Multiplication in Three Pieces: 20 pts. In class, we saw a divide and conquer algorithm for multiplication that divided each $n$ bit integer into high and low positions, each $n/2$ bits long. Consider algorithms that break the integers up into three pieces instead, the high order, mid order, and low order pieces, each $n/3$ bits long. What is the best divide-and-conquer multiplication algorithm of this type you can find? Is it better or worse than the two piece algorithm from class?

Base conversion: 20 points Consider the problem of converting a base 3 integer into decimal. Give an efficient algorithm (in terms of single digit operations) for this problem, trying to beat the $O(n^2)$ algorithm on the calibration homework. You can use the multiplication algorithm from class (or above) as a subroutine.

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 $DFS$ 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 $LCA[x,y]$. 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. $LCA[r,r] ← r$
2. IF $r.left ≠ NIL$ THEN
3.   LeastCommonAncestor($r.left$);
4.   $L_1 ← DFS(r.left)$;
5. IF $r.right ≠ NIL$ THEN
6.   LeastCommonAncestor($r.right$);
7.   $L_2 ← DFS(r.right)$;
8. FOR each $x ∈ L_1$
9.   FOR each $y ∈ L_2$
First, give a recurrence relation for the time of this algorithm when the input is a \emph{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 \emph{worst-case} analysis for the time, not making any assumptions about the input tree.

**Back-tracking: Hamiltonian path** Consider the following algorithm for deciding whether a graph has a Hamiltonian Path from \( x \) to \( y \), i.e., a simple path in the graph from \( x \) to \( y \) going through all the nodes in \( G \) exactly once. (\( N(x) \) is the set of neighbors of \( x \), i.e. nodes directly connected to \( x \) in \( G \)).

1. \( \text{HamPath}(G, x : \text{node}, y : \text{node}) \)
2. If \( x = y \) is the only node in \( G \) return \text{True}.
3. If no node in \( G \) is connected to \( x \), return \text{false}.
4. For each \( z \in N(x) \) do:
   5. If \( \text{HamPath}(G - \{x\}, z, y) \), return \text{true}.
6. Return \text{false}

a. Explain (informally) why this algorithm is correct. (5 points) b. If every node of the graph \( G \) has degree (number of neighbors) at most 3, how long will this algorithm take at most? (15 points) (Hint: you can get a tighter bound than the most obvious one.)

**Implementation: 20 pts** Implement a back-tracking algorithm for maximum independent set (such as from class). Run your algorithm on random graphs with edge probability \( 1/2 \) (as in the previous assignment) for \( n \) as many different powers of 2 as you can without using more than a day computer time on any one instance. How does the actual maximum independent set size compare to the size found by the greedy heuristic last assignment?