(a) Truthfully write the phrase “I have read and understand the course policies.”

Solution:  *I have read and understand the course policies.*

Rubric: 1 point total.

(b) Give as tight an upper bound as you can for the amount of time it takes to perform \( \text{BIRDSONG}(\text{peep}[1 .. n]) \).

Solution: Each loop iterates over at most \( n \) values, and the “peep” loops is nested three levels in. Therefore, it takes at most \( n \cdot n \cdot n = O(n^3) \) time to sing the song.

Rubric: 3 points total.

(c) Give as tight a lower bound as possible for the amount of time it takes to perform \( \text{BIRDSONG}(\text{peep}[1 .. n]) \). correct.

Solution: Even if all the bits are set to FALSE, there are still \( \lfloor n/2 \rfloor \) iterations of the outer loop where the robin sings at least \( n/2 \) “chirps”. The singing time is at least \( \lfloor n/2 \rfloor \cdot n/2 = \Omega(n^2) \).

Rubric: 3 points total.

(d) If possible, give an asymptotically tight bound for the amount of time it takes to perform \( \text{BIRDSONG}(\text{peep}[1 .. n]) \), stating it as \( \Theta(g(n)) \) for some simple function \( g(n) \). If not possible, give a brief explanation for why not.

Solution: There is no tight bound that is correct for all singing times. Thanks to the peep bits, there is no matching lower and upper bound.

Rubric: 3 points total.
Prove the following claims (preferably using induction):

(a) For all non-negative integers $k$, a binomial tree of order $k$ has exactly $2^k$ nodes.

**Solution:** Let $k$ be any non-negative integer. Assume for any non-negative integer $k'$ with $k' < k$, a binomial tree of order $k'$ has exactly $2^{k'}$ nodes. If $k = 0$, then a binomial tree of order $k$ has exactly $1 = 2^k$ nodes. Otherwise, a binomial tree of order $k$ consists of two binomial trees of order $k - 1$. By the induction hypothesis, it has a total of $2^{k-1} + 2^{k-1} = 2^k$ nodes.

**Rubric:** 3 points total. -1/2 for missing the base case.

(b) For all positive integers $k$, attaching a new leaf to every node in a binomial tree of order $k - 1$ results in a binomial tree of order $k$.

**Solution:** Let $k$ be any positive integer. Assume for any positive integer $k'$ with $k' < k$, attaching a new leaf to every node of a binomial tree of order $k' - 1$ results in a binomial tree of order $k'$. If $k = 1$, then attaching a new leaf to the one node in a binomial tree of order 0 results in a binomial tree of order $1 = k$. Otherwise, suppose we add a new leaf to every node in a binomial tree of order $k - 2$ results in two binomial trees of order $k - 1$, again with one tree connected as a new child of the root of the other. By the induction hypothesis, attaching our new leaves to these two trees of order $k - 2$ results in two binomial trees of order $k - 1$ connected in such a way make a binomial tree of order $k$.

**Rubric:** 4 points total. -1 for missing the base case.

(c) For all non-negative integers $k$ and $d$, a binomial tree of order $k$ has exactly $\binom{k}{d}$ nodes with depth $d$. (Hence the name!)

**Solution:** For simplicity, we’ll use the convention that $\binom{k}{d} = 0$ if $d > k$. Let $k$ be any non-negative integer. Assume for any non-negative integer $k'$ with $k' < k$, a binomial tree of order $k'$ has exactly $\binom{k'}{d}$ nodes with depth $d$. If $k = 0$, then the binomial tree of order $k$ has exactly $1 = \binom{k}{0}$ nodes of depth 0 and $0 = \binom{k}{d}$ nodes of any depth $d > 0$. Otherwise, a binomial tree $T$ of order $k$ consists of two binomial trees $T_1, T_2$ of order $k - 1$, with the root of $T_2$ connected as a new child of the root of $T_1$. For any non-negative integer $d$, the nodes of depth $d$ in $T$ are nodes of depth $d$ in $T_1$ and nodes of depth $d - 1$ in $T_2$. Therefore, by the induction hypothesis, there are $\binom{k-1}{d} + \binom{k-1}{d-1} = \binom{k}{d}$ nodes with depth $d$ in $T$.

**Rubric:** 3 points total. -1/2 for missing the base case.
Using $\Theta$-notation, provide asymptotically tight bounds in terms of $n$ for the solution to each of the following recurrences.

(a) $T(n) = 8T(n/4) + n^{1.5}$

**Solution:** Each level of the recursion tree sums to $n$. There are $\Theta(\log n)$ levels, so $T(n) = \Theta(n^{1.5} \log n)$. ■

(b) $T(n) = 7T(n/2) + n^3$

**Solution:** The $i$th level of the recursion tree sums to $(7/8)^i n$. The level sums form a decreasing geometric series bounded by the largest term at the root level, so $T(n) = \Theta(n^3)$. ■

(c) $T(n) = 5T(n/3) + n$

**Solution:** The $i$th level of the recursion tree sums to $(5/3)^i n$. The level sums form an increasing geometric series bounded by the largest term at the leaf level. There are $\Theta(5^{\log_3 n}) = \Theta(n^{\log_3 5})$ leaves, so $T(n) = \Theta(n^{\log_3 5})$. ■

(d) $T(n) = 3T(n/5) + T(2n/5) + n$

**Solution:** Each level of the recursion tree sums to $n$. There are at most $\log_{5/2} n = O(\log n)$ levels in the tree, and at least $\log_5 n = \Omega(\log n)$ of these levels are full and actually sum up to $n$. Therefore, $T(n) = \Theta(n \log n)$. ■

(e) $T(n) = 3T(n/3) + n \log n$

**Solution:** The $i$th level of the recursion tree sums up to $n \log(n/3^i) = n \log n - i \log 3$. The level sums form an arithmetic series with $\Theta(\log n)$ terms whose largest term is $n \log n$. Therefore, $T(n) = \Theta(n \log^2 n)$. ■
Our goal for this problem is to design an algorithm to compute the depth of largest complete subtree of a given binary tree.

(a) What is the depth of the largest complete subtree of $T$ in terms of $\ell_1$, $r_1$, $\ell$, and $r$? Be sure to justify your answer.

Solution: If the largest complete subtree of $T$ includes $T$'s root, then it includes complete subtrees rooted at the left and right children of $T$'s root. However, these subtrees must be the same depth, and therefore cannot be deeper than either of the largest complete subtrees rooted at the left and right children of $T$'s root. In this case, the largest complete subtree in $T$ has depth $1 + \min\{\ell_1, r_1\}$.

If the largest complete subtree of $T$ does not include $T$'s root, then it must be the largest complete subtree existing entirely within either $T$'s left or right subtree and has depth $\ell$ or $r$. We want the largest complete subtree in $T$ without assumptions, so we need the best of all three options. The largest complete subtree of $T$ has depth $\max\{1 + \min\{\ell_1, r_1\}, \ell, r\}$.

Rubric: 3 points total: 2 points for expression; 1 point for justification.

(b) Describe a recursive algorithm that computes the depth of the largest complete subtree of a given binary tree $T$.

Solution: Taking a cue from part (a), we'll use a recursive algorithm that returns two pieces of information for each subtree of $T$. Procedure \textsc{FindDepth}(v) takes as its input a node $v$ of $T$ and returns a pair $(d_1, d)$ where $d_1$ is the depth of the largest complete subtree rooted at $v$ and $d$ is the depth of the largest complete subtree anywhere in $v$'s subtree.

\begin{verbatim}
\textbf{FindDepth}(v):
  if $v$ is a leaf
    return $(0, 0)$
  else if $v$ has exactly one child $c$
    $(d_1, d) \leftarrow \textsc{FindDepth}(c)$
    return $(0, d)$
  else
    $c_l \leftarrow v$'s left child
    $(\ell_1, \ell) \leftarrow \textsc{FindDepth}(c_l)$
    $c_r \leftarrow v$'s right child
    $(r_1, r) \leftarrow \textsc{FindDepth}(c_r)$
    return $(1 + \min\{\ell_1, r_1\}, \max\{1 + \min\{\ell_1, r_1\}, \ell, r\})$
\end{verbatim}

If $v$ is a leaf, then it is by itself the only complete subtree of its own subtree. The depth is 0. If $v$ has exactly one child $c$, then the only complete subtree rooted at $v$ is $v$ itself and has depth 0. However, there may be a larger complete subtree within $c$'s subtree. Procedure
FINDDEPTH(c) successfully returns the depth of its subtree’s largest complete subtree by induction on the number of nodes in a subtree. Finally, if v has left and right children cᵢ and cᵣ, respectively, then the two recursive calls to FINDDEPTH successfully find the four values from part (a) by induction on the number of nodes in a subtree. From the arguments in part (a), the algorithm then successfully computes both $d_i$ and $d$.

(c) What is the running time of your algorithm in terms of $n$, the number of nodes in $T$?

Solution: The algorithm performs a postorder traversal over the tree, spending constant time per each of the $n$ nodes. The running time is $O(n)$. 

Rubric: 5 points total: 4 points for the algorithm; 1 point for justification.
Suppose more than half the delegates belong to the same political party. Describe an efficient algorithm that identifies all members of this majority party.

**Solution:** We describe a procedure \textsc{FindMajority}(A[1 .. n]) that takes an array of \(n\) delegates where \(n \geq 1\). If more than half the delegates belong to the same political party, it will return a list containing all members of that party. Otherwise, it returns a list with at least one member of an arbitrary party. The procedure relies on a subroutine \textsc{SameParty}(x, y) that returns \textsc{True} if and only if delegates \(x\) and \(y\) belong to the same political party.

\begin{verbatim}
\textbf{FindMajority}(A[1 .. n]):
    majority ← an empty list
    if \(n = 1\)
        add \(A[1]\) to majority
        return majority
    else
        \(m \leftarrow \lfloor n/2 \rfloor\)
        \(majority \leftarrow \text{FindMajority}(A[1 .. m])\)
        \(rep_{l} \leftarrow \text{first member of } majority_{l}\)
        for \(i \leftarrow 1\) to \(n\)
            if \textsc{SameParty}(rep_{l}, A[i])
                add \(A[i]\) to majority
                if \(majority\) has more than \(n/2\) members
                    return \(majority\)
        \(majority \leftarrow \text{an empty list}\)
        \(majority_{r} \leftarrow \text{FindMajority}(A[m + 1 .. n])\)
        \(rep_{r} \leftarrow \text{first member of } majority_{r}\)
        for \(i \leftarrow 1\) to \(n\)
            if \textsc{SameParty}(rep_{r}, A[i])
                add \(A[i]\) to \(majority\)
        return \(majority\)
\end{verbatim}

We may assume by induction that recursive calls to \textsc{FindMajority} work correctly given they take subarrays of size less than \(n\). Now, suppose it is not true that more than half the delegates in \(A[1 .. n]\) belong to the same political party. In this case, \(n > 1\), and the attempt to return \(majority\) after the first recursive call fails. However, the algorithm will return at least one member of the party represented at the front \(majority_{r}\).

Now, suppose more than half the delegates in \(A[1 .. n]\) belong to the same political party. If \(n = 1\), the algorithm returns the one delegate and sole member of the only party. Otherwise, if strictly greater than \(m/2\) majority party members belong to \(A[1 .. m]\), then the first recursive call will return a list of those members. The algorithm will then find all members of that party and return them. If at most \(m/2 = \lfloor n/2 \rfloor/2\) members of the majority party belong to \(A[1 .. m]\), then more than \(n/2 - \lfloor n/2 \rfloor/2 \geq \lceil n/2 \rceil/2\) members of the majority party belong to \(A[m + 1 .. n]\). Even if the first recursive call fails to return a list of majority party members for \(A[1 .. n]\), the second one will, and the algorithm will successfully find all the majority party members for \(A[1 .. n]\).
The algorithm does $O(n)$ work outside of at most two recursive calls on arrays of size approximately $n/2$. Therefore, it follows the familiar runtime recurrence of $T(n) \leq 2T(n/2) + n$, which we know from MERGESORT solves to $O(n \log n)$.

**Rubric:** 10 points total: 5 points for the algorithm; 3 points for justification; 2 points for runtime analysis.