Describe and analyze iterative ***dynamic programming*** algorithms to evaluate the following recurrences from Wednesday's lab. Each recurrence refers to a fixed array $A[1..n]$ of integers.

Warning: Any dynamic-programming solution to a self-contained algorithm-design problem must include an English description of the underlying recursive function(s), even if the solution is presented using iterative pseudocode. (See the solved problems in Homeworks 6 and 7.) But these problems aren't self-contained; we're giving you the recurrences, without telling you what the functions represent.

---

1. Add a sentinel value $A[0] = -\infty$, and then compute $LIS(0,1)$, where

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

Solution (design): We can memoize the function LIS into an array $LIS[0..n, 1..n+1]$.

Each entry $LIS[i,j]$ depends only on entries in the next column $LIS[\cdot, j+1]$.

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

```
LIS(A[1..n]):
  A[0] ← −∞                          ⟨⟨Add a sentinel⟩⟩
  for i ← 0 to n                     ⟨⟨Base cases⟩⟩
      LIS[i, n+1] ← 0

  for j ← n down to 1
      for i ← j − 1 down to 0
          if A[i] ≥ A[j]
              LIS[i, j] ← LIS[i, j+1]
          else
              LIS[i, j] ← max{LIS[i, j+1], 1 + LIS[j, j+1]}
  return LIS[0, 1]
```

■

2. Add a sentinel value $A[n+1] = \infty$, and then compute $LIS(n, n+1)$, where

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

Solution (design): We can memoize the function LIS into an array $LIS[0..n, 1..n+1]$.

Each entry $LIS[i,j]$ depends only on entries in the previous row $LIS[i-1,\cdot]$.

So we can fill the array in standard row-major order, scanning top to bottom (increasing i) ion the outer loop and left to right (increasing j) in the inner loop.

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

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

```
LIS(A[1..n]):
  A[n+1] ← ∞                    ⟨⟨Add a sentinel⟩⟩
  for j ← 1 to n+1              ⟨⟨Base cases⟩⟩
      LIS[0, j] ← 0

  for i ← 1 to n
      for j ← i+1 to n+1
          if A[i] ≥ A[j]
              LIS[i, j] ← LIS[i−1, j]
          else
              LIS[i, j] ← max{LIS[i−1, j], 1 + LIS[i−1, i]}
  return LIS[n, n+1]
```

■

3. Compute $\max_i LIS(i)$, where

$$LIS(i) = \begin{cases} 1 & \text{if } A[j] \le A[i] \text{ for all } j > i \\ 1 + \max\left\{ LIS(j) \;\middle|\; j > i \text{ and } A[j] > A[i] \right\} & \text{otherwise} \end{cases}$$

Solution (design): We can memoize the function *LIS* into a one-dimensional array, which we can fill in reverse index order. For each index i, we compute $LIS[i]$ using a for-loop over all $j > i$, so filling the array the array takes $O(n^2)$ time. Then we find the maximum entry in the *LIS* array in $O(n)$ additional time. The overall algorithm runs in $\boldsymbol{O(n^2)}$ ***time***. ■

Solution (pseudocode): The following algorithm runs in $\boldsymbol{O(n^2)}$ ***time***.

```
LIS(A[1..n]):
  maxLIS ← ∞
  for i ← n down to 0
      LIS[i] ← 1
      for j ← i + 1 to n
          if A[j] > A[i] and LIS[i] < 1 + LIS[j]
              LIS[i] ← 1 + LIS[j]
      if maxLIS < LIS[i]
          maxLIS ← LIS[i]
  return maxLIS
```

■

4. Add a sentinel value $A[0] = \infty$, and then compute $LDS(0, 1)$, where

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

Solution (design): We can memoize LDS into a 2D array $LIS[0..n, 1..n+1]$. Each entry $LDS[i, j]$ depends only on entries in the next column $LDS[\cdot, j+1]$, 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, in $\boldsymbol{O(n^2)}$ ***time***. ■

Solution (pseudocode): The following algorithm runs in $\boldsymbol{O(n^2)}$ ***time***:

```
LDS(A[1..n]):
  A[0] ← −∞                          ⟨⟨Add a sentinel⟩⟩
  for i ← 0 to n                     ⟨⟨Base cases⟩⟩
      LDS[i, n+1] ← 0

  for j ← n down to 1
      for i ← j − 1 down to 0
          if A[i] ≤ A[j]
              LDS[i, j] ← LDS[i, j+1]
          else
              LDS[i, j] ← max{LDS[i, j+1], 1 + LDS[j, j+1]}
  return LDS[0, 1]
```

■

5. Add a sentinel value $A[0] = -\infty$ and compute $LAS^+(0,1)$, where

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

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

Solution (design): We can memoize these functions into two 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(n²) time***. ■

Solution (pseudocode): The following algorithm runs in ***O(n²) time***.

```
LAS(A[1..n]):
  A[0] ← −∞                          ⟨⟨Add a sentinel⟩⟩
  for i ← 0 to n                     ⟨⟨Base cases⟩⟩
      LAS⁺[i, n+1] ← 0
      LAS⁻[i, n+1] ← 0

  for j ← n down to 1
      for i ← j − 1 down to 1
          LAS⁺[i, j] ← LAS⁺[i, j+1]
          LAS⁻[i, j] ← LAS⁻[i, j+1]
          if A[i] < A[j]
              LAS⁺[i, j] ← max{LAS⁺[i, j], 1 + LAS⁻[j, j+1]}
          if A[i] > A[j]
              LAS⁻[i, j] ← max{LAS⁻[i, j], 1 + LAS⁺[j, j+1]}

  return LAS⁺[0, 1]
```

■

6. Add a sentinel value $A[0] = -\infty$ and compute $LAS^+(0,1)$, where

$$LAS^+(i) = 1 + \max\left\{LAS^-(j) \;\middle|\; j > i \text{ and } A[j] < A[i]\right\}$$

$$LAS^-(i) = 1 + \max\left\{LAS^+(j) \;\middle|\; j > i \text{ and } A[j] > A[i]\right\}$$

To establish base cases, we define $\max \varnothing = 0$.

Solution (design): We can memoize these functions into two one-dimensional arrays $LAS^+[0..n]$ and $LAS^-[0..n]$. Each entry $LAS^\pm[i]$ depends only on later entries in the same array or the other array. So we can fill both arrays in parallel, in reverse index order. Evaluating each entry $LAS^\pm[i]$ takes $O(n)$ time, so the resulting algorithm runs in $\boldsymbol{O(n^2)}$ ***time***. ■

Solution (pseudocode): The following algorithm runs in $\boldsymbol{O(n^2)}$ ***time***.

```
LAS(A[1..n]):
  A[0] ← −∞                    ⟨⟨Add a sentinel⟩⟩
  for i ← n down to 1
      LAS⁺[i, j] ← 1           ⟨⟨= 1 + max ∅⟩⟩
      LAS⁻[i, j] ← 1
      for j ← i + 1 to n
          if A[i] < A[j]
              LAS⁺[i] ← max{LAS⁺[i], 1 + LAS⁻[j]}
          if A[i] > A[j]
              LAS⁻[i, j] ← max{LAS⁻[i], 1 + LAS⁺[j]}
  return LAS⁺[0, 1]
```

■

7. Compute $\max_{1 \le i < j \le n} LCS(i, j)$, where

$$LCS(i, j) = 1 + \max \left\{ LCS(j, k) \;\middle|\; j < k \le n \text{ and } A[i] + A[k] > 2A[j] \right\}$$

To establish a base case, we define $\max \varnothing = 0$.

Solution (design): 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***. Finally, we can compute and return the largest entry in the memoization array in $O(n^2)$ additional time. ■

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

```
LCS(A[1..n]):
  maxLCS ← 0
  for i ← n − 1 down to 1
      for j ← n down to i + 1
          LCS[i, j] ← 1        ⟨⟨= 1 + max ∅⟩⟩
          for k ← j + 1 to n
              if A[i] + A[k] > 2A[j]
                  LCS[i, j] ← max{LCS[i, j], 1 + LCS[j, k]}
          maxLCS ← max{ℓ, LCS[i, j]}
  return ℓ
```

■

8. Compute $LPS(1, n)$, where

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

Solution (design): We can memoize the function *LPS* into a two-dimensional array. Each entry depends on the $LPS[i,j]$ 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 $\boldsymbol{O(n^2)}$ ***time***. ■

Solution (pseudocode): The following algorithm runs in $\boldsymbol{O(n^2)}$ ***time***.

```
LPS(A[1..n]):
  for i ← n down to 1
      LPS[i, i−1] ← 0
      LPS[i, i] ← 1
      for j ← i+1 to n
          LPS[i, j] ← max{LPS[i+1, j], LPS[i, j−1]}
          if A[i] = A[j]
              LPS[i, j] ← max{LPS[i, j], 2 + LPS[i+1, j−1]}
  return LPS[1, n]
```

■