CS 61A Fall 2017 — Midterm 2 Review

October 13, 2017

Instructions

Form a small group. Start on the first problem. Check off with a helper or discuss your *solution process* with another group once everyone understands *how to solve* the first problem and then repeat for the second problem ...

You may not move to the next problem until you check off or discuss with another group and *everyone understands why the solution is what it is.* You may use any course resources at your disposal: the purpose of this review session is to have everyone learning together as a group.

0.1 What would Python display?

```
>>> pikachu, charmander = 'electric', 'fire'
>>> ash = [[pikachu], [charmander], [[pikachu]]]
>>> pikachu, charmander = 2, 0
>>> ash[pikachu] = [ash, ash[pikachu][charmander]]
>>> ash
```

```
[['electric'], ['fire'], [[...], ['electric']]]
```

1 Lists & Tree Recursion

Mutative (*destructive*) operations change the state of a list by adding, removing, or otherwise modifying the list itself.

- `lst.append(element)`
- `lst.extend(lst)`
- `lst.pop(index)`
- `lst += lst` (**not** `lst = lst + lst`)
- `lst[i] = x`
- `lst[i:j] = lst`

Non-mutative (*non-destructive*) operations include the following.

- `lst + lst`
- `lst * n`
- `lst[i:j]`
- `list(lst)`

Recall: To execute assignment statements,

- Evaluate all expressions to the right of the `=` sign
- Bind all names to the left of the `=` to those resulting values

The **Golden Rule of Equals** describes how this rule behaves with composite values. *Composite values*, such as functions and lists, are connected by a pointer. When an expression evaluates to a composite value, we are returned the pointer to that value, rather than the value itself.

In an environment diagram, we can summarize this rule with,

Copy *exactly* what is in the box!

1.1 Write a list comprehension that accomplishes each of the following tasks.

(a) Square all the elements of a given list, `lst`.

```
[x ** 2 for x in lst]
```

(b) Compute the dot product of two lists `lst1` and `lst2`. *Hint*: The dot product is defined as $\text{lst1}[0] \cdot \text{lst2}[0] + \text{lst1}[1] \cdot \text{lst2}[1] + \ldots + \text{lst1}[n] \cdot \text{lst2}[n]$. The Python **`zip`** function may be useful here.

```
sum([x * y for x, y in zip(lst1, lst2)])
```

(c) `[[0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4]]`

```
[[x for x in range(y)] for y in range(1, 6)]
```

(d) Return the same list as above, except now excluding every instance of the number 2: `[[0], [0, 1], [0, 1], [0, 1, 3], [0, 1, 3, 4]]`.

```
[[x for x in range(y) if x != 2] for y in range(1, 6)]
```

1.2 Draw the environment diagram that results from running the following code.

```
pom = [16, 15, 13]
pompom = pom * 2
pompom.append(pom[:])
pom.extend(pompom)
```

https://goo.gl/ZU1V7h

1.3 Draw the environment diagram that results from running the following code.

```
bless, up = 3, 5
another = [1, 2, 3, 4]
one = another[1:]

another[bless] = up
another.append(one.remove(2))
another[another[0]] = one
one[another[0]] = another[1]
one = one + [another.pop(3)]
another[1] = one[1][1][0]
one.append([one.pop(1)])
```

https://goo.gl/FyMmbJ

1.4
```
def jerry(jerry):
    def jerome(alex):
        alex.append(jerry[1:])
        return alex
    return jerome

ben = ['nice', ['ice']]
jerome = jerry(ben)
alex = jerome(['cream'])
ben[1].append(alex)
ben[1][1][1] = ben
print(ben)
```

https://goo.gl/uhSClr

1.5 Implement `subset_sum`, which takes in a list of integers and a number k and returns whether there is a subset of the list that adds up to k? *Hint*: The **in** operator can determine if an element belongs to a list.

```
def subset_sum(seq, k):
    """
    >>> subset_sum([2, 4, 7, 3], 5)         # 2 + 3 = 5
    True
    >>> subset_sum([1, 9, 5, 7, 3], 2)
    False
    """

    if len(seq) == 0:
        return False
    elif k in seq:
        return True
    else:
        return subset_sum(seq[1:], k - seq[0]) or subset_sum(seq[1:], k)
```

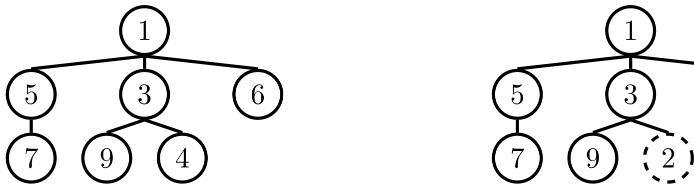
2 Trees

```
def tree(label, branches=[]):
    return [label] + list(branches)

def label(tree):
    return tree[0]

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

2.1 A **min-heap** is a tree with the special property that every node's value is less than or equal to the values of all of its branches.



Implement `is_min_heap` which takes in a `tree` data abstraction and returns whether the tree satisfies the min-heap property or not.

```
def is_min_heap(t):
    for b in branches(t):
        if label(t) > label(b) or not is_min_heap(b):
             return False
    return True
```

3 Growth

3.1 Give a tight asymptotic runtime bound for the following functions in $\Theta(\cdot)$ notation, or "Infinite" if the program does not terminate.

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

$\Theta(\log n)$

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

$\Theta(n^2)$

(c)
```
def three(n):
    i = 1
    while i <= n:
        for j in range(i):
            print(j)
        i *= 2
```

$\Theta(n)$

4 Nonlocals & OOP

4.1 Draw the environment diagram that results from running the code.

```
def campa(nile):
    def ding(ding):
        nonlocal nile
        def nile(ring):
            return ding
    return nile(ding(1914)) + nile(1917)

ring = campa(lambda nile: 103)
```

https://goo.gl/G1Kmbw

4.2 Implement the classes so that the code to the right runs.

```
class Plant:
    def __init__(self):
        self.leaf = Leaf(self)
        self.materials = []
        self.height = 1

    def absorb(self):
        self.leaf.absorb()

    def grow(self):
        for sugar in self.materials:
            sugar.activate()
            self.height += 1

class Leaf:
    def __init__(self, plant):
        self.alive = True
        self.sugars_used = 0
        self.plant = plant

    def absorb(self):
        if self.alive:
            self.plant.materials.append(Sugar(self, self.plant))

    def __repr__(self):
        return 'Leaf'

class Sugar:
    sugars_created = 0

    def __init__(self, leaf, plant):
        self.leaf = leaf
        self.plant = plant
        Sugar.sugars_created += 1

    def activate(self):
        self.leaf.sugars_used += 1
        self.plant.materials.remove(self)

    def __repr__(self):
        return 'Sugar'
```

```
>>> p = Plant()
>>> p.height
1
>>> p.materials
[]
>>> p.absorb()
>>> p.materials
[Sugar]
>>> Sugar.sugars_created
1
>>> p.leaf.sugars_used
0
>>> p.grow()
>>> p.materials
[]
>>> p.height
2
>>> p.leaf.sugars_used
1
```

5 Exam Preparation *Extra Practice*

5.1 Implement `slice_reverse` which takes a linked list `s` and mutatively reverses the elements on the interval, $[i, j)$ (including i but excluding j). Assume `s` is zero-indexed, $i > 0$, $i < j$, and that `s` has at least j elements.

```
def slice_reverse(s, i, j):
    """
    >>> s = Link(1, Link(2, Link(3)))
    >>> slice_reverse(s, 1, 2)
    >>> s
    Link(1, Link(2, Link(3)))
    >>> s = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> slice_reverse(s, 2, 4)
    >>> s
    Link(1, Link(2, Link(4, Link(3, Link(5)))))
    """
    start = s

    for _ in range(i - 1):

        start = start.rest

    reverse = Link.empty

    current = start.rest

    for _ in range(j - i):

        rest = current.rest

        current.rest = reverse

        reverse = current

        current = rest

    start.rest.rest = current

    start.rest = reverse
```

5.2 A **Binary Search Tree** is a tree where each node contains either 0, 1, or 2 nodes and where the left branch (if present) contains values *strictly less than* (<) the root value, and the right branch (if present) contains values *strictly greater than* (>) the root value. The definition is recursive: both the left and right branches must also be BSTs for the entire tree to be a BST.

Implement `is_binary` which that takes in a Tree `t`, and returns `True` if `t` is a Binary Search Tree and `False` otherwise. Trees can contain any number of branches, but if a tree contains only one branch, interpret it as a left branch.

```
def is_binary(t):
    def binary(t, lo, hi):

        if lo < t.label < hi:

            if t.is_leaf():
                return True

            elif len(t.branches) == 1 and t.branches[0].label < t.label:

                return binary(t.branches[0], lo, t.label)

            elif len(t.branches) == 2 and t.branches[0].label < t.label < t.branches[1].
label:

                return binary(t.branches[0], lo, t.label) and binary(t.branches[1], t.label,
 hi)

        return False
    return binary(t, float('-inf'), float('inf'))
```

5.3 Give a tight asymptotic runtime bound for the following scenarios in $\Theta(\cdot)$ notation, or "Infinite" if the program does not terminate. Assume the implementation of `is_binary` is optimal.

(a) `is_binary` on a well-formed binary search tree with n nodes.

$\Theta(n)$

(b) `is_binary` on a tree where each node contains 3 branches and the overall height of the tree is n.

$\Theta(1)$