1. Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string.

Solution: Let $A[1..n]$ be the input string. We define two functions:

- *IsPal?*(i, j) is True if the substring $A[i..j]$ is a palindrome, and False otherwise.
- *MinPals*(k) is the minimum number of palindromes that make up the suffix $A[k..n]$.

We need to compute *MinPals*(1).

First consider the support function *IsPal?*. Every string with length at most 1 is a palindrome. A string of length 2 or more is a palindrome if and only if its first and last characters are equal *and* the rest of the string is a palindrome. Thus:

$$IsPal?(i,j) = \begin{cases} \text{True} & \text{if } i \geq j \\ (A[i] = A[j]) \wedge IsPal?(i+1, j-1) & \text{otherwise} \end{cases}$$

We can memoize the *IsPal?* function into a two-dimensional array *IsPal?*$[1..n, 0..n]$. Each entry *IsPal?*$[i, j]$ depends only on *IsPal*$[i+1, j-1]$. Thus, we can fill this array row-by-row, from the bottom row upward.

```
FindPals(A[1..n]):
  for i ← n down to 1
      IsPal?[i, i−1] ← True
      IsPal?[i, i] ← True
      for j ← i+1 to n
          if A[i] = A[j]
              IsPal?[i, j] ← IsPal?[i+1, j−1]
          else
              IsPal?[i, j] ← False
```

This algorithm clearly runs in $O(n^2)$ time. Notice that FindPals doesn't return anything; it just memoizes the function for use by the main algorithm.

Now let's handle the main function *MinPals*. The empty string can be partitioned into zero palindromes. Otherwise, the best palindrome decomposition has at least one palindrome. If the first palindrome in the *optimal* decomposition of $A[1..n]$ ends at index ℓ, the remainder must be the *optimal* decomposition for the remaining characters $A[\ell+1..n]$. The following recurrence considers all possible values of ℓ.

$$MinPals(k) = \begin{cases} 0 & \text{if } k > n \\ 1 + \min\left\{ MinPals(\ell+1) \;\middle|\; k \leq \ell \leq n \text{ and } IsPal?(k,\ell) \right\} & \text{otherwise} \end{cases}$$

We can memoize the *MinPals* function into a one-dimensional array *MinPals*$[1..n]$. Each entry *MinPals*$[k]$ depends only on entries *MinPals*$[\ell+1]$ with $\ell \geq k$, so we can fill this array from right to left.

```
MinPals(A[1..n]):
  FindPals(A[1..n])
  MinPals[n+1] ← 0
  for k ← n down to 1
      MinPals[k] ← ∞
      for ℓ ← k to n
          if IsPal?[k, ℓ]
              MinPals[k] ← min{MinPals[k], 1 + MinPals[ℓ+1]}
  return MinPals[1]
```

The subroutine FINDPALS runs in $O(n^2)$ time, and the rest of the algorithm also clearly runs in $O(n^2)$ time. So the overall algorithm runs in $\boldsymbol{O(n^2)}$ ***time.***

If we had used the obvious iterative algorithm to test whether each substring is a palindrome, instead of precomputing the array *IsPal?*, our algorithm would have run in $O(n^3)$ time. ■

Rubric: Standard dynamic programming rubric. This is more detail than necessary for full credit. Max 8 points for an $O(n^3)$-time algorithm; scale partial credit.

2. (a) Describe an algorithm to compute the minimum number of rounds required for the message to be delivered to all nodes in a ***binary*** tree.

Solution: See part (b). ■

Solution (self-contained): Let T be the input tree. For each node v in T, let $MinRounds(v)$ denote the minimum number of rounds required, after v learns the message, to inform every descendant of v. We need to compute $MinRounds(root(T))$. This function obeys the following recurrence:

$$MinRounds(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ 1 + MinRounds(v.left) & \text{if } v \text{ has no right child} \\ 1 + MinRounds(v.right) & \text{if } v \text{ has no left child} \\ \min \left\{ \begin{array}{l} \max \left\{ \begin{array}{c} 1 + MinRounds(v.left) \\ 2 + MinRounds(v.right) \end{array} \right\} \\ \max \left\{ \begin{array}{c} 1 + MinRounds(v.right) \\ 2 + MinRounds(v.left) \end{array} \right\} \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function in the tree itself, by adding a new field $v.MinRounds$ to each node record. But in fact, the following recursive algorithm evaluates this function purely recursively, essentially by memoizing $MinRounds(v)$ at the *parent* of v, in the temporary variables ℓ and r.

```
MinRounds(v):
  if v is a leaf
      return 0
  else if v.right = Null
      return 1 + MinRounds(v.left)
  else if v.left = Null
      return 1 + MinRounds(v.right)
  else
      ℓ ← MinRounds(v.left)
      r ← MinRounds(v.right)
      if ℓ < r
          return r + 1
      else if ℓ > r
          return ℓ + 1
      else
          return ℓ + 2
```

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

Rubric: 4 points: standard DP rubric (scaled). “See part (b)” is worth *exactly* as many points as the submitted algorithm for part (b).

(b) Describe an algorithm to compute the minimum number of rounds required for the message to be delivered to all nodes in an ***arbitrary rooted*** tree.

Solution: Let T be the input tree. For each node v in T, let $\deg(v)$ denote its degree (number of children), and let $\textsc{MinRounds}(v)$ denote the minimum number of rounds required, after v learns the message, to inform every descendant of v. We need to compute $MinRounds(root(T))$. We can compute this function recursively as follows.

```
MinRounds(v):
  for i ← 1 to deg(v)
      w ← ith child of v
      R[i] ← MinRounds(w)
  sort R[1 .. deg(v)] in decreasing order
  rounds ← 0
  for i ← 1 to deg(v)
      rounds ← max{rounds, i + R[i]}
  return rounds
```

Assuming we use an $O(n \log n)$-time algorithm like mergesort or heapsort to sort the array R, this algorithm spends $O(\deg(v) \cdot \log \deg(v))$ time at each node v, not counting recursive calls. Thus, the overall running time of the algorithm is

$$\begin{aligned}\sum_v O(\deg(v) \cdot \log \deg(v)) &\le \sum_v O(\deg(v) \log n) \\ &= O(\log n) \cdot \sum_v \deg(v) \;=\; O(n \log n).\end{aligned}$$

To prove the algorithm correct, we need to justify sorting R downward. The order of R determines the order in which v broadcasts to its children. Assuming all descendants of v broadcast optimally (thanks to the Recursion Fairy), the total number of rounds to reach all descendants of v is

$$r := \max_i \left\{ i + R[i] \right\}.$$

Now suppose $R[i-1] < R[i]$ for some index i, and let r' be the number of rounds required to reach all descendants of v we swap children $i-1$ and i in the broadcast schedule. In the new schedule, we reach all descendants of child $i-1$ after $(i-1)+R[i]$ rounds, we reach all descendants of child i after $i + R[i-1]$ rounds, and we reach all descendants of any other child j after $j + R[j]$ rounds. We immediately observe that

$$\begin{aligned}(i-1) + R[i] < i + R[i] &\le r, \\ i + R[i-1] < i + R[i] &\le r, \qquad \text{and} \\ j + R[j] &\le r \qquad \text{for all } j,\end{aligned}$$

which implies that $r' \le r$. Thus sorting R in decreasing order yields a schedule that is no worse than any other permutation. In other words, our algorithm is correct. ■

Rubric: 6 points = 4 for the algorithm (standard DP rubric, scaled) + 2 for the proof of correctness.

3. Suppose you are given an $m \times n$ bitmap, represented by an array $M[1..m, 1..n]$ of 0s and 1s.

   (a) Describe and analyze an algorithm to compute a guillotine subdivision of M of minimum *size*.

Solution: We define two functions for every quadruple of indices i, i', j, j':

- *Solid?*(i, i', j, j') is TRUE if the subarray $M[i, i' .., j .. j']$ is a solid block, and FALSE otherwise.
- *Size*(i, i', j, j') denotes the size of the smallest guillotine subdivision of the subarray $M[i, i' .., j .. j']$.

We need to compute *Size*$(1, m, 1, n)$. We consider these functions one at a time, just like in problem 1.

The function *Solid?* obeys the following recurrence. Informally, a block is solid if it is either a single pixel or it can be split into two solid blocks with the same color.

$$\mathit{Solid?}(i,i',j,j') = \begin{cases} \textsc{True} & \text{if } i = i' \text{ and } j = j' \\ \mathit{Solid?}(i,i,j+1,j') \wedge (M[i,j] = M[i,j+1]) & \text{if } i = i' \text{ and } j < j' \\ \begin{array}{l} \mathit{Solid?}(i,i,j,j') \wedge \mathit{Solid?}(i+1,i',j,j') \\ \qquad \wedge (M[i,j] = M[i+1,j]) \end{array} & \text{if } i < i' \text{ and } j < j' \end{cases}$$

This function can be memoized into a four-dimensional array, which we can fill in the following order in $O(m^2n^2)$ time.

```
FINDSOLID(M):
  for i ← 1 to m
      for i′ ← i to m
          for j ← 1 to n
              for j′ ← j to n
                  ⟨⟨evaluate recurrence in O(1) time⟩⟩
```

The main function *Size* obeys the following recurrence:

$$\mathit{Size}(i,i',j,j') = \begin{cases} 1 & \text{if } \mathit{Solid?}(i,i',j,j') \\ \min \left\{ \begin{array}{l} \displaystyle\min_{i \le h < i'} \big(\mathit{Size}(i,h,j,j') + \mathit{Size}(h+1,i',j,j')\big) \\ \displaystyle\min_{j \le v < j'} \big(\mathit{Size}(i,i',j,v) + \mathit{Size}(i,i',v+1,j')\big) \end{array} \right\} & \text{otherwise} \end{cases}$$

(The recurrence correctly handles the cases where either $i = i'$ or $j = j'$. If both $i = i'$ and $j = j'$, then *Solid?*$(i, i', j, j') = $ TRUE. Otherwise, if $i = i'$, there are no indices h such that $i \le h < i'$, so the first min expression evaluates to ∞.) We can memoize this function into a four-dimensional array, which we can fill in the following order:

```
MinSize(M):
  FindSolid(M)
  for i ← m down to 1
      for i′ ← i to m
          for j ← n down to 1
              for j′ ← j to n
                  ⟨⟨evaluate recurrence in O(m + n) time⟩⟩
  return Size[1, m, 1, n]
```

Evaluating the recurrence for any particular values of i, i', j, j' requires two for-loops—one considering $i' - i - i \leq m$ possible horizontal splits h, and the other considering $j' - j - 1 \leq n$ possible vertical splits v—and thus requires $O(m + n)$ time. The overall algorithm runs in $\boldsymbol{O(m^2 n^2 (m + n))}$ ***time***.

Without the preprocessing phase to find all solid blocks, the algorithm would run in $O(m^3 n^3)$ time. ■

Rubric: 5 points, standard DP rubric (scaled). Max 4 points for an algorithm that runs in $O(m^3 n^3)$ time; scale partial credit.

(b) Describe and analyze an algorithm to compute a guillotine subdivision of M of minimum *depth*.

Solution: The solution is nearly identical to part (a); the only difference is that we use the recursive definition of depth instead of the recursive definition of size. The base case is 0 instead of 1, and we take the max of the children's depths rather than the sum of their sizes.

Specifically, the function $Depth(i, i', j, j')$, which denotes the *depth* of the *shallowest* guillotine subdivision of the subarray $M[i, i' .., j .. j']$, obeys the recurrence

$$Depth(i, i', j, j') = \begin{cases} 0 & \text{if } Solid?(i, i', j, j') \\ \min \left\{ \begin{array}{l} \min\limits_{i \leq h < i'} \left(1 + \max \left\{ \begin{array}{c} Depth(i, h, j, j') \\ Depth(h+1, i', j, j') \end{array} \right\} \right) \\ \min\limits_{j \leq v < j'} \left(1 + \max \left\{ \begin{array}{c} Depth(i, i', j, v) \\ Depth(i, i', v+1, j') \end{array} \right\} \right) \end{array} \right\} & \text{otherwise} \end{cases}$$

All other aspects of the algorithm, including the running time, are identical. ■

Rubric: 5 points, standard DP rubric (scaled). Yes, this solution is enough for full credit. Max 4 points for a $O(m^3 n^3)$-time algorithm; scale partial credit.