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

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

Solution:

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

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

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

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 in the outer loop, and bottom to top in the inner loop.

```
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]
```

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

Solution:

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

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

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

We can memoize the function LIS into a one-dimensional array, which we can fill in reverse order as follows:

```
LIS(A[1..n]):
A[0] = −∞          Add a sentinel
for i ← n downto 0
   LIS[i] ← 1
   for j ← i + 1 to n
      if A[j] > A[i] and 1 + LIS[j] > LIS[i]
         LIS[i] ← 1 + LIS[j]
return LIS[0] − 1    Don't count the sentinel
```

The resulting algorithm runs in $\boldsymbol{O(n^2)}$ ***time***.

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

Solution:

[two parameters] Add a sentinel value $A[0] = \infty$. Let $LIS(i, j)$ denote the length of the longest decreasing subsequence of $A[j \,..\, n]$ where every element is smaller 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 $LDS[\cdot, j+1]$, so we can fill the array in reverse column-major order, scanning right to left in the outer loop, and bottom to top in the inner loop.

```
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]
```

The resulting algorithm runs in $\boldsymbol{O(n^2)}$ ***time***.

Solution:

[clever] The following algorithm runs in $\boldsymbol{O(n^2)}$ ***time***.

```
LDS(A[1 .. n]):
for i ← 1 to n
    Z[i] ← −A[i]
return LIS(Z)
```

Here **LIS** is the longest-*increasing*-subsequence algorithm we developed for problem 1.

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

Solution:

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:

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

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

The length of the longest alternating subsequence is

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

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 in the outer loop, and bottom to top in the inner loop.

```
LAS(A[1..n]):
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]}

ℓ ← 0
for j ← 1 to n
    ℓ ← max{ℓ, 1 + LAS+[j, j+1], 1 + LAS−[j, j+1]}
return ℓ
```

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

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
m ← 1
B[1] ← A[1]
for i ← 2 to n
  if A[i] ≠ B[m]
    m ← m + 1
    B[m] ← A[i]

Count local extrema
ℓ ← 2
for i ← 2 to m − 1
  if B[i] < min {B[i − 1], B[i + 1]}  or  B[i] > max {B[i − 1], B[i + 1]}
    ℓ ← ℓ + 1
return ℓ
```

We need to prove that this greedy algorithm is correct. (Greedy algorithms ***always*** require a proof of correctness, even on exams, because greedy algorithms without proofs are almost always incorrect. Greedy algorithms without proofs will receive ***zero*** credit, even if they are correct. Premature optimization is the root of all evil!) 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 $1 = x_1 < x_2 < x_3 < \cdots < x_\ell = n$ 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 0.1. *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 a local extremum of A.*

Proof: The local extrema $A[x_j]$ divide A into $\ell - 1$ contiguous *blocks* $A_j = A[x_{j-1} \,..\, x_j]$, which overlap at their endpoints and which 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.

After performing this modification inside every block, S contains only local extrema of A, as required.■

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

Solution:

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:

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

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

We can memoize the function LCS into a two-dimensional array, which we can fill in reverse row-major order in $\boldsymbol{O(n^3)}$ ***time*** as follows:

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

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

Solution:

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

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

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

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 in the outer loop, and from left to right in inner loop, as follows:

```
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]
```

The resulting algorithm runs in $\boldsymbol{O(n^2)}$ ***time***.

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

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 in the outer loop, and from left to right in inner loop, as follows:

```
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
        if A[i] = A[j]
            LPS[i, j] ← 2 + LPS[i + 1, j − 1]
        else
            LPS[i, j] ← max {LPS[i + 1, j], LPS[i, j − 1]}
return LPS[1, n]
```

The resulting algorithm runs in $\boldsymbol{O(n^2)}$ ***time***. See, the optimization didn't actually help!