This document gives a sample problem and solution in order to demonstrate what constitutes a clearly-written and complete solution.

**Problem:** You’re a summer intern at Giigle. On your first day at work, the CEO comes to your office and tells you about your first task: “I like to throw company parties,” she says. “Your job is to write a program to help me figure out who to invite. As you know, Giigle has a hierarchical structure. You can think of it as a tree. The CEO, that’s me, is at the root of the tree. Below the root are supervisors, below them are managers, below them are team leaders, etc., etc., until you get down to the leaves - the summer interns. The tree is not necessarily binary; some non-leaf nodes may have one “child”, others two, and others even more. Anyhow, to make the party fun, I thought it best that we don’t invite an employee along with their immediate boss (their parent in the tree). My objective is to invite as many employees as possible without inviting an employee and their boss.”

Describe a **greedy algorithm** for this problem in clear English. Your algorithm should take a tree as input and return the largest set of employees in the tree that can be invited to the party subject to the constraint that if an employee is invited, their boss is not invited. You should assume that the tree is stored as a collection of nodes where each node has a reference to its parent, a count of the number of its children, and a list of references to its children. You may assume that the size of the tree is also given. Prove the correctness of your algorithm and give its worst-case running time.

**Solution:** Our greedy algorithm is based on the observation, which we will prove below, that it’s always good to invite the leaves of the tree to the party. Our greedy algorithm, therefore, begins by choosing some arbitrary leaf, \( v \), and inviting it. Next, the parent of \( v \), denoted \( p(v) \), must be removed from consideration. Thus, we remove \( p(v) \) and all its incident edges. This results in fragmenting the original tree into one or more smaller trees (including some that might be single vertices - e.g., any leaf siblings of \( v \)). Each of these smaller trees is now solved independently using the greedy algorithm.

We prove the correctness of this greedy algorithm using strong induction on the number of vertices \( n \) in the tree.

The basis is when \( n = 1 \). There is only one optimal solution (take the vertex!) and our greedy algorithm finds it.
The induction hypothesis is that the greedy algorithm is optimal for any tree with \( n \) or fewer vertices.

Now, in the induction step, we consider a tree with \( n + 1 \) vertices. The greedy algorithm chooses a leaf \( v \). To begin, we will show that the leaf \( v \) that we chose is part of some optimal solution. To show this, consider some optimal solution \( OPT \). Note that \( OPT \) is a set of vertices in the tree that can be invited to the party and, by assumption, there is no larger set than \( OPT \) that can be invited to the party. If \( v \in OPT \), we have shown that \( v \) is in some optimal solution. If \( v \notin OPT \) then consider \( p(v) \), the parent of \( v \). If \( p(v) \notin OPT \) then \( OPT \) is not optimal since we could invite \( v \) after all and increase the size of the solution. So, \( p(v) \in OPT \). Then, removing \( p(v) \) and adding \( v \) to \( OPT \) is still a valid solution and it’s a solution of the same size as \( OPT \) and is thus an optimal solution. So, we now know that there exists an optimal solution that includes \( v \).

The inclusion of \( v \) requires that we remove \( p(v) \) and all edges incident on \( p(v) \). This results in some number of smaller trees. Each of these smaller trees is independent of the other trees since they have no edges in common (we’ve removed those edges when we removed \( p(v) \)). Therefore, an optimal solution should choose the largest number of employees from each of those smaller trees. Our greedy algorithm will recurse on each of those trees and we know, by the induction hypothesis, that the greedy algorithm will find the optimal solution in each of those trees. Thus, the greedy algorithm finds an optimal solution to the original problem. Q.E.D.

Next, we describe an implementation of this greedy algorithm. We use depth-first-search (DFS) to find all the leaves in the tree and we place those leaves in a queue. While the leaf queue is not empty, we dequeue a leaf, \( v \), and add it to our solution. We find the parent (if it exists), \( p(v) \), of that leaf and remove it and its incident edges from the tree. Similarly, we find the grandparent of \( v \) (if it exists), \( p(p(v)) \), and decrease its number of children by 1. If it now has no children, it is added to the leaf queue.

Finally, we analyze the running time of the algorithm. We know that the DFS step takes time \( O(n + m) \) where \( n \) and \( m \) denote the number of vertices and edges, respectively. In a tree, \( m \in O(n) \), so this DFS step takes time \( O(n) \). Subsequently, each vertex is enqueued at most once and dequeued at most once, for a total of \( O(n) \) work maintaining the queue. In addition, the algorithm traverses each of the \( O(n) \) edges at most once. Thus, the total running time of this algorithm is \( O(n) \).