CS 61A Spring 2021

More Trees and OOP

Lost 5: March 5, 2021

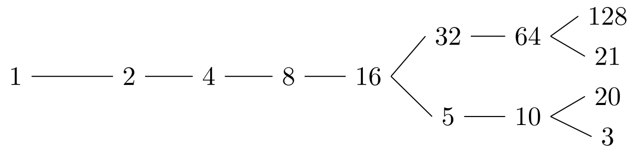
1 Learning Goals

- Learn to approach complex tree questions
- Touch on iterators and generators in the context of other topics
- Learn the basics of object-oriented programming

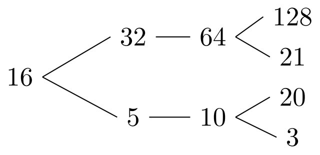
2 Trees (Exam-Level)

2.1 We can represent the hailstone sequence as a tree in the figure below, showing the route different numbers take to reach 1. Remember that a hailstone sequence starts with a number n, continuing to $n/2$ if n is even or $3n + 1$ if n is odd, ending with 1. Write a function `hailstone_tree(n, h)` which generates a tree of height `h`, containing hailstone numbers that will reach `n`.

Hint: A node of a hailstone tree will always have at least one, and at most two branches (which are also hailstone trees). Under what conditions do you add the second branch?



`hailstone_tree(1, 7)`



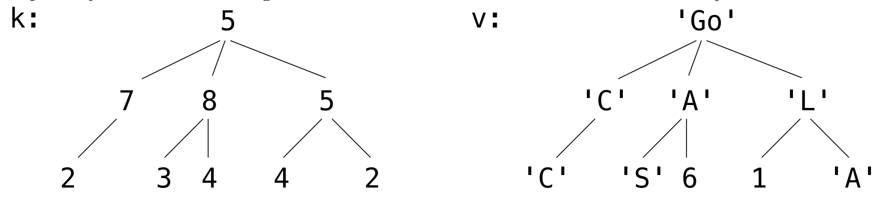
`hailstone_tree(16, 3)`

```
def hailstone_tree(n, h):
    """Generates a tree of hailstone numbers that will
       reach N, with height H.
    >>> hailstone_tree(1, 0)
    [1]
    >>> hailstone_tree(1, 4)
    [1, [2, [4, [8, [16]]]]]
    >>> hailstone_tree(8, 3)
    [8, [16, [32, [64]], [5, [10]]]]
    """
```

2.2 **Fall 2018 Midterm 2, Question 7: Trictionary or Treat** A trictionary is a pair of *Tree* instances *k* and *v* that have identical structure: each node in *k* has a corresponding node in *v*. The labels in *k* are called keys. Each key may be the label for multiple nodes in *k*, and the values for that key are the labels of all the corresponding nodes in *v*.

A lookup function returns one of the values for a key. Specifically, a lookup function for a node in *k* is a function that takes *v* as an argument and returns the label for the corresponding node in *v*.

Implement the generator function *lookups*, which takes a *Tree* instance *k* and some *key*. It yields all lookup functions for nodes in k that have key as their label.



key	*values*
2	'C' , 'A'
3	'S'
4	6 , 1
5	'Go', 'L'
7	'C'
8	'A'

```
k = Tree(5, [Tree(7, [Tree(2)]), Tree(8, [Tree(3), Tree(4)]), Tree(5, [Tree(4), Tree(2)])])
v = Tree('Go', [Tree('C', [Tree('C')]), Tree('A', [Tree('S'), Tree(6)]), Tree('L', [Tree(1), Tree('A'
    )])])

def lookups(k, key):
    """Yield one lookup function for each node of k that has the label key.
    >>> [f(v) for f in lookups(k, 2)]
    ['C', 'A']
    >>> [f(v) for f in lookups(k, 3)]
    ['S']
    >>> [f(v) for f in lookups(k, 6)]
    []
    """

    if ___________________________________________________________________________:

        yield lambda v: _____________________________________________________________

    for i in range(len(k.branches)):

        __________________________________________________________________________:

            yield new_lookup(i, lookup)

def new_lookup(i, f):
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

```
def g(v):

    return ____________________________________________________________________________________

return
```

3 Object-Oriented Programming

3.1 What is the relationship between a class and an ADT?

3.2 What is the definition of a Class? What is the definition of an Instance?

3.3 What is a Class Attribute? What is an Instance Attribute?

3.4 What Would Python Display?

```
class Foo():
    x = 'bam'
    def __init__(self, x):
        self.x = x
    def baz(self):
        return self.x

class Bar(Foo):
        x = 'boom'
        def __init__(self, x):
                Foo.__init__(self, 'er' + x)
        def baz(self):
                return Bar.x + Foo.baz(self)

foo = Foo('boo')

Foo.x

foo.x

foo.baz()

Foo.baz()

Foo.baz(foo)

bar = Bar('ang')
Bar.x

bar.x

bar.baz()
```

3.5 What Would Python Display?

```
class Student:
        def __init__(self, subjects):
                self.current_units = 16
                self.subjects_to_take = subjects
                self.subjects_learned = {}
                self.partner = None

        def learn(self, subject, units):
                print('I just learned about ' + subject)
                self.subjects_learned[subject] = units
                self.current_units -= units

        def make_friends(self):
                if len(self.subjects_to_take) > 3:
                        print('Whoa! I need more help!')
                        self.partner = Student(self.subjects_to_take[1:])
                else:
                        print("I'm on my own now!")
                        self.partner = None

        def take_course(self):
                course = self.subjects_to_take.pop()
                self.learn(course, 4)
                if self.partner:
                        print('I need to switch this up!')
                        self.partner = self.partner.partner
                        if not self.partner:
                                print('I have failed to make a friend :(')

tim = Student(['Chem1A', 'Bio1B', 'CS61A', 'CS70', 'CogSci1'])
tim.make_friends()

print(tim.subjects_to_take)

tim.partner.make_friends()

tim.take_course()

tim.partner.take_course()

tim.take_course()

tim.make_friends()
```

3.6 Fill in the implementation for the Cat and Kitten classes. When a cat meows, it should say "Meow, (name) is hungry" if it is hungry, and "Meow, my name is (name)" if not. Kittens do the same thing as cats, except they say "i'm baby" instead of "meow", and they say "I want mama (parent's name)" after every call to meow().

```
>>>cat = Cat('Tuna')
>>>kitten = kitten('Fish', cat)
>>>cat.meow()
meow, Tuna is hungry
>>>kitten.meow()
i'm baby, Fish is hungry
I want mama Tuna
>>>cat.eat()
meow
>>>cat.meow()
meow, my name is Tuna
>>>kitten.eat()
i'm baby
>>>kitten.meow()
meow, my name is Fish
I want mama Tuna

class Cat():
    noise = 'meow'
    def __init__(self, name):







    def meow(self):







    def eat(self):
        print(self.noise)
        self.hungry = False

class Kitten(Cat):
```