

Efficiency

Announcements

Tree Class

Tree Class

A Tree has a label 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])
```

```
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])
```

Tree Practice

Example: Count Twins

Implement `twins`, which takes a Tree `t`. It return the number of pairs of sibling nodes whose labels are equal.

```
def twins(t):
```

```
    """Count the pairs of sibling nodes with equal labels.
```

```
    >>> t1 = Tree(3, [Tree(4, [Tree(5), Tree(6)]), Tree(4, [Tree(5), Tree(5)])])
```

```
    >>> twins(t1) # 4 and 5
```

```
    2
```

```
    >>> twins(Tree(1, [Tree(1, [Tree(2)]), Tree(2, [Tree(2)])]))
```

```
    0
```

```
    >>> twins(Tree(8, [t1, t1, t1])) # 3 pairs of twins at the top, plus 2 in each branch
```

```
    9
```

```
    """
```

```
    count = 0
```

```
    n = len(t.branches)
```

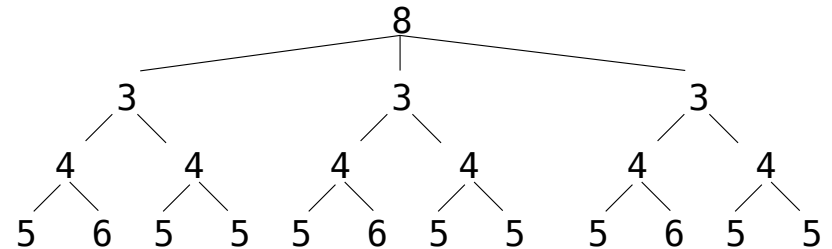
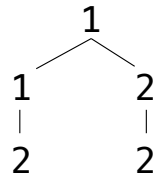
```
    for i in range(n-1):
```

```
        for j in range(i+1, n):
```

```
            if t.branches[i].label == t.branches[j].label:
```

```
                count += 1
```

```
    return count + sum([twins(b) for b in t.branches])
```



Spring 2023 Midterm 2 Question 4(b)

You have already implemented `exclude(t, x)`, which takes a `Tree` instance `t` and a value `x`. It returns a `Tree` containing the root node of `t` as well as each non-root node of `t` with a label not equal to `x`. The parent of a node in the result is its nearest ancestor node that is not excluded. The input `t` is not modified.

Implement `remove`, which takes a `Tree` instance `t` and a value `x`. It removes all non-root nodes from `t` that have a label equal to `x`, then returns `t`. The parent of a node in `t` is its nearest ancestor that is not removed.

```
def remove(t, x):  
    """Remove all non-root nodes labeled x from t.
```

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

```
>>> remove(t, 2)
```

```
Tree(1, [Tree(3), Tree(4)])
```

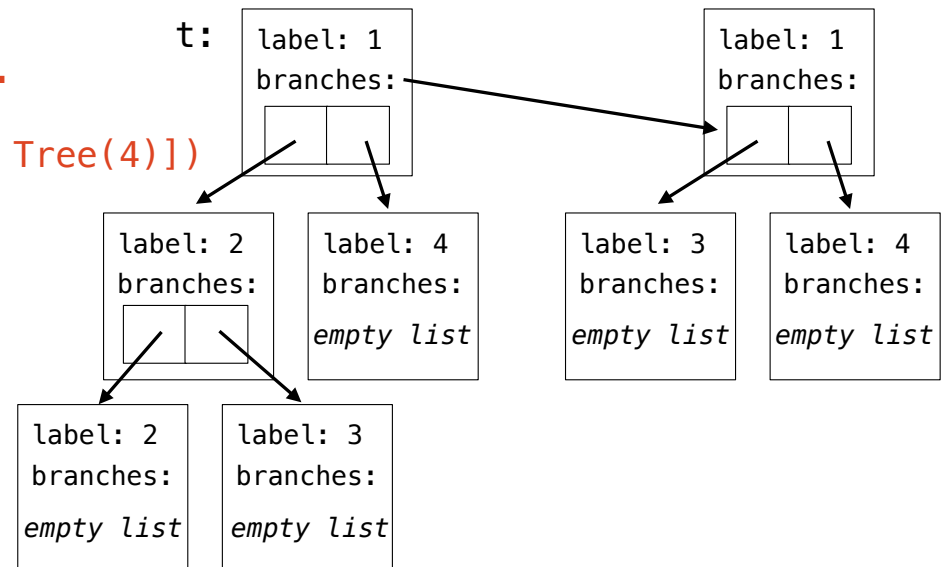
```
>>> remove(t, 3)
```

```
Tree(1, [Tree(4)])
```

```
"""
```

```
t.branches = exclude(t, x).branches
```

```
return t
```

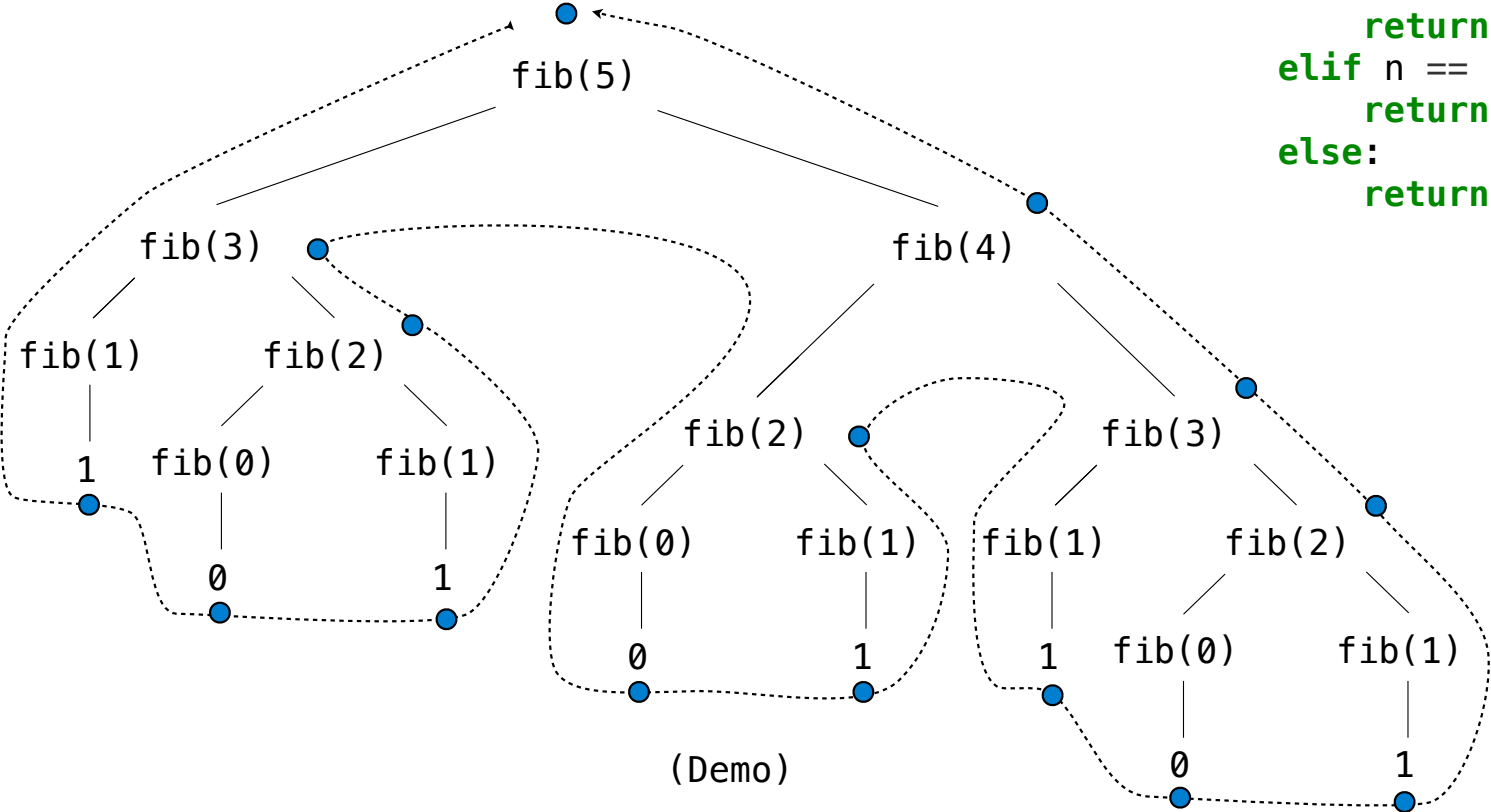


Measuring Efficiency

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fib(n-2) + fib(n-1)
```



Memoization

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):  
    cache = {}  
    def memoized(n):  
        if n not in cache:  
            cache[n] = f(n)  
        return cache[n]  
    return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)

Orders of Growth

Common Orders of Growth

Exponential growth. E.g., recursive `fib`

Incrementing n multiplies *time* by a constant

Quadratic growth.

Incrementing n increases *time* by n times a constant

Linear growth.

Incrementing n increases *time* by a constant

Logarithmic growth.

Doubling n only increments *time* by a constant

Constant growth. Increasing n doesn't affect time

Spring 2023 Midterm 2 Question 3(a) Part (iii)

Definition. A *prefix sum* of a sequence of numbers is the sum of the first n elements for some positive length n .

(1 pt) What is the order of growth of the time to run `prefix(s)` in terms of the length of `s`? Assume `append` takes one step (constant time) for any arguments.

```
def prefix(s):  
    "Return a list of all prefix sums of list s."  
    t = 0  
    result = []  
    for x in s:  
        t = t + x  
        result.append(t)  
    return result
```