Using $\Theta$-notation, provide asymptotically tight bounds in terms of $n$ for the solution to each of the following recurrences.

(a) $T(n) = 5T(n/2) + n$

**Solution:** Node values at level $i$ of the recursion tree sum to $(5/2)^i n$, so the sum of level sums is an increasing geometric series. The largest term is proportional to the number of leaves, so $T(n) = \Theta(n^{\log_{5/2} 5})$. (Recall, $\lg n = \log_2 n$.)

(b) $T(n) = 4T(n/2) + n^2$

**Solution:** Node values at level $i$ of the recursion tree sum to $n^2$. There are $O(\log n)$ levels in the recursion tree, so summing the level sums results in $T(n) = \Theta(n^2 \log n)$.

(c) $T(n) = T(n/4) + T(n/2) + n$

**Solution:** Node values at level $i$ of the recursion tree sum to at most $(3/4)^i n$, so the sum of level sums is bounded from above by a decreasing geometric series. The largest term is the root sum of $n$, so $T(n) = O(n)$. The root alone is already $\Omega(n)$, so $T(n) = \Theta(n)$.

**Rubric:** 2 points for each of parts (a), (b), and (c). −1/2 per part for failing to provide some kind of justification.

Consider the procedure `StoogeSort(A[1..n])` which sorts the given array $A[1..n]$.

(d) State a recurrence for the running time of `StoogeSort(A[1..n])`.

**Solution:** Let $T(n)$ be the worst-case running time of `StoogeSort(A[1..n])`. The procedure makes three recursive calls, each on an array of size $[2n/3]$ (yes, $n - (n - [2n/3] + 1) + 1 = [2n/3]$). Everything else takes constant time, and we can safely ignore ceilings and constant factors in a divide-and-conquer recurrence, so $T(n) = 3T(2n/3) + 1$.

**Rubric:** 2 points total. −1/2 point for a $+O(n)$ instead of a $+O(1)$. −1/2 point for no justification. 0 points total for claiming two differently sized recursive calls.

(e) Solve your recurrence from part (d) in order to determine the running time of `StoogeSort(A[1..n])`.

**Solution:** Node values at level $i$ of the recursion tree sum to $3^i$, so the sum of level sums is an increasing geometric series. The largest term is proportional to the number of leaves, so the running time is $T(n) = \Theta(n^{\log_3 3})$ (which is $\Omega(n^{2/7})$).
(f) (Extra credit worth half a question.) Prove \textsc{stoogesort}(A[1..n]) actually sorts its input.

\textbf{Solution:} If \( n \leq 1 \), then the array is already sorted and the procedure is correct to do nothing. If \( n = 2 \), then the procedure correctly swaps the two elements only if they are out of order. Suppose \( n \geq 3 \). Call an element of rank at most \([2n/3]\) small and the remaining elements large. Initially, there are at most \( n - [2n/3] = [n/3] \) elements lying outside of \( A[1..[2n/3]] \). In particular, this collection contains at most \([n/3]\) small elements. Therefore, \( A[1..[2n/3]] \) contains at least \([2n/3] - [n/3] \geq [n/3]\) small elements.

The first recursive call will inductively move the small elements of \( A[1..[2n/3]] \) to fill the first \([n/3]\) (or more) positions of \( A \), so every large element will be outside \( A[1..[n/3]] \) and therefore in \( A[[n/3] + 1..n] \). Therefore, the second recursive call will move every large element to the last \([n/3]\) positions of \( A \), their final destination. At this point, all the small elements are in \( A[1..[2n/3]] \), and the final recursive call will inductively put them in their sorted order.

\textbf{Rubric:} 5 extra credit points total. −1 point for missing base cases. Students may not use outside sources for extra credit.
(a) Describe and analyze an implementation of \textsc{CountAndMerge}(A[1..n], m) based on the Merge\footnote{Note that Erickson uses the \textsc{MergeSort} procedure implementation in Erickson Figure 1.6.} procedure in Erickson Figure 1.6.\footnote{Note that Erickson uses the \textsc{MergeSort} procedure implementation in Erickson Figure 1.6.}

**Solution:** The algorithm is described in pseudocode below.

```
count ← 0
i ← 1; j ← m + 1
for k ← 1 to n
    if j > n
        B[k] ← A[i]; i ← i + 1
    else if i > m
        B[k] ← A[j]; j ← j + 1
    else if A[i] ≤ A[j]
        B[k] ← A[i]; i ← i + 1
    else
        B[k] ← A[j]
    count ← count + (m − i + 1)
    j ← j + 1
for k ← 1 to n
    A[k] ← B[k]
return count
```

New lines are depicted in **red**. These lines do not affect the merging of the arrays, so the arrays must be merged correctly. We will now prove inductively that for any \( k \) between \( 1 \) and \( n + 1 \), the last \( n − k + 1 \) iterations of the main for loop increase \textit{count} by the number of inversions \((i', j')\) where \( i \leq i' \leq m \) and \( m + 1 \leq j' \leq n \). If \( k = n + 1 \), there are no such inversions, and we are correct to stop looping and increasing \textit{count}. Suppose \( k \leq n \). If \( j > n \), then there are no \( j' \) between \( m + 1 \) and \( n \) and in particular no such inversions \((i', j')\) where \( i' = i \). Inductively, the remaining iterations will count the (zero) remaining inversions. Similarly, if \( i > m \), there are no applicable inversions \((i', j')\) with \( j' = j \), and the remaining iterations will count the (zero) remaining inversions.

If neither case holds but \( A[i] \leq A[j] \), then \( A[i] \leq A[j] \leq A[j'] \) for all \( j \leq j' \leq n \). There are no inversions \((i', j')\) to count with \( i' = i \), so it suffices just to count just those with \( i + 1 \leq i' \leq m \), which we do inductively. In the final case, \( A[i'] \geq A[i] > A[j] \) for all \( i \leq i' \leq m \). There are exactly \( m − i + 1 \) applicable inversions \((i', j')\) with \( j' = j \), and we increase \textit{count} by this amount. We have now counted all possible inversions with \( j' = j \), so it suffices to inductively count the remaining ones with \( j' > j \).

For the running time analysis, we still have two for loops over \( n \) values where iterations take constant time each. **The running time is \( O(n) \).**

**Rubric:** 5 points total: 3 points for a correct algorithm. 1 point for the proof of correctness. 1 point for the running time analysis.

(Here and more generally, our proofs of correctness will be more detailed than what is needed for full credit. You should still take them seriously, though, to convince both yourself and us that your algorithm is correct.)
(b) Describe and analyze an implementation of \texttt{CountAndMergeSort}(A[1..n]) based on the \texttt{MergeSort}(A[1..n]) procedure in Erickson Figure 1.6.

**Solution:** The algorithm is described in pseudocode below.

```
\begin{verbatim}
CountAndMergeSort(A[1..n], m):
    count ← 0
    if n > 1
        m ← \lfloor n/2 \rfloor
        count ← count + CountAndMergeSort(A[1..m])
        count ← count + CountAndMergeSort(A[m+1..n])
        count ← count + CountAndMerge(A[1..n], m)
    return count
\end{verbatim}
```

Modified lines are depicted in red. The lines do not affect the sorting of the array, so the array must be sorted correctly. If \( n \leq 1 \), then there can be no inversions, and the algorithm correctly returns \( \text{count} = 0 \). Otherwise, there are three kinds of inversions \((i, j)\) possible: 1) those with \( 1 \leq i < j \leq m \); 2) those with \( m + 1 \leq i < j \leq n \); and 3) those with \( 1 \leq i \leq m \) and \( m + 1 \leq j \leq n \). Inductively, we count the first type during the first recursive call while also sorting \( A[1..m] \). Inductively, we count the second type during the second recursive call while also sorting \( A[m+1..n] \). Inversions of the third type survive shuffling of the two disjoint subarrays, so we successfully take the two sorted subarrays and count inversions of the third type during the call to \texttt{CountAndMerge} as explained in part (a).

We added some constant time operations on top of what \texttt{MergeSort} was already doing, so the running time is still \( O(n \log n) \). (Alternatively we could set up a recurrence \( T(n) = 2T(n/2) + O(n) \) which solves to \( O(n \log n) \).)

\[\blacksquare\]

**Rubric:** 5 points total: 3 points for a correct algorithm. 1 point for the proof of correctness. 1 point for the running time analysis.