Final Examples
Announcements
Trees
Tree-Structured Data
Tree-Structured Data

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

def label(tree):
    return tree[0]

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

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)
Tree-Structured Data

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

def label(tree):
    return tree[0]

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

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)
```
**Tree-Structured Data**

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

def label(tree):
    return tree[0]

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

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

A tree can contain other trees:
Tree-Structured Data

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

def label(tree):
    return tree[0]

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

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

A tree can contains other trees:
[5, [6, 7], 8, [[9], 10]]
Tree-Structured Data

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

def label(tree):
    return tree[0]

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

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

A tree can contain other trees:
[5, [6, 7], 8, [[9], 10]]
(+ 5 (- 6 7) 8 (* (- 9) 10))
Tree-Structured Data

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

def label(tree):
    return tree[0]

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

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

A tree can contains other trees:
[5, [6, 7], 8, [[9], 10]]
(+ 5 (- 6 7) 8 (* (- 9) 10))
(S
 (NP (JJ Short) (NNS cuts))
  (VP (VBP make)
    (NP (JJ long) (NNS delays)))
  (. .))
```
Tree-Structured Data

```python
def tree(label, branches=[]):
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)
```

A tree can contain other trees:

```
[5, [6, 7], 8, [[9], 10]]
(+ 5 (- 6 7) 8 (* (- 9) 10))
(S
  (NP (JJ Short) (NNS cuts))
  (VP (VBP make)
    (NP (JJ long) (NNS delays)))
  (.
   .))
```

```
<ul>
  <li>Midterm <b>1</b></li>
  <li>Midterm <b>2</b></li>
</ul>
```
Tree-Structured Data

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

def label(tree):
    return tree[0]

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

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

A tree can contain other trees:

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

(+ 5 (- 6 7) 8 (* (- 9) 10))

(S
   (NP (JJ Short) (NNS cuts))
   (VP (VBP make)
      (NP (JJ long) (NNS delays)))
   (. .))

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

Tree processing often involves recursive calls on subtrees
Tree Processing
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
"""
```

```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
"""
```
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
    """
```
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
    """
```
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 tree structure is as follows:

```
        1
         |
        3
         |
        4
         |   0
         |
        4
```

- 1
- 3
- 4
- 0
- 4
- 5 (checked)
- 2
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
    """
```

```plaintext
    1
   /|
  / 3
 /|
/ 0
/|
/|
/ 2
```

```plaintext
1
4
5
```
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
    """
```
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
    """
```
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
    """
```
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
```
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

    if t.is_leaf():
        return ___
    else:
        return ___([___ for b in t.branches])
```

```plaintext
<table>
<thead>
<tr>
<th>1</th>
<th>□</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>✔</td>
</tr>
<tr>
<td>5</td>
<td>✔</td>
</tr>
</tbody>
</table>
```

The root label is always larger than all of its ancestors.
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."""
    if t.is_leaf():
        return ___
    else:
        return ___([___ for b in t.branches])
```

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

    if t.is_leaf():
        return ___
    else:
        return ___([___ for b in t.branches])
```

Somehow increment the total count.
Implement \texttt{bigs}, which takes a Tree instance \texttt{t} containing integer labels. It returns the number of nodes in \texttt{t} whose labels are larger than any labels of their ancestor nodes.

\begin{verbatim}
def bigs(t):
    """Return the number of nodes in \texttt{t} that are larger than all their ancestors."
    if \texttt{t}.is_leaf():
        return ___
    else:
        return ___([___ for \texttt{b} in \texttt{t}.branches])

>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)]),])])

Somehow increment the total count
\end{verbatim}
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
if t.is_leaf():
    return ___
else:
    return ___([___ for b in t.branches])
```

```python
if node.label > max(ancestors):
    Somehow track a list of ancestors
    Somehow increment the total count
```
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."

    if t.is_leaf():
        return ___
    else:
        return ___([___ for b in t.branches])
```

The root label is always larger than all of its ancestors

```python
>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)]),)])
>>> bigs(a)
4
""
```

Somehow track a list of ancestors

```python
if node.label > max(ancestors):
    Somehow increment the total count
```

```python
if node.label > max_ancestors:
```

Somehow track a list of ancestors

```python
```

If the node is a leaf, return ___

```python
```
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
if t.is_leaf():
    return ___
else:
    return ___([___ for b in t.branches])
```

```python
if node.label > max(ancestors):
    somehow track a list of ancestors

if node.label > max_ancestors:
    somehow track the largest ancestor
```

```python
if node.label > max_ancestors:
    somehow increment the total count
```
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

    def f(a, x):
        if ________________:
            return 1 + _______________________

        else:
            return __________________________

    return ________________________________
```

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
    ""
    def f(a, x):
        if ____________________________________________:
            return 1 + ______________________________________
        else:
            return ______________________________________
    return ___________________________________________
```

Solving Tree Problems

Somehow track the largest ancestor
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
    """
    def f(a, x):
        if node.label > max_ancestors:
            return 1 + ____________________________:
                  node.label > max_ancestors
        else:
            return ______________________________
    return ________________________________
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."

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4

    def f(a, x):
        """Somehow track the largest ancestor""
        if a.label > x:
            return 1 + node.label > max_ancestors
        else:
            return

    return
```

[Diagram of a tree with labels and nodes marked with ✔️]
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
```

```python
def f(a, x):
    """A node in t
    
    node.label > max_ancestors
    
    if a.label > x:
        return 1 + ____________________________
    else:
        return ____________________________
    return ____________________________
```

Solving Tree Problems
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

    """  
    def f(a, x):
        """A node in `t` max_ancestor somehow track the largest ancestor"
        if a.label > x:
            return 1 + node.label > max_ancestors
        else:
            return _______________________

        return _______________________

    return f(t, ___________________)
```

A node in `t` max_ancestor somehow track the largest ancestor

`node.label > max_ancestors`

if `a.label > x`
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."""

    def f(a, x):
        """A node in t max_ancestor
        if a.label > x: node.label > max_ancestors"
        return 1 + ___________________________

        else:
            return ______________________________

    return f(t, ______________________________)
```

Some initial value for the largest ancestor so far...
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.""

    def f(a, x):
        if a.label > max_ancestors:
            return 1 + node.label > max_ancestors
        else:
            return Somehow increment the total count

    return Somehow track the largest ancestor

>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
>>> bigs(a)
4
```

A node in `t`... max_ancestor

Somehow increment the total count

Some initial value for the largest ancestor so far...
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
    ""

    def f(a, x):
        if a.label > x:  # Somehow track the largest ancestor
            return 1 + sum([f(b, a.label) for b in a.branches])
        else:
            return _ # Somehow increment the total count

    return ________________________________
```

Some initial value for the largest ancestor so far...
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
    """
    def f(a, x):
        if a.label > x:
            return 1 + sum([f(b, a.label) for b in a.branches])
        else:
            return sum([f(b, x) for b in a.branches])
    
    return f(t, )
```

Solving Tree Problems

A node in `t` with label `node.label` greater than the largest ancestor so far `max_ancestors`.

Somehow track the largest ancestor so far...

Somehow increment the total count so far...

Some initial value for the largest ancestor so far...
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
```

```python
def f(a, x):
    """A node in `t` has largest ancestor if `node.label > max_ancestors`.
    """  
    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])

return f(t, x)
```

Some initial value for the largest ancestor so far...
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.
    ""
    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])
    return f(t, t.label - 1)
```

Some initial value for the largest ancestor so far...

Somehow track the largest ancestor

A node in `t` max_ancestor

Somehow increment the total count

Root label is always larger than its ancestors

Some initial value for the largest ancestor so far...
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.""
    def f(a, x):
        if a.label > x:
            return 1 + sum([f(b, a.label) for b in a.branches])
        else:
            return sum([f(b, x) for b in a.branches])
    return f(t, t.label - 1)
```

```python
>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
>>> bigs(a)
4
```
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
    """

def f(a, x):
    """A node in t
    max_ancestor
    node.label > max_ancestors
    if a.label > x:
        return 1 + sum([f(b, a.label) for b in a.branches])
    else:
        return sum([f(b, x) for b in a.branches])
    return f(t, t.label - 1)
```

Somehow track the largest ancestor

Somehow increment the total count

Some initial value for the largest ancestor so far...

Root label is always larger than its ancestors

Somehow increment the total count

Some initial value for the largest ancestor so far...

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.
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
```

```python
def f(a, x):
    """A node in t

    >>> node.label > max_ancestors
    node.label > max_ancestors
    
    If `a.label > x`, return:

    return 1 + sum([f(b, a.label) for b in a.branches])

    Else, somehow increment the total count:

    return sum([f(b, x) for b in a.branches])

return f(t, t.label - 1)
```

Some initial value for the largest ancestor so far...

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

Somehow increment the total count

Some initial value for the largest ancestor so far...
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."
    def f(a, x):
        """A node in t
        node.label > max_ancestors
        """
        if a.label > x:
            return 1 + sum([f(b, a.label) for b in a.branches])
        else:
            return sum([f(b, x) for b in a.branches])
    return f(t, t.label - 1)
```

Some initial value for the largest ancestor so far...

Somehow track the largest ancestor

A node in `t`

`max_ancestor`

Root label is always larger than its ancestors

Somehow increment the total count
**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."

    def f(a, x):
        """A node in `t` is the largest ancestor if its label is greater than `x`.
        """
        if a.label > x:
            return 1 + sum([f(b, a.label) for b in a.branches])
        else:
            return sum([f(b, x) for b in a.branches])
        return f(t, t.label - 1)

>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
>>> bigs(a)
4
```

Solving Tree Problems

Somehow track the largest ancestor

Somehow increment the total count

Some initial value for the largest ancestor so far...
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.
    """
    if ____________________________:
        return 1 + _____________________
    else:
        return _______________________
    return _________________________

>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)]), [Tree(3)])])
>>> bigs(a)
4
```

Somehow track the largest ancestor

A node in `t`'s largest ancestor

Somehow increment the total count

Some initial value for the largest ancestor so far...

Root label is always larger than its ancestors

Some initial value for the largest ancestor so far...
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
    """
    def f(a, x):
        """A node in t
        if a.label > x:
            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])
        return f(t, t.label - 1)
```

Somehow track the largest ancestor

Somehow increment the total count

Some initial value for the largest ancestor so far...
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.
    ""
    def f(a, x):
        """A node in `t` whose label is larger than any of its ancestor labels.
        ""
        if a.label > x:
            return 1 + sum( f(b, a.label) for b in a.branches )
        else:
            return sum( f(b, x) for b in a.branches )
    return f(t, t.label - 1)
```

Some initial value for the largest ancestor so far...

Somehow track the largest ancestor.

Somehow increment the total count.

Root label is always larger than its ancestors.

Some node in `t` whose label is larger than any of its ancestor labels.

Solving Tree Problems
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.

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."""
    n = 0

    def f(a, x):
        _____________________________
        if __________________________:
            n += 1
        _____________________________:
        f(_______________________)
        _________________________________

    return n
```
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."""
    n = 0
    def f(a, x):
        # Somehow track the largest ancestor
        if a < x:
            n += 1
        f(a, a) # Ancestor
        f(f(a, x)) # Larger ancestor
    return n
```

```
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."""
    n = 0
    def f(a, x):
        Somehow track the largest ancestor

        if node.label > max_ancestors:
            n += 1

        f(a, x):

    return n
```

Somehow track the largest ancestor
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."""
    n = 0
    def f(a, x):
        # Somehow track the largest ancestor
        if node.label > max_ancestors:
            n += 1
        # Somehow increment the total count
        f()
    return n
```

Somehow track the largest ancestor

Somehow increment the total count
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."
    n = 0
    def f(a, x):
        """Somehow track the largest ancestor"
        if node.label > max_ancestors:
            n += 1
        else:
            f(_____________________)  # Somehow increment the total count
    return n  # Root label is always larger than its ancestors
```
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."""
    n = 0
    def f(a, x):
        if a.label > x:
            n += 1
        # Somehow track the largest ancestor
        # node.label > max_ancestors
        f(a, max(a.label, x))
        # Somehow increment the total count
        # Root label is always larger than its ancestors
    return n
```
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."""
    n = 0
    def f(a, x):
        if a.label > x:
            n += 1  # Somehow track the largest ancestor
        if f(a, x):
            f(___________________________)

    return n
f(t, t.label - 1)  # Root label is always larger than its ancestors
```

Somehow track the largest ancestor

Somehow increment the total count

Somehow track the largest ancestor

Root label is always larger than its ancestors
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."""
    n = 0
    def f(a, x):
        if a.label > x:
            n += 1
        for b in a.branches:
            f(b, x)
        f(t, t.label - 1)
    return n
```

- Somehow track the largest ancestor
- `node.label > max_ancestors`
- Somehow increment the total count
- `Root label is always larger than its ancestors`
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."""
    n = 0
    def f(a, x):
        if a.label > x:
            n += 1
        for b in a.branches:
            f(b, max(a.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
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."""
    n = 0

    def f(a, x):
        nonlocal n

        if a.label > x:
            n += 1

        for b in a.branches:
            f(b, max(a.label, x))

    f(t, t.label - 1)

    return n
```
Designing Functions
How to Design Programs

https://htdp.org/2018-01-06/Book/
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.

https://htdp.org/2018-01-06/Book/
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.

https://htdp.org/2018-01-06/Book/
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.

https://htdp.org/2018-01-06/Book/
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.

https://htdp.org/2018-01-06/Book/
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.

https://htdp.org/2018-01-06/Book/
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.

https://htdp.org/2018-01-06/Book/
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.

https://htdp.org/2018-01-06/Book/
Applying the Design Process
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.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        process(t)
    return result
```
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.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        process(t)
    return result
```

**Signature:** `Tree -> List of Trees`
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.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)]),])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]

    """
    result = []
    def process(t):

    process(t)
    return result
```

**Signature:** `Tree` $\rightarrow$ `List of Trees`
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.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        process(t)
    return result
```

*Signature: Tree -> List of Trees*
Implement \texttt{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.

\begin{verbatim}
def smalls(t):
    """Return the non-leaf nodes in \( t \) that are smaller than all their descendants."

    result = []
def process(t):
    process(t)
    return result

process(t)
return result
\end{verbatim}
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.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."""

    result = []
    def process(t):
        pass
    process(t)
    return result

>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
```

**Signature:** `Tree -> List of Trees`
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.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        process(t)
    return result
```

**Signature:** Tree → List of Trees

```python
>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
```
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.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]

    result = []
def process(t):

    process(t)
    return result
```

**Signature:** \( \text{Tree} \rightarrow \text{List of Trees} \)
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."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        process(t)
    return result
```

**Signature:** `Tree -> List of Trees`

**Signature:** `Tree -> number`
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.

```python
def smalls(t):
    """Return the non-leaf nodes in \( t \) that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        """Find smallest label in \( t \) & maybe add \( t \) to result"
        process(t)
    return result
```

**Signature:** Tree -> List of Trees

**Signature:** Tree -> number
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.

```python
def smalls(t):
    """Return the non-leaf nodes in `t` that are smaller than all their descendants.""
    result = []
    def process(t):
        """Find smallest label in `t` & maybe add `t` to result"
        if t.is_leaf():
            return t.label
        else:
            return min(...)
    process(t)
    return result
```

**Signature: Tree -> List of Trees**

```python
>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
```

**Signature: Tree -> number**
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.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)]]))])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        """Find smallest label in t & maybe add t to result""
        if t.is_leaf():
            return _________________________
        else:
            smallest = _______________________
            if ________________________________:
                _______________________________
                return min(smallest, t.label)
            process(t)
            return result
    return process(t)
```

**Signature:** `Tree -> List of Trees`

```python
>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)]]))])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
```

**Signature:** `Tree -> number`

```python
"Find smallest label in t & maybe add t to result"
```
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.

```python
def smalls(t):
    """Return the non-leaf nodes in `t` that are smaller than all their descendants."
    result = []
    def process(t):
        """Find smallest label in `t` & maybe add `t` to result"
        if t.is_leaf():
            return t.label
        else:
            smallest = [ ]
            if ____________________________:
                ________________________________
            return min(smallest, t.label)
    process(t)
    return result
```

**Signature:** `Tree -> List of Trees`

```python
>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
```

**Signature:** `Tree -> number`

"Find smallest label in `t` & maybe add `t` to result"
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.

```python
def smalls(t):
    """Return the non-leaf nodes in \( t \) that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])  # [0, 2]
```

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

```python
result = []
def process(t):
    """Find smallest label in \( t \) & maybe add \( t \) to result"
    if t.is_leaf():
        return t.label
    smallest = [t.label for t in smalls(t)]
    if smallest:
        return min(smallest, t.label)
    process(t)
return result
```

**Signature:** \( \text{Tree} \rightarrow \text{number} \)

"Find smallest label in \( t \) & maybe add \( t \) to result"
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.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    result = []
    def process(t):
        """Find smallest label in t & maybe add t to result"
        if t.is_leaf():
            return t.label
        else:
            smallest = t.label
            if t.label < smallest:
                return min(smallest, t.label)
            process(t)
    process(t)
    return result
```

**Signature:** `Tree -> List of Trees`

```python
>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
```

**Signature:** `Tree -> number`

```python
Signature: Tree -> number
if t.is_leaf():
    return t.label
else:
    smallest = smallest label in a branch of t
    if t.label < smallest:
        return min(smallest, t.label)
    process(t)
return result
```
Designing a Function

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

\begin{verbatim}
def smalls(t):
    """Return the non-leaf nodes in \texttt{t} that are smaller than all their descendants."
    result = []
    def process(t):
        """Find smallest label in \texttt{t} & maybe add \texttt{t} to \texttt{result}"""
        if t.is_leaf():
            return t.label
        else:
            smallest = process(t)
            if t.label < smallest:
                result.append(t.label)
            return min(smallest, t.label)
    process(t)
    return result
\end{verbatim}

\begin{verbatim}
>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])

>>> sorted([t.label for t in smalls(a)])
[0, 2]
\end{verbatim}
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.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."
    result = []
    def process(t):
        """Find smallest label in t & maybe add t to result"
        if t.is_leaf():
            return t.label
        else:
            smallest = 0
            if t.label < smallest:
                result.append(t)
            return min(smallest, t.label)
    process(t)
    return result
```

---

**Signature: Tree -> List of Trees**

```
>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
```

---

**Signature: Tree -> number**

```
>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> process(a)
[0, 2]
```

---

**Tree Representation**

```
1
  2       3
   |       | 4
  4
```

---

**Design Elements**

- **smallest label in a branch of t**
- **Return the non-leaf nodes in t that are smaller than all their descendants.**
- **Find smallest label in t & maybe add t to result**

---

**Commentary**

- The `smallest` variable is used to keep track of the smallest label encountered so far.
- The `process` function is a helper function that recursively processes the tree, updating the `result` list with non-leaf nodes that meet the criteria.

---

**Visual Aids**

- Diagram illustrating the tree structure and label values.
- Code examples demonstrating the function's usage and output.

---

**Additional Notes**

- The function is designed to handle nested trees efficiently, ensuring all non-leaf nodes with smaller labels are identified.
- The provided examples showcase the function's correctness and utility in practical scenarios.
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.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants.
    ""
    result = []
    def process(t):  # Find smallest label in t & maybe add t to result
        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)
            return min(smallest, t.label)
    process(t)
    return result
```

**Signature:** `Tree -> List of Trees`

```python
>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])[0]]))]
>>> sorted([t.label for t in smalls(a)])
[0, 2]
```

**Signature:** `Tree -> number`

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants.
    ""
    result = []
    def process(t):  # Find smallest label in t & maybe add t to result
        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)
            return min(smallest, t.label)
    process(t)
    return result
```
Expression Trees
Interpreter Analysis

How many times does scheme_eval get called when evaluating the following expressions?

```
(define x (+ 1 2))
```

```
(define (f y) (+ x y))
```

```
(f (if (> 3 2) 4 5))
```
Interpreter Analysis

How many times does scheme_eval get called when evaluating the following expressions?

```scheme
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```
How many times does scheme_eval get called when evaluating the following expressions?

\[
\text{(define x } (\text{+ 1 2}))
\]

\[
\text{(define (f y) } (\text{+ x y}))
\]

\[
(f \text{ (if (> 3 2) 4 5))}
\]
Interpreter Analysis

How many times does scheme_eval get called when evaluating the following expressions?

```
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```
Interpreter Analysis

How many times does scheme_eval get called when evaluating the following expressions?

```scheme
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```