MUTABLE TREES AND MIDTERM REVIEW

COMPUTER SCIENCE MENTORS 61A

October 24, 2022–October 28, 2022

1 Trees

For the following problems, use this definition for the `Tree` class:

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

    def is_leaf(self):
        return self.branches == []

    # Implementation ommitted
```

Here are a few key differences between the `Tree` class and the Tree abstract data type, which we have previously encountered:

- Using the constructor: Capital T for the `Tree` class and lowercase t for tree ADT `t = Tree(1)` vs. `t = tree(1)`
- In the class, `label` and `branches` are instance variables and `is_leaf()` is an instance method. In the ADT, all of these were globally defined functions.

  `t.label` vs. `label(t)`

  `t.branches` vs. `branches(t)`

  `t.is_leaf()` vs. `is_leaf(t)`
- A `Tree` object is mutable while the tree ADT is not mutable. This means we can change attributes of a `Tree` instance without making a new tree. In other words, we can solve tree class problems non-destructively and destructively, but can only solve tree ADT problems non-destructively.

Created by Gabe Classon, Aditya Balasubramanian, Alyssa Smith, Esther Shen, Maya Romero, and Manas Khatore

`t.label = 2` is allowed but `label(t)= 2` would error.

Apart from these differences, we can largely take the approaches we used for the tree ADT and apply them to the `Tree` class!

1. Implement `tree_sum`, which takes in a `Tree` object and replaces the label of the tree with the sum of all the values in the tree. `tree_sum` should also return the new label.

```
def tree_sum(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4)])
    >>> tree_sum(t)
    10
    >>> t.label
    10
    >>> t.branches[0].label
    5
    >>> t.branches[1].label
    4
    """
```

2. Define `delete_path_duplicates`, which takes in `t`, a tree with non-negative labels. If there are any duplicate labels on any path from root to leaf, the function should mutate the label of the occurrences deeper in the tree (i.e. farther from the root) to be the value `-1`.

```
def delete_path_duplicates(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(1), Tree(1)])])
    >>> delete_path_duplicates(t)
    >>> t
    Tree(1, [Tree(2, [Tree(-1), Tree(-1)])])
    >>> t2 = Tree(1, [Tree(2), Tree(2, [Tree(2, [Tree(1,
       [Tree(5)])])])])
    >>> delete_path_duplicates(t2)
    >>> t2
    Tree(1, [Tree(2), Tree(2, [Tree(-1, [Tree(-1,
       [Tree(5)])])])])
    """
    def helper(_________________, _________________):

        if ____________________________________:

            ____________________________________

        else:

            ____________________________________

        for _______ in _____________________:

            ____________________________________

    ____________________________________________
```

3. Given a tree `t`, mutate the tree so that each leaf's label becomes the sum of the labels of all nodes in the path from the leaf node to the root node.

```
def replace_leaves_sum(t):
    """
    >>> t = Tree(1, [Tree(3, [Tree(2), Tree(8)]), Tree(5)])
    >>> replace_leaves_sum(t)
    >>> t
    Tree(1, [Tree(3, [Tree(6), Tree(12)]), Tree(6)])
    """
    def helper(_____________ , ___________________):

        if t.is_leaf():

            ________________________________________

        for b in t.branches:

            ________________________________

    ____________________________________________________
```

4. Write a function that returns `True` if there exists a path from root to leaf that contains at least `n` instances of `elem` in a tree `t`.

   Hint: recall that the built-in function ***any*** *takes in an iterable and returns* `True` *if any of the iterable's elements are truthy.*

```
def contains_n(elem, n, t):
    """
    >>> t1 = Tree(1, [Tree(1, [Tree(2)])])
    >>> contains_n(1, 2, t1)
    True
    >>> contains_n(2, 2, t1)
    False
    >>> contains_n(2, 1, t1)
    True
    >>> t2 = Tree(1, [Tree(2), Tree(1, [Tree(1), Tree(2)])])
    >>> contains_n(1, 3, t2)
    True
    >>> contains_n(2, 2, t2) # Not on a path
    False
    """
    if n == 0:

        return True

    elif ____________________________________________________:

        return ____________________________________________

    elif ____________________________________________________:

        return ____________________________________________

    else:

        return ____________________________________________
```

2 Efficiency

1. What is the order of growth for `foo`?

   (a)
   ```
   def foo(n):
       for i in range(n):
           print('hello')
   ```

   (b) What's the order of growth of `foo` if we change `range(n)` to

   i. `range(n/2)`?

   ii. `range(n**2 + 5)`?

   iii. `range(10000000)`?

2. What is the order of growth for `belgian_waffle`?

```
def belgian_waffle(n):
    total = 0
    while n > 0:
        total += 1
        n = n // 2
    return total
```

3 Higher Order Functions

1. Write a function, `make_digit_remover`, which takes in a single digit `i`. It returns another function that takes in an integer and, scanning from right to left, removes all digits from the integer up to and including the first occurrence of `i`. If `i` does not occur in the integer, the original number is returned.

```
def make_digit_remover(i):
    """
    >>> remove_two = make_digit_remover(2)
    >>> remove_two(232018)
    23
    >>> remove_two(23)
    0
    >>> remove_two(99)
    99
    """
    def remove(________):

        removed = ____________________________

        while ____________________________ > 0:

            ___________________________________

            removed = removed // 10

            if _________________________________:

                ____________________________

        return _____________________________

    return ______________________________
```

4 Recursion

1. Write a function that takes as input a number `n` and a list of numbers `lst` and returns `True` if we can find a subset of `lst` that sums to `n`.

```
def add_up(n, lst):
    """
    >>> add_up(10, [1, 2, 3, 4, 5])
    True
    >>> add_up(8, [2, 1, 5, 4, 3])
    True
    >>> add_up(-1, [1, 2, 3, 4, 5])
    False
    >>> add_up(100, [1, 2, 3, 4, 5])
    False
    """
    if _____________________________________________:

        return True

    if lst == []:

        ____________________________________________________________

    else:

        first, rest = _____________________, _____________________

        return _____________________________________________
```

5 Lists

1. Draw the box-and-pointer diagram.

```
>>> violet = [7, 77, 17]
>>> violet.append([violet.pop(1)])

>>> dash = violet * 2
>>> jack = dash[3:5]
>>> jackjack = jack.extend(jack)

>>> helen = list(violet)
>>> helen += [jackjack]
>>> helen[2].append(violet)
```

2. Implement `subsets`, which takes in a list of values and an integer `n` and returns all subsets of the list of size exactly `n` in any order. You may not need to use all the lines provided.

```
def subsets(lst, n):
    """
    >>> three_subsets = subsets(list(range(5)), 3)
    >>> for subset in sorted(three_subsets):
    ...     print(subset)
    [0, 1, 2]
    [0, 1, 3]
    [0, 1, 4]
    [0, 2, 3]
    [0, 2, 4]
    [0, 3, 4]
    [1, 2, 3]
    [1, 2, 4]
    [1, 3, 4]
    [2, 3, 4]
    """
    if n == 0:
        _____________________________________

    if _____________________________________:

        _____________________________________

    _____________________________________________________

    _____________________________________________________

    return _____________________________________________
```

6 Iterators and Generators

1. Write a generator function `num_elems` that takes in a possibly nested list of numbers `lst` and yields the number of elements in each nested list before finally yielding the total number of elements (including the elements of nested lists) in `lst`. For a nested list, yield the size of the inner list before the outer, and if you have multiple nested lists, yield their sizes from left to right.

```
def num_elems(lst):
    """
    >>> list(num_elems([3, 3, 2, 1]))
    [4]
    >>> list(num_elems([1, 3, 5, [1, [3, 5, [5, 7]]]]))
    [2, 4, 5, 8]
    """

    count = _________________

    for _____________________________:

        if ____________________________:

            for ____________________________________:

                yield _________________________________

            _____________________________________
        else:

            _____________________________________

    yield _________________________________
```

7 Object Oriented Programming

1. Let's use OOP to help us implement our good friend, the ping-pong sequence!

   As a reminder, the ping-pong sequence counts up starting from 1 and is always either counting up or counting down.

   At element `k`, the direction switches if `k` is a multiple of 7 or contains the digit 7.

   The first 30 elements of the ping-pong sequence are listed below, with direction swaps marked using brackets at the 7th, 14th, 17th, 21st, 27th, and 28th elements:

   ```
   1 2 3 4 5 6 [7] 6 5 4 3 2 1 [0] 1 2 [3] 2 1 0 [-1] 0 1 2 3 4
   [5] [4] 5 6
   ```

   Assume you have a function `has_seven(k)` that returns True if k contains the digit 7.

   ```
   >>> tracker1 = PingPongTracker()
   >>> tracker2 = PingPongTracker()
   >>> tracker1.next()
   1
   >>> tracker1.next()
   2
   >>> tracker2.next()
   1
   ```

   ```
   class PingPongTracker:
       def __init__(self):




       def next(self):
   ```

8 Linked Lists

1. Write a function `combine_two`, which takes in a linked list of integers `lnk` and a two-argument function `fn`. It returns a new linked list where every two elements of `lnk` have been combined using `fn`.

```
def combine_two(lnk, fn):
    """
    >>> lnk1 = Link(1, Link(2, Link(3, Link(4))))
    >>> combine_two(lnk1, add)
    Link(3, Link(7))
    >>> lnk2 = Link(2, Link(4, Link(6)))
    >>> combine_two(lnk2, mul)
    Link(8, Link(6))
    """
    if _____________________________________________:

        return _____________________________________

    elif _____________________________________________

        return _____________________________________

    combined = _____________________________________

    return _____________________________________________
```

2. Write a recursive function `insert_all` that takes as input two linked lists, `s` and `x`, and an index `index`. `insert_all` should return a new linked list with the contents of `x` inserted at index `index` of `s`.

```
def insert_all(s, x, index):
    """
    >>> insert = Link(3, Link(4))
    >>> original = Link(1, Link(2, Link(5)))
    >>> insert_all(original, insert, 2)
    Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> start = Link(1)
    >>> insert_all(original, start, 0)
    Link(1, Link(1, Link(2, Link(5))))
    """
    if ____________________ and ____________________:

        ____________________________________________

    if ____________________ and ____________________:

        ____________________________________________

    ____________________________________________________
```