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
Tree-Structured Data

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

def label(t):
    return t[0]

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

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

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

    def is_leaf(self):
        return not self.branches
```

A tree can contain other trees:
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]]
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

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

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

```python
defsolving_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.

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
    """
```

---

```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
"""
```
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)])[0]])])
    >>> 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
    """

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])
```
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.
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 all labels of their ancestor nodes.

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

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 all labels of their ancestor nodes.

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

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

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

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

Somehow increment the total count.
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

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

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])
```

The root label is always larger than all of its ancestors

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

Sometimes track a list of ancestors

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

if 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 all labels of their ancestor nodes.

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

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

    The root label is always larger than all of its ancestors

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

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

Somehow increment the total count
Solving Tree Problems

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

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

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

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

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

def f(a, x):
    if ________________________________________________________________:
        return 1 + ______________________________________________________
    else:
        return ______________________________________________________

    return ______________________________________________________
```

Solving Tree Problems

Somehow track the largest ancestor
Implement \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.

```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 ___________________________
```

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)
```

Somehow track the largest ancestor

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

Implement the function `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.
    """

    # Some example tree structure:
    >>> 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
        if a.label > x < node.label > max_ancestors:
            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)])])])
    print(bigs(a))

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

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 + node.label > max_ancestors
        else:
            return Somehow increment the total count
    return Somehow track the largest ancestor
```

A node in `t` with label `x`...

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

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 f(t, node.label > max_ancestors)
```

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

A node in `t`:

- **max_ancestor**: The largest ancestor so far.
- **node.label > max_ancestors**: A node in `t` whose label is greater than the largest ancestor so far.

```python
def f(a, x):
    """Return 1 + sum([f(b, a.label) for b in a.branches]) if a.label > x, else sum([f(b, x) for b in a.branches])."
    return 1 + sum([f(b, a.label) for b in a.branches]) if a.label > x else sum([f(b, x) for b in a.branches])
```

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:  # 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 sum([f(t, node.label) for node in t])
```

Somehow track the largest ancestor.

A node in `t`.

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.
    ""

    >>> 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 sum([f(b, t.label - 1) for b in t.branches])
```

Somehow track the largest ancestor

A node in `t`

$max_\text{ancestor}$ node.label > max_ancestors

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

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

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

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

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` 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)

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

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

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

```python
def f(a, x):
    """A node in t that are larger than all their ancestors
    if a.label > x: node.label > max_ancestors
    return 1 + sum([f(b, a.label) for b in a.branches])
    else: Somehow increment the total count
    return sum([f(b, x) for b in a.branches])
    return f(t, t.label)  # Root label is always larger than its ancestors
```

Somehow track the largest ancestor
Somehow increment the total count
Some initial value for the largest ancestor so far...
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."
```

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

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

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

Some initial value for the largest ancestor so far...
Somehow track the largest ancestor
Somehow increment the total count
Root label is always larger than its ancestors
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.
    \"""
    if \__\:
        return 1 + \__\n    else:
        return \__\n    return f(t, t.label - 1)

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

>>> 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

    def f(a, x):
        """A node in \( t \) has its label larger than its largest ancestor if \( \text{node.label} > \text{x} \)."
        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

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.

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 `node.label` is greater than `max_ancestors`.
        ""
        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)

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

The `f(1,0)` node is labeled 1, greater than all its ancestors.

f(1,0)

Return the number of nodes in `t` that are larger than all their ancestors.

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

Somehow track the largest ancestor

Somehow increment the total count

A node in `t` whose `node.label` is greater than `max_ancestors`. If a node's label is greater than its maximum ancestor, increment the total count by 1 plus the sum of nodes greater than the node's label in its branches. Otherwise, sum of nodes greater than the current label in its branches.

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):
        ________________
        if ________________:
            n += 1
        ________________:
            f(_______________)
        ________________
    return n
```

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."""
    n = 0
    def f(a, x):
        _____________________________
        if __________________________:
            n += 1
            _____________________________:
                f(_________________________)
        _____________________________:
            f(_________________________)
    return n
```

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."""
    n = 0
    def f(a, x):
        Somehow track the largest ancestor
        if node.label > max_ancestors:
            n += 1
        return n
    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 node.label > max_ancestors:
            n += 1
        else:
            f(a, c)
    f(t, None):
    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 node.label > max_ancestors:
            n += 1
        if root label is always larger than its ancestors:
            f(
        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):
        # Somehow track the largest ancestor
        if a.label > x:
            n += 1
        # Somehow increment the total count
        f(
            # Root label is always larger than its 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):
    """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, a.label)

    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 \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):
        # 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
\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."""
    n = 0
    def f(a, x):
        # Somehow track the largest ancestor
        if a.label > x:
            n += 1
        for b in a.branches:
            f(b, max(a.label, x))
    f(t, t.label - 1)  # 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):
    
    
    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
```

- Somehow track the largest ancestor
- Somehow increment the total count
- Node label > max_ancestors
- Root label is always larger than its ancestors
Designing Functions
How to Design Programs

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

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.
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.
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.
How to Design Programs

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

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

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

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

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

Testing
Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.

https://htdp.org/2018-01-06/Book/
Applying the Design Process
Implement the function `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]
"""
Designing a Function

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

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

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

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

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

*Signature: Tree -> List of Trees*
Implement `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):
        return result
    def process(t):
        return 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]
```

```
4 5 6
```

```
3
```

```
2
```

```
1
```

```
0
```
Designing a Function

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

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

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

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

Signature: Tree -> List of Trees
Designing a Function

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

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

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

Signature: Tree → 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 \( \rightarrow \) List of Trees**
Designing a Function

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

\begin{verbatim}
def smalls(t):
    
    return

def process(t):
    process(t)
    return result

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

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

"""
result = []
def process(t):
    process(t)
    return result
\end{verbatim}
**Designing a Function**

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

```python
def smalls(t):
    
    
        
    result = []
    def process(t):
        """Find smallest label in t & maybe add t to result"
        pass
    process(t)
    return result
```

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

**Signature:** `Tree -> number

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

```
1
  
  3

  2

  0

[ 4 5 , 6 ]
```
Designing a Function

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

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

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

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

**Signature:** `Tree -> List of Trees`
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 = 
            if 
                return min(smallest, t.label)
    process(t)
    return result
```

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

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

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

"Find smallest label in `t` & maybe add `t` to `result""
Designing a Function

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

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

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

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

"Find smallest label in t & maybe add t to result"
Designing a Function

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

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

    return result
```

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

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

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

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

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

    >>> 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 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`
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."
    return []

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

process(t)
return result
```

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

```python
t.label < smallest
result.append(t)
```

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

```python
result.append(t)
```

```python
[4, 5, 6]
```
Designing a Function

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

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

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

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

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

Compute the smallest label in a branch of `t` if it is smaller than the smallest label found in the branches of `t`.

```python
smallest label in a branch of `t`
```