

Problem Solving

Announcements

Tree Processing

Using the Tree Abstraction

For a tree `t`, you can **only**:

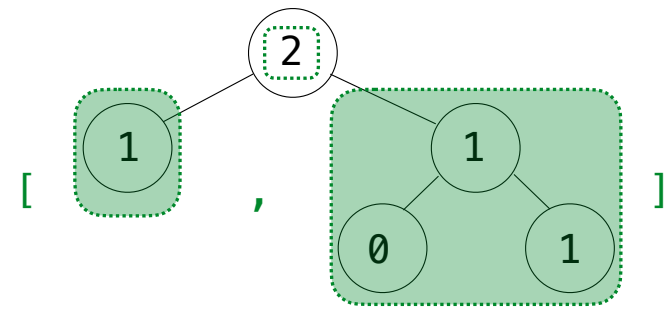
- Get the label for the root of the tree: `t.label`
- Get the list of branches for the tree: `t.branches`
- Get the branch at index `i`, which is a Tree: `t.branches[i]`
- Determine whether the tree is a leaf: `is_leaf(t)`
- Treat `t` as a value: `return t, f(t), [t], s = t`, etc.

Create a tree by calling `Tree` on a label and a list of branches

```
t = Tree(2, [Tree(1), Tree(1, [Tree(0), Tree(1)])])
```

```
isinstance(t, Tree) evaluates to True
```

An example tree `t`:



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 all labels of their ancestor nodes. (Assume the root label is always larger than all its ancestors, since it has no ancestors.)

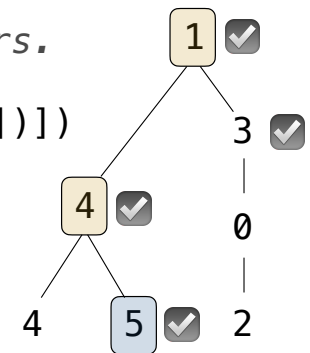
```
def bigs(t: Tree[int]):
```

```
    """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
```



```
if is_leaf(t):  
    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_ancestor:
```

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 all labels of their ancestor nodes. (Assume the root label is always larger than all its ancestors, since it has no ancestors.)

```
def bigs(t: Tree[int]):  
    """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):
```

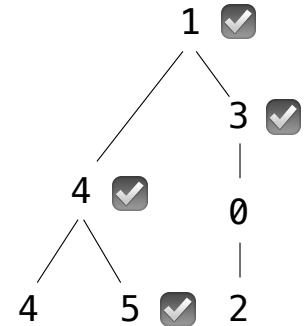
```
    if _____:
```

```
        return 1 + _____
```

```
    else:
```

```
        return _____
```

```
    return _____
```



Solving Tree Problems

Implement **big**s, 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. (Assume the root label is always larger than all its ancestors, since it has no ancestors.)

```
def bigs(t: Tree[int]):
```

```
    """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 ↑ max_ancestor a.label > x node.label > max_ancestor
```

```
    if _:
```

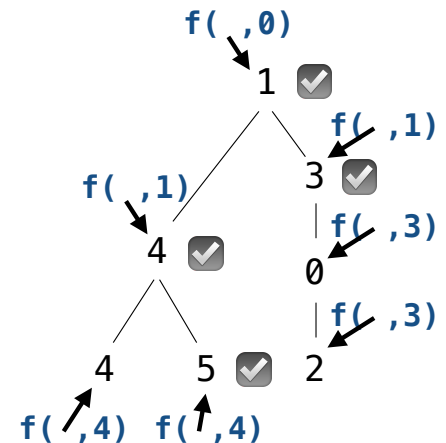
```
        return 1 + sum([f(b, a.label) for b in a.branches])
```

```
    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...

How to Design Programs

Problem Solving Process from *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/2026-5-28//Book/part_preface.html#%28part._sec~3asystematic-design%29

Applying the Design Process

61A Fall 2015 Final Question 3 [Extended Remix]

Definition. A *path* through a Tree is a list of adjacent node labels that starts with the root label and ends with a leaf label.

def **count_big**(t, n): *Signature: (Tree, minimum path sum) -> number of paths*
 """Return the number of paths in t that have a sum larger or equal to n.

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

```
>>> count_big(t, 3)
```

```
3
```

```
>>> count_big(t, 6)
```

```
2
```

```
>>> count_big(t, 9)
```

```
1
```

```
"""
```

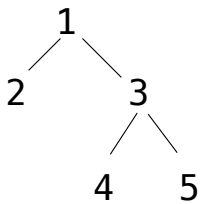
```
if is_leaf(t):
```

```
    return one(_____)
```

```
else:
```

```
    return _____
```

```
def one(b):  
    if b:  
        return 1  
    else:  
        return 0
```



61A Fall 2015 Final Question 3 [Extended Remix]

Definition. A *path* through a Tree is a list of adjacent node labels that starts with the root label and ends with a leaf label.

def **count_big**(t, n): *Signature: (Tree, minimum path sum) -> number of paths*
 """Return the number of paths in t that have a sum larger or equal to n.

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

```
>>> count_big(t, 3)
```

```
3
```

```
>>> count_big(t, 6)
```

```
2
```

```
>>> count_big(t, 9)
```

```
1
```

```
"""
```

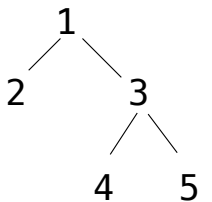
```
if is_leaf(t):
```

```
    return one(_____ t.label >= n _____)
```

```
else:
```

```
    return _____ sum([count_big(b, n-t.label) for b in t.branches]) _____
```

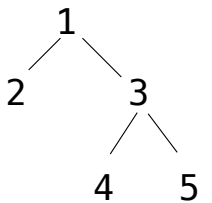
```
def one(b):  
    if b:  
        return 1  
    else:  
        return 0
```



61A Fall 2015 Final Question 3 [Extended Remix]

```
def print_big(t, n):  
    """Print the paths in t that have a sum larger or equal to n.
```

```
>>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])  
>>> print_big(t, 3)  
[1, 2]  
[1, 3, 4]  
[1, 3, 5]  
>>> print_big(t, 6)  
[1, 3, 4]  
[1, 3, 5]  
>>> print_big(t, 9)  
[1, 3, 5]  
"""
```



```
def helper(t, p):
```

Signature: (Tree, list[int]) -> None

"For a node & partial path, extend the path & maybe print it"

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

```
        if _____:  
            print(p)
```

```
    else:
```

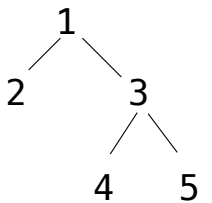
```
        _____:
```

```
    helper(t, _____)
```

61A Fall 2015 Final Question 3 [Extended Remix]

```
def print_big(t, n):  
    """Print the paths in t that have a sum larger or equal to n.
```

```
>>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])  
>>> print_big(t, 3)  
[1, 2]  
[1, 3, 4]  
[1, 3, 5]  
>>> print_big(t, 6)  
[1, 3, 4]  
[1, 3, 5]  
>>> print_big(t, 9)  
[1, 3, 5]  
"""
```



```
def helper(t, p):  
    p = p + [t.label]
```

```
    if is_leaf(t):  
        if sum(p) >= n:  
            print(p)
```

```
    else:  
        for b in t.branches:  
            helper(b, p)
```

```
helper(t, [])
```

Signature: (Tree, list[int]) -> None

"For a node & partial path, extend the path & maybe print it"

Only Paths (Fall 2023 61A Midterm 2)

Implement `only_paths`, which takes a Tree of integers `t` and an int `n`. It returns a Tree with only the nodes of `t` that are on a path from the root to a leaf with labels that sum to `n`, or `None` if no path sums to `n`.

```
def only_paths(t: Tree[int], n: int) -> Tree[int] | None:
```

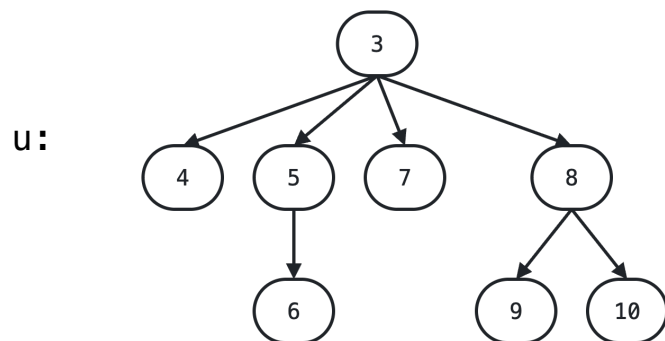
```
    if _____:
```

```
        return t
```

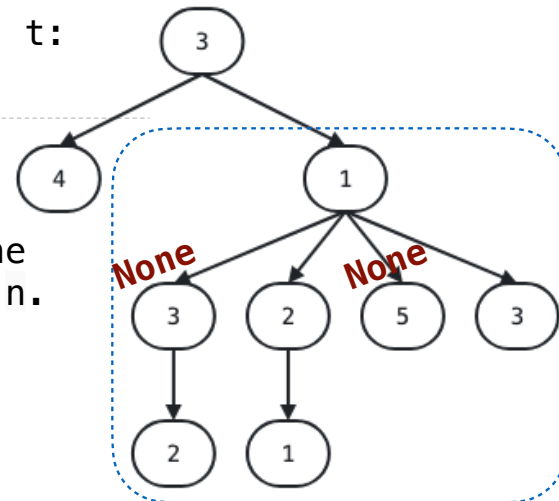
```
    new_branches = [_____ for b in t.branches]
```

```
    if _____:
```

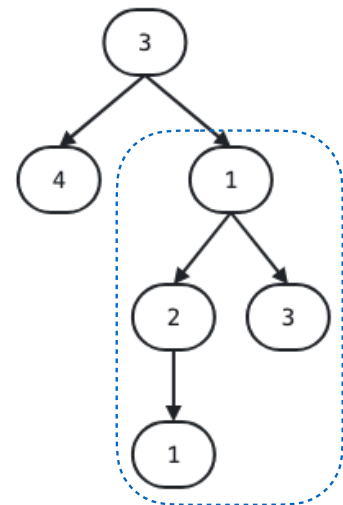
```
        return Tree(t.label, [b for b in new_branches if _____])
```



`only_paths(u, 2): None`



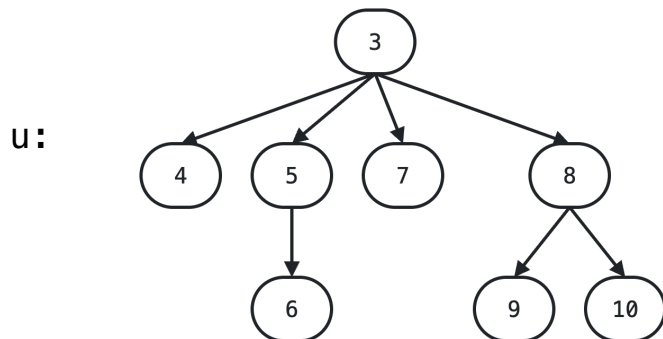
`only_paths(t, 7):`



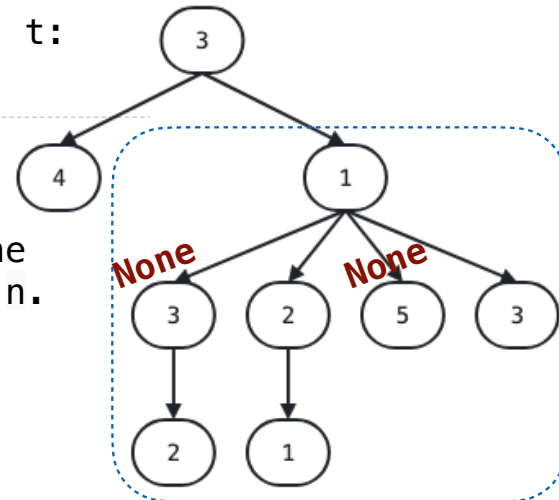
Only Paths (Fall 2023 61A Midterm 2)

Implement `only_paths`, which takes a Tree of integers `t` and an `int n`. It returns a Tree with only the nodes of `t` that are on a path from the root to a leaf with labels that sum to `n`, or `None` if no path sums to `n`.

```
def only_paths(t: Tree[int], n: int) -> Tree[int] | None:
    if is_leaf(t) and t.label == n:
        return t
    new_branches = [only_paths(b, n - t.label) for b in t.branches]
    if any(new_branches):
        return Tree(t.label, [b for b in new_branches if b != None])
```



`only_paths(u, 2): None`



`only_paths(t, 7):`

