A palindrome is any string that is exactly the same as its reversal, like I, or DEED, or RACECAR, or AMANAPLANACATACANALPANAMA. Note that a palindrome may have an odd number of characters.

(a) Let $MaxPalSub(i,j)$ be the length of the longest subsequence of $X[i .. j]$ that is also a palindrome. (For simplicity, we treat $X[i .. j]$ as empty if $i > j$.) Fill in the blanks to complete the following recursive definition of $MaxPalSub(i,j)$.

Solution:

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

Rubric: 4 points total: 1 point per blank.

(b) Use dynamic programming to write a method in Java that takes as its one parameter an array $x$ of characters and returns the length of the longest palindrome subsequence in $x$. Your method should be based on the above recurrence and run in $O(n^2)$ time given an array of length $n$.

Solution:

```java
public static void longestPalindromeLength(char[] x) {
    int n = x.length;

    // Second base case prevents i = n + 1 or j = 0.
    int[][] maxPalSub = new int[n][n];
    for (int i = n - 1; i >= 0; i--) {
        for (int j = 0; j < n; j++) {
            if (i > j) {
                maxPalSub[i][j] = 0;
            } else if (i == j) {
                maxPalSub[i][j] = 1;
            } else if (x[i] == x[j]) {
                maxPalSub[i][j] = 2 + maxPalSub[i+1][j-1];
            } else {
                maxPalSub[i][j] = Math.max(maxPalSub[i][j-1], maxPalSub[i+1][j]);
            }
        }
    }

    return maxPalSub[0][n-1];
}
```
Rubric: 6 points total: −2 points for not returning correct entry; −3 points for not filling table in correct order.
Suppose you are given a directed graph \( G = (V, E) \) where each edge has an integer weight between 0 and some small value \( M > 0 \) along with a designated vertex \( s \in V \). You may assume every vertex is reachable from \( s \).

(a) What is the maximum possible distance a vertex can have from \( s \)?

**Solution:** Weights are non-zero, so shortest paths need not repeat any vertices. The longest simple path has \(|V| - 1\) edges, so the maximum distance is \( M(|V| - 1) \).

**Rubric:** 2 points total. Any value larger but asymptotically equal to \( M(|V| - 1) \) is worth full credit.

(b) Describe how to implement Dijkstra’s algorithm so that it runs in \( O(M|V| + |E|) \) time.

**Solution:** Instead of a priority queue, we’ll maintain an array of doubly linked lists where the list in position \( i \) contains links to all vertices \( v \) such that \( v.dist = i \). We’ll also have each vertex maintain a link to its linked list node so the vertex can be removed and then inserted into a different list in \( O(1) \) time. All \( dist \) values are either \( \infty \) or integers between 0 and \( M(|V| - 1) \), so the array should have length \( M(|V| - 1) + 1 \).

Initially, only vertex \( s \) belongs to a list, and that list is the one at position 0. Whenever some value \( v.dist \) is updated, we remove \( v \) from the list containing it (if \( v.dist \neq \infty \)) and add \( v \) to the list at position \( v.dist \).

For the outer loop, we maintain a single \( \text{finger} \) integer equal to the largest distance of any known vertex. To pick the next vertex to process, we increment the \( \text{finger} \) until it equals the position of a non-empty list, remove a vertex \( v \) from the list, and process \( v \). Distances for the vertices we process only increase over the course of the algorithm, so we’ll never “skip” a vertex choosing them in this way.

Creating the array and incrementing the \( \text{finger} \) at empty positions takes \( O(M|V|) \) time total. Processing vertices takes \( O(|E|) \) time total as each distance check and update takes only \( O(1) \) time. The total time used is \( O(M|V| + |E|) \).

**Rubric:** 8 points total.