Announcements
Tree Class
Tree Review
Recursive description (wooden trees):

Relative description (family trees):
Recursive description (wooden trees):
A tree has a label value and a list of branches

Relative description (family trees):
Recursive description (wooden trees):
A tree has a label value and a list of branches

Relative description (family trees):
**Recursive description (wooden trees):**
A tree has a label value and a list of branches

**Relative description (family trees):**
**Recursive description** *(wooden trees):*

A **tree** has a **label** value and a list of **branches**

Each branch is a **tree**

---

**Relative description** *(family trees):*

---
**Recursive description (wooden trees):**
A **tree** has a **label** value and a list of **branches**
Each branch is a **tree**
A tree with zero branches is called a **leaf**

**Relative description (family trees):**
**Tree Review**

**Recursive description (wooden trees):**
A tree has a label value and a list of branches.
Each branch is a tree.
A tree with zero branches is called a leaf.

**Relative description (family trees):**
**Recursive description (wooden trees):**
A tree has a label value and a list of branches.
Each branch is a tree.
A tree with zero branches is called a leaf.

**Relative description (family trees):**
Each location in a tree is called a node.
**Recursive description** (wooden trees):
A **tree** has a **label** value and a list of **branches**
Each branch is a **tree**
A tree with zero branches is called a **leaf**

**Relative description** (family trees):
Each location in a tree is called a **node**
Recursive description (wooden trees):
A tree has a label value and a list of branches
Each branch is a tree
A tree with zero branches is called a leaf

Relative description (family trees):
Each location in a tree is called a node
Each node has a value
Recursive description (wooden trees):
A tree has a label value and a list of branches.
Each branch is a tree.
A tree with zero branches is called a leaf.

Relative description (family trees):
Each location in a tree is called a node.
Each node has a value.
Recursive description (wooden trees):
A tree has a label value and a list of branches.
Each branch is a tree.
A tree with zero branches is called a leaf.

Relative description (family trees):
Each location in a tree is called a node.
Each node has a value.
One node can be the parent/child of another.
**Recursive description** (wooden trees):  
A **tree** has a **label** value and a list of **branches**  
Each branch is a **tree**  
A tree with zero branches is called a **leaf**

<table>
<thead>
<tr>
<th>Label Values</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Relative description** (family trees):  
Each location in a tree is called a **node**  
Each **node** has a **value**  
One node can be the **parent/child** of another  
Top node of tree is its **root**
**Recursive description (wooden trees):**

A tree has a label value and a list of branches.
Each branch is a tree.
A tree with zero branches is called a leaf.

**Relative description (family trees):**

Each location in a tree is called a node.
Each node has a value.
One node can be the parent/child of another.
Top node of tree is its root.
**Recursive description** *(wooden trees):*

A **tree** has a **label** value and a list of **branches**

Each branch is a **tree**

A tree with zero branches is called a **leaf**

---

**Relative description** *(family trees):*

Each location in a tree is called a **node**

Each **node** has a **value**

One node can be the **parent/child** of another

Top node of tree is its **root**
A Tree has a label value and a list of branches; each branch is a Tree
Tree Class

A Tree has a label value and a list of branches; each branch is a Tree

class Tree:
A Tree has a label value and a list of branches; each branch is a Tree

class Tree:
    def __init__(self, label, branches=[]):
A Tree has a label value and a list of branches; each branch is a Tree

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

A Tree has a label value and a list of branches; each branch is a Tree

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
A Tree has a label value and a list of branches; each branch is a Tree

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
A Tree has a label value and a list of branches; each branch is a Tree

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
A Tree has a label value and a list of branches; each branch is a Tree

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = left.label + right.label
        return Tree(fib_n, [left, right])
A Tree has a label value and a list of branches; each branch is a Tree

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = left.label + right.label
        return Tree(fib_n, [left, right])
### Tree Class

A Tree has a label value and a list of branches; each branch is a Tree.

```python
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = left.label + right.label
        return Tree(fib_n, [left, right])
```

```python
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

def label(tree):
    return tree[0]

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

def fib_tree(n):
    if n == 0 or n == 1:
        return tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = label(left) + label(right)
        return tree(fib_n, [left, right])
```

(Demo)
Side Excursion: Equality

If x and y are two objects, the equality test, x == y, does not automatically mean what you want it to mean.

For example, Tree(4) != Tree(4) but after performing x = Tree(4), we do have x == x

The reason for this is that in Python,

• All values (conceptually, at least) are in fact pointers to objects, and

• By default, == on pointers compares the pointers themselves (“are these pointing at exactly the same object?”).

• That is, by default == and != are the same as the is and is not operators.

• That can be changed on a class-by-class basis. For example, == on numbers, lists, tuples, strings, sets, and dictionaries means what we expect: the contents are the same.
Tree Mutation
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*.

Prune branches before recursive processing.
Example: Pruning Trees

Removing subtrees from a tree is called "pruning".

Prune branches before recursive processing.
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*.

Prune branches before recursive processing.

```python
def prune(t, n):
    """Prune sub-trees whose label value is n."""
    t.branches = [________________ for b in t.branches if ____________________]
    for b in t.branches:
        prune(_________________________, _______________________________)
```
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

```python
def prune(t, n):
    """Prune sub-trees whose label value is n."""
    t.branches = [______________ for b in t.branches if ______b.label != n____]
    for b in t.branches:
        prune(_______________________________, _______________________________)
```
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*.

Prune branches before recursive processing.

```
def prune(t, n):
    #"""Prune sub-trees whose label value is n."""
    t.branches = [_________ for b in t.branches if _______b.label != n______]
    for b in t.branches:
        prune(______________b_____________________, ___________n_____________)
```
Example: Pruning Trees

Removing subtrees from a tree is called pruning

Prune branches before recursive processing

```
def prune(t, n):
    """Prune sub-trees whose label value is n."""
    t.branches = [________ for b in t.branches if __________]
    for b in t.branches:
        prune(__________________________, _______________
        (Demo))
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.
Example: Pruning Trees

Removing subtrees from a tree is called **pruning**.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.

Memoization:
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.

**Memoization:**

- Returned by `fib`

```
fib(5) → fib(4) → fib(3) → fib(2) → fib(1)
1 1 1
0 1
fib(0) fib(1) fib(2) fib(3)
0 1
fib(0) fib(1) fib(2) fib(1) fib(0) fib(1)
0 1
```
Example: Pruning Trees

Removing subtrees from a tree is called pruning

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:
- Returned by fib
- Found in cache
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.

**Memoization:**
- Returned by fib
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a tree is called \textit{pruning}.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.

\textbf{Memoization:}

- Returned by \textit{fib}
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.

**Memoization:**
- Returned by fib
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

**Memoization:**
- Returned by `fib`
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a tree is called pruning.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.

Memoization:
- Returned by fib
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.

**Memoization:**
- Returned by fib
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a tree is called **pruning**

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

**Memoization:**
- Returned by fib
- Found in cache
- Skipped

![Diagram of Fibonacci sequence with memoization](image)
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.

**Memoization:**
- Returned by `fib`
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a tree is called **pruning**

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

**Memoization:**
- Returned by fib
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a tree is called **pruning**

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

**Memoization:**
- Returned by fib
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a tree is called **pruning**

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

**Memoization:**
- Returned by `fib`
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.

**Memoization:**
- Returned by `fib`
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a tree is called prunning.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.

Memoization:
- Returned by fib
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.

**Memoization:**
- Returned by fib
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a tree is called pruning.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.

Memoization:
- Returned by fib
- Found in cache
- Skipped

Diagram:

```
fib(5)  
   /     
  /      
fib(4)  fib(3)  
  /      /     
/ fib(2) fib(1) fib(0) 1
  / 1
/ 0 1
/ 0
/ 1
```
Example: Pruning Trees

Removing subtrees from a tree is called pruning.

Prune branches before recursive processing.

E.g., want to prune cached (previously memorized) values.

Memoization:
- Returned by fib
- Found in cache
- Skipped
Example: Pruning Trees

Removing subtrees from a
tree is called \textit{pruning}.

Prune branches before
recursive processing.

E.g., want to prune
cached (previously
memorized) values.

\textbf{Memoization:}
- Returned by \texttt{fib}
- Found in cache
- Skipped

(Demo)
Hailstone Trees
Hailstone Trees
Hailstone Trees

Pick a positive integer n as the start
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2

If $n$ is odd, multiply it by 3 and add 1
Hailstone Trees

Pick a positive integer $n$ as the start
If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1
Hailstone Trees

Pick a positive integer $n$ as the start
If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1

(Demo)
Hailstone Trees

Pick a positive integer \( n \) as the start
If \( n \) is even, divide it by 2
If \( n \) is odd, multiply it by 3 and add 1
Continue this process until \( n \) is 1

(Demo)
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1

(Demo)
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1

Continue this process until \( n \) is 1

(Demo)
Hailstone Trees

Pick a positive integer $n$ as the start
If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1

(Demo)
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1

Continue this process until \( n \) is 1

(Demo)
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2

If $n$ is odd, multiply it by 3 and add 1

Continue this process until $n$ is 1

(Demo)
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2

If $n$ is odd, multiply it by 3 and add 1

Continue this process until $n$ is 1

(Demo)
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2

If $n$ is odd, multiply it by 3 and add 1

Continue this process until $n$ is 1

(Demo)
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1

Continue this process until \( n \) is 1

(Demo)

\[
\begin{align*}
1 \\
2 \\
4 \\
8 \\
16 \\
32 \\
64 \\
128
\end{align*}
\]
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2

If $n$ is odd, multiply it by 3 and add 1

Continue this process until $n$ is 1

(Demo)

```
<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>128</td>
</tr>
</tbody>
</table>
```
Hailstone Trees

Pick a positive integer $n$ as the start
If $n$ is even, divide it by 2
If $n$ is odd, multiply it by 3 and add 1
Continue this process until $n$ is 1

(Demo)
Hailstone Trees

Pick a positive integer $n$ as the start

If $n$ is even, divide it by 2

If $n$ is odd, multiply it by 3 and add 1

Continue this process until $n$ is 1

(Demo)
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1

Continue this process until \( n \) is 1

(Demo)
Hailstone Trees

Pick a positive integer n as the start
If n is even, divide it by 2
If n is odd, multiply it by 3 and add 1
Continue this process until n is 1

(Demo)

All possible n that start a length-8 hailstone sequence
Hailstone Trees

Pick a positive integer \( n \) as the start

If \( n \) is even, divide it by 2

If \( n \) is odd, multiply it by 3 and add 1

Continue this process until \( n \) is 1

(Demo)

def hailstone_tree(k, n=1):
    """Return a Tree in which the paths from the leaves to the root are all possible hailstone sequences of length \( k \) ending in \( n \).""

All possible \( n \) that start a length-8 hailstone sequence
Hailstone Trees

Pick a positive integer n as the start
If n is even, divide it by 2
If n is odd, multiply it by 3 and add 1
Continue this process until n is 1

```
def hailstone_tree(k, n=1):
    """Return a Tree in which the paths from the leaves to the root are all possible hailstone sequences of length k ending in n."""
    # Code implementation
```

All possible n that start a length-8 hailstone sequence

(Demo)