Final Examples
Announcements
Trees
Tree-Structured Data
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 contains other trees:
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
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 contains other trees:

```
[5, [6, 7], 8, [[9], 10]]
(+ 5 (- 6 7) 8 (* (- 9) 10))
```

```
(S
  (NP (JJ Short) (NNS cuts))
  (VP (VBP make)
    (NP (JJ long) (NNS delays)))
  (. .))
```
Tree-Structured Data

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)))
  )

<ul>
  <li>Midterm <b>1</b></li>
  <li>Midterm <b>2</b></li>
</ul>
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 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 **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
    ""
```
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
    """
```
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
"""
```
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 0 2
\|--
  \|
\|--
  \|
4 5
```
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
    """

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

☑

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.
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])

>>> 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
```

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 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])
```

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

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

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

Implement `bigs`, which takes a Tree instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than 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 node.label > max(ancestors):
    somehow track a list of ancestors
    somehow increment the total count
```

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

```python
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
    """

    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

if node.label > max(ancestors):
    ```
    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 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 ___________________________________________________
```

---

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

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

def f(a, x):
    if ______________________________________________________:
        return 1 + __________________________________________
    else:
        return ______________________________________________
    return _____________________________________________________
```
Solving Tree Problems

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

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."
    
    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    
    def f(a, x):
        if node.label > max_ancestors:
            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):
        """Somehow track the largest ancestor
        node.label > max_ancestors"

        if a.label > x:
            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):
        """A node in t that is larger than all its ancestors."
        if a.label > x:
            return 1 + ___________________________
        else:
            return _______________________________
    return _____________________________________
```

**Solving Tree Problems**

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

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."
    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)]]))])
    >>> bigs(a)
    4
    >>>
    def f(a, x):
        """A node in t that is larger than all its ancestors."
        if a.label > x:
            return 1 + ___________________________
        else:
            return _______________________________
    return _____________________________________
```
Solving Tree Problems

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

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

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

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

    return __________________________
```

Somehow track the largest ancestor

A node in `t` has a `max ancestor` which is the largest ancestor of the node.

- `node.label > max_ancestors` checks if the node's label is larger than the `max ancestor`.
- If true, return 1 and add to the count.
- If false, return the result of calling `f` recursively.

Return the result of `f(t, )`.
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."

    >>> 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 \( t \) maximally ancestor
    if \( a \).label > x < node.label > max_ancestors:
        return 1 + ______________________________
    else:
        return _______________________________

    return _______________________________

return f(t, _______________________________
\end{verbatim}

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 1 + ____________________________

    elif ________________________________:
        return ______________________________

    return _______________________________
```

```python
>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
>>> bigs(a)  # Some initial value for the largest ancestor so far...
```

---

-A node in \( t \) containing integer labels. It returns the number of nodes in \( t \) whose labels are larger than any labels of their ancestor nodes.

```python
def f(a, x):
    
    elif node.label > max_ancestors:
        return ______________________________

    return _______________________________
```

---

A node in \( t \) containing integer labels. It returns the number of nodes in \( t \) whose labels are larger than any labels of their ancestor nodes.

---

-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."

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

return f(t, )
```

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

```python
def bigs(t):
    """Return the number of nodes in `t` that are larger than all their ancestors.
    """
    def f(a, x):
        if 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, max_ancestor if a.label > x else x)
```

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

    def f(a, x):
        """A node in \( t \)\'s largest ancestor

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

    return f(t, node.\text{label} > max_ancestors)
\end{verbatim}

Somehow track the largest ancestor

\begin{itemize}
  \item \texttt{sum([f(b, a.label) for b in a.branches])}
  \item \texttt{sum([f(b, x) for b in a.branches])}
  \item Some initial value for the largest ancestor so far...
\end{itemize}

A node in \( t \)\'s largest ancestor

Root label is always larger than its ancestors

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."
    return sum([f(b, a.label) for b in a.branches])
```

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

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

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

\begin{verbatim}
def \texttt{bigs}(t):
    ""
    Return the number of nodes in \texttt{t} that are larger than all their ancestors.
    ""
    def \texttt{f}(a, x):
        ""
        A node in \texttt{t} \ \texttt{max\_ancestor}
        if \texttt{a.label} > \texttt{x}:
            return 1 + \texttt{sum([f(b, a.label) for b in a.branches])}
        else:
            return \texttt{sum([f(b, x) for b in a.branches])}
    return \texttt{f(t, t.label - 1)}

>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
>>> \texttt{bigs(a)}
4
\end{verbatim}
Solving Tree Problems

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

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

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

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

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

Somehow track the largest ancestor

Somehow increment the total count

Some initial value for the largest ancestor so far...
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."
    return 1 + sum([f(b, a.label) for b in a.branches])

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

Somehow track the largest ancestor:

\[ \text{node.label} > \text{max}_\text{ancestors} \]

Somehow increment the total count:

\[ \sum \text{[f(b, \text{node.label}) for b in \text{node.branches}]} \]

Root label is always larger than its ancestors:

\[ \text{t.label - 1} \]

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 $a$.label > x node.label > max_ancestors:
        return sum([f(b, a.label) for b in a.branches])
        else:
            Somehow increment the total count
            return sum([f(b, x) for b in a.branches])
    return f(t, t.label - 1)  Root label is always larger than its ancestors

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

Somehow track the largest ancestor

Some initial value for the largest ancestor so far...
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 \( t \) whose label is larger than any ancestor.
        ""
        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):
        """Return the number of nodes in t that are larger than all their ancestors."
        max_ancestor = node.label > max_ancestors
        return 1 + sum([f(b, a.label) for b in a.branches])

    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...
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 increment the total count""
        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...

A node in \( t \) whose label is always larger than its ancestors

Somehow track the largest ancestor

A node in \( t \) whose label is larger than all its ancestors

Somehow track the largest ancestor

Root label is always larger than its ancestors

Somehow increment the total count

Root label is always larger than its ancestors

Some initial value for the largest ancestor so far...
Recursive Accumulation
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):
        _____________________________
        _____________________________:
        n += 1
        _____________________________:
        f(_______________________)
        _____________________________
    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):
        # 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(a, x)
        # f(a, x)

    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):
        if node.label > max_ancestors:
            n += 1
        if somehow track the largest ancestor:
            f(____________________)
        if somehow increment the total count:
            f(____________________)
    return n
```

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

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

Somehow track the largest ancestor

Somehow increment the total count

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

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."""
    n = 0
    def f(a, x):
        if a.label > x:
            n += 1
        if node.label > max_ancestors:
            f(_____________)
            return n
```

Somehow track the largest ancestor
Somehow increment the total count
Root label is always larger than its ancestors
Implement `bigs`, which takes a Tree instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than any labels of their ancestor nodes.

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."""
    n = 0
def f(a, x):
    """Somehow track the largest ancestor"""
    if a.label > x:
        n += 1
    else:
        f(a.label > max_ancestors)
        f(t, t.label - 1)
    return n
f(t, t.label - 1)
```

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

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

Somehow track the largest ancestor

Somehow increment the total count

Root label is always larger than its ancestors
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.
How to Design Programs

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

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

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

https://htdp.org/2018-01-06/Book/
How to Design Programs

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

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

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

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

https://htdp.org/2018-01-06/Book/
How to Design Programs

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

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

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

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

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

https://htdp.org/2018-01-06/Book/
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 **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
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):
        # ...
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 **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 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]

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]
```

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

**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""
        result
    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**

```
[[2], [0], [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:
            return min(...)  
    process(t)
    return result
```

Signature: `Tree --> List of Trees`

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

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

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

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

```
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants.
    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    ""

    result = []
    def process(t):
        """Find smallest label in t & maybe add t to result"
        if t.is_leaf():
            return __________________________
        else:
            smallest = __________________________
            if __________________________:
                __________________________
            return min(smallest, t.label)
    process(t)
    return result
```
Implement `smallss`, 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 smallss(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants.
    \[0, 2\]
    """
    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
```
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
            in a branch of t
            if True:
                return min(smallest, t.label)
            process(t)
    return result
```

```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 -> List of Trees`

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

Find smallest label in `t` & maybe add `t` to `result`
Implement `smalls`, which takes a Tree instance `t` containing integer labels. It returns the non-leaf nodes in `t` whose labels are smaller than any labels of their descendant nodes.

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

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

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

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

---

**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 for t in smalls(t)]
            if smallest:
                return min(smallest, t.label)
            else:
                return t.label
    return process(t)
```

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

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

---

**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 for t in smalls(t)]
            if smallest:
                return min(smallest, t.label)
            else:
                return t.label
    return process(t)
```

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

```python
>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
```
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:
            smallest = t.label
            if t.label < smallest:
                result.append(t.label)
            return min(smallest, t.label)
    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):
        """Find smallest label in `t` & maybe add `t` to result""
        if t.is_leaf():
            return t.label
        else:
            smallest = None
            if t.label < smallest:
                result.append(t)
            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."

    >>> 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 = 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**

```text
1
   3
  2 0
4 5 6
```

- **smallest label in a branch of t**
- **if** `t.label < smallest`
- **return** `min(smallest, t.label)`

---

14
Society
Privacy Policies and Laws
Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010

"People have really gotten comfortable not only sharing more information and different kinds, but more openly and with more people. That social norm is just something that has evolved over time."
Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010
"People have really gotten comfortable not only sharing more information and different kinds, but more openly and with more people. That social norm is just something that has evolved over time."

Tim Cook in Brussels, October 24, 2018
Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010
"People have really gotten comfortable not only sharing more information and different kinds, but more openly and with more people. That social norm is just something that has evolved over time."

Tim Cook in Brussels, October 24, 2018
"We at Apple are in full support of a comprehensive federal privacy law in the United States. There, and everywhere, it should be rooted in four essential rights:
Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010
"People have really gotten comfortable not only sharing more information and different kinds, but more openly and with more people. That social norm is just something that has evolved over time."

Tim Cook in Brussels, October 24, 2018
"We at Apple are in full support of a comprehensive federal privacy law in the United States. There, and everywhere, it should be rooted in four essential rights:
• First, the right to have personal data minimized. Companies should challenge themselves to de-identify customer data, or not to collect it in the first place."
Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010

"People have really gotten comfortable not only sharing more information and different kinds, but more openly and with more people. That social norm is just something that has evolved over time."

Tim Cook in Brussels, October 24, 2018

"We at Apple are in full support of a comprehensive federal privacy law in the United States. There, and everywhere, it should be rooted in four essential rights:

• First, the right to have personal data minimized. Companies should challenge themselves to de-identify customer data, or not to collect it in the first place.
• Second, the right to knowledge. Users should always know what data is being collected and what it is being collected for. This is the only way to empower users to decide what collection is legitimate and what isn’t. Anything less is a sham."
Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010
"People have really gotten comfortable not only sharing more information and different kinds, but more openly and with more people. That social norm is just something that has evolved over time."

Tim Cook in Brussels, October 24, 2018
"We at Apple are in full support of a comprehensive federal privacy law in the United States. There, and everywhere, it should be rooted in four essential rights:

• First, the right to have personal data minimized. Companies should challenge themselves to de-identify customer data, or not to collect it in the first place.
• Second, the right to knowledge. Users should always know what data is being collected and what it is being collected for. This is the only way to empower users to decide what collection is legitimate and what isn’t. Anything less is a sham.
• Third, the right to access. Companies should recognize that data belongs to users, and we should all make it easy for users to get a copy of, correct, and delete their personal data."
Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010

"People have really gotten comfortable not only sharing more information and different kinds, but more openly and with more people. That social norm is just something that has evolved over time."

Tim Cook in Brussels, October 24, 2018

"We at Apple are in full support of a comprehensive federal privacy law in the United States. There, and everywhere, it should be rooted in four essential rights:

• First, the right to have personal data minimized. Companies should challenge themselves to de-identify customer data, or not to collect it in the first place.

• Second, the right to knowledge. Users should always know what data is being collected and what it is being collected for. This is the only way to empower users to decide what collection is legitimate and what isn’t. Anything less is a sham.

• Third, the right to access. Companies should recognize that data belongs to users, and we should all make it easy for users to get a copy of, correct, and delete their personal data.

• And fourth, the right to security. Security is foundational to trust and all other privacy rights."
Perils of Sharing
Perils of Sharing

A persistent source of privacy breaches: sending a message to an unintended recipient
Perils of Sharing

A persistent source of privacy breaches: sending a message to an unintended recipient

Grandmas keep accidentally tagging themselves as Grandmaster Flash on Facebook
Perils of Sharing

A persistent source of privacy breaches: sending a message to an unintended recipient

Grandmas keep accidentally tagging themselves as Grandmaster Flash on Facebook
Software
Automated Decision Making
Automated Decision Making

What should the self-driving car do?
Automated Decision Making

What should the self-driving car do?
Life