For these questions, you are given a tree as a linked data structure. Each tree node has a list of children. You may add one or more fields to the node data structure to help your algorithm.

1. Describe a recursive algorithm to find the maximum independent set of a tree. Your algorithm should take a root node as its input and output the size of the maximum independent set of the tree. (Unlike some past problems, you should not need to add any extra arguments to the recursive function, i.e., your recursive calls should all be of the form \( \text{MIS}(n) \) for some tree node \( n \)).

**Solution:**

```python
def MIS(root):
    """compute the size of the maximum independent set of
    the tree rooted at 'root' ""
    # base case
    if len(root.children) == 0:
        return 1
    # compute MIS that excludes root
    excludes = sum(MIS(child) for child in root.children)
    # compute MIS that includes root
    includes = 1 + sum(MIS(grandchild) for child in root.children
                       for grandchild in child.children)
    return max(includes, excludes)
```

2. Design and analyze a dynamic programming version of your algorithm. Hint: store the intended result in \( \text{node.mis} \)

**Solution:** We perform a pre-order traversal of the tree, computing the \( \text{node.mis} \) field of each node, which represents the MIS of the subtree rooted at that node. Note that we still use recursion to traverse the tree but the algorithm computes the MIS of each subtree exactly once.

```python
def MIS_DP(root):
    """compute the size of the maximum independent set of
    the tree rooted at 'root' and store it in 'root.mis' ""
    # base case
    if len(root.children) == 0:
        return 1
    for child in root.children:
        # compute MIS of the children
```
A slightly tricky part here is analyzing the performance. Note that \( \text{MIS\_DP} \) gets called exactly once for each tree node. However, the execution of the function is not constant time, since it iterates over all children and grandchildren. In the worst case, where a tree with \( n \) nodes has a single root node with \( n-1 \) children, we execute \( n-1 \) additions in \( \text{MIS\_DP}(\text{root}) \).

However, notice that each node participates in at most two sums: one in the \( \text{MIS\_DP} \) call of its parent, and one in the \( \text{MIS\_DP} \) call of its grandparent. (Except for the root and its children). Therefore, the total number of additions is still \( \Theta(n) \).

3. Consider now that each node has a weight, \( \text{node.w} \). Design an algorithm for finding (the weight of) the maximum weight independent set.

**Solution:** This is essentially the same algorithm, except that we use the weight of the node instead of 1.

```python
def MIS_weighted(root):
    # compute the weight of the maximum-weight independent set
    # of the tree rooted at 'root' and store it in 'root.wmis'

    # base case
    if len(root.children) == 0:
        return root.w
    for child in root.children:
        # compute MIS of the children
        MIS_DDP(child)
        # compute MIS that excludes root
        excludes = sum(child.wmis for child in root.children)
        # compute MIS that includes root
        includes = root.w + sum(grandchild.wmis for child in root.children
                                  for grandchild in child.children)
        root.wmis = max(includes, excludes)
```

4. Your goal in this problem is to assign each node one of three labels, 1, 2, or 3. Each node must have a different label than its parent. The cost of an assignment is the number red nodes that have a label smaller than their parent. For example, in the figure below, the red nodes are ones that have a label than their parent, and therefore the cost of this labeling would be 9.
Your goal is to find the (cost of) the minimal cost labeling of a given tree. (The labeling above is *not* the minimal cost one.)

**Solution:** We define 3 variables: `node1`, `node2`, and `node3` to represent the minimum cost of a tree rooted at node, where the root is labeled 1, 2, or 3, respectively. We can then recursively compute the costs based on the costs of the children:

```python
def min_cost_label(root):
    root1 = 0
    root2 = 0
    root3 = 0
    # base case
    if len(root.children) == 0:
        return 0
    for child in root.children:
        min_cost_label(child)
    root1 += min(child2, child3)
    root2 += min(1+child1, child3)
    root3 += min(1+child1, 1+child2)
    return min(root1, root2, root3)
```

Page 3