Final Examples
Announcements
Trees
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
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

A tree can contain other trees:
Tree-Structured Data

```python
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

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(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

A tree can contains 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(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

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)))
  (.)())
```
Tree-Structured Data

```python
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

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)))
  (. .))

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

```python
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
```

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)))
  (.
   .))
```

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 all 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 \texttt{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.

\begin{verbatim}
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
\end{verbatim}
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.

```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)])])])
3
>>> 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 all 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 all 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
```

```latexegin{itemize}
\item \textbf{Solving Tree Problems}
\end{itemize}

Implement \textbf{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.

```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 all 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
4
5
1
```

```plaintext
4
5
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 all 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  
      |
   4 ☑  0
      |
     4  5 ☑ 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 all 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
```

```mermaid
diagram Tree
    1
    3
    ___
    0
    |   ___
    |   4
    |   5
    |       ___
    |       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 all labels of their ancestor nodes.

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."
    return 1 + sum(bigs(child) for child in t)
```

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

The root label is always larger than all of its ancestors

```python
1 ✓
4 ✓
0
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

The root label is always larger than all of its ancestors

```python
1 ✓
4 ✓
0
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
1 ✓
4 ✓
0
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
1 ✓
4 ✓
0
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
1 ✓
4 ✓
0
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])
```
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 all 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])
```

```python
>>> `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])
```
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 all 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 increment the total count

```python
if node.label > max(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 all 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])  # Somehow track a list of ancestors

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

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 all 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])

    if node.label > max(ancestors):
        Somehow track a list of ancestors
        Somehwow increment the total count
        if node.label > max_ancestors:

    4 5 2

```

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 all 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 track the largest ancestor
```

Somehow increment the total count

```python
if node.label > max_ancestors:
    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 all 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 all 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
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 all 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 + 
    else:
        return 

return 
```

Somehow track the largest ancestor

node.label > max_ancestors

return 

---

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.

```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 + 
    else:
        return 

return 
```

Somehow track the largest ancestor

node.label > max_ancestors

return 

---

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.

```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 + 
    else:
        return 

return 
```

Somehow track the largest ancestor

node.label > max_ancestors

return 

---

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.

```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 + 
    else:
        return 

return 
```

Somehow track the largest ancestor

node.label > max_ancestors

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 all 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 + node.label > max_ancestors
        else:
            return
    return
```

Somehow track the largest ancestor

![Tree Diagram]

```python
def f(a, x):
    if a.label > x:
        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 all 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 max_ancestor
        if node.label > max_ancestor:
            return 1 + node.label > max_ancestors
        else:
            return node.label > max_ancestors
        return
    return 1 + node.label > max_ancestors
```

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 all 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's label > max_ancestors"
        if a.label > max_ancestors:
            return 1 + f(t, )
        else:
            return f(t, )

    return f(t, )
```

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 all 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
        
        if a.label > max_ancestor:
            return 1 + node.label > max_ancestors
        else:
            return f(t, _______)

    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 all 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
        if a.label > x:
            return 1 + node.label > max_ancestors
        else:
            return # Somehow increment the total count
    return f(t, # Some initial value for the largest ancestor so far...)
```

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

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 all 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 max_ancestor
        if node.label > max_ancestor:
            return 1 + sum([f(b, a.label) for b in a.branches])
        else:
            return 1
        return f(t, some_initial_value_for_the_largest_ancestor_so_far...)
```

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 all 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's label is larger than `x` if its label is greater than the maximum ancestor."
        if a.label > x:
            node.label > max_ancestors
        return 1 + sum([f(b, a.label) for b in a.branches])
        # Somehow track the largest ancestor

        else:
            return sum([f(b, x) for b in a.branches])
            # Somehow increment the total count

    return sum([f(b, x) for b in a.branches])
    # 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 all 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, node.label > max_ancestors)
```

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 all 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
        **node** -> **max_ancestor**
        if `node.label` > `max_ancestor`:
            return 1 + \(\sum ([f(b, a.label) \text{ for } b \text{ in } a.branches])\)
        else:
            return \(\sum ([f(b, x) \text{ for } b \text{ in } a.branches])\)
        return f(t, t.label - 1)
    return f(a, 0)

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

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 all labels of their ancestor nodes.

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their 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, t.label - 1)
```

Somehow track the largest 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 all 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 node.label > x
    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

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 all 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 max_ancestor node.label > 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)
    return f
```

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
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 all 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
    max_ancestor
    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) Root label is always larger than its ancestors
```

Somehow track the largest 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 all 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 + 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)
```
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 all 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 with label `a.label` is big if its label is larger than `x` and all its 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)
```

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

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 all 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

        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 sum([f(b, t.label) for b in t.branches])
```

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

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 ________________:
            ________________
            ________________:
            ________________:
                f(______________)
            ________________
        ________________
    return n[0]
```
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 ________________________:
            ______________________
        ________________________:
            ________________________:
                f(____________________)
                ______________________
        return n[0]
```
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 node.label > max_ancestors:
            Somehwo track the largest ancestor
            f(_____________________)  
    f(_______________________)
    return n[0]
```
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 node.label > max_ancestors:
            Somehow track the largest ancestor
            Somehow increment the total count
            f(node)
            f(a)

    return n[0]
```

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

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:  # node.label > max_ancestors
            n[0] += 1  # Somehow increment the total count
            f(a)  # Somehow track the largest ancestor
        else:
            f(a)
    return n[0]
```

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):
    
    def f(a, x):
        if a.label > x:
            n[0] += 1
        elif f(a.left):
            return n[0]
        return n[0]

    n = [0]
    return f(t, -10000000000)
```

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):
    if a.label > x:
        n[0] += 1

    some_thing:
        f(some_thing)

f(t, t.label - 1)  # Root label is always larger than its ancestors
return n[0]
```
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.label > x:
        n[0] += 1

    # Somehow increment the total count
    for b in a.branches:
        f(b, max_ancestors)

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

return n[0]
```
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[0] += 1
        for b in a.branches:
            f(b, max(a.label, x))
    f(t, t.label - 1)
    return n[0]
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.

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.
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/
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/
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.

[link](https://htdp.org/2018-01-06/Book/)
Applying the Design Process
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
```

```
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 \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."

    >>> 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
\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."

    >>> 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 $\rightarrow$ List of Trees
Designing a Function

Implement `small`, 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
```

*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]
```

```
result = []
def process(t):
    process(t)
    return result
```
Designing a Function

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

>>> 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 \rightarrow \text{List of Trees}
\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):
        process(t)
        return result
```

**Signature:** Tree $\rightarrow$ 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 $\rightarrow$ number

```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]
```

- Tree node labels: 0, 1, 2, 3, 4, 5, 6.
- Non-leaf nodes smaller than all descendent labels: 0 and 2.

Diagram of tree structure:
```
        1
         |
         3
        /|
       2 0
      /  |
     4  5
    /    |
   4  5  6
```

The Diagram shows the tree structure with marked nodes 0 and 2 as the non-leaf nodes smaller than all descendent labels.
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"
        ... # process function
    process(t)
    return result
```

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

```python
def process(t):
    """Find smallest label in t & maybe add t to result"
```

**Signature:** `Tree -> number`
Designing a Function

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."

    >>> 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 t.label
        else:
            return min(...)
    process(t)
    return result
\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 ________________________________
        else:
            smallest = ________________________________
            if ________________________________:
                ________________________________
            return min(smallest, t.label)
    process(t)
    return result
```

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

```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.""
    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 _________________________:
                _________________________
            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`

```python
if t.is_leaf():
    return t.label
else:
    smallest = _________________________
    if _________________________:
        _________________________
    return min(smallest, t.label)
```

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):
        """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

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."
    result = []
    def process(t):
        """Find smallest label in \( t \) & maybe add \( t \) to result"
        if t.is_leaf():
            return t.label
        else:
            smallest = None
            if t.label < smallest:
                return min(smallest, t.label)
            process(t)
        return result
    return process(t)
```

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

Signature: `Tree` → `List of Trees`

Signature: `Tree` → `number`

```
>>> 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]
```

Diagram:

```
  1
 /|
/  |
3   \
/    |
2     0
/     |
4     5 6
 / \
4   5
 / \
[ 4 5 , 6 ]
```
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)
            return min(smallest, t.label)
    process(t)
    return result
```

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

**Signature:** Tree -> number
Interpreters
Interpreter Analysis

What expressions are passed to scheme_eval when evaluating the following expressions?

(define x (+ 1 2))

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

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

What expressions are passed to scheme_eval when evaluating the following expressions?

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

What expressions are passed to scheme_eval when evaluating the following expressions?

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

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

(f (if (> 3 2) 4 5))
```
What expressions are passed to `scheme_eval` when evaluating the following expressions?

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

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

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

What expressions are passed to scheme_eval when evaluating the following expressions?

(define x ((+ 1 2)))

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

(f (if (> 3 2) 4 5))
What expressions are passed to `scheme_eval` when evaluating the following expressions?

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

What expressions are passed to scheme_eval when evaluating the following expressions?

(\begin{align*}
&\text{(define x (+ 1 2))} \\
&\text{(define (f y) (+ x y))} \\
&\text{(f (if (> 3 2) 4 5))}
\end{align*})
Interpreter Analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

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

What expressions are passed to scheme_eval when evaluating the following expressions?

```
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```
What expressions are passed to scheme_eval when evaluating the following expressions?

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

What expressions are passed to `scheme_eval` when evaluating the following expressions?

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

What expressions are passed to `scheme_eval` when evaluating the following expressions?

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

What expressions are passed to `scheme_eval` when evaluating the following expressions?

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