

## 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

---

```
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:

## 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]]
```



## 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>
```

## 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>
```

Tree processing often involves recursive calls on subtrees

# Tree Processing

## Solving Tree Problems

---

Implement `big`, 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 big(t):
```

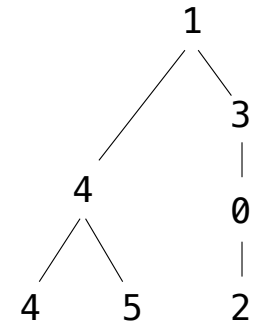
```
    """Return the number of nodes in t that are larger than all their ancestors.
```

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

```
    >>> big(a)
```

```
4
```

```
    """
```



## Solving Tree Problems

---

Implement `big`, 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 big(t):
```

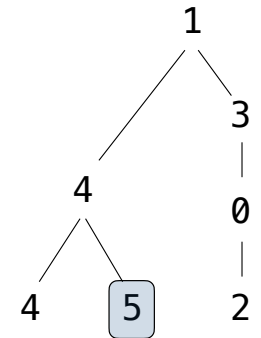
```
    """Return the number of nodes in t that are larger than all their ancestors.
```

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

```
    >>> big(a)
```

```
4
```

```
    """
```



## Solving Tree Problems

---

Implement `big`, 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 big(t):
```

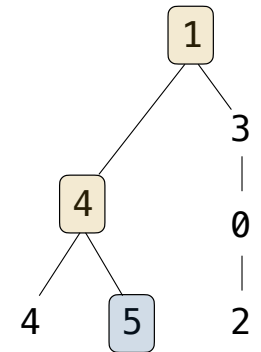
```
    """Return the number of nodes in t that are larger than all their ancestors.
```

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

```
    >>> big(a)
```

```
4
```

```
    """
```





## Solving Tree Problems

---

Implement `big`, 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 big(t):
```

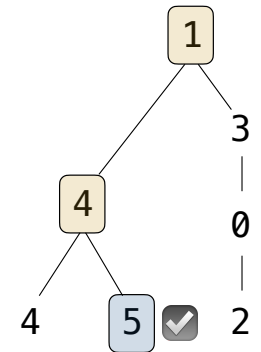
```
    """Return the number of nodes in t that are larger than all their ancestors.
```

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

```
    >>> big(a)
```

```
4
```

```
    """
```



## Solving Tree Problems

---

Implement `big`, 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 big(t):
```

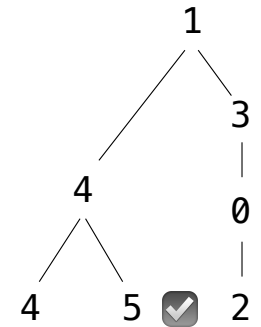
```
    """Return the number of nodes in t that are larger than all their ancestors.
```

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

```
    >>> big(a)
```

```
4
```

```
    """
```



## Solving Tree Problems

---

Implement `big`, 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 big(t):
```

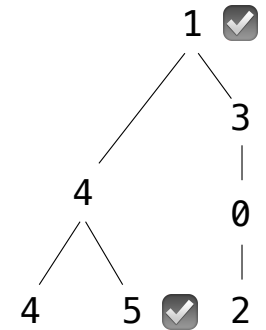
```
    """Return the number of nodes in t that are larger than all their ancestors.
```

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

```
    >>> big(a)
```

```
4
```

```
    """
```



## Solving Tree Problems

---

Implement `big`, 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 big(t):
```

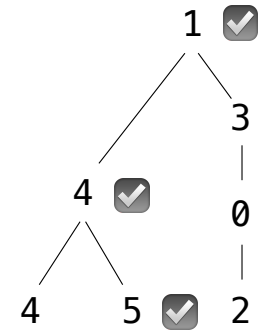
```
    """Return the number of nodes in t that are larger than all their ancestors.
```

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

```
    >>> big(a)
```

```
4
```

```
    """
```



## Solving Tree Problems

---

Implement `big`, 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 big(t):
```

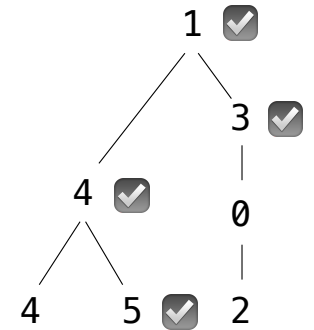
```
    """Return the number of nodes in t that are larger than all their ancestors.
```

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

```
    >>> big(a)
```

```
4
```

```
    """
```



## Solving Tree Problems

---

Implement `big`, 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 big(t):
```

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

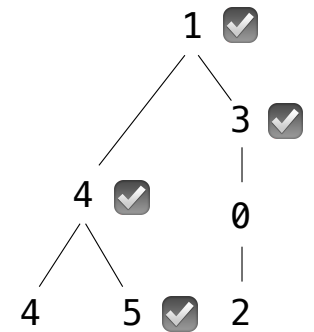
```
    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

The root label is always larger than all of its ancestors



## Solving Tree Problems

---

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
4
```

```
    """
```

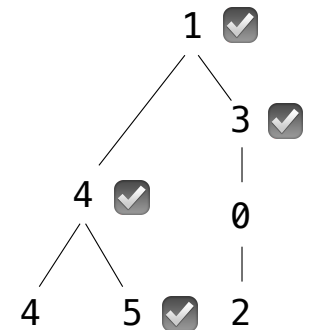
The root label is always larger than all of its ancestors

```
if t.is_leaf():
```

```
    return ____
```

```
else:
```

```
    return ____([____ for b in t.branches])
```



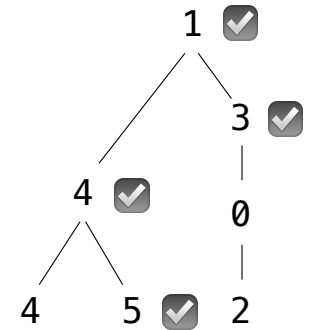
## Solving Tree Problems

Implement `big`, 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 big(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)])])])  
    >>> big(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 `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
4
```

```
    """
```

The root label is always larger than all of its ancestors

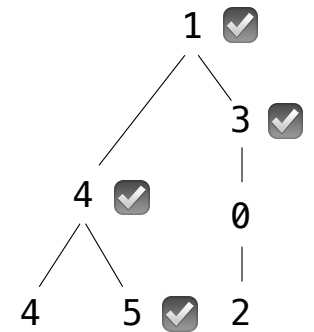
```
if t.is_leaf():
```

```
    return ____
```

```
else:
```

```
    return ____([__ for b in t.branches])
```

Somehow increment  
the total count

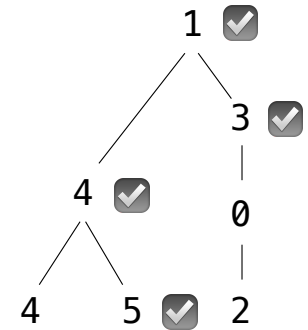


## Solving Tree Problems

Implement `big`, 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 big(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)])])])  
    >>> big(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 increment  
the total count

## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

The root label is always larger than all of its ancestors

```
if t.is_leaf():
```

```
    return ____
```

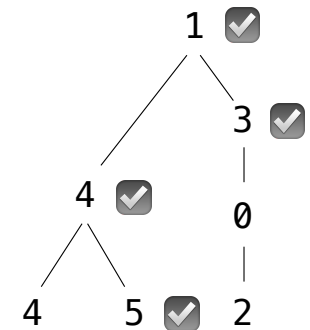
```
else:
```

```
    return ____([__ for b in t.branches])
```

Somehow increment  
the total count

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

Somehow track a  
list of ancestors



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

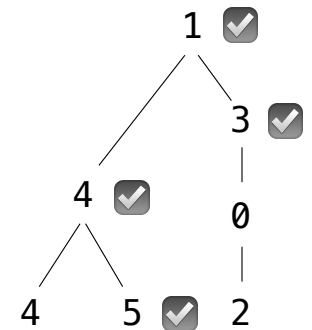
The root label is always larger than all of its ancestors

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

Somehow increment  
the total count

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

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



Somehow track a  
list of ancestors

## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

The root label is always larger than all of its ancestors

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

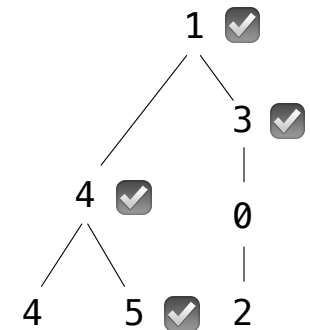
Somehow increment  
the total count

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

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

Somehow track a  
list of ancestors

Somehow track the  
largest ancestor



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
4
```

```
    """
```

```
def f(a, x):
```

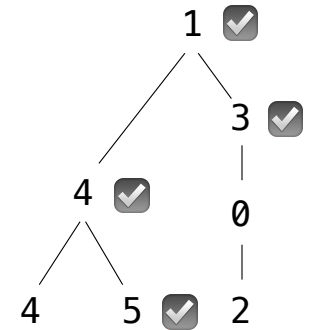
```
    if _____:
```

```
        return 1 + _____
```

```
    else:
```

```
        return _____
```

```
return _____
```



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

```
    def f(a, x):
```

Somehow track the largest ancestor

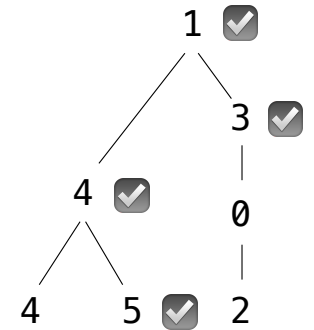
```
        if _____:
```

```
            return 1 + _____
```

```
        else:
```

```
            return _____
```

```
    return _____
```



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
4
```

```
    """
```

```
def f(a, x):
```

```
    if _____:
```

```
        return 1 + _____
```

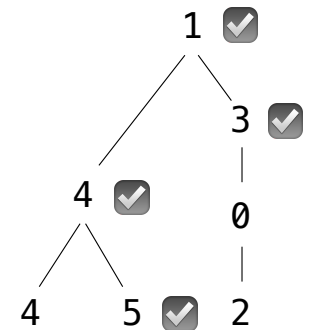
```
    else:
```

```
        return _____
```

```
return _____
```

Somehow track the largest ancestor

node.label > max\_ancestors





## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

```
    def f(a, x):
```

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

```
            node.label > max_ancestors
```

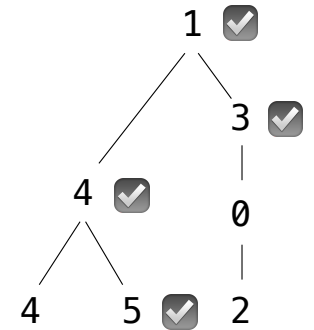
```
            :
```

```
            return 1 +
```

```
        else:
```

```
            return
```

```
    return
```



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

```
    def f(a, x):
```

Somehow track the largest ancestor

```
        if a.label > x:
```

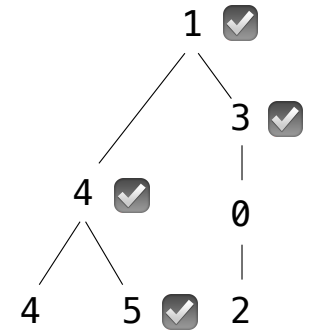
node.label > max\_ancestors

```
            return 1 + _____
```

```
        else:
```

```
            return _____
```

```
    return _____
```



## Solving Tree Problems

Implement **big**, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

```
    def f(a, x):
```

Somehow track the largest ancestor

```
        if a.label > x:
```

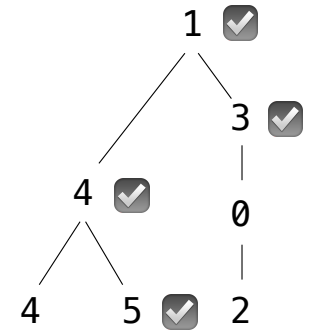
node.label > max\_ancestors

```
            return 1 + _____
```

```
        else:
```

```
            return _____
```

```
    return f(t, _____)
```



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

```
    def f(a, x):
```

Somehow track the largest ancestor

```
        if a.label > x:
```

node.label > max\_ancestors

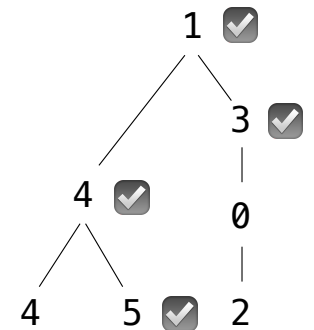
```
            return 1 + _____
```

```
        else:
```

```
            return _____
```

```
    return f(t, _____)
```

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



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

```
    def f(a, x):
```

Somehow track the largest ancestor

```
        if a.label > x:
```

node.label > max\_ancestors

```
            return 1 +
```

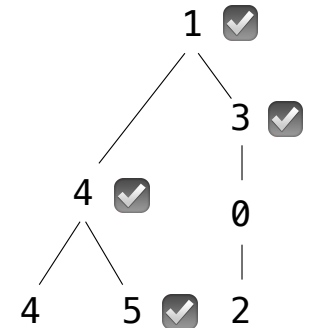
```
        else:
```

Somehow increment the total count

```
            return
```

```
    return f(t,
```

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



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

```
    def f(a, x):
```

Somehow track the largest ancestor

```
        if a.label > x:
```

A node in t

max\_ancestor

node.label > max\_ancestors

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

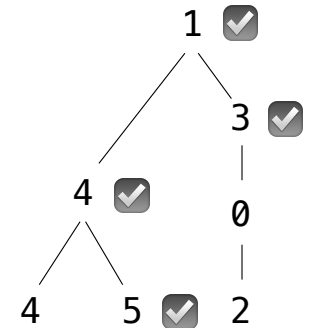
```
        else:
```

Somehow increment the total count

```
            return
```

```
    return f(t, )
```

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



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

```
    def f(a, x):
```

Somehow track the largest ancestor

```
        if a.label > x:
            node.label > max_ancestors
```

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

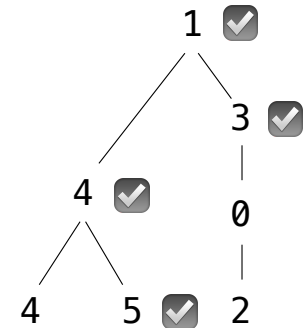
```
        else:
```

Somehow increment the total count

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

```
    return f(t, )
```

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



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

```
    def f(a, x):
```

Somehow track the largest ancestor

```
        if a.label > x:
            node.label > max_ancestors
```

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

```
        else:
```

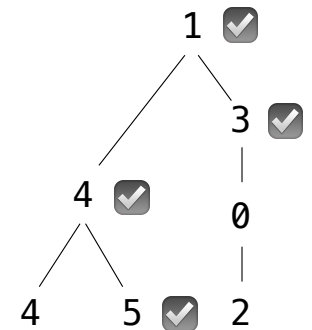
Somehow increment the total count

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

```
    return f(t, )
```

Root label is always larger than its ancestors

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





## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

```
    def f(a, x):
```

Somehow track the largest ancestor

```
        if a.label > x:
            node.label > max_ancestors
```

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

```
        else:
```

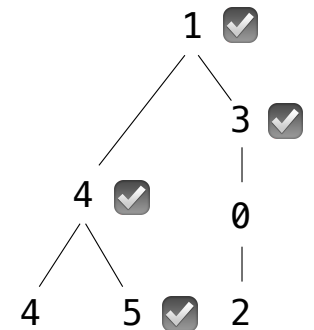
Somehow increment the total count

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

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

Root label is always larger than its ancestors

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



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
>>> big(a)
```

```
4
```

```
"""
```

```
def f(a, x):
```

Somehow track the largest ancestor

```
    if a.label > x:  # node.label > max_ancestors
```

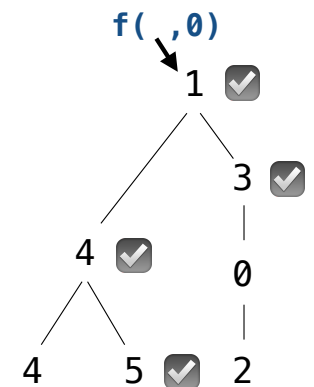
```
        return 1 + sum([f(b, a.label) for b in a.branches])
```

```
    else:  # Somehow increment the total count
```

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

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

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



## Solving Tree Problems

Implement `big`, 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 big(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)])])])  
>>> big(a)
```

```
4  
"""  
def f(a, x):  
    """  
    Somehow track the largest ancestor
```

```
    A node in t → max_ancestor  
    if a.label > x: node.label > max_ancestors
```

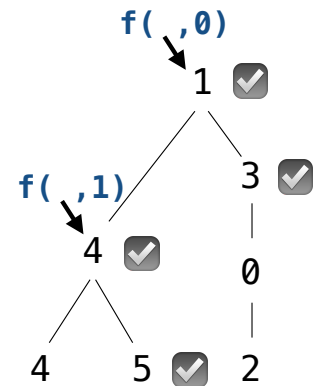
```
        return 1 + sum([f(b, a.label) for b in a.branches])
```

```
    else:  
        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
```

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



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

```
    def f(a, x):
```

Somehow track the largest ancestor

```
        if a.label > x:
            node.label > max_ancestors
```

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

```
        else:
```

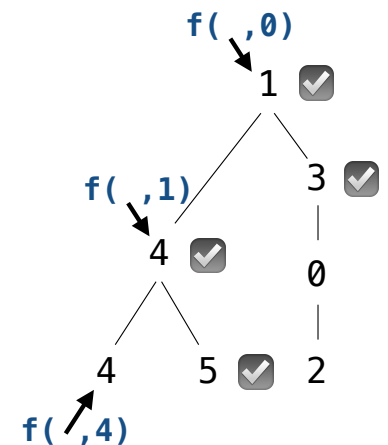
Somehow increment the total count

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

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

Root label is always larger than its ancestors

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



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

```
    def f(a, x):
```

Somehow track the largest ancestor

```
        if a.label > x:
            node.label > max_ancestors
```

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

```
        else:
```

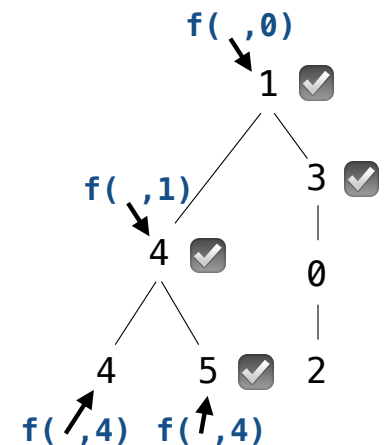
Somehow increment the total count

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

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

Root label is always larger than its ancestors

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



## Solving Tree Problems

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

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

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

```
4
```

```
"""
```

```
def f(a, x):
```

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:
```

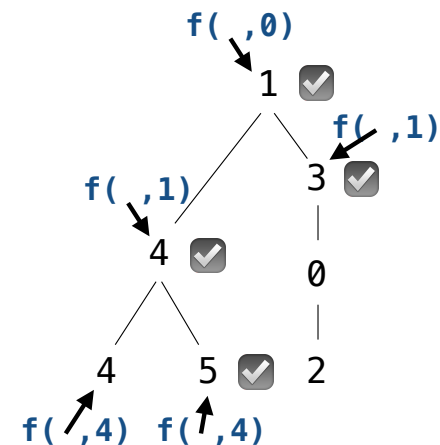
Somehow increment the total count

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

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

Root label is always larger than its ancestors

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



## Solving Tree Problems

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

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

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

```
4
"""
```

Somehow track the largest ancestor

```
def f(a, x):
    if a.label > x:
```

A node in t  $\rightarrow$   $\uparrow$   $\uparrow$  *max\_ancestor*  $\rightarrow$  node.label > max\_ancestors

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

```
    else:
```

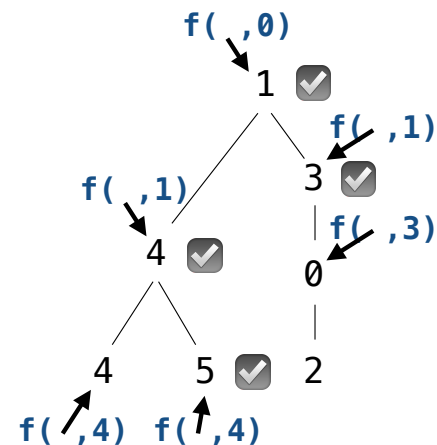
Somehow increment the total count

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

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

Root label is always larger than its ancestors

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



## Solving Tree Problems

Implement `big`, 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 big(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)])])])
```

```
    >>> big(a)
```

```
    4
```

```
    """
```

```
    def f(a, x):
```

Somehow track the largest ancestor

```
        if a.label > x:
            node.label > max_ancestors
```

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

```
        else:
```

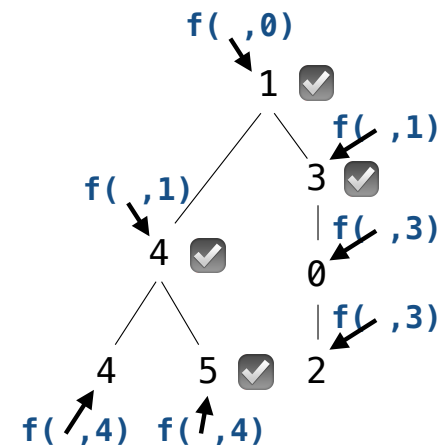
Somehow increment the total count

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

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

Root label is always larger than its ancestors

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







## Recursive Accumulation

## Solving Tree Problems

---

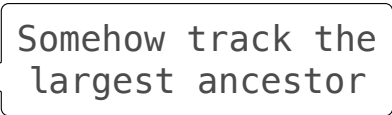
Implement `big`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than any labels of their ancestor nodes.

```
def big(t):  
    """Return the number of nodes in t that are larger than all their ancestors."""  
    n = 0  
  
    def f(a, x):  
  
        _____  
  
        if _____:  
            n += 1  
  
        _____:  
            f(_____)  
  
        _____  
  
    return n
```

## Solving Tree Problems

---

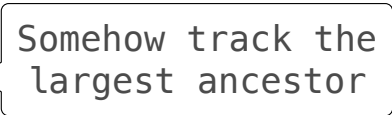

Implement `big`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than any labels of their ancestor nodes.

```
def big(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

---

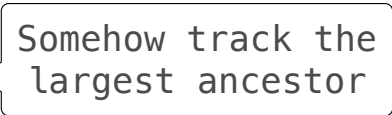

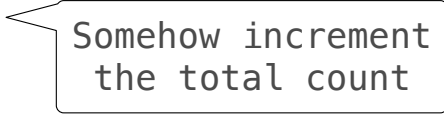
Implement `big`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than any labels of their ancestor nodes.

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

## Solving Tree Problems

---

Implement `big`, 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 big(t):  
    """Return the number of nodes in t that are larger than all their ancestors."""  
    n = 0  
    def f(a, x):  
          
        _____  
        if _____:   
            n += 1   
            _____:  
            f(_____)  
        _____  
    return n
```

## Solving Tree Problems

---

Implement `big`, 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 big(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 label is always larger than its ancestors

```
    return n
```

---

## Solving Tree Problems

---

Implement `big`, 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 big(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:
```

node.label > max\_ancestors

```
            n += 1
```

Somehow increment the total count

```
            f(a.left, a.label)
```

Root label is always larger than its ancestors

```
    return n
```

---



## Solving Tree Problems

---

Implement `big`, 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 big(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:
```

node.label > max\_ancestors

```
            n += 1
```

Somehow increment the total count

```
            f(a, a.label)
```

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

Root label is always larger than its ancestors

```
    return n
```

## Solving Tree Problems

---

Implement `big`, 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 big(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:
```

node.label > max\_ancestors

```
            n += 1
```

```
        for b in a.branches:
```

Somehow increment the total count

```
            f(b, x)
```

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

Root label is always larger than its ancestors

```
    return n
```

---

## Solving Tree Problems

---

Implement `big`, 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 big(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:
```

node.label > max\_ancestors

```
            n += 1
```

```
        for b in a.branches:
```

Somehow increment the total count

```
            f(b, max(a.label, x))
```

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

Root label is always larger than its ancestors

```
    return n
```

---

## Solving Tree Problems

---

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

Somehow track the largest ancestor

node.label > max\_ancestors

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.

## How to Design Programs

---

### **From Problem Analysis to Data Definitions**

Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

### **Signature, Purpose Statement, Header**

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

### **Functional Examples**

Work through examples that illustrate the function's purpose.

### **Function Template**

Translate the data definitions into an outline of the function.

## How to Design Programs

---

### **From Problem Analysis to Data Definitions**

Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

### **Signature, Purpose Statement, Header**

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

### **Functional Examples**

Work through examples that illustrate the function's purpose.

### **Function Template**

Translate the data definitions into an outline of the function.

### **Function Definition**

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

## 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.

## 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.

```
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.

```
def smalls(t): Signature: Tree -> List of Trees
    """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.

```
def smalls(t): Signature: Tree -> List of Trees
    """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.

```
def smalls(t): Signature: Tree -> List of Trees
    """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
```

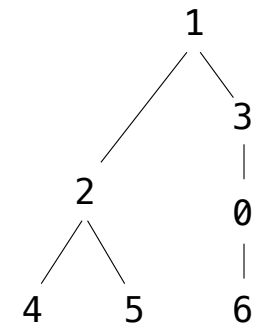
## Designing a Function

---

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.

```
def smalls(t): Signature: Tree -> List of Trees
    """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
```

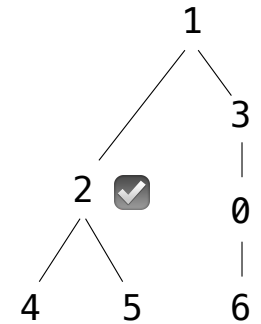
## Designing a Function

---

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.

```
def smalls(t): Signature: Tree -> List of Trees
    """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
```

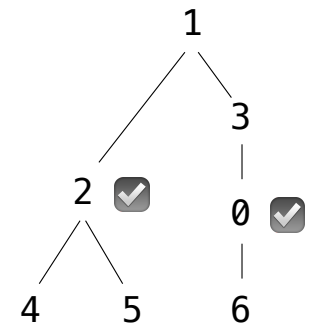
## Designing a Function

---

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.

```
def smalls(t): Signature: Tree -> List of Trees
    """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
```

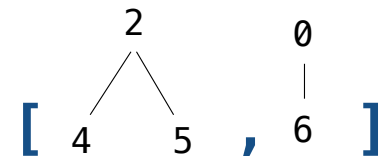
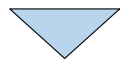
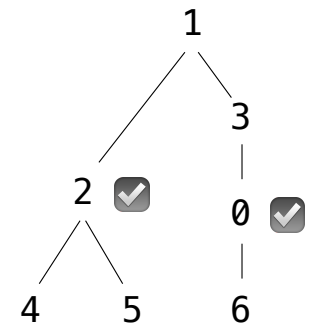
## Designing a Function

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.

```
def smalls(t): Signature: Tree -> List of Trees
    """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

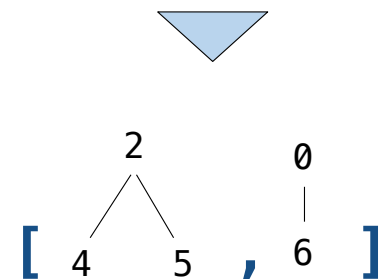
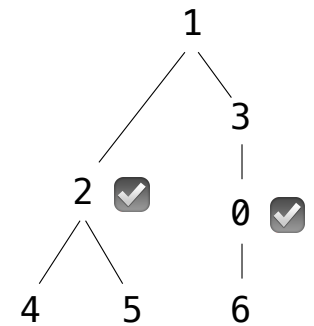
Implement `smalls`, which takes a `Tree` instance `t` containing integer labels. It returns the non-leaf nodes in `t` whose labels are smaller than any labels of their descendant nodes.

```
def smalls(t): Signature: Tree -> List of Trees  
    """Return the non-leaf nodes in t that are smaller than all their descendants.
```

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

```
    """ Signature: Tree -> number  
    result = []  
    def process(t):
```

```
        process(t)  
    return result
```



## Designing a Function

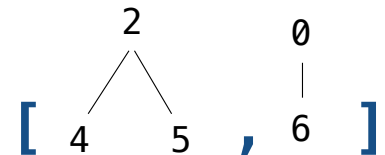
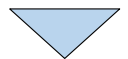
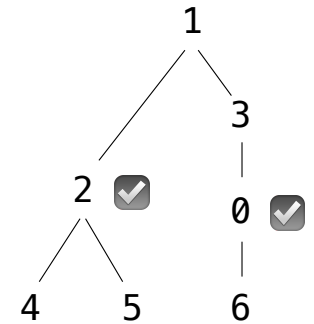
Implement `smalls`, which takes a `Tree` instance `t` containing integer labels. It returns the non-leaf nodes in `t` whose labels are smaller than any labels of their descendant nodes.

```
def smalls(t): Signature: Tree -> List of Trees  
    """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): Signature: Tree -> number  
        "Find smallest label in t & maybe add t to result"
```

```
    process(t)  
    return result
```





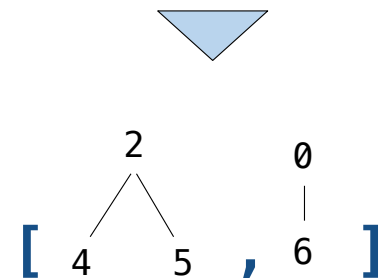
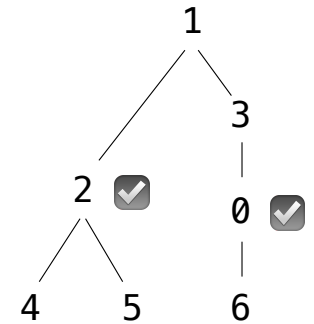
## Designing a Function

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.

```
def smalls(t): Signature: Tree -> List of Trees
    """Return the non-leaf nodes in t that are smaller than all their descendants.

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

    """
    Signature: Tree -> number
    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
```



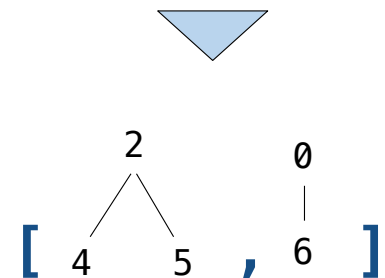
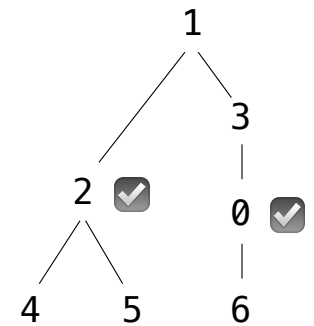
## Designing a Function

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): Signature: Tree -> List of Trees
    """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 = []
    Signature: Tree -> number
    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
```



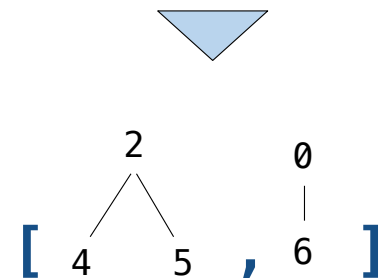
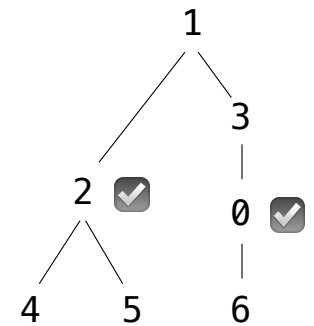
## Designing a Function

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): Signature: Tree -> List of Trees
    """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 = []
    Signature: Tree -> number
    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
```



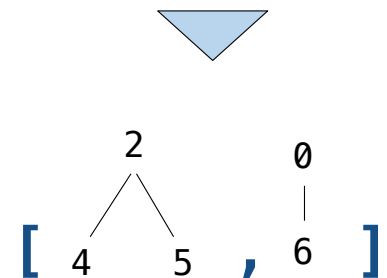
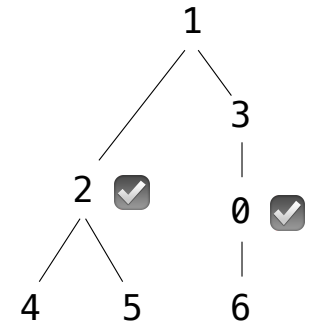
## Designing a Function

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): Signature: Tree -> List of Trees
    """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 = []
    Signature: Tree -> number
    def process(t): "Find smallest label in t & maybe add t to result"
        if t.is_leaf():
            return t.label
        else:
            smallest = _____
            smallest label in a branch of t if _____:
                return min(smallest, t.label)
    process(t)
    return result
```



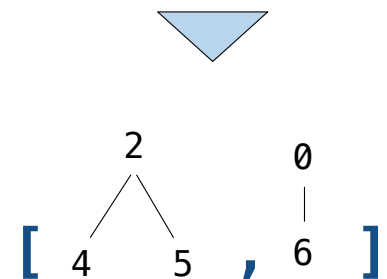
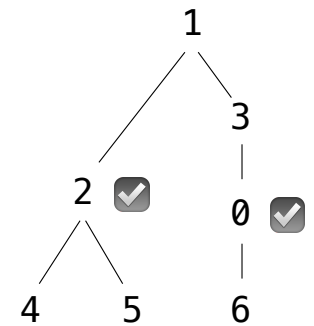
## Designing a Function

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.

```
def smalls(t): Signature: Tree -> List of Trees
    """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 = []
    Signature: Tree -> number
    def process(t): "Find smallest label in t & maybe add t to result"
        if t.is_leaf():
            return _____
        else:
            smallest = _____
            smallest label in a branch of t if _____:
                return min(smallest, t.label)
    process(t)
    return result
```



## Designing a Function

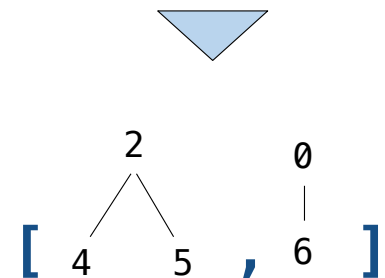
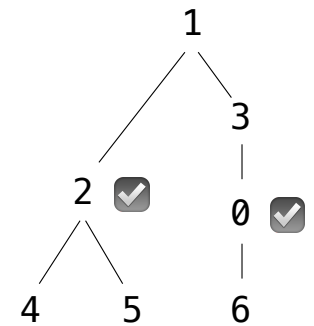
Implement `smalls`, which takes a `Tree` instance `t` containing integer labels. It returns the non-leaf nodes in `t` whose labels are smaller than any labels of their descendant nodes.

```
def smalls(t): Signature: Tree -> List of Trees
    """Return the non-leaf nodes in t that are smaller than all their descendants.

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

    """
    Signature: Tree -> number
    result = []
    def process(t): "Find smallest label in t & maybe add t to result"
        if t.is_leaf():
            return _____
        else:
            smallest = _____
            if _____:
                result.append(_____)
            return min(smallest, t.label)
    process(t)
    return result
```

*smallest label in a branch of t* →



## Designing a Function

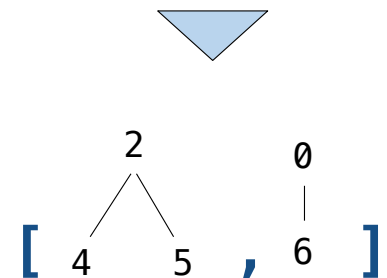
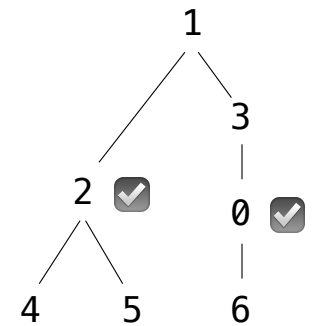
Implement `smalls`, which takes a `Tree` instance `t` containing integer labels. It returns the non-leaf nodes in `t` whose labels are smaller than any labels of their descendant nodes.

```
def smalls(t): Signature: Tree -> List of Trees
    """Return the non-leaf nodes in t that are smaller than all their descendants.

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

    """
    Signature: Tree -> number
    result = []
    def process(t): "Find smallest label in t & maybe add t to result"
        if t.is_leaf():
            return _____
        else:
            smallest = _____
            if _____:
                result.append( t )
            return min(smallest, t.label)
    process(t)
    return result
```

*smallest label in a branch of 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.

```
def smalls(t): Signature: Tree -> List of Trees
    """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 = []
    Signature: Tree -> number
    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])
            smallest label in a branch of t
            if t.label < smallest:
                result.append( t )
            return min(smallest, t.label)
    process(t)
    return result
```

