

Here, $LDS[i]$ stores the length of the longest decreasing subsequence of $A[i..n]$ that begins with $A[i]$. ■

Solution (clever): The following algorithm runs in $O(n^2)$ time.

```
LDS( $A[1..n]$ ):  
  for  $i \leftarrow 1$  to  $n$   
     $Z[i] \leftarrow -A[i]$   
  return LIS( $Z$ )
```

Here, LIS is the longest-*increasing*-subsequence algorithm described in Erickson's Algorithms Chapter 3. ■

Solution (reverse): The following algorithm runs in $O(n^2)$ time.

```
LDS( $A[1..n]$ ):  
  for  $i \leftarrow 1$  to  $n$   
     $Z[n + 1 - i] \leftarrow A[i]$   
  return LIS( $Z$ )
```

Here LIS is the longest-*increasing*-subsequence algorithm described in Erickson's Algorithms Chapter 3. ■

2. Given an array $A[1..n]$ of integers, compute the length of a longest **alternating** subsequence of A .

Solution (design): We define two functions:

- Let $LAS^+(i, j)$ denote the length of the longest alternating subsequence of $A[j..n]$ whose first element (if any) is larger than $A[i]$ and whose second element (if any) is smaller than its first.
- Let $LAS^-(i, j)$ denote the length of the longest alternating subsequence of $A[j..n]$ whose first element (if any) is smaller than $A[i]$ and whose second element (if any) is larger than its first.

These two functions satisfy the following mutual recurrences:

$$LAS^+(i, j) = \begin{cases} 0 & \text{if } j > n \\ LAS^+(i, j+1) & \text{if } j \leq n \text{ and } A[j] \leq A[i] \\ \max\{LAS^+(i, j+1), 1 + LAS^-(j, j+1)\} & \text{otherwise} \end{cases}$$

$$LAS^-(i, j) = \begin{cases} 0 & \text{if } j > n \\ LAS^-(i, j+1) & \text{if } j \leq n \text{ and } A[j] \geq A[i] \\ \max\{LAS^-(i, j+1), 1 + LAS^+(j, j+1)\} & \text{otherwise} \end{cases}$$

The length of the longest alternating subsequence is

$$\max_j \max\{1 + LAS^+(j, j+1), 1 + LAS^-(j, j+1)\}.$$

Here j is the index of the first entry in the longest alternating subsequence.

We can memoize these functions into two-dimensional arrays $LAS^+[0..n, 1..n+1]$ and $LAS^-[0..n, 1..n+1]$. Each entry $LAS^\pm[i, j]$ depends only on entries in the next column of either the same array or the other array. So we can fill both arrays in parallel, scanning right to left (decreasing j) in the outer loop, and bottom to top (decreasing i) in the inner loop. The resulting algorithm runs in $O(n^2)$ time. ■

Solution (pseudocode): The following algorithm runs in $O(n^2)$ time.

```

LAS( $A[1..n]$ ):
  for  $i \leftarrow 0$  to  $n$                                  $\langle\langle$ Base cases $\rangle\rangle$ 
     $LAS^+[i, n+1] \leftarrow 0$ 
     $LAS^-[i, n+1] \leftarrow 0$ 
  for  $j \leftarrow n$  down to 1
    for  $i \leftarrow j-1$  down to 1
       $LAS^+[i, j] \leftarrow LAS^+[i, j+1]$ 
       $LAS^-[i, j] \leftarrow LAS^-[i, j+1]$ 
      if  $A[i] < A[j]$ 
         $LAS^+[i, j] \leftarrow \max\{LAS^+[i, j], 1 + LAS^-[j, j+1]\}$ 
      if  $A[i] > A[j]$ 
         $LAS^-[i, j] \leftarrow \max\{LAS^-[i, j], 1 + LAS^+[j, j+1]\}$ 
   $\ell \leftarrow 0$ 
  for  $j \leftarrow 1$  to  $n$ 
     $\ell \leftarrow \max\{\ell, 1 + LAS^+[j, j+1], 1 + LAS^-[j, j+1]\}$ 
  return  $\ell$ 

```

The algorithm maintains two arrays:

- $LAS^+[i, j]$ stores the length of the longest alternating subsequence of $A[j..n]$ whose first element (if any) is larger than $A[i]$ and whose second element (if any) is smaller than its first.
- $LAS^-[i, j]$ stores the length of the longest alternating subsequence of $A[j..n]$ whose first element (if any) is smaller than $A[i]$ and whose second element (if any) is larger than its first.

■

Solution (greedy): The following greedy algorithm computes the length of the longest alternating subsequence in $O(n)$ time.

```

GREEDYLAS( $A[1..n]$ ):
    «Elide runs of the same element; unnecessary if elements are distinct»
     $m \leftarrow 1$ 
     $B[1] \leftarrow A[1]$ 
    for  $i \leftarrow 2$  to  $n$ 
        if  $A[i] \neq B[m]$ 
             $m \leftarrow m + 1$ 
             $B[m] \leftarrow A[i]$ 

    «Count local minima and maxima»
     $\ell \leftarrow 2$ 
    for  $i \leftarrow 2$  to  $m - 1$ 
        if  $B[i] < \min\{B[i-1], B[i+1]\}$  or  $B[i] > \max\{B[i-1], B[i+1]\}$ 
             $\ell \leftarrow \ell + 1$ 
    return  $\ell$ 

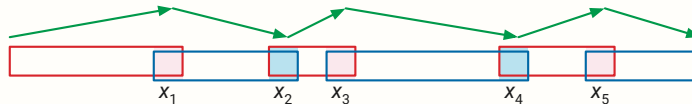
```

We need to prove that this greedy algorithm is correct, because we *always* need to prove that greedy algorithms are correct.^a Assume without loss of generality that $A[i] \neq A[i+1]$ for all i ; any alternating subsequence contains at most one element from any run of equal values.

Let $1 = x_1 < x_2 < x_3 < \dots < x_\ell = n$ be the indices of all local minima and local maxima of A ; these are the elements counted in the final for-loop of GREEDYLAS. The following claim immediately implies that no alternating subsequence of A has length greater than ℓ .

Claim 1. *For any alternating subsequence S of A , there is an alternating subsequence of A with the same length as S , in which every element is either a local minimum of A or a local maximum of A .*

Proof: The local extrema $A[x_j]$ divide A into $\ell - 1$ contiguous blocks $A_j = A[x_{j-1}..x_j]$ that overlap at their endpoints and that alternate between increasing and decreasing.



Let S be an arbitrary subsequence of A . For each index j from 1 to ℓ , we modify S as follows to obtain a new alternating subsequence with the same length as S . Assume without loss of generality that $A[x_{j-1}] < A[x_j]$; the other case is symmetric.

- If S contains no elements of block A_j , there is nothing to do.
- Suppose S contains exactly one element of A_j . If that element is a local maximum of S , replace it with $A[x_j]$. Similarly, if that element is a local minimum of S , replace it with $A[x_{j-1}]$.

- Suppose S contains exactly two elements of A_j ; the first must be a local minimum of S and the second must be a local maximum of S . Replace those two elements with $A[x_{j-1}]$ and $A[x_j]$.
- S cannot contain more than two elements of A_j , because S is alternating.

Only local maxima of S are moved to local maxima of A . Because S cannot have two local maxima in a row, at most one element of S is moved to any local maximum of A , and similarly for local minima. Thus, after performing this modification inside every block, S is still an alternating subsequence of A , the length of S has not changed, and S contains only local extrema of A . □

■

^a**Yes, even on exams!** Submitting a greedy algorithm *without* a proof of correctness is a Deadly Sin, even worse than weak induction or not describing problems in English. Greedy algorithms without proofs will be penalized, **even if** they are perfectly correct. Premature optimization is the root of all evil!

3. Given an array $A[1..n]$ of integers, compute the length of a longest **convex** subsequence of A .

Solution (design): Let $LCS(i, j)$ denote the length of the longest convex subsequence of $A[i..n]$ whose first two elements are $A[i]$ and $A[j]$. This function obeys the following recurrence:

$$LCS(i, j) = 1 + \max \{LCS(j, k) \mid j < k \leq n \text{ and } A[i] + A[k] > 2A[j]\}$$

Here we define $\max \emptyset = 0$; this gives us a working base case. The length of the longest convex subsequence is $\max_{1 \leq i < j \leq n} LCS(i, j)$.

We can memoize the function LCS into a two-dimensional array, which we can fill in reverse row-major order in $O(n^3)$ time. ■

Solution (pseudocode): The following algorithm runs in $O(n^3)$ time:

```

LCS( $A[1..n]$ ):
   $\ell \leftarrow 0$ 
  for  $i \leftarrow n - 1$  down to 1
    for  $j \leftarrow n$  down to  $i + 1$ 
       $LCS[i, j] \leftarrow 1$ 
      for  $k \leftarrow j + 1$  to  $n$ 
        if  $A[i] + A[k] > 2A[j]$ 
           $LCS[i, j] \leftarrow \max \{LCS[i, j], 1 + LCS[j, k]\}$ 
       $\ell \leftarrow \max \{\ell, LCS[i, j]\}$ 
  return  $\ell$ 

```

Here $LCS[i, j]$ stores the length of the longest convex subsequence of $A[i..n]$ whose first two elements are $A[i]$ and $A[j]$. ■

4. Given an array $A[1..n]$, compute the length of a longest *palindrome* subsequence of A .

Solution (design): Let $LPS(i, j)$ denote the length of the longest palindrome subsequence of $A[i..j]$. This function obeys the following recurrence:

$$LPS(i, j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ \max \begin{Bmatrix} LPS(i+1, j) \\ LPS(i, j-1) \end{Bmatrix} & \text{if } i < j \text{ and } A[i] \neq A[j] \\ \max \begin{Bmatrix} 2 + LPS(i+1, j-1) \\ LPS(i+1, j) \\ LPS(i, j-1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

We need to compute $LPS(1, n)$.

We can memoize the function LPS into a two-dimensional array. Each entry $LPS[i, j]$ depends on (at most) three entries $LPS[i+1, j]$, $LPS[i, j-1]$, and $LPS[i+1, j-1]$ immediately below and/or to the left. Thus, we can fill the array from bottom to top (decreasing i) in the outer loop, and from left to right (increasing j) in the inner loop, in $O(n^2)$ time. ■

Solution (pseudocode): The following algorithm runs in $O(n^2)$ time.

```

LPS( $A[1..n]$ ):
  for  $i \leftarrow n$  down to 1
     $LPS[i, i-1] \leftarrow 0$ 
     $LPS[i, i] \leftarrow 1$ 
    for  $j \leftarrow i+1$  to  $n$ 
       $LPS[i, j] \leftarrow \max \{LPS[i+1, j], LPS[i, j-1]\}$ 
      if  $A[i] = A[j]$ 
         $LPS[i, j] \leftarrow \max \{LPS[i, j], 2 + LPS[i+1, j-1]\}$ 
  return  $LPS[1, n]$ 

```

Here $LPS[i, j]$ stores the length of the longest palindrome subsequence of the substring $A[i..j]$. ■

Solution (greedy optimization): Let $LPS(i, j)$ denote the length of the longest palindrome subsequence of $A[i..j]$. This function obeys the following recurrence:

$$LPS(i, j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ 2 + LPS(i + 1, j - 1) & \text{if } i < j \text{ and } A[i] = A[j] \\ \max\{LPS(i + 1, j), LPS(i, j - 1)\} & \text{otherwise} \end{cases}$$

See the Lab 7a solutions for a proof that the case in red is correct. We need to compute $LPS(1, n)$.

We can memoize the function LPS into a two-dimensional array. Each entry $LPS[i, j]$ depends on (at most) three entries $LPS[i + 1, j]$, $LPS[i, j - 1]$, and $LPS[i + 1, j - 1]$ immediately below and/or to the left. Thus, we can fill the array from bottom to top (decreasing i) in the outer loop, and from left to right (increasing j) in the inner loop, in $O(n^2)$ time. The greedy optimization didn't actually speed up the algorithm, so we shouldn't have bothered. ■

Solution (greedy optimization, pseudocode): The following algorithm runs in $O(n^2)$ time.

```

LPS( $A[1..n]$ ):
  for  $i \leftarrow n$  down to 1
     $LPS[i, i - 1] \leftarrow 0$ 
     $LPS[i, i] \leftarrow 1$ 
    for  $j \leftarrow i + 1$  to  $n$ 
      if  $A[i] = A[j]$ 
         $LPS[i, j] \leftarrow 2 + LPS[i + 1, j - 1]$ 
      else
         $LPS[i, j] \leftarrow \max\{LPS[i + 1, j], LPS[i, j - 1]\}$ 
  return  $LPS[1, n]$ 

```

Here $LPS[i, j]$ stores the length of the longest palindrome subsequence of the substring $A[i..j]$. We incorporated a greedy optimization in the 'if' case of the inner loop—see the solutions for Lab 7a for a proof of correctness—but it didn't actually make the algorithm faster, so we shouldn't have bothered. ■

5. Given an array $A[1..n]$, compute the length of a *shortest* palindrome *supersequence* of A .

Solution (design): Let $SPS(i, j)$ denote the length of the shortest palindrome supersequence of $A[i..j]$. This function obeys the following recurrence:

$$SPS(i, j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ \min \begin{Bmatrix} 2 + SPS(i+1, j) \\ 2 + SPS(i, j-1) \end{Bmatrix} & \text{if } i < j \text{ and } A[i] \neq A[j] \\ \min \begin{Bmatrix} 2 + SPS(i+1, j-1) \\ 2 + SPS(i+1, j) \\ 2 + SPS(i, j-1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

We need to compute $SPS(1, n)$.

We can memoize the function SPS into a two-dimensional array. Each entry $SPS[i, j]$ depends on (at most) three entries $SPS[i+1, j]$, $SPS[i, j-1]$, and $SPS[i+1, j-1]$ immediately below and/or to the left. Thus, we can fill the array from bottom to top (decreasing i) in the outer loop, and from left to right (increasing j) in the inner loop, in $O(n^2)$ time. ■

Solution (pseudocode): The following algorithm runs in $O(n^2)$ time.

```
SPS( $A[1..n]$ ):
  for  $i \leftarrow n$  down to 1
     $SPS[i, i-1] \leftarrow 0$ 
     $SPS[i, i] \leftarrow 1$ 
    for  $j \leftarrow i+1$  to  $n$ 
       $SPS[i, j] \leftarrow \min \{2 + SPS[i+1, j], 2 + SPS[i, j-1]\}$ 
      if  $A[i] = A[j]$ 
         $SPS[i, j] \leftarrow \min \{SPS[i, j], 2 + SPS[i+1, j-1]\}$ 
  return  $SPS[1, n]$ 
```

Here $SPS[i, j]$ stores the length of the shortest palindrome supersequence of the substring $A[i..j]$. ■