Trees

A tree can contain other trees:

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

Tree processing often involves recursive calls on subtrees.

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

```
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
    """
```

```
def f(a, x):
    if node.label > max_ancestors:
        return 1 + sum(f(b, a.label) for b in a.branches)
    else:
        return sum(f(b, x) for b in a.branches)

f(t, 0)
```

```
if node.label > max_ancestors:
    Somehow track the largest ancestor
    Somehow increment the total count
else:
    Somehow track a list of ancestors
```

```
if True
    return ___
else:
    return ___ + sum(f(b, a.label) for b in a.branches)
```

```python
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)

def is_leaf(t):
    return not branches(t)

def branches(t):
    return t[1:]

def label(t):
    return t[0]

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

```
def label(t):
    return t[0]

def branches(t):
    return t[1:]

def is_leaf(t):
    return not branches(t)

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

Tree-Structured Data

```
def tree(label, branches=[]):
    return [label] + list(branches)
def label(t):
    return t[0]
def branches(t):
    return t[1:]
def is_leaf(t):
    return not branches(t)
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)
def is_leaf(self):
    return not self.branches
```

```
def tree(label, branches=[]):
    return [label] + list(branches)
def label(t):
    return t[0]
def branches(t):
    return t[1:]
def is_leaf(t):
    return not branches(t)
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)
def is_leaf(self):
    return not self.branches
```

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 f(a, x):
    if node.label > max_ancestors:
        return 1 + sum(f(b, a.label) for b in a.branches)
    else:
        return sum(f(b, x) for b in a.branches)

f(t, 0)
```

```
def f(a, x):
    if node.label > max_ancestors:
        return 1 + sum(f(b, a.label) for b in a.branches)
    else:
        return sum(f(b, x) for b in a.branches)
```

```
if node.label > max_ancestors:
    Somehow track the largest ancestor
    Somehow increment the total count
else:
    Somehow track a list of ancestors
```

Solving Tree Problems
Solving Tree Problems

Implement \( \text{bigs} \), 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 \text{bigs}(t):
    """Return the number of nodes in \( t \) that are larger than all their ancestors."""
    n = 0
    def f(a, x):
        nonlocal n
        if x < a.label:
            n += 1
        for b in a.branches:
            f(b, max(a.label, x))
    return n
```

Somehow track the largest ancestor node.\( \text{bigs}(\text{Tree}(1, [\text{Tree}(2, [\text{Tree}(4), \text{Tree}(5)]), \text{Tree}(3, [\text{Tree}(0, [\text{Tree}(6)])])])) \)

Somehow increment the total count.

Root label is always larger than its ancestors.

Designing Functions

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.

Applying the Design Process

Designing a Function

Implement \( \text{smalls} \), 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.

```python
def \text{smalls}(t):
    """Return the non-leaf nodes in \( t \) that are smaller than all their descendants."""
    result = []
    def process(t):
        process(t)
        return result
    return process(t)
```

Signature: \( \text{Tree} \to \text{List of Trees} \)

```
f(t, t.label) \text{ smallest label in a branch of t}
min([process(b) for b in t.branches])
```