# CS 61A OOP, Nonlocal, Trees, LLs, Growth
# Spring 2019 Guerrilla Section 3: March 16, 2019 Solutions

# 1 OOP

## Questions

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

In general, we can think of an abstract data type as defined by some collection of selectors and constructors, together with some behavior conditions. As long as the behavior conditions are met (such as the division property above), these functions constitute a valid representation of the data type.

There are two different layers to the abstract data type:

1) The program layer, which uses the data, and
2) The concrete data representation that is independent of the programs that use the data. The only communication between the two layers is through selectors and constructors that implement the abstract data in terms of the concrete representation.

Classes are a way to implement an Abstract Data Type. But, ADTs can also be created using a collection of functions, as shown by the rational number example. (See Composing Programs 2.2)

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

Class: a template for all objects whose type is that class that defines attributes and methods that an object of this type has.
Instance: A specific object created from a class. Each instance shares class attributes and stores the same methods and attributes. But the values of the attributes are independent of other instances of the class. For example, all humans have two eyes but the color of their eyes may vary from person to person.

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

Class Attribute: A static value that can be accessed by any instance of the class and is shared among all instances of the class.
Instance Attribute: A field or property value associated with that specific instance of the object.

1.4 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
```

'bam'

```
foo.x
```

'boo'

```
foo.baz()
```

'boo'

```
Foo.baz()
```

Error

```
Foo.baz(foo)
```

'boo'

```
bar = Bar('ang')
Bar.x
```

'boom'

```
bar.x
```

'erang'

```
bar.baz()
```

'boomerang'

## 1.5 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()
```

```
Whoa! I need more help!
```

```
print(tim.subjects_to_take)
```

```
['Chem1A', 'Bio1B', 'CS61A', 'CS70', 'CogSci1']
```

```
tim.partner.make_friends()
```

```
Whoa! I need more help!
```

```
tim.take_course()
```

```
I just learned about CogSci1
I need to switch this up!
```

```
tim.partner.take_course()
```

```
I just learned about CogSci1
```

```
tim.take_course()
```

```
I just learned about CS70
I need to switch this up!
I have failed to make a friend :(
```

```
tim.make_friends()
```

```
I?m on my own now!
```

# 2 Nonlocal

## Questions

2.1 Draw an environment diagram for the following code:

```
ore = "settlers"
def sheep(wood):
    def ore(wheat):
        nonlocal ore
        ore = wheat
    ore(wood)
    return ore
sheep(lambda wood: ore)("wheat")
```

https://goo.gl/dfYjcT

2.2 Draw an environment diagram for the following code:

```
aang = 120
def airbend(zuko):
    aang = 2
    def katara(aang):
        nonlocal zuko
        zuko = lambda sokka : aang + 4
        return aang
    if zuko(10) == 1:
        katara(aang + 9)
    return zuko(airbend)
airbend(lambda x: aang + 1)
```

https://goo.gl/NKCbnD

2.3 Write **make_max_finder**, which takes in no arguments but returns a function which takes in a list. The function it returns should return the maximum value it?s been called on so far, including the current list and any previous list. You can assume that any list this function takes in will be nonempty and contain only non-negative values.

```
def make_max_finder():
    """
        >>> m = make_max_finder()
        >>> m([5, 6, 7])
        7
        >>> m([1, 2, 3])
        7
        >>> m([9])
        9
        >>> m2 = make_max_finder()
        >>> m2([1])
        1
        """

    max_so_far = 0
    def find_max_overall(lst):
        nonlocal max_so_far
        if max(lst) > max_so_far:
            max_so_far = max(lst)
        return max_so_far
    return find_max_overall
```

# 3 Object Oriented Trees

## Questions

3.1 Define **filter_tree**, which takes in a tree **t** and one argument predicate function **fn**. It should mutate the tree by removing all branches of any node where calling **fn** on its label returns **False**. In addition, if this node is not the root of the tree, it should remove that node from the tree as well.

```
def filter_tree(t, fn):
    """
    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4)]), Tree(6, [Tree(7)])])
    >>> filter_tree(t, lambda x: x % 2 != 0)
    >>> t
    tree(1, [Tree(3)])
    >>> t2 = Tree(2, [Tree(3), Tree(4), Tree(5)])
    >>> filter_tree(t2, lambda x: x != 2)
    >>> t2
    Tree(2)
    """

    if not fn(t.label):
        t.branches = []
    else:
        for b in t.branches[:]:
            if not fn(b.label):
                t.branches.remove(b)
            else:
                filter_tree(b, fn)
```

3.2 Fill in the definition for **nth_level_tree_map**, which also takes in a function and a tree, but mutates the tree by applying the function to every nth level in the tree, where the root is the 0th level.

```
def nth_level_tree_map(fn, tree, n):
   """Mutates a tree by mapping a function all the elements of a tree.
   >>> tree = Tree(1, [Tree(7, [Tree(3), Tree(4), Tree(5)]),
                Tree(2, [Tree(6), Tree(4)])])
   >>> nth_level_tree_map(lambda x: x + 1, tree, 2)
   >>> tree
   Tree(2, [Tree(7, [Tree(4), Tree(5), Tree(6)]),
            Tree(2, [Tree(7), Tree(5)])])
   """

    def helper(tree, level):
          if level % n == 0:
               tree.label = fn(tree.label)
          for b in tree.branches:
```

```
            helper(b, level + 1)
helper(tree, 0)
```

# 4 Linked Lists

## Questions

4.1 What is a linked list? Why do we consider it a naturally recursive structure?

A linked list is a data structure with a first and a rest, where the first is some arbitrary element and the rest MUST be another linked list (or Link.empty)

4.2 Draw a box and pointer diagram for the following:

```
Link('c', Link(Link(6, Link(1, Link('a'))), Link('s')))
```

4.3 The Link class can represent lists with cycles. That is, a list may contain itself as a sublist. Implement **has_cycle** that returns whether its argument, a Link instance, contains a cycle. There are two ways to do this: iteratively with two pointers, or keeping track of Link objects we've seen already. Try to come up with both!

```
def has_cycle(link):
    """
    >>> s = Link(1, Link(2, Link(3)))
    >>> s.rest.rest.rest = s
    >>> has_cycle(s)
    True
    """


    # solution 1
    tortoise = link
    hare = link.rest
    while tortoise.rest and hare.rest and hare.rest.rest:
        if tortoise is hare:
            return True
        tortoise = tortoise.rest
        hare = hare.rest.rest
    return False

    # solution 2
    seen = []
    while link.rest:
        if link in seen:
            return True
        seen.append(link)
        link = link.rest
    return False
```

4.4 Fill in the following function, which checks to see if **sub_link**, a particular sequence of items in one linked list, can be found in another linked list (the items have to be

in order, but not necessarily consecutive).

```
def seq_in_link(link, sub_link):
    """
    >>> lnk1 = Link(1, Link(2, Link(3, Link(4))))
    >>> lnk2 = Link(1, Link(3))
    >>> lnk3 = Link(4, Link(3, Link(2, Link(1))))
    >>> seq_in_link(lnk1, lnk2)
    True
    >>> seq_in_link(lnk1, lnk3)
    False
    """

    if sub_link is Link.empty:
        return True
    if link is Link.empty:
        return False
    if link.first == sub_link.first:
        return seq_in_link(link.rest, sub_link.rest)
    else:
        return seq_in_link(link.rest, sub_link)
```

# 5 Growth

## Questions

5.1 What is the runtime of the following function?

```
def one(n):
    if 1 == 1:
        return None
    return n
```

a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

Theta(1) - the function always returns None, because 1 == 1 is always True. And even if it was a false statement, the function would just return n. So since the runtime of the function doesn't change with respect to the size of the input, it is constant time.

5.2 What is the runtime of the following function?

```
def two(n):
    for i in range(n):
        print(n)
```

a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

Theta(n) - the function iterates n times; if n increases by 1, the function loops 1 additional time. Therefore there is a linear relationship between the input size and runtime.

5.3 What is the runtime of the following function?

```
def three(n):
    while n > 0:
       n = n // 2
```

a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

Theta(log n) - The function continues to loop as long as n ¿ 0. Inside the while loop, we divide n by 2 every loop. So to get the function to loop one additional time, we need to double our original input size. This is a logarithmic relationship between input size and runtime.

5.4 What is the runtime of the following function?

```
def four(n):
    for i in range(n):
        for j in range(i):
            print(str(i), str(j))
```

a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

d. Theta($n^2$) - The outer loop loops through every number from 0 to n. The inner loop loops corresponding to the outer loop. So the total number of loops from the

inner loop looks like this: 0 + 1 + 2 + 3 + 4 ... + n. This is the summation of the first n natural numbers = n(n + 1)/2, which asymptotically is Theta($n^2$)

5.5 What is the runtime of the following function?

```
def five(n):
    if n <= 0:
        return 1
    return five(n - 1) + five(n - 2)
```

a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

e. Theta($2^n$) - Draw out the tree of recursive calls. You should see that every node branches out into 2 more nodes. Since the base case returns when n <= 0, and each recursive call subtracts 1 or 2 from n, the height of our tree is n. We're branching out by a factor of 2 each layer for n layers – that means we'll have $2^n$ nodes in our tree of recursive calls. Each 'node' represents 1 'unit of work' as all the function does is return something. So 1 unit of work across $2^n$ nodes is $2^n$ total.

5.6 What is the runtime of the following function?

```
def five(n):
    if n <= 0:
        return 1
    return five(n//2) + five(n//2)
```

a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

c. Theta(n) - Draw out the tree of recursive calls. You should see that every node branches out into 2 more nodes. Since the base case returns when n <= 0, and each recursive call divides n by 2, the height of our tree is logn (by the same logic as three(n): if we want one additional layer in our tree, our original input has to be doubled, which is a logarithmic relationship). We're branching out by a factor of 2 each layer for logn layers – that means we'll have $2^(logn) = n$ nodes in our tree of recursive calls. Each 'node' represents 1 'unit of work' as all the function does is return something. So 1 unit of work across n nodes is n total.