**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 \textit{Multiply2} from class and the text; and the grade school linear time \((O(n))\) \textit{Add} algorithm as sub-routines. We assume Add and 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 \textit{ConstructPower2}, given \( n \), construct \( 2^n \) in binary in time \( O(n) \).

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

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 (10 points). Finally, think of a modification to this algorithm that would improve its running time (5 points, somewhat tricky).

**Binary Tree Isorphism**—25 points Consider the following recursive algorithm, which makes the following assumptions. \( x, y \) are the roots of two binary trees, \( T_x \) and \( T_y \). \textit{Left}(\( z \)) is a pointer to the left child of node \( z \) in either tree, and \textit{Right}(\( z \)) points to the right child. If the node doesn’t have a left or right child, the pointer returns “NIL”. Each node \( z \) also has a field \textit{Size}(\( z \)) which returns the number of nodes in the sub-tree rooted at \( z \).
\( \text{Size}(\text{NIL}) \) is defined to be 0. The algorithm \( \text{SameTree}(x, y) \) returns a boolean answer that says whether or not the trees rooted at \( x \) and \( y \) are the same if you ignore the difference between left and right pointers.

1. Program: \( \text{SameTree}(x, y: \text{Nodes}): \text{Boolean}; \)
2. \( \text{IF} \ \text{Size}(x) \neq \text{Size}(y) \ \text{THEN return} \ False; \text{halt}. \)
3. \( \text{IF} \ x = \text{NIL} \ \text{THEN return} \ True; \text{halt}. \)
4. \( \text{IF} \ (\text{SameTree}(\text{Left}(x), \text{Left}(y)) \ \text{AND} \ \text{SameTree}(\text{Right}(x), \text{Right}(y))) \)
\( \text{OR} \ (\text{SameTree}(\text{Right}(x), \text{Left}(y)) \ \text{AND} \ \text{SameTree}(\text{Left}(x), \text{Right}(y))) \)
\( \text{THEN return} \ True; \text{halt}. \)
5. \( \text{Return} \ False; \text{halt}. \)

Give a time analysis (up to order) for this program for the case when the trees rooted at \( x \) and \( y \) are both complete balanced trees with \( n \) nodes. (Every node \( z \) in a complete balanced tree has \( \text{Size}(\text{Left}(z)) = \text{Size}(\text{Right}(z)). \)) Then give a time analysis for the case of arbitrary trees of size \( n \).

**Bimodular array maximum** Say that an array of integers \( A[1...n] \) is bimodular if there is some \( 1 \leq I \leq n \) so that the array is increasing up to \( I \), and decreasing thereafter, i.e., \( A[1] < A[2] < A[I] > A[I+1] > .. > A[n] \). Give an efficient algorithm that given a bimodular array finds the maximum element (which will be the \( A[I] \) in the definition). Your algorithm should take substantially less than linear time, not even looking at most of the array. (10 points correctness, 15 points efficiency and time analysis.)

**Implementation: 25 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 \( O(n\log^2 n) \) 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.