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>
```

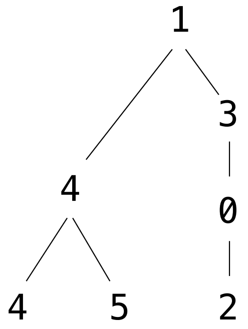
Tree processing often involves recursive calls on subtrees

Tree Processing

Solving Tree Problems

Implement **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.

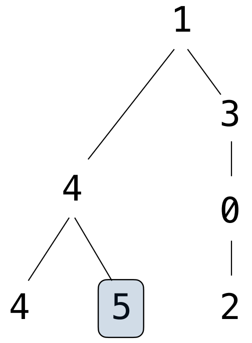
```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> 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.

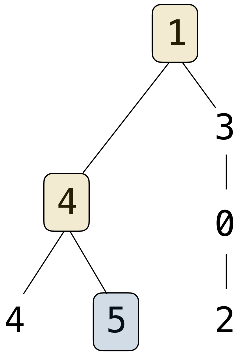
```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> 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.

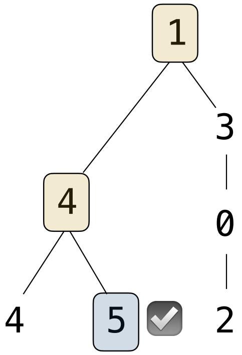
```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> 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.

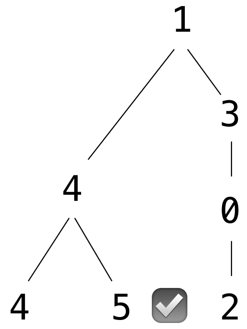
```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> 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.

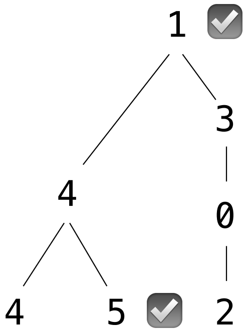
```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> 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.

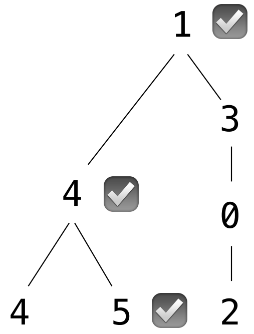
```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> 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.

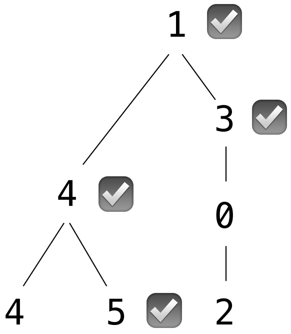
```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> 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.

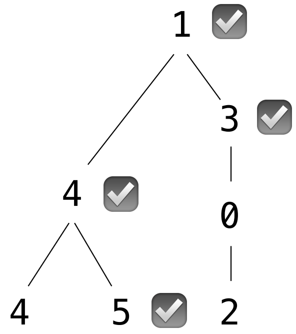
```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> 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.

```
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
    """
```

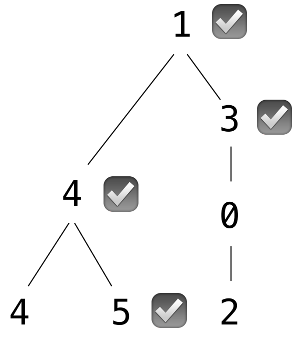


The root label is always larger than all of its ancestors

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.

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

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



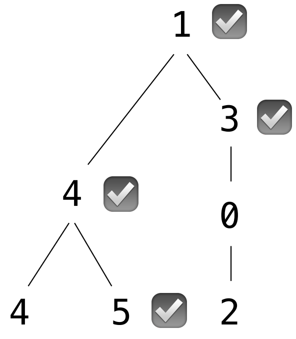
The root label is always larger than all of its ancestors

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

Solving Tree Problems

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

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

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



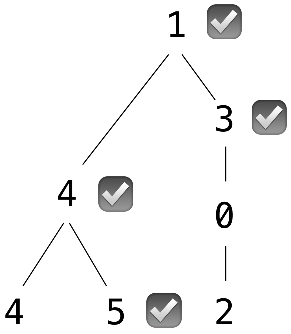
The root label is always larger than all of its ancestors

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

Solving Tree Problems

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

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

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



The root label is always larger than all of its ancestors

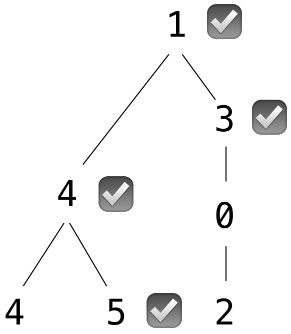
```
~~if t.is_leaf():
    return ___
else:
    return ___([___ for b in t.branches])~~
```

Somehow increment the total count

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.

```
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])~~
```

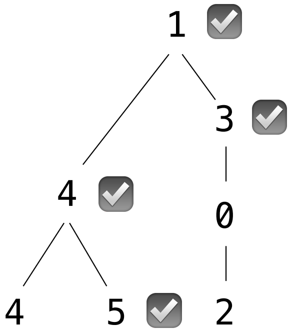
```
if node.label > max(ancestors):
```

Somehow 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.

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

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



The root label is always larger than all of its ancestors

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

Somehow track a list of ancestors

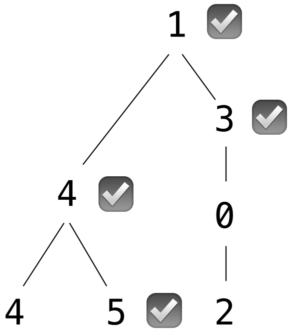
```
if node.label > max(ancestors):
```

Somehow increment the total count

Solving Tree Problems

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



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

Somehow track a list of ancestors

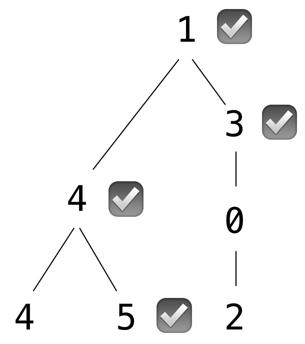
```
if node.label > max(ancestors):
```

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

Solving Tree Problems

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



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

Somehow track a list of ancestors

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

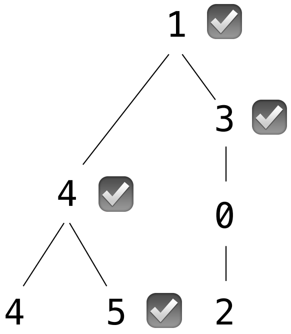
Somehow track the largest ancestor

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

Solving Tree Problems

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

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

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

        if _____________________________________________________:

            return 1 + __________________________________________

        else:

            return ______________________________________________

    return __________________________________________________
```

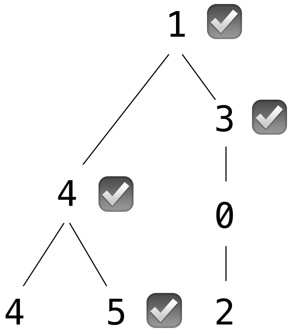


Solving Tree Problems

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

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

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

        if _____________________________________________________:

            return 1 + __________________________________________

        else:

            return ______________________________________________

    return _____________________________________________________
```

Somehow track the largest ancestor



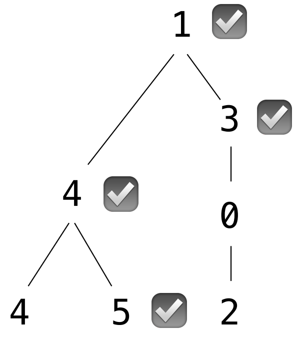
Solving Tree Problems

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

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

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

        if _____________________________________________________:

            return 1 + __________________________________________

        else:

            return ______________________________________________

    return ______________________________________________________
```

Somehow track the largest ancestor

node.label > max_ancestors



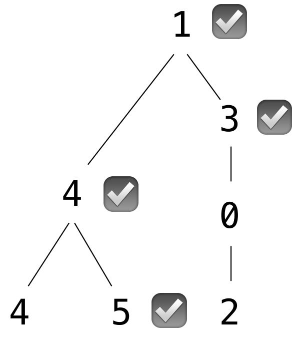
Solving Tree Problems

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

```
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 + __________________________________

        else:

            return ______________________________________

    return ______________________________________________
```

Somehow track the largest ancestor

node.label > max_ancestors



Solving Tree Problems

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

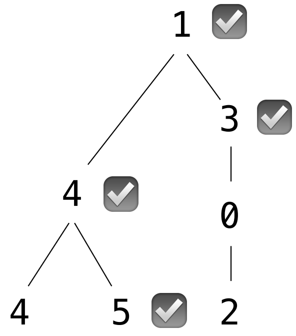
```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> 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 + _________________________________________

        else:

            return _____________________________________________

    return _____________________________________________________
```

Somehow track the largest ancestor

A node in t

max_ancestor

node.label > max_ancestors



Solving Tree Problems

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

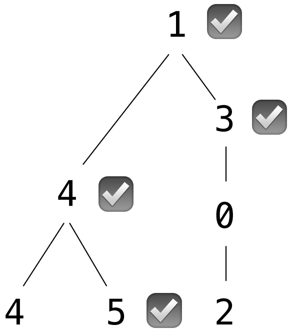
```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> 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 + ________________________________________

        else:

            return ________________________________________

    return f(t,            )______________________________________
```

Somehow track the largest ancestor

A node in t

max_ancestor

node.label > max_ancestors



Solving Tree Problems

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

```
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 + ______________________________

        else:

            return __________________________________

    return f(t,            )_____________________________
```

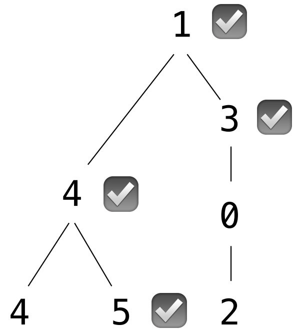
Somehow track the largest ancestor

A node in t

max_ancestor

node.label > max_ancestors

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



Solving Tree Problems

Implement **bigs**, which takes a Tree instance t containing integer labels. It returns the number of nodes in t whose labels are larger than 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):
        if a.label > x ____________________________________:

            return 1 + ____________________________________

        else:

            return ____________________________________

    return f(t,            )_______________________________
```

Somehow track the largest ancestor

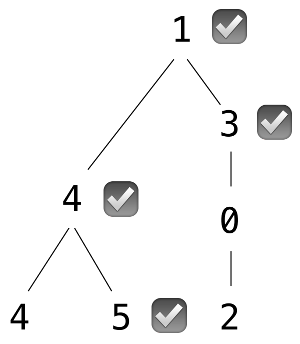
A node in t

max_ancestor

node.label > max_ancestors

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.

    >>> 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 _________________________________

    return f(t,            ) ______________________________
```

Somehow track the largest ancestor

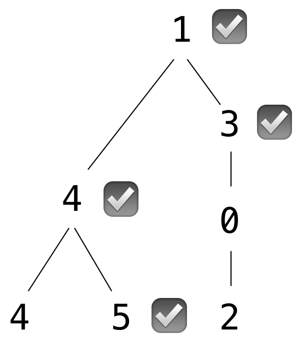
A node in t

max_ancestor

node.label > max_ancestors

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.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):
        if a.label > x                                  :

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

        else:

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

    return f(t,           )
```

A node in t → a

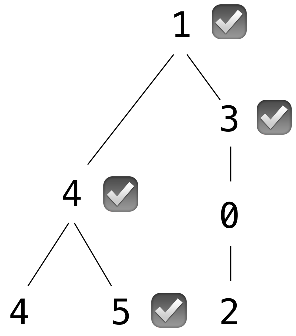
max_ancestor → x

Somehow track the largest ancestor

node.label > max_ancestors

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.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):
        if a.label > x                                          :

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

        else:

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

    return f(t,             )
```

Somehow track the largest ancestor

A node in t

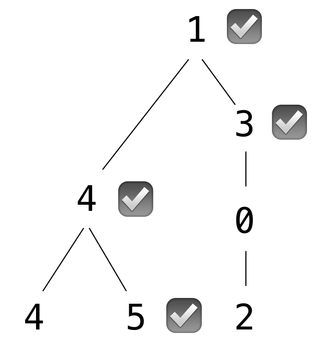
max_ancestor

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...



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.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):
        if a.label > x:
            return 1 + sum([f(b, a.label) for b in a.branches])
        else:
            return sum([f(b, x) for b in a.branches])
    return f(t, t.label - 1)
```

Somehow track the largest ancestor

A node in t

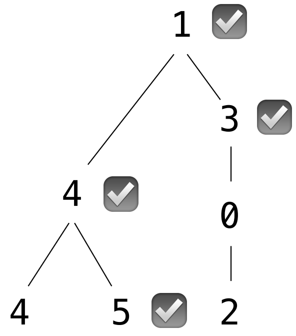
max_ancestor

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...



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.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):
        if a.label > x:
            return 1 + sum([f(b, a.label) for b in a.branches])
        else:
            return sum([f(b, x) for b in a.branches])
    return f(t, t.label - 1)
```

Somehow track the largest ancestor

A node in t

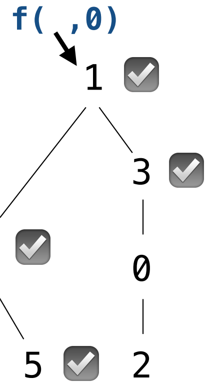
max_ancestor

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...



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.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):
        if a.label > x:
            return 1 + sum([f(b, a.label) for b in a.branches])
        else:
            return sum([f(b, x) for b in a.branches])
    return f(t, t.label - 1)
```

Somehow track the largest ancestor

A node in t → a

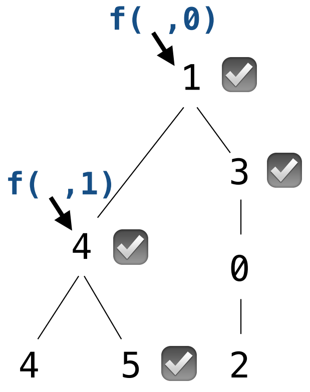
max_ancestor → x

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...



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.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):
        if a.label > x:
            return 1 + sum([f(b, a.label) for b in a.branches])
        else:
            return sum([f(b, x) for b in a.branches])
    return f(t, t.label - 1)
```

A node in t

max_ancestor

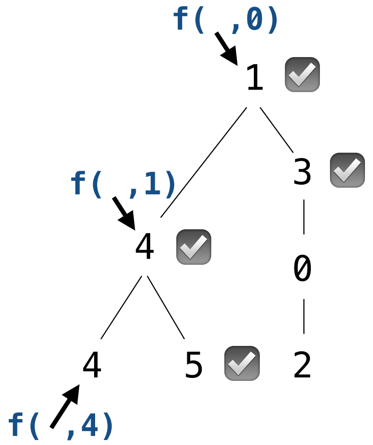
Somehow track the largest ancestor

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...



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.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):
        if a.label > x                                              :

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

        else:

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

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

Somehow track the largest ancestor

A node in t

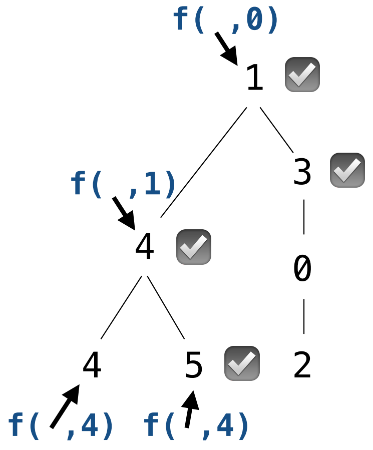
max_ancestor

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...



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.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):
        if a.label > x:
            return 1 + sum([f(b, a.label) for b in a.branches])
        else:
            return sum([f(b, x) for b in a.branches])
    return f(t, t.label - 1)
```

Somehow track the largest ancestor

A node in t

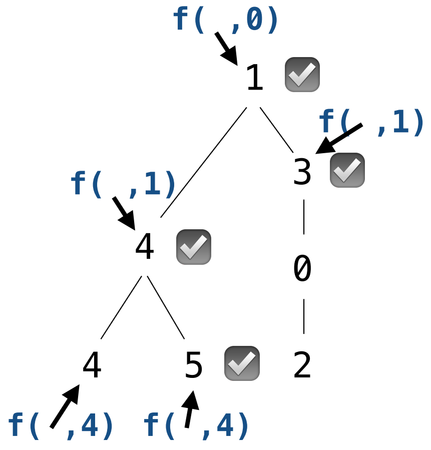
max_ancestor

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...



Solving Tree Problems

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

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

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):
        if  a.label > x  :
            return 1 + sum([f(b, a.label) for b in a.branches])
        else:
            return     sum([f(b, x) for b in a.branches])
    return  f(t, t.label - 1)
```

A node in t

max_ancestor

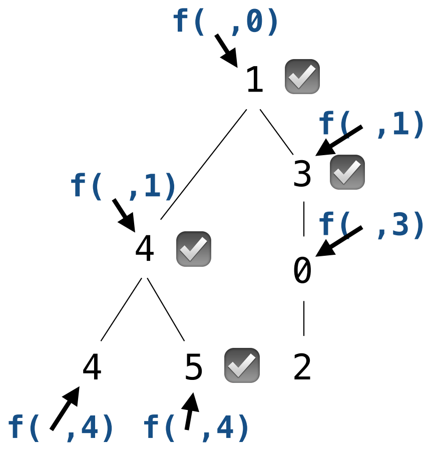
Somehow track the largest ancestor

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...



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.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):
        if a.label > x                                        :
            return 1 + sum([f(b, a.label) for b in a.branches])
        else:
            return     sum([f(b, x) for b in a.branches])
    return f(t, t.label - 1)
```

Somehow track the largest ancestor

A node in t

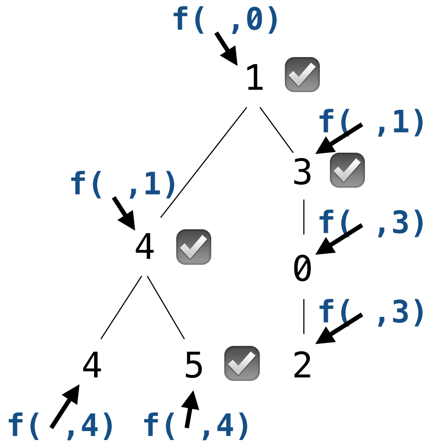
max_ancestor

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...



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.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):
        if a.label > x :
            return 1 + sum( f(b, a.label) for b in a.branches )
        else:
            return sum( f(b, x) for b in a.branches )
    return f(t, t.label - 1)
```

Somehow track the largest ancestor

A node in t

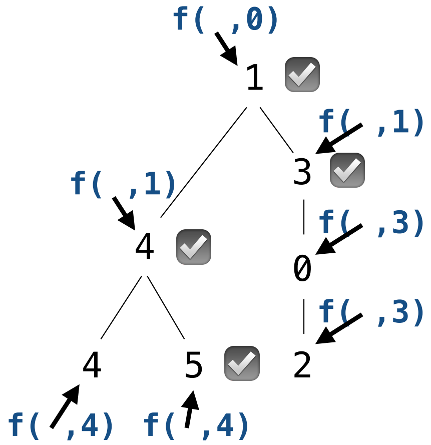
max_ancestor

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...



Recursive Accumulation

Solving Tree Problems

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

```
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

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."""
    n = 0

    def f(a, x):

        ___________________________

        if ________________________:

            n += 1

        ___________________________:

            f(_____________________)

    _______________________________

    return n
```

Somehow track the largest ancestor

Solving Tree Problems

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

```
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
```

Somehow track the largest ancestor

node.label > max_ancestors

Solving Tree Problems

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

```
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
```

Somehow track the largest ancestor

node.label > max_ancestors

Somehow increment the total count

Solving Tree Problems

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

```
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
```

Somehow track the largest ancestor

node.label > max_ancestors

Somehow increment the total count

Root label is always larger than its ancestors

Solving Tree Problems

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

```
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(_____________________)

    _______________________________

    return n
```

Somehow track the largest ancestor

node.label > max_ancestors

Somehow increment the total count

Root label is always larger than its ancestors

Solving Tree Problems

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

```
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(_____________________)

    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

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."""
    n = 0

    def f(a, x):

        ___________________________

        if a.label > x_____________:

            n += 1

        for b in a.branches________:

            f(_____________________)

    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

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."""
    n = 0

    def f(a, x):

        ____________________________

        if a.label > x             :

            n += 1

        for b in a.branches        :

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

    f(t, t.label - 1)

    return n
```

Somehow track the largest ancestor

node.label > max_ancestors

Somehow increment the total count

Root label is always larger than its ancestors

Solving Tree Problems

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

```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."""
    n = 0

    def f(a, x):

        nonlocal n

        if a.label > x:

            n += 1

        for b in a.branches:

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

    f(t, t.label - 1)

    return n
```

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.

https://htdp.org/2018-01-06/Book/

How to Design Programs

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

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

https://htdp.org/2018-01-06/Book/

How to Design Programs

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

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

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

https://htdp.org/2018-01-06/Book/

How to Design Programs

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

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

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

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

https://htdp.org/2018-01-06/Book/

How to Design Programs

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

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

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

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

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

https://htdp.org/2018-01-06/Book/

How to Design Programs

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

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

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

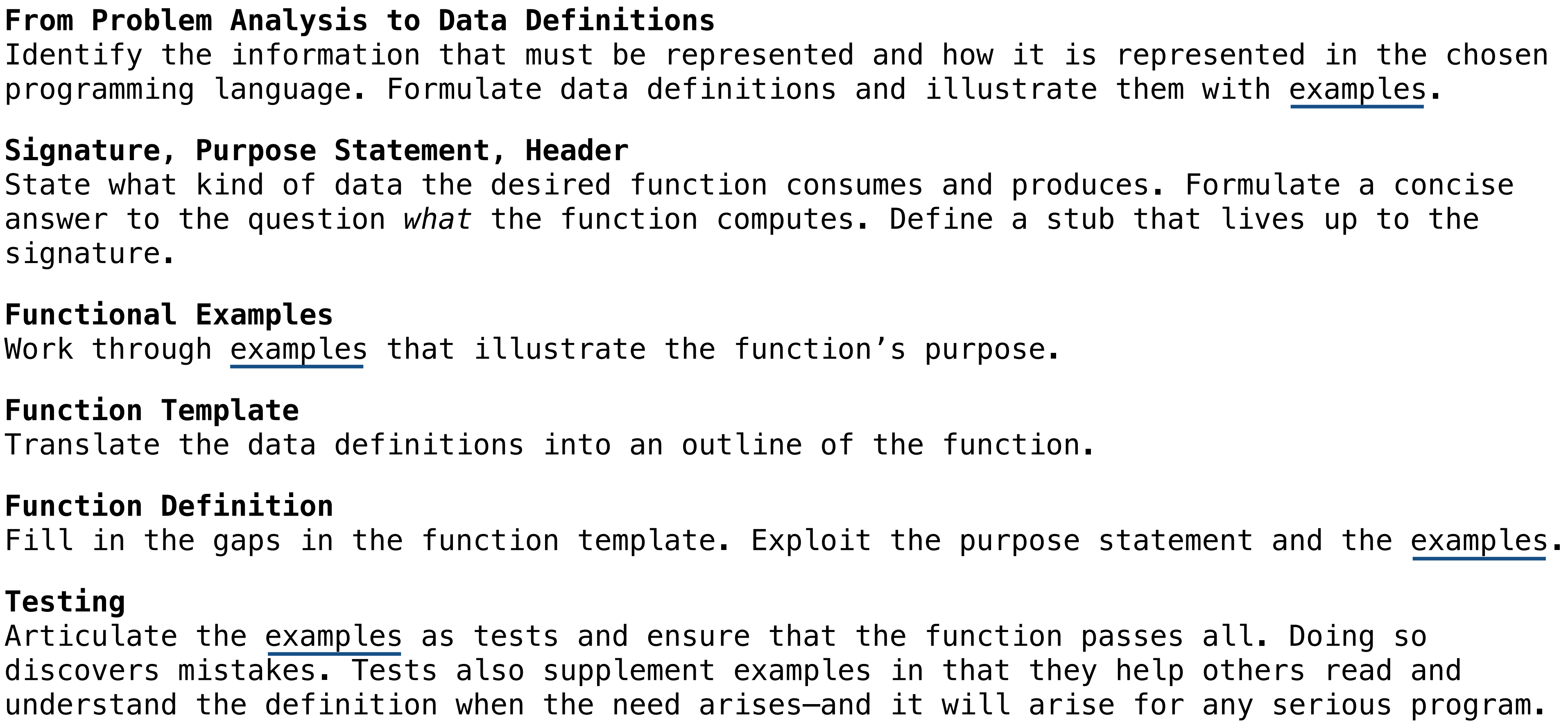
Function Template
Translate the data definitions into an outline of the function.

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

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

https://htdp.org/2018-01-06/Book/

How to Design Programs



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

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

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

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

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

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

https://htdp.org/2018-01-06/Book/

Applying the Design Process

Designing a Function

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

```
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.

Signature: Tree -> List of Trees

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

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







    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.

Signature: Tree -> List of Trees

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

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







    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.

Signature: Tree -> List of Trees

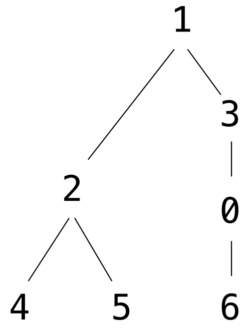
```
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants.

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







    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.

Signature: Tree -> List of Trees

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

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




    process(t)
    return result
```

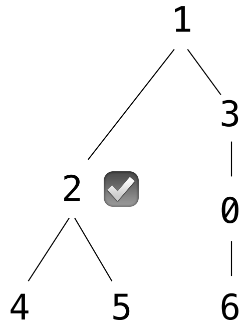


Designing a Function

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.

Signature: Tree -> List of Trees

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

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




    process(t)
    return result
```

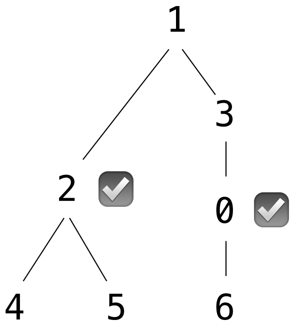


Designing a Function

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.

Signature: Tree -> List of Trees

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

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




    process(t)
    return result
```

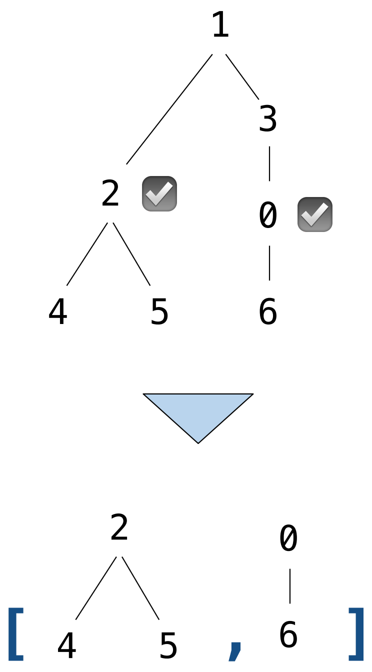


Designing a Function

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.

Signature: Tree -> List of Trees

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

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




    process(t)
    return result
```



Designing a Function

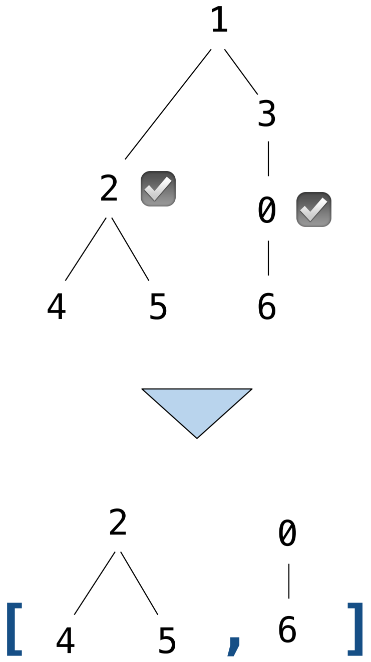
Implement **smalls**, which takes a Tree instance t containing integer labels. It returns the non-leaf nodes in t whose labels are smaller than any labels of their descendant nodes.

Signature: Tree -> List of Trees

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

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

Signature: Tree -> number

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




    process(t)
    return result
```



Designing a Function

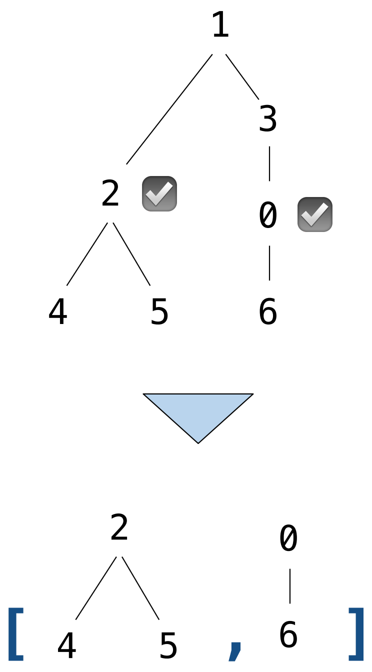
Implement **smalls**, which takes a Tree instance t containing integer labels. It returns the non-leaf nodes in t whose labels are smaller than any labels of their descendant nodes.

Signature: Tree -> List of Trees

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

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




    process(t)
    return result
```

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.

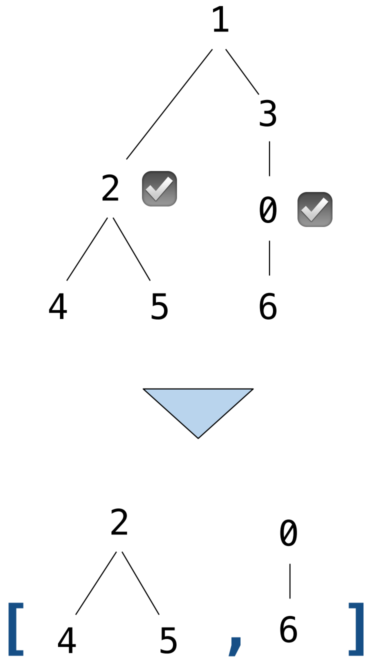
Signature: Tree -> List of Trees

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

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

Signature: Tree -> number

"Find smallest label in t & maybe add t to result"

```
    result = []
    def process(t):
        if t.is_leaf():
            return t.label
        else:



            return min(...)
    process(t)
    return result
```



Designing a Function

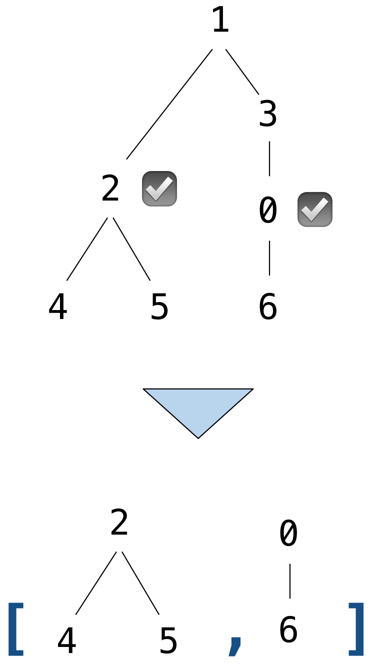
Implement **smalls**, which takes a Tree instance t containing integer labels. It returns the non-leaf nodes in t whose labels are smaller than any labels of their descendant nodes.

Signature: Tree -> List of Trees

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

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        if t.is_leaf():
            return ___________________________________________
        else:
            smallest = _______________________________________
            if ______________________________________________:
                _____________________________________________
            return min(smallest, t.label)
    process(t)
    return result
```

Signature: Tree -> number

"Find smallest label in t & maybe add t to result"



Designing a Function

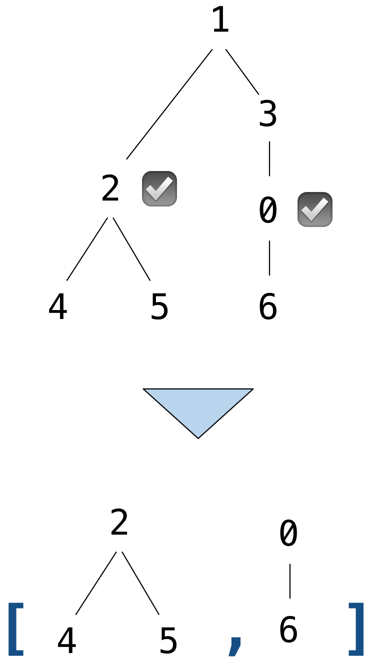
Implement **smalls**, which takes a Tree instance t containing integer labels. It returns the non-leaf nodes in t whose labels are smaller than any labels of their descendant nodes.

Signature: Tree -> List of Trees

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

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        if t.is_leaf():
            return __________t.label__________
        else:
            smallest = ________________________________
            if ________________________________________:
                ______________________________________
            return min(smallest, t.label)
    process(t)
    return result
```

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.

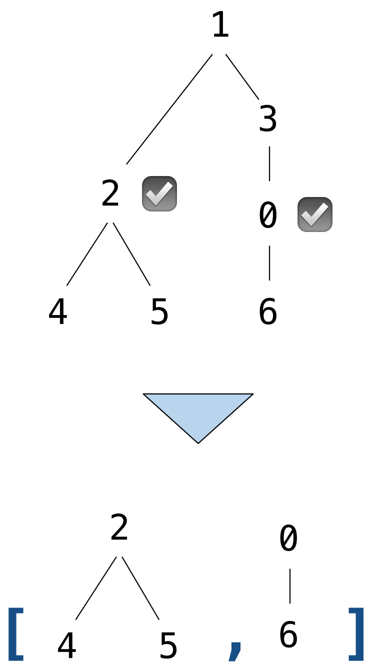
Signature: Tree -> List of Trees

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

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        if t.is_leaf():
            return ____________t.label____________
        else:
            smallest = ______________________________
            if ______________________________________:
                _____________________________________
            return min(smallest, t.label)
    process(t)
    return result
```

Signature: Tree -> number

"Find smallest label in t & maybe add t to 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.

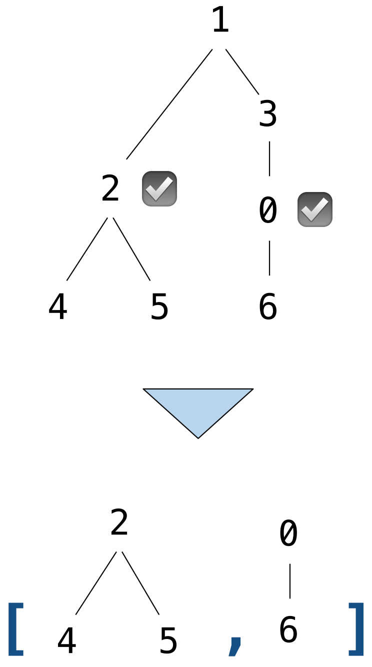
Signature: Tree -> List of Trees

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

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        if t.is_leaf():
            return _______________t.label_______________
        else:
            smallest = ______________________________
            if ____t.label < smallest____________________:
                __________________________________________
            return min(smallest, t.label)
    process(t)
    return result
```

Signature: Tree -> number

"Find smallest label in t & maybe add t to 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.

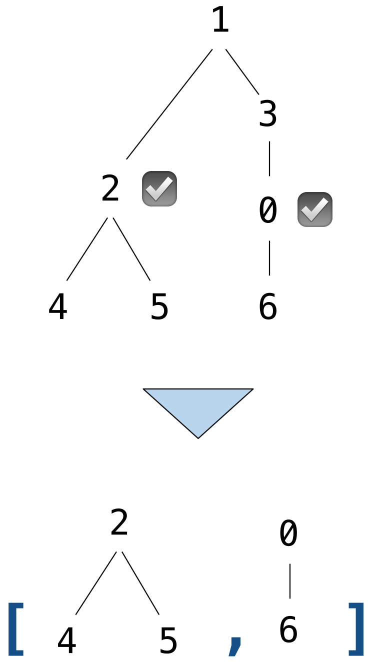
Signature: Tree -> List of Trees

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

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        if t.is_leaf():
            return _______________t.label_______________
        else:
            smallest = __________________________________
            if _____t.label < smallest___________________:
                ___result.append(    )___________________
            return min(smallest, t.label)
    process(t)
    return result
```

Signature: Tree -> number

"Find smallest label in t & maybe add t to 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.

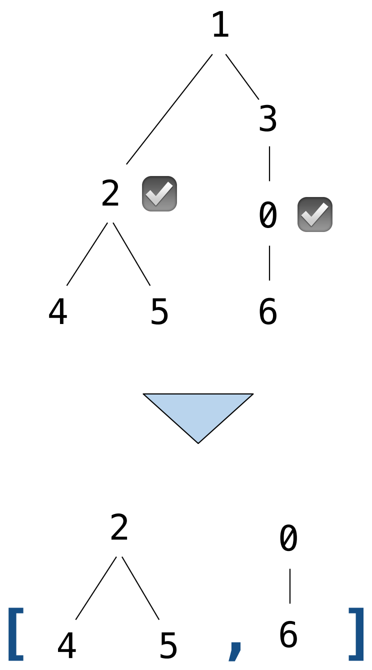
Signature: Tree -> List of Trees

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

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

"Find smallest label in t & maybe add t to 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.

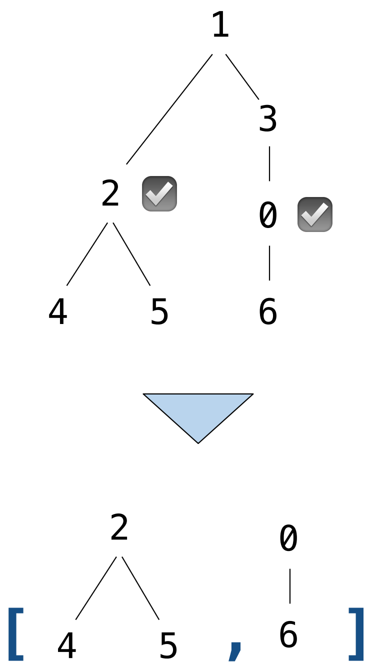
Signature: Tree -> List of Trees

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

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        if t.is_leaf():
            return t.label
        else:
            smallest = min([process(b) for b in t.branches])
            if t.label < smallest:
                result.append( t )
            return min(smallest, t.label)
    process(t)
    return result
```

Signature: Tree -> number

"Find smallest label in t & maybe add t to result"

smallest label in a branch of t



Society

Privacy Policies and Laws

Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010

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

Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010

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

Tim Cook in Brussels, October 24, 2018

Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010

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

Tim Cook in Brussels, October 24, 2018

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

Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010

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

Tim Cook in Brussels, October 24, 2018

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

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

Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010

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

Tim Cook in Brussels, October 24, 2018

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

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

Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010

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

Tim Cook in Brussels, October 24, 2018

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

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

Privacy Policies and Laws

Mark Zuckerberg in San Francisco, January 8, 2010

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

Tim Cook in Brussels, October 24, 2018

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

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

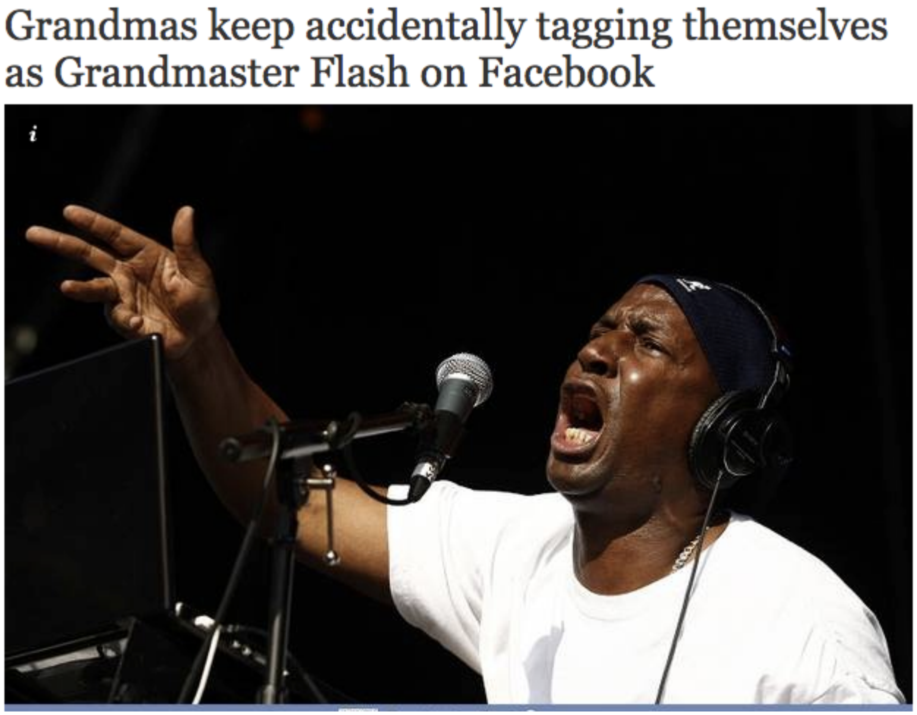
Perils of Sharing

Perils of Sharing

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

Perils of Sharing

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

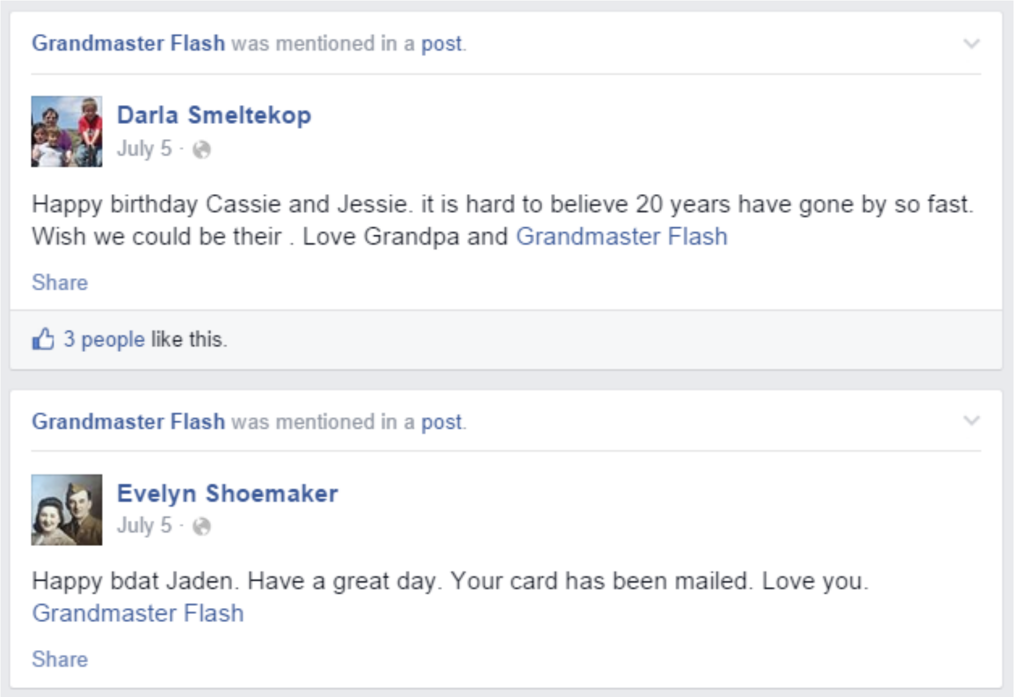


Grandmas keep accidentally tagging themselves as Grandmaster Flash on Facebook

Perils of Sharing

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

Grandmas keep accidentally tagging themselves as Grandmaster Flash on Facebook



Grandmaster Flash was mentioned in a **post**.

Darla Smeltekop
July 5 ·

Happy birthday Cassie and Jessie. it is hard to believe 20 years have gone by so fast. Wish we could be their . Love Grandpa and Grandmaster Flash

Grandmaster Flash was mentioned in a **post**.

Evelyn Shoemaker
July 5 ·

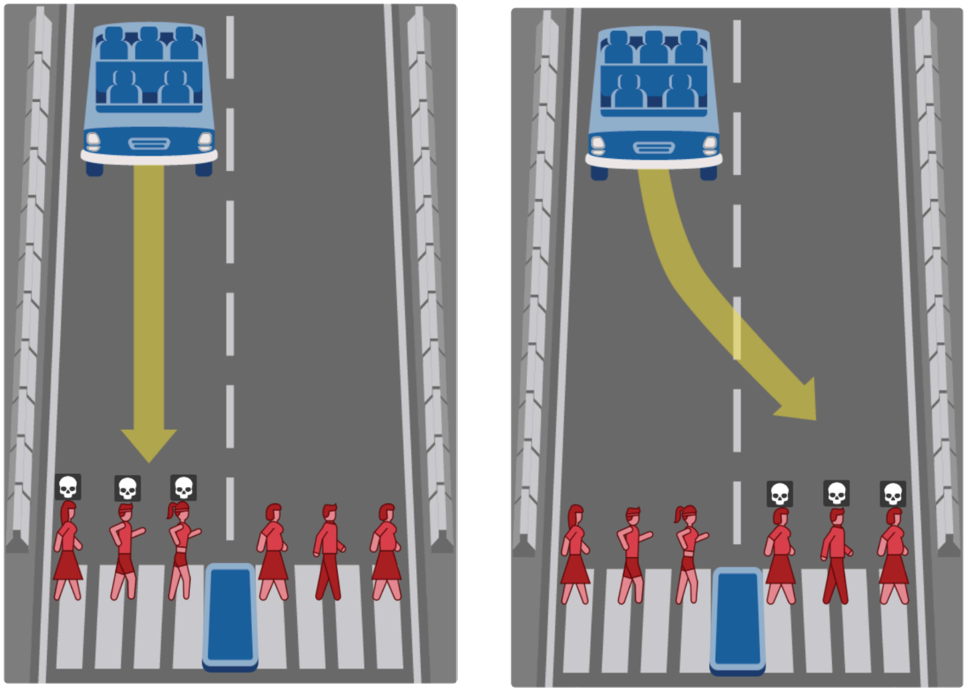
Happy bdat Jaden. Have a great day. Your card has been mailed. Love you. Grandmaster Flash

Software

Automated Decision Making

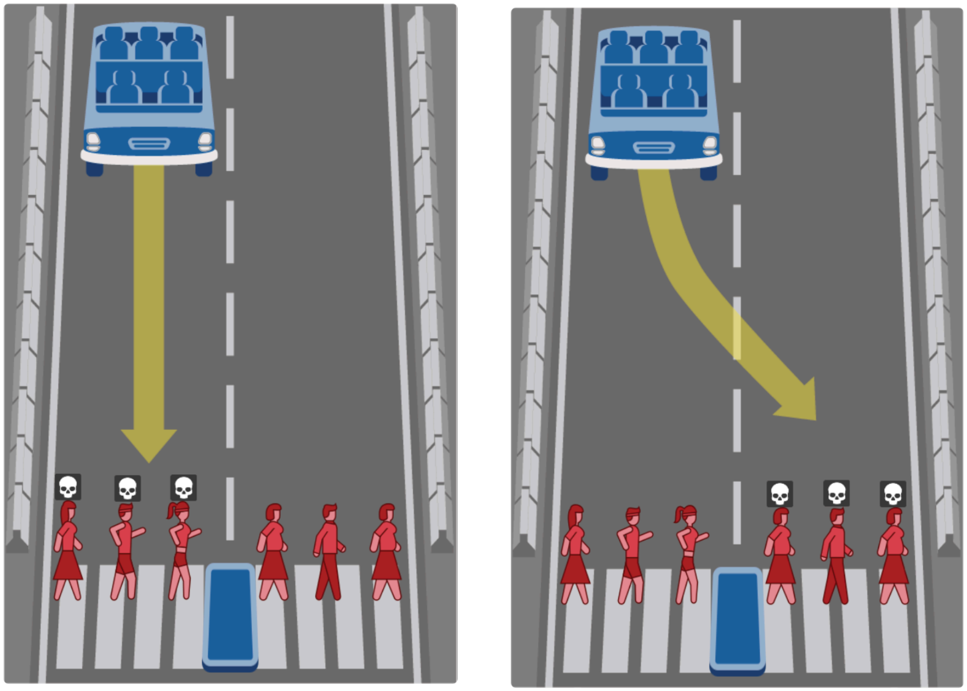
Automated Decision Making

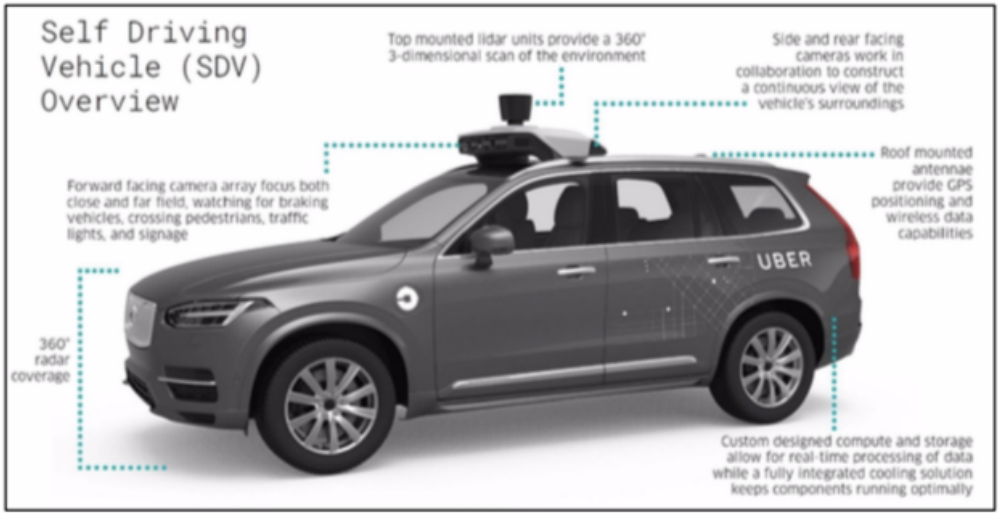
What should the self-driving car do?



Automated Decision Making

What should the self-driving car do?





Self Driving Vehicle (SDV) Overview

Top mounted lidar units provide a 360° 3-dimensional scan of the environment

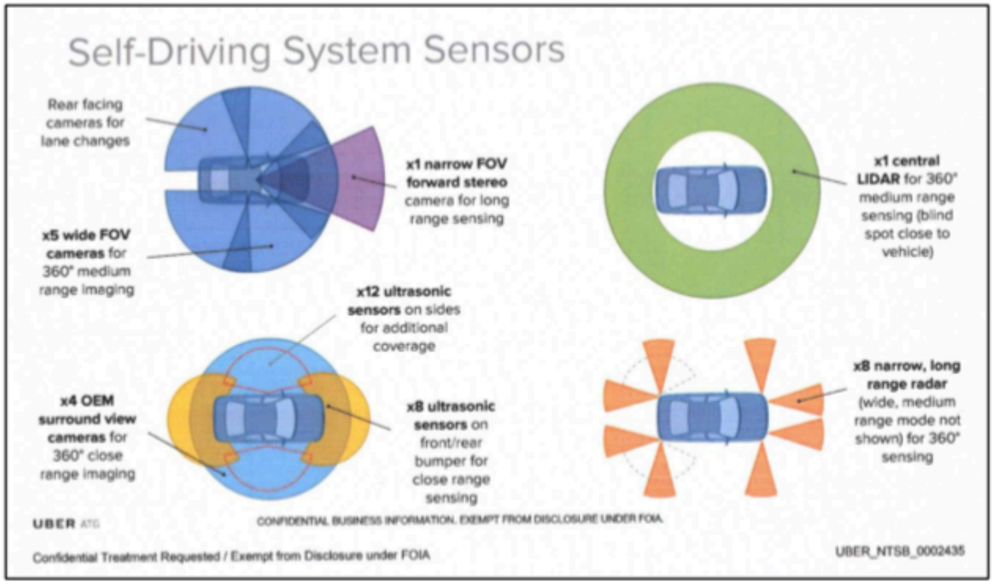
Side and rear facing cameras work in collaboration to construct a continuous view of the vehicle's surroundings

Forward facing camera array focus both close and far field, watching for braking vehicles, crossing pedestrians, traffic lights, and signage

Roof mounted antennae provide GPS positioning and wireless data capabilities

360° radar coverage

Custom designed compute and storage allow for real-time processing of data while a fully integrated cooling solution keeps components running optimally



Self-Driving System Sensors

Rear facing cameras for lane changes

x1 narrow FOV forward stereo camera for long range sensing

x5 wide FOV cameras for 360° medium range imaging

x1 central LIDAR for 360° medium range sensing (blind spot close to vehicle)

x12 ultrasonic sensors on sides for additional coverage

x4 OEM surround view cameras for 360° close range imaging

x8 ultrasonic sensors on front/rear bumper for close range sensing

x8 narrow, long range radar (wide, medium range mode not shown) for 360° sensing

UBER ATG

CONFIDENTIAL BUSINESS INFORMATION. EXEMPT FROM DISCLOSURE UNDER FOIA.

Confidential Treatment Requested / Exempt from Disclosure under FOIA

UBER_NTSB_0002435

Life