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

```python
def tree(label, branches=[]):
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
```
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)
def tree(label, branches=[]):
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
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)

class BTree(Tree):
    empty = Tree(None)
    def __init__(self, label, left=empty, right=empty):
        Tree.__init__(self, label, [left, right])

@property
def left(self):
    return self.branches[0]

@property
def right(self):
    return self.branches[1]
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)

class BTree(Tree):
    empty = Tree(None)
    def __init__(self, label, left=empty, right=empty):
        Tree.__init__(self, label, [left, right])

@property
def left(self):
    return self.branches[0]

@property
def right(self):
    return self.branches[1]

A tree can contains 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)

class BTree(Tree):
    empty = Tree(None)
    def __init__(self, label, left=empty, right=empty):
        Tree.__init__(self, label, [left, right])

@property
def left(self):
    return self.branches[0]

@property
def right(self):
    return self.branches[1]

A tree can contains other trees:
[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)
class BTree(Tree):
    empty = Tree(None)
    def __init__(self, label, left=empty, right=empty):
        Tree.__init__(self, label, [left, right])
    @property
def left(self):
        return self.branches[0]
    @property
def right(self):
        return self.branches[1]
```

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)
class BTree(Tree):
    empty = Tree(None)
    def __init__(self, label, left=empty, right=empty):
        Tree.__init__(self, label, [left, right])
@property
def left(self):
    return self.branches[0]
@property
def right(self):
    return self.branches[1]
```

A tree can contain other trees:

```plaintext
[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

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)

class BTree(Tree):
    empty = Tree(None)
    def __init__(self, label, left=empty, right=empty):
        Tree.__init__(self, label, [left, right])
    @property
def left(self):
    return self.branches[0]
    @property
def right(self):
    return self.branches[1]

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>
```
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))
    (. .)))
```

Midterm <b>1</b>

Midterm <b>2</b>

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

---
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
    """
```

![Tree Diagram]

```plaintext
     1
    /|
   / |
  4  3
 /   |
4   5
```

4

1
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 any 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 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
"""
```
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

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
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
class Tree:
    def __init__(self, label, children):
        self.label = label
        self.children = children
```

```python
import sys

if __name__ == '__main__':
    from sys import stdin
    costs = [int(x) for x in stdin]
    n = len(costs)
    dp = [0] * (n + 1)
    for i in range(1, n + 1):
        dp[i] = costs[i - 1] + max(dp[j] for j in range(1, i) if costs[j] < costs[i - 1])
    print(dp[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."
    >>> 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.

```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 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])
```
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.
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

```python
if 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."

    >>> 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
        Somehow increment the total count
```

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

>>> 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])
```

- **Somehow track a list of ancestors**
- **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."

    >>> 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 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 _______________________________________________________
```

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."

    >>> 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
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):
        """Somehow track the largest ancestor\n        \[\text{node.label} > \text{max_ancestors}\]
        \[\text{if} \ a.\text{label} > x\]
        
        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)

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

        Somehow track the largest ancestor

        return 1 + node.label > max_ancestors

        else:

        return

        return
```

Somehow track the largest ancestor

A node in `t`
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.
    """
    # Some code here...

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

A node in `t` is labeled by `max_ancestor` if its label is greater than the label of any of its ancestors.

```python
def f(a, x):
    """A node in `t` is labeled by `max_ancestor` if its label is greater than the label of any of its ancestors.
    """
    if a.label > x:
        return 1 + node.label > max_ancestors
    else:
        return

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

    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):
    """A node in t max_ancestor
    if node.label > max_ancestors:
        return 1 + 
    else:
        Somehow increment the total count
    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.
    """
    >>> 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 None
    return f(t, None)  # Some initial value for the largest ancestor 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

    def f(a, x):
        """"""
        A node in t
        if node.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

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, node.label > max_ancestors)
```

Somehow track the largest ancestor

A node in `t` max_ancestor

If \(a.\text{label} > x\):

- Return \(1 + \sum(f(b, a.\text{label}) \text{ for } b \in a.\text{branches})\)

Else:

- Somehow increment the total count

Return \(\sum(f(b, x) \text{ for } b \in a.\text{branches})\)

Return \(f(t, \text{node.label} > \text{max_ancestors})\)

Root label is always larger than its 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 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, t.label - 1)
```

Somehow track the largest ancestor

A node in \( t \) if \( a.\text{label} > x \) \( \text{node.} \text{label} > \text{max_ancestors} \):

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

else:
\[
\text{return } \text{sum}([f(b, x) \text{ for } b \text{ in } a.\text{branches}])
\]

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

Root label is always larger than its ancestors

Somehow increment the total count

Somehow track the largest ancestor

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):
        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 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 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)

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

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):
        """A node in t whose label is larger than 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)

>>> 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 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 \( \text{max}_\text{ancestor} \) 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)
```

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

Somehow track the largest ancestor

Somehow increment the total count

A node in `t` max_ancestor

Node label is always larger than its ancestors

Root label is always larger than its ancestors
Solving Tree Problems

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 any labels of their ancestor nodes.

\begin{verbatim}
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 > \text{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)

>>> 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 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 node.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
```

Somehow track the largest ancestor

A node in `t` node.label > max_ancestors

if node.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)  # 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):
        '''Somehow track the largest ancestor

        A node in t
        ```
        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)
```

```python
>>> 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

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):
        """A node in `t` has 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)
```

```python
>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
>>> bigs(a)
4
```
Recursive Accumulation
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 any labels of their ancestor nodes.

\begin{verbatim}
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
\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."""
    n = 0
    def f(a, x):
        # Somehow track the largest ancestor
        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 node.label > max_ancestors:
            n += 1
        f(
        return n
```

Somehow track the largest ancestor

[node.label > max_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):
        # 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
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 x.label > max_ancestors:
            n += 1
        # Somehow increment the total count
        f(a, a.parent)
    # 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
        f(a, node.label > max_ancestors)
    f(a, node.label > max_ancestors)
    return n
```

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 any labels of their ancestor nodes.

```python
def bigs(t):
    
    n = 0
    
    def f(a, x):
        
        if a.label > x:
            n += 1
        
        f(a.right, max(a.label, x))
        
        f(t, t.label - 1)
        
    return n
```

Root label is always larger than its ancestors

Somehow track the largest ancestor

Somehow increment the total count

Node label is always larger than 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 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 += 1
        for b in a.branches:
            f(b, a.label - 1)
        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
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 \texttt{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.

\begin{verbatim}
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
\end{verbatim}
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.
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.
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.

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 `smallst`, 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

>>> 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`
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):
        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]
    ""

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."

    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.""
    result = []
    def process(t):
        # Your implementation here
    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."

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

    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 -> 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.""

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

    def sort(t):
        return [t.label for t in smalls(a)]

    >>> 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
    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"
        for t in smalls(a))
    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`

**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 t.label
        else:
            return min(...)
    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 \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."

    >>> 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 \texttt{t} & maybe add \texttt{t} to result"
        if t.is_leaf():
            return ________________________________
        else:
            smallest = ________________________________
            if ________________________________:
                ________________________________
            return min(smallest, t.label)
    process(t)
    return result
\end{verbatim}

\textbf{Signature: Tree $\rightarrow$ List of Trees}

\textbf{Signature: Tree $\rightarrow$ 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 t.label
        else:
            smallest =
            if smallest:
                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."""

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

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

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

>>> 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 = min(t.label, *[process(node) for node in t])
            if t.label < smallest:
                result.append(t.label)
            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.""

    # Signature: Tree -> List of Trees
    if t.is_leaf():
        return [t.label]
    else:
        smallest = []
        for t in smalls(t):
            if t.label < smallest:
                result.append(t)
        return min(smallest, t.label)

    # Signature: Tree -> number
    """Find smallest label in t & maybe add t to result"
    return smallest label in a branch of t

>>> 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]
```

---

1. 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.""

    # Signature: Tree -> List of Trees
    if t.is_leaf():
        return [t.label]
    else:
        smallest = []
        for t in smalls(t):
            if t.label < smallest:
                result.append(t)
        return min(smallest, t.label)

    # Signature: Tree -> number
    """Find smallest label in t & maybe add t to result"
    return smallest label in a branch of t

>>> 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]
```

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

The function `smalls(t)` finds the non-leaf nodes in a Tree `t` that are smaller than all their descendants. It returns these nodes as a list. The implementation involves iterating through each node, checking if it is a leaf, and if not, comparing its label with the smallest label of its children. If the current node's label is smaller, it is added to the result list. Finally, it returns the minimum of the smallest label and the current node's label.