For the first two parts, our goal is to find the length of the longest subsequence that is also a palindrome of a given string $A[1..n]$.

(a) Complete the recurrence for $LPS(i,j)$.

Solution:

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

If $i > j$, then the string $A[i..j]$ is empty and all subsequences are empty. If $i = j$, then the entire string $A[i..j]$ is a palindrome of length 1. If $i < j$ and $A[i] = A[j]$, then we can include $A[i] = A[j]$ as the first and last characters of a palindrome subsequence. In fact, we should, because ignoring both characters hurts us, and we can always exchange the one character we ignore for the first or last character of a palindrome subsequence that ignores exactly one of $A[i]$ or $A[j]$. The rest of the subsequence would come from $A[i+1..j-1]$, and we want to make it as long as possible as well. Finally, if $i < j$ but $A[i] \neq A[j]$, then we cannot include both $A[i]$ and $A[j]$ as extreme characters on a palindrome subsequence. If we ignore $A[i]$, then we’re left to work with $A[i+1..j]$. We work with $A[i..j-1]$ if we ignore $A[j]$. Either way, we want to create the longest subsequence we can from either of those two substrings.

(b) Describe and analyze a dynamic programming algorithm that fills a 2-dimensional array $LPS[1..n+1,0..n]$ with the solution to each subproblem $LPS(i,j)$ and then returns the length of the longest palindrome subsequence of $A[1..n]$.

Solution: Each subproblem $LPS(i,j)$ depends upon those with larger first parameter or smaller second parameter. Therefore, we can fill the array from larger first parameter down as the outer loop and smaller first parameter up as the inner loop. By definition of $LPS$, we want to return $LPS(1,n)$. There are $O(n) \cdot O(n) = O(n^2)$ subproblems to solve in $O(1)$ time each, so the algorithm will take $O(n^2)$ time and use $O(n^2)$ space.
LongestPalindromeSubsequence(A[1..n]):
for i ← n + 1 to 1
    for j ← 0 to n
        if i > j
            LPS[i, j] ← 0
        else if i = j
            LPS[i, j] ← 1
        else 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]

Rubric: 3 points total: 1 point for filling the table correctly. 1 point for returning the correct entry. 1 point for running time analysis.

(c) Describe and analyze a dynamic programming algorithm that returns the length of the shortest supersequence that is also a palindrome of a given string A[1 .. n].

Solution: Per the advice, let SPS(i, j) denote the length of the shortest palindrome supersequence of A[i .. j]. By definition, we want to compute SPS(1, n).

If i > j, string A[i .. j] is empty, and the empty string is its shortest supersequence. If i = j, then A[i] is its own shortest palindrome supersequence. If i < j and A[i] = A[j], it would be wasteful to start and end the supersequence with anything other than A[i] = A[j]. The rest of the supersequence must be a palindrome supersequence of A[i + 1 .. j − 1], and we naturally want to minimize its length. Finally, if i < j but A[i] ≠ A[j], then we need to pair up A[i] or A[j] with a brand new copy of that character and continue working with the one we haven’t paired up yet. That means creating a shortest palindrome supersequence of remaining substring A[i + 1 .. j] or A[i .. j − 1]. We have the following recurrence.

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

Everything said about evaluation order and the \(O(n^2)\) running time from part (b) applies here almost verbatim.
ShortestPalindromeSupersequence(A[1 .. n]):
    for i ← n + 1 to 1
        for j ← 0 to n
            if i > j
                SPS[i, j] ← 0
            else if i = j
                SPS[i, j] ← 1
            else if A[i] = A[j]
                SPS[i, j] ← 2 + SPS[i + 1, j − 1]
            else
                SPS[i, j] ← 2 + min{SPS[i + 1, j], SPS[i, j − 1]}
    return SPS[1, n]

Rubric: 5 points total: 2 points for the recurrence. 2 points for the rest of the algorithm. 1 point for running time analysis.

A correct $O(n^2)$ time algorithm that does not use dynamic programming is worth full credit.
Describe and analyze an efficient algorithm to compute your maximum possible score in a game of Candy Swap Saga XV. Your input is an array $C[1..n]$, where $C[i]$ is the type of candy that the $i$th animal is holding.

**Solution:** The structure of the game itself asks us to perform a sequence of decisions for whether or not to swap candy with each animal. After deciding what to do with one animal, our goal is to maximize our score with the remaining animals, given the current candy we’re holding. That’s practically the same problem!

Let $MaxScore(i, c)$ denote the maximum score we can achieve playing with animals $i$ through $n$ given we start with a piece of candy of type $c \in \{peanut, bar, truffle\}$. By definition, we want our algorithm to compute $MaxScore(1, peanut)$.

If $i > n$, there are no more animals left, and we score a total of 0. If $i \leq n$ and $c = C[i]$, we’re going to leave with a piece of candy of type $c$ no matter what. We should certainly swap for the 1 point then, and try to maximize our score for the remaining animals $i + 1$ through $n$. If $i \leq n$ and $c \neq C[i]$, we should take the better of two choices: swap for $-1$ point and then play with the remaining animals while holding the piece of candy of type $C[i]$ or don’t swap and keep our piece of type $c$. We have the following recurrence:

$$MaxScore(i, c) = \begin{cases} 
0 & \text{if } i > n \\
1 + MaxScore(i + 1, c) & \text{if } i \leq n \text{ and } c = C[i] \\
\max\{-1 + MaxScore(i + 1, C[i]), MaxScore(i + 1, c)\} & \text{otherwise}
\end{cases}$$

We need to solve subproblems with $1 \leq i \leq n + 1$, so we can store subproblem solutions in an array/table $MaxScore[1..n+1, \{peanut, bar, truffle\}]$ (yes, I’m treating candy type as an index to simplify the final code). Each subproblem depends upon those with higher first parameter, so we’ll loop from higher first parameter to lower and then go over the candy types in arbitrary order. There are $O(n) \cdot 3 = O(n)$ subproblems to solve in $O(1)$ time each, so the algorithm will take $O(n)$ time and use $O(n)$ space.

```python
CandySwap(C[1..n]):
for i ← n + 1 to 1
  for c ∈ {peanut, bar, truffle}
    if i > n
      MaxScore[i, c] ← 0
    else if c = C[i]
      MaxScore[i, c] ← 1 + MaxScore[i + 1, c]
    else
      MaxScore[i, c] ← max\{-1 + MaxScore[i + 1, C[i]], MaxScore[i + 1, c]\}
return MaxScore[1, peanut]
```

**Rubric:** 10 points total: 5 points for the recurrence. 3 points for the rest of the algorithm. 2 point for running time analysis.