Binary Conversion} The following recursive algorithm uses the divide and conquer method to convert an $n$ bit binary integer $x_{n-1}...x_0$ into decimal. It uses the $O(n^{\log_2 3})$ time divide-and-conquer multiplication algorithm $\text{Multiply2}$ from class and the text; and the grade school linear time ($O(n)$) $\text{Add}$ algorithm as sub-routines. We assume $\text{Add}$ and $\text{Multiply}$ are defined to take decimal integers as input and output. Note that $2^n$, in binary, is a 1 followed by $n$ 0’s, so is easy to construct as a binary integer in linear time. Let $\text{ConstructPower2}$, given $n$, construct $2^n$ in binary in time $O(n)$.

$\text{ConvertToDecimal}(x_{n-1}...x_0$: Binary integer represented as an array of bits): decimal integer;

1. IF $n = 1$ return $x_0$.
2. $y \leftarrow x_{n-1}...x_{n/2}$
3. $z \leftarrow x_{n/2-1}...x_0$
4. $w \leftarrow \text{ConstructPower2}(n/2 - 1)$ (in binary)
5. $a \leftarrow \text{ConvertToDecimal}(y)$
6. $b \leftarrow \text{ConvertToDecimal}(z)$
7. $c \leftarrow \text{ConvertToDecimal}(w)$
8. $e \leftarrow \text{Add}(c, c)$
9. $d \leftarrow \text{Multiply2}(a, c)$
10. $e \leftarrow \text{Add}(d, b)$
11. Return $e$

First, give a proof that this algorithm is correct, by strong induction (5 points). Second, give a recurrence for the time this algorithm takes (5 points). Third, solve the recurrence to give a time analysis for this algorithm (5 points). Finally, think of a modification to this algorithm that would improve its running time (5 points, somewhat tricky).

The following recursive algorithm uses the divide and conquer method to convert an $n$ bit binary integer $x_{n-1}...x_0$ into decimal. It uses the $O(n^{\log_2 3})$ time divide-and-conquer multiplication algorithm $\text{Multiply2}$ from class and the text; and the grade school linear time ($O(n)$) $\text{Add}$ algorithm as sub-routines. We assume $\text{Add}$ and $\text{Multiply}$ are defined to take decimal integers as input and output. Note that $2^n$, in binary, is a 1 followed
by $n$ 0’s, so is easy to construct as a binary integer in linear time. Let ConstructPower2, given $n$, construct $2^n$ in binary in time $O(n)$.

ConvertToBinary($x_{n-1}$...$x_0$: Binary integer represented as an array of bits): decimal integer;
1. IF $n = 1$ return $x_0$.
2. IF $n = 2$ return $2x_1 + x_0$ as a single digit.
3. $y ← x_{n-1}$...$x_{n/2}$
4. $z ← x_{n/2-1}$..$x_0$
5. $w ←$ ConstructPower2($n/2 − 1$) (in binary)
6. $a ←$ ConvertToBinary($y$)
7. $b ←$ ConvertToBinary($z$)
8. $c ←$ ConvertToBinary($w$)
9. $c ←$ Add($c, c$
10. $d ←$ Multiply2($a, c$)
11. $e ←$ Add($d, b$
12. Return $e$

For the time analysis, we make three recursive calls, on $w, y$ and $z$. Now, $w, y, z$ are all $n/2$ bit binary integers. The time to construct them is $O(n)$. The results are decimal versions, and so have fewer digits (by about a log 10 factor). Thus $a, b, c, d$ are at most $O(n)$ digits each, so the time for the two Adds is $O(n)$ and the Multiply is $O(n \log 3)$. SO the total time out of the recursion is $O(n \log 3)$.

This gives $T(n) = 3T(n/2) + O(n \log 3)$ as the recurrence. This meets the format of the Master Theorem with $A = 3, B = 2, K = \log 3$. Then since $3 = 2^{\log 3}$, we are in the steady-state case, so $T(n) ∈ O(n^{\log 3} \log n)$.

We can prove that this algorithm is correct by strong induction on the size of the input. If the input is a single digit, the base two digit is also the base 10 representation. If it has two digits, the value of the binary representation is $2x_1 + x_0$. Since this is at most 3, and in particular, less than 10, we can output it as a single digit. If $n > 2$, assume the algorithm correctly outputs the number in decimal for inputs of all sizes 1..$n − 1$. In particular, it correctly converts numbers of size $n/2$. The value of the number $x$ is the value of $y$ times $2^n/2$, plus the value of $z$. By the induction hypothesis, $a$ represents the value of $y$ in decimal, as $b$ does $z$ and $c$ does the value of $2^n/2$ (after it is doubled). Therefore, by the correctness of the multiplication and addition algorithms, our final answer is the representation of $y2^{n/2} + z = x$, as required.
One thing we could do better is observe that for \( n = 2^k \), all the powers of 2 we use in the divide and conquer are actually for \( 2^i \), \( 0 \leq i \leq \log n \). So we could pre-compute all of these using \( \text{Exponent}[i] = \text{Multiply}[	ext{Exponent}(i - 1), \text{Exponent}(i - 1)] \) and then replace the recursive call to get \( c \) by setting \( c \) to be the precomputed element. This will remove the \( \log n \) factor from the order, which isn’t much. However, if we improve the Multiply algorithm, we also get the improvement in this algorithm, unlike the one we did first. So the limit might be using Fast-Fourier Multiplication in the modified algorithm sketched above.

**Least Common Ancestor: 20 points** Consider the following recursive algorithm that takes as input a binary tree \( T \). Each non-leaf in \( T \), \( x \), has left-child \( x.left \), and right child \( x.right \), and each non-root has parent \( x.parent \). (Child pointers at leaves and the parent pointer at the root return NIL). It uses a depth-first search procedure \( \text{DFS} \) that is linear-time in the size of the sub-tree and returns the list of nodes in the sub-tree. It computes, for each pair of nodes \( x \) and \( y \) in \( T \), the deepest node that is an ancestor of both \( x \) and \( y \), and stores it in an array \( \text{LCA}[x,y] \). The main idea is that if \( x \) is in the left sub-tree of the root, and \( y \) is in the right sub-tree, then the only common ancestor of \( x \) and \( y \) is the root. Otherwise, the least common ancestor is in the subtree that contains both \( x \) and \( y \).

LeastCommonAncestor(r: node)

1. \( \text{LCA}[r,r] \leftarrow r \)
2. IF \( r.left \neq \text{NIL} \) THEN
   
   3. \( \text{LeastCommonAncestor}(r.left) \)
   
   4. \( L_1 \leftarrow \text{DFS}(r.left) \)
5. IF \( r.right \neq \text{NIL} \) THEN
   
   6. \( \text{LeastCommonAncestor}(r.right) \)
   
   7. \( L_2 \leftarrow \text{DFS}(r.right) \)
8. FOR each \( x \in L_1 \)
   
   9. FOR each \( y \in L_2 \)
   
   10. \( \text{LCA}[x,y] \leftarrow r \)

First, give a recurrence relation for the time of this algorithm when the input is a complete binary tree of size \( n = 2^d - 1 \), where \( d \) is the depth of the tree. (Note that such a complete binary tree is always perfectly balanced, with left and right sub-trees of the same size.), and solve it to give a time analysis for the algorithm in the complete binary tree case. Then give a worst-case analysis for the time, not making any assumptions about the input tree.
Let \( l \) be the size of the left sub-tree and \( r \) the size of the right subtree. In all cases, \( n = l + r + 1 \).

Since we make one recursive call to the left sub-tree and one to the right sub-tree, and the rest of the algorithm is dominated by the two nested for loops, that go through all \( l \) vertices on the left and \( r \) vertices on the right, we get \( T(n) \leq T(l) + T(r) + c(lr + n) \). The \( O(n) \) term was added in case one of \( l, r \) is 0, and we need to take the \( DFS \) time into account. In the complete binary tree case, \( l = r = (n-1)/2 \) and this amounts to \( T(n) \leq 2T(n/2) + O(n^2) \), which meets our general formula with \( a = 2, b = 2, k = 2 \). Since \( b^k = 2^2 = 4 > 2 = a \), we are in the top-heavy case, and the total time is \( O(n^2) \).

We’ll use the guess-and-verify method to show that this is also the worst-case bound. Pick \( d \) to be the maximum of \( c \) and \( T(1) \). (This value of \( d \) was chosen after I wrote down the induction proof, to make each step go through.) We’ll show by strong induction that \( T(n) \leq dn^2 \) for \( n \geq 1 \). The base case is true by our choice of \( d \), \( T(1) \leq d1^2 \) since \( d \geq T(1) \). (And that was the reason why we chose \( d \) to be at least \( T(1) \).)

Assume that \( T(m) \leq dm^2 \) for any \( 1 \leq m \leq n-1 \). We’ll show that \( T(n) \leq dn^2 \). Then \( T(n) \leq T(l) + T(r) + c(lr + cn) \), where \( n = l + r + 1 \). If one of \( l \) or \( r \) are 0, we have \( T(n) \leq T(n-1) + cn = d(n-1)^2 + cn \leq dn^2 - 2dn + d + cn \), which is \( \leq dn^2 \) since \( n \geq 1 \) and \( d \geq c \) (one reason we picked \( d \) to be \( \geq c \).)

If both \( l \geq 1, r \geq 1 \), then we can apply the induction assumption to get

\[
T(l) \leq dl^2 \quad \text{and} \quad T(r) \leq dr^2.
\]

Then \( T(n) \leq T(l) + T(r) + c(lr + cn) \leq dl^2 + dr^2 + c(n) \leq d(l + r)^2 - 2dlr + c(n) \leq d(n - 1)^2 + cn \leq dn^2 - 2dn + d + cn \leq dn^2. \) (The first step is our recurrence for \( T \), the second step the induction hypothesis, the third step uses \( 2d > d \geq c \) and \( l + r = n - 1 \), and the last step uses \( n \geq 1 \) and \( d \geq c \).)

Thus, by strong induction, \( T(n) \leq dn^2 \) for all \( n \). Since \( d \) was a fixed constant, \( T(n) \in O(n^2) \).

**Triangles: 20 pts** On the calibration homework, we saw an \( O(nm) \) algorithm to compute whether a graph \( G \) had a triangle, three distinct nodes \( x, y, z \) so that any two were connected by an edge in \( G \). For large \( m \), this is \( O(n^3) \).

Use the Strassen Matrix Multiply algorithm in the text as a subroutine to give a faster algorithm for this problem, assuming the graph \( G \) is presented in adjacency matrix form.

First, I claim that there is a path of length 3 from \( x \) to itself in the graph if and only if the graph has a triangle containing \( x \). Suppose \( x, y, z \) is a triangle; then \( x \leftarrow y \leftarrow z \leftarrow x \) is such a path. Conversely, suppose \( x \leftarrow a \leftarrow b \leftarrow x \) is such a path. Then \( x \) must be connected to \( a \), \( a \) must be connected to \( b \), and \( B \) to \( x \), so they must all be different, and form a triangle.
Second, I claim that the \((u, v)\) th co-ordinate in \(M^k_G\) is positive if and only if there is a path of length \(k\) from \(u\) to \(v\) in \(G\). Note that no negative values occur in \(M^k_G\), so there will only be non-negative values in \(M^k_G\).

Then \(M^k[u, v] = \sum_w M^{k-1}[u, w] \cdot M[w, v]\) is non-zero if and only if there is some \(w\) so that \(M^{k-1}[u, w] > 0\) and \(M[w, v] > 0\). By induction on \(k\), this happens if and only if there is a path of length \(k - 1\) from \(u\) to \(w\) and an edge from \(w\) to \(v\), i.e., if and only if there is a path of length \(k\) from \(u\) to \(v\) whose penultimate step is \(w\).

Together, there is a triangle in \(G\) if and only if there is a non-zero co-ordinate \(M^3_G[x, x]\).

Therefore we can use the following algorithm:

Step 1: Multiply \(M\) by itself twice, using Strassen’s matrix multiplication algorithm from Section 2.5. This takes \(O(n^{\log_73}) = O(n^{2.8...})\) time. Step 2: For each \(i\), test whether the \(i, i\) th co-ordinate in the resulting product matrix is greater than 0. If so, halt and output, “Has a Triangle”. This step is \(O(n)\). Step 4: If not, output “No Triangles”.

The runtime is dominated by Step 2, so the total time is \(O(n^{\log_73})\).

**Implementation: 20 pts** Often, divide-and-conquer algorithms only become superior to asymptotically slower algorithms for large inputs, and are slower for smaller. A simple technique for improving their performance on small inputs is to use a threshold. Put in a larger base case in the recurrence, using a simpler but asymptotically slower algorithm when we fall below this threshold. In other words, for some threshold \(T\), we would use the recurrence if \(n > T\) and use a simpler algorithm if \(n \leq T\). Implement the \(O(n^{\log_33})\) time divide-and-conquer multiplication algorithm from class, and the grade-school multiplication algorithm. Then consider a hybrid algorithm using the technique above, where you replace the base-case of the recursion with the grade-school method for inputs of size less than some threshold \(T\). For the different thresholds \(T\), plot the average times to multiply random \(n\) bit numbers using the two methods (on log-log scale). Experimentally determine the best value of \(T\). Does this method only improve the running time on small inputs, or on all inputs? Explain.

The exact answer depends on details such as what programming language and operating system you use. However, a general phenomenon is that the best threshold is MUCH less than the size when the grade school method has the same time as the D&C recurrence. If you use this crossover point as the threshold, your time will roughly be the better of the two algorithms. But if you use a place where the grade school method is much better than the recurrence, that improves the recursive algorithm by almost the same ratio from then on.