1. Call a sequence $X[1..n]$ of numbers weakly bitonic if there is an index $i$ with $1 \leq i \leq n$, such that the prefix $X[1..i]$ is increasing and the suffix $X[i..n]$ is decreasing. Describe and analyze an $O(n^2)$ time algorithm to compute the length of the longest weakly bitonic subsequence of an arbitrary array $A$ of integers. Your analysis should explain how much time and space your algorithm uses.

**Solution:** We'll design a dynamic programming algorithm based on the one for longest increasing subsequence. The main difference for the recurrence is that when we consider a suffix $A[j..n]$ of $A[1..n]$, the option to include $A[j]$ in our subsequence is not only dependent on the previous element in the subsequence, but also on whether or not we've passed the maximum element of the subsequence.

For simplicity, let $A[0] = -\infty$. Given a boolean $z$ and two integers $i$ and $j$ with $0 \leq i < j \leq n + 1$, let $LBS(z, i, j)$ denote the length of the longest subsequence $X[1..k]$ of $A[j..n]$ that meets one of the following conditions: if $z = \text{False}$, then $X$ is weakly bitonic and $X[1] > A[i]$, and if $z = \text{True}$, then $X$ is decreasing and $X[1] < A[i]$. The length of the longest weakly bitonic subsequence of $A$ is therefore $LBS(\text{False}, 0, 1)$. $LBS(z, i, j)$ can be defined recursively as follows:

$$
LBS(z, i, j) = \begin{cases} 
-\infty & \text{if } z = \text{False} \text{ and } j > n \\
0 & \text{if } z = \text{True} \text{ and } j > n \\
LBS(\text{False}, i, j + 1) & \text{if } z = \text{False} \text{ and } A[i] \geq A[j] \\
\max(LBS(\text{False}, i, j + 1), 1 + LBS(\text{False}, j, j + 1)) & \text{if } z = \text{False} \text{ and } A[i] < A[j] \\
LBS(\text{True}, i, j + 1) & \text{if } z = \text{True} \text{ and } A[i] \leq A[j] \\
\max(LBS(\text{True}, i, j + 1), 1 + LBS(\text{True}, j, j + 1)) & \text{otherwise}
\end{cases}
$$

If $j > n$, $A[j..n]$ is empty. There are no weakly bitonic subsequences of 0 length, so we use $-\infty$ as an error sentinel if $z = \text{False}$; otherwise, it is correct to say the length of the longest decreasing subsequence of an empty sequence is 0. The first two cases are correct.

Now, consider any $j \leq n$ and assume inductively the recurrence is correct for all $LBS(\cdot, \cdot, j')$ where $j' > j$ (i.e., $n - j' + 1 < n - j + 1$). We need the length of a longest subsequence $X[1..k]$ of $A[j..n]$ subject to some conditions. If $z = \text{False}$, then $X[1] > A[i]$. Therefore, $A[j]$ cannot be in $X$ if $A[i] \geq A[j]$, so the third case of the recurrence correctly considers only the later elements $A[j + 1..n]$. If $A[i] < A[j]$, then $X$ may or may not begin with $A[j]$. If it does begin with $A[j]$, then it may or may not be the largest member of $X$. If it is not the largest member of $X$, then the rest of $X$ is weakly bitonic, so we should count $A[j]$ toward the length of $X$ and ask for the maximum length for the remainder of $X$ which comes from $A[j + 1..n]$ and starts with an element larger than $A[j]$. If $A[j]$ is the largest member of $X$, then the rest of $X$ must be a decreasing subsequence with elements smaller than $A[j]$, so we should count $A[j]$ and then ask for the maximum length for the remainder of $X$ which is a decreasing subsequence with starting element smaller than $A[j]$. The forth case correctly considers the best of these three possibilities.
The final two cases are correct for similar reasons. In fact, they are the same recurrence we used for longest increasing subsequence, just with the inequalities flipped.

At this point, we may observe that our dynamic programming data structure has three indices. One of these indices takes on only two values, but the others take on $O(n)$ values each. The space usage will be $O(n^2)$. Each subproblem takes $O(1)$ time to solve given its dependencies, so the time usage is $O(n^2)$ as well.

For the data structure itself, we’ll use a 3-dimensional array $LBS[0..1][0..n][1..n+1]$ where the first index is 0 for $z = \text{FALSE}$ and 1 for $z = \text{TRUE}$. Each entry depends upon strictly larger $j$ indices and $z$ indices that are at least as large. Therefore, one option for filling the array is to use decreasing $z$ values for an outermost loops, decreasing $j$ values in a middle loop, and increasing $i$ values in an innermost loop.

**Rubric:** 10 points total: 6 points for recurrence and proof, 2 points for filling a data structure, 1 point each for space and time analysis. The recurrence must be correct to get credit for the data structure or analyzes.
2. A palindrome is any string that is exactly the same as its reversal, like I, or DEED, or RACECAR, or AMANAPLANACATACANALPANAMA.

Describe and analyze an efficient algorithm to find the length of the longest palindrome subsequence of a given string/array $A[1 .. n]$. Your analysis should explain how much time and space your algorithm uses.

**Solution:** We'll design a dynamic programming algorithm based on the following observation. Either the longest palindrome subsequence of $A[1 .. n]$ contains both $A[1]$ and $A[n]$ as its endpoints, or it doesn’t.

Given integers $i$ and $j$ such that $1 \leq i \leq n + 1$ and $0 \leq j \leq n$, define $LPS(i, j)$ to be the length of the longest palindrome subsequence of $A[i .. j]$ if $i \leq j$. For $i > j$, let $LPS(i, j) = 0$. The length of the longest palindrome subsequence of $A[1 .. n]$ is $LPS(1, n)$. $LPS(i, j)$ can be defined recursively as follows.

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

The first case is correct by definition. A string with one character is itself a palindrome and is itself its only non-empty subsequence, so the second case is correct. For other cases, we’ll assume inductively that the recurrence is correct for all $LPS(i’, j’)$ where $j’ - i’ < j - i$. If $A[i]$ differs from $A[j]$, they can’t both be in a palindrome subsequence of $A[i .. j]$; otherwise, that subsequence’s endpoints would not match. Therefore, the third case correctly finds the maximum length using only substrings of $A[i .. j]$ that result from removing one of $A[i]$ or $A[j]$. For the final case, $A[i]$ and $A[j]$ can start and end a palindrome subsequence since they match. The inside of the longest palindrome subsequence would then be the longest palindrome subsequence from characters between $A[i]$ and $A[j]$. The final case considers this option in the third part of the max while also considering the two options where one of $A[i]$ or $A[j]$ does not appear in the longest palindrome subsequence.

At this point, we may observe that our dynamic programming data structure has two indices, each taking on $O(n)$ different values. The space usage will be $O(n^2)$. Each subproblem takes $O(1)$ time to solve given its dependencies, so the time usage is $O(n^2)$ as well.

For the data structure itself, we’ll use a 2-dimensional array $LPS[1 .. n + 1][0 .. n]$. Each entry $LPS[i][j]$ depends upon entries $LPS[i’][j’]$ with $i’ > i$, $j’ < j$, or both. In other words, $j’ - i’ < j - i$. Therefore, we should fill in all the entries below the main diagonal (where $i > j$) with 0’s, fill the main diagonal (where $i = j$) with 1’s, and then fill the array in diagonal by diagonal where each diagonal is above or to the right of the previous one. In other words, the $k$th iteration of the outer loop should consider all entries where $j - i = k$. The entries within a diagonal can be filled in an arbitrary order.

**Rubric:** 10 points total: 5 points for recurrence and proof, 3 points for filling a data structure, 1 point each for space and time analysis. The recurrence must be correct to get credit for the data structure or analyzes.
3. **Extra credit** (worth 1/2 a normal question): Describe and analyze an $O(n^2)$ time algorithm to find the smallest number of palindromes that make up a given input string/array $A[1..n]$. Your analysis should explain how much time and space your algorithm uses.

**Solution:** We'll use main two ideas to obtain a dynamic programming solution. First, the optimal decomposition of $A$ uses some first palindrome followed by what remains of $A$ after removing that first palindrome. Second, we can preprocess $A$, again using dynamic programming, so we can quickly look up which substrings of $A$ are palindromes.

Given integers $i$ and $j$ such that $1 \leq i \leq j \leq n$, let $Pal(i, j) = \text{True}$ if $A[i..j]$ is a palindrome. Otherwise, $Pal(i, j) = \text{False}$. For now, assume we can compute $Pal(i, j)$ in constant time; later, we'll discuss how to preprocess $A$ in order to do so. Given integer $i$ with $1 \leq i \leq n + 1$, let $PD(i)$ denote the smallest number of palindromes that make up $A[i..n]$. The smallest number of palindromes that make up $A$ itself is $PD(1)$. $PD(i)$ can be defined recursively as follows.

\[
PD(i) = \begin{cases} 
0 & \text{if } i > n \\
\min_{j \text{ s.t. } i \leq j \leq n \text{ and } Pal(i, j) = \text{True}} 1 + PD(j + 1) & \text{otherwise}
\end{cases}
\]

The first case is correct, because there are no characters in $A[n+1..n]$ so there are no palindromes needed. For the second case, we may assume inductively that the recurrence is correct for $PD(i')$ where $i' > i$ (i.e. $n - i' < n - i$). The second case is correctly considering all first palindromes in the decomposition of $A[i..n]$ and adding on the number of palindromes needed to decompose the remaining string $A[j+1..n]$ given a particular choice.

The data structure for $PD$ takes only one index between $1$ and $n + 1$ so it uses $O(n)$ space. It takes $O(n)$ time to check all prefixes of $A[i..n]$ to find which ones are palindromes, so the total time spent filling in entries of the data structure should be $O(n^2)$. We can use an array $PD[1..n+1]$ for the data structure. Since each entry relies on those to the right (with higher index), we can fill it from right to left (in decreasing order of index).

So, how do we precompute all $Pal(i, j)$ efficiently? First, we note how $Pal(i, j)$ can be defined using a recurrence.

\[
Pal(i, j) = \begin{cases} 
\text{True} & \text{if } i = j \\
\text{True} & \text{if } i + 1 = j \text{ and } A[i] = A[j] \\
\text{False} & \text{if } A[i] \neq A[j] \\
Pal(i + 1, j - 1) & \text{otherwise}
\end{cases}
\]

Indeed, a single character string is a palindrome, a two character string is a palindrome if its characters match, and in all other cases, a string is a palindrome if and only if its outermost characters match and the rest of its characters form a palindrome. We can precompute $Pal(i, j)$ for all $1 \leq i \leq j \leq n$ by storing the values in an array $Pal[1..n][1..n]$. Doing so takes $O(n^2)$ space and, since each entry is dependent on a constant number of other entries, $O(n^2)$ time. So in total, the whole algorithm uses $O(n^2)$ space and time. Each entry of $Pal[1..n][1..n]$ is dependent on a subset of the entries needed for Problem 2, so as in that case, we fill in the main diagonal first, and then fill in the array diagonal-by-diagonal where each subsequent diagonal is above or to the right of the previous one.

\[\blacksquare\]
Rubric: 5 points extra credit total: 3 points for recurrences, 1 point for filling the data structures, 1 point for space and time analysis. The recurrences must be correct to get credit for the data structures or analyzes.