**Final Examples**

**Announcements**

**Trees**

**Tree-Structured Data**

```python
def tree(label, branches=[]):
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]
```

A tree can contain other trees:

```
[5, [6, [7]], [8, [9], 10]]
```

```
NP (JJ Short) (NNS cuts)
```

```
VP (VBP make) (NP (JJ long) (NNS delays))
```

Tree processing often involves recursive calls on subtrees.

```
<ul>
  <li>Midterm 1</li>
  <li>Midterm 2</li>
</ul>
```

**Solving Tree Problems**

Implement `bigs`, which takes a Tree instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than any labels of their ancestor nodes.

```python
def bigs(t):
    # Return the number of nodes in t that are larger than all their ancestors.
    >>>
    a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0), Tree(2)])])
    >>> bigs(a)
    4
```

The root label is always larger than all of its ancestors.

```python
# Somewhere track a list of ancestors
# Somewhere track the largest ancestor so far...
```

```python
f(t, )
```

```
Somehow track the largest ancestor so far...
```

```
Somehow increment the total count
```

**Recursive Accumulation**
Solving Tree Problems

Implement `bigs`, which takes a Tree instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than any labels of their ancestor nodes.

def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."""
    n = 0
    def f(a, x):
        # Somehow track the largest ancestor
        node, label > max_ancestors
        n += 1
        # Somehow increment the total count
        for b in a.branches:
            f(b, min(label, x))
    f(t, t.label - 1)
    return n

Somehow track the largest ancestor
Somehow increment the total count
Root label is always larger than its ancestors

How to Design Programs

From Problem Analysis to Data Definitions
Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

Signature, Purpose Statement, Header
State what kind of data the desired function consumes and produces. Formulate a concise answer to the question what the function computes. Define a stub that lives up to the signature.

Functional Examples
Work through examples that illustrate the function’s purpose.

Function Template
Translate the data definitions into an outline of the function.

Function Definition
Fill in the gaps in the function template. Exploit the purpose statement and the examples.

Testing
Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.

Designing a Function

Implement `smalls`, which takes a Tree instance `t` containing integer labels. It returns the non-leaf nodes in `t` whose labels are smaller than any labels of their descendant nodes.

def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."
    result = []
    def process(t):
        if t.is_leaf():
            return t.label
        else:
            smallest = min([process(b) for b in t.branches])
            if t.label < smallest:
                result.append(t.label)
            result += [t.label]
        return min(result)
    process(t)
    return result