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

(a) \(T(n) = 4T(n/2) + n^2\)

**Solution:** Each level of the recursion tree sums to \(n^2\) and there are \(\lg n\) levels, so \(T(n) = \Theta(n^2 \log n)\).

(b) \(T(n) = 7T(n/3) + n\)

**Solution:** The \(i\)th level of the recursion tree sums to \((7/3)^i n\). The level sums form an increasing geometric series asymptotically bounded by its largest term, the number of leaves. There are \(7^{\log_3 n} = n^{\log_3 7}\) leaves, so \(T(n) = \Theta(n^{\log_3 7})\).

(c) \(T(n) = 3T(n/2) + n^2\)

**Solution:** The \(i\)th level of the recursion tree sums to \((3/4)^i n\). The level sums form a decreasing geometric series asymptotically bounded by its largest term, so \(T(n) = \Theta(n^2)\).

(d) \(T(n) = 3T(n/4) + T(n/5) + n\)

**Solution:** The \(i\)th level of the recursion tree sums to \((19/20)^i n\). The level sums form a decreasing geometric series asymptotically bounded by its largest term, so \(T(n) = \Theta(n)\).

(e) \(T(n) = 2T(n/2) + n \lg^2 n\)

**Solution:** Each level of the recursion tree sums to at most \(n \lg^2 n\). There are \(\lg n\) levels, so \(T(n) = O(n \log^3 n)\). The top \((1/2)\lg n\) levels each sum to at least \((1/4)n \lg^2 n\), so \(T(n) = \Omega(n \lg^3 n)\). Combining the upper and lower bounds, we get \(T(n) = \Theta(n \lg^3 n)\).

**Rubric:** 2 points each: 1 point for correct answer, 1 point for justification.
Describe an algorithm to transfer a stack of $n$ disks from one *vertical* needle to the other *vertical* needle, using the smallest possible number of moves.

**Solution:** We’ll use two mutually recursive procedures: $\text{Hanoi}(n, src, dst, tmp)$ moves a stack of $n$ disks from a vertical needle $src$ to a vertical needle $dst$ using the leaning needle $tmp$ as a temporary placeholder. $\text{Hanoi}(n, src, dst, tmp)$ moves a stack of $n$ disks from a vertical needle $src$ to the leaning needle $tmp$ using the vertical needle $tmp$ as a temporary placeholder. Our algorithm calls $\text{Hanoi}(n, src, dst, tmp)$ using the starting vertical needle as the initial $src$ and the other vertical needle as the initial $dst$.

$\text{Hanoi}(n, src, dst, tmp)$:

* if $n = 1$
  * move disk $n$ from $src$ to $dst$
* else if $n > 1$
  * $\text{Hanoi}(n - 1, src, tmp, dst)$
  * move disk $n$ from $src$ to $dst$
  * move disks 1 through $n - 1$ from $tmp$ to $dst$

$\text{Hanoi}(n, src, dst, tmp)$:

* if $n > 0$
  * $\text{Hanoi}(n - 1, src, tmp, dst)$
  * move disk $n$ from $src$ to $dst$
  * $\text{Hanoi}(n - 1, tmp, dst, src)$

For $\text{Hanoi}$, we either do nothing when $n = 0$, move only the one disk from source to destination when $n = 1$, or we have to move the $n - 1$ disks on top of the biggest disk before we can move the biggest disk from the source to destination. The call to $\text{Hanoi}(n - 1, src, tmp, dst)$ does that first move correctly by induction. We then get to move the whole stack of $n - 1$ in a single move since the disks are leaving a leaning needle.

For $\text{Hanoi}$, we don’t need to do anything when $n = 0$. Otherwise, we need to move all $n - 1$ disks from off the bigger disk to the other vertical needle, move the largest disk to the leaning needle, and then move the $n - 1$ disks back to the leaning needle. The two recursive calls do these bigger moves correctly by induction.

For the number of moves, we can write a pair of recurrence relations: $HV(n)$ denotes the number of moves for $\text{Hanoi}(n, src, dst, tmp)$ and $HL(n)$ denotes the number of moves for $\text{Hanoi}(n, src, dst, tmp)$. From the recursive call sizes in the psuedocode, we get

$$HV(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
HL(n - 1) + 2 & \text{otherwise}
\end{cases}$$

and

$$HL(n) = \begin{cases} 
0 & \text{if } n = 0 \\
HV(n - 1) + HL(n - 1) + 1 & \text{otherwise}
\end{cases}$$

Enumerating the first few cases, we can guess $HV(n) = 2F_{n+1} - 1$ and $HL(n) = 2F_{n+2} - 3$ for all $n \geq 1$. A simple inductive proof verifies the guesses, so we may conclude the algorithm performs $2F_{n+1} - 1$ moves for any non-empty stack of disks.

■
**Rubric:** 10 points total: 6 points for the algorithm, 2 points for justification, 2 points for number of moves analysis. Any analysis that results in one or more correct recurrences for the given algorithm is worth full credit. Any algorithm that uses $\Theta(F_n)$ moves is worth full credit.
Let \( n = 2^\ell - 1 \) for some positive integer \( \ell \). Suppose someone claims to hold an unsorted array \( A[1..n] \) of distinct \( \ell \)-bit strings; thus, exactly one \( \ell \)-bit string does not appear in \( A \). Suppose further that the only way we can access \( A \) is by calling the function \( \text{FetchBit}(i, j) \), which returns the \( j \)th bit of string \( A[i] \) in \( O(1) \) time. Describe a recursive algorithm to find the missing string in \( A \) using only \( O(n) \) calls to \( \text{FetchBit} \).

**Solution:** Our high level strategy will be to take a pass through the array checking the first bit and recursively search within the subset of strings that have one too few 0s or 1s in that first bit. To aid with the recursion, we’ll pass along an array of indices that we’re currently interested in (we don’t need random access, so a linked list of indices will do just as well). Observe that when we’re recursively checking the \( j \)th bit, there are only \( 2^{\ell-j+1} - 1 \) strings remaining. Procedure \( \text{FindString}(\text{elements}[1..m], j) \) takes a list of indices for strings to be searched and the current bit \( j \) (we’ll guarantee \( m = 2^{\ell-j+1} - 1 \)). We initially call \( \text{FindString}(\text{elements}, 1) \) where \( \text{elements} \) is an array containing the numbers 1 through \( n \).

```
\text{FindString}(\text{elements}[1..m], j):
  (m = 2^{\ell-j+1} - 1)
  if j = \ell
    return \text{elements}[1] with its last bit flipped
  else
    zeroes ← 0; ones ← 0
    for k ← 1 to m
      if \text{FetchBit}(\text{elements}[k], j) = 0
        zeroes ← zeroes + 1
        zeroElements[zeroes] ← \text{elements}[k]
      else
        ones ← ones + 1
        oneElements[ones] ← \text{elements}[k]
    if zeroes < ones
      return \text{FindString}(\text{zeroElements}, j + 1)
    else
      return \text{FindString}(\text{oneElements}, j + 1)
```

We will prove by induction on \( \ell - j \) that \( \text{FindString}(\text{elements}[1..m], j) \) finds the one string missing from a set of \( 2^{\ell-j+1} - 1 \) substrings pointed to by \( \text{elements} \) where each substring uses the last \( \ell - j + 1 \) bits. For \( \ell - j = 0 \), there is exactly one string missing, the one with a differing bit from the string present, and the algorithm returns it. For \( \ell - j > 0 \), there is exactly one fewer substrings starting with 0 or 1 than there are substrings starting with the other bit. That less represented bit must be the starting bit of the missing substring, and from the collection of substrings starting with that bit, exactly one is missing. The algorithm is correct to build a list of indices for that less represented bit and recursively look for the missing substring from the collection that starts with that bit. \( \ell - (j + 1) < \ell - j \), so the recursive calls work correctly by induction.

For the running time, observe that we spend \( O(m) \) time performing work outside the one recursive call on a set of elements half as large. We get a runtime recurrence of \( T(m) = T(m/2) + O(m) \). The \( i \)th level of the recursion tree sums to \( (1/2)^i m \), so the level sums form a decreasing geometric series bounded asymptotically by its largest term. We conclude
$T(m) = O(m)$. Since we initially call the procedure with $m = n$, the total running time is $O(n)$. \hfill \blacksquare

**Rubric:** 10 points total: 5 points for the algorithm, 3 points for justification, 2 points for the running time analysis.