MUTABLE TREES AND MIDTERM REVIEW Solutions

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.

Vibha Tantry and Jordan Schwartz, with
Gabe Classon, Aditya Balasubramanian, Oliver Yu, Hanze Tan, Gene Pan, Carolyn Wang, Raman Varma, Noor Haq, Angela Zhang, Abigail Brooks, Ted Kim, Aiden Joshua Vehemente, Marie Chorpita, Ashley Chiu, Bill Hu, Omar Yu, Angel Alberto Aldaco, Andrew Park, Chase Clements, Grace Yi, Preshtha Garg, Laksith Venkatesh Prabu, Aditya Murali, Irene Yang, Andy Chen, and Hailey Park

`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
    """

    for b in t.branches:
        t.label += tree_sum(b)
    return t.label
```

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 ____________________:

            ________________________________

    ________________________________________
```

```
def helper(t, seen_so_far):
    if t.label in seen_so_far:
      t.label = -1
    else:
        seen_so_far = seen_so_far + [t.label]
    for b in t.branches:
        helper(b, seen_so_far)
helper(t, [])
```

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:

            _________________________________

    _____________________________________________
```

```
def replace_leaves_sum(t):
    def helper(t, total):
        if t.is_leaf():
            t.label = total + t.label
        else:
            for b in t.branches:
                helper(b, total + t.label)
    helper(t, 0)
```

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 ______________________________________
    if n == 0:
        return True
    elif t.is_leaf():
        return n == 1 and t.label == elem
    elif t.label == elem:
        return any([contains_n(elem, n - 1, b) for b in
          t.branches])
    else:
        return any([contains_n(elem, n, b) for b in
          t.branches])
```

Base cases: The simplest case we have is when `n == 0`, or when we want at least 0 instances of `elem` in `t`. In this case, we always return `True`. The other simple case we consider is when the tree is only a leaf — there is nothing left to recurse on. In that case, we simply check to see that both `n == 1` and that `t.label == elem`, meaning that we have one element left to satisfy, and the leaf label satisfies the final element we are looking for. If we have more elements to search for (ie. n ¿ 1), then we will not satisfy that many elements at the leaf node; conversely, if we have fewer (ie. `n == 0`), then the case would already be covered by the first base case.

Recursive cases: If the current node isn't a leaf, then there's two different cases we should consider. Either the label of the current node is equal to `elem` or the label is not equal to `elem`. For the former, we would have to search for `n` more `elems` in each branch of `t` and return `True` if any of the branches contain `n` elems. For the latter, we would have `(n - 1)` elements remaining, so we would search for `(n - 1)` more `elems` in each branch of `t` and return `True` if any of the branches contain `(n - 1)` `elems`. Since there is not room to do a for loop, we can use a list comprehension to recursively call the function on each branch. Thus, our two list comprehension statements would be `[contains_n(elem, n, b)for b in t.branches]` and `[contains_n(elem, n - 1, b)for b in t.branches]`. To determine if any of the branches contain either `n` elems or `(n - 1)` elems, we can check if there's a `True` element in the respective lists.

2 Efficiency

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

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

   Linear. This is a for loop that will run `n` times.

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

   i. `range(n/2)`?

   Linear. The loop runs `n/2` times, but the runtime still scales linearly proportionally to `n`.

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

   Quadratic. The number of times the loop runs is proportional to n^2.

   iii. `range(10000000)`?

   Constant. No matter the size of `n`, we will run the loop the same number of times.

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

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

Logarithmic. Notice that with each pass through the while loop, the value of `n` is halved. Since we are halving till 0, this would be a logarithmic runtime.

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 __________________________
```

```
def make_digit_remover(i):
    def remove(n):
        removed = n
        while removed > 0:
            digit = removed % 10
            removed = removed // 10
            if digit == i:
                return removed
        return n
    return remove
```

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 n == 0:
        return True
    if lst == []:
         return False
    else:
        first, rest = lst[0], lst[1:]
        return add_up(n - first, rest) or add_up(n, rest)
```

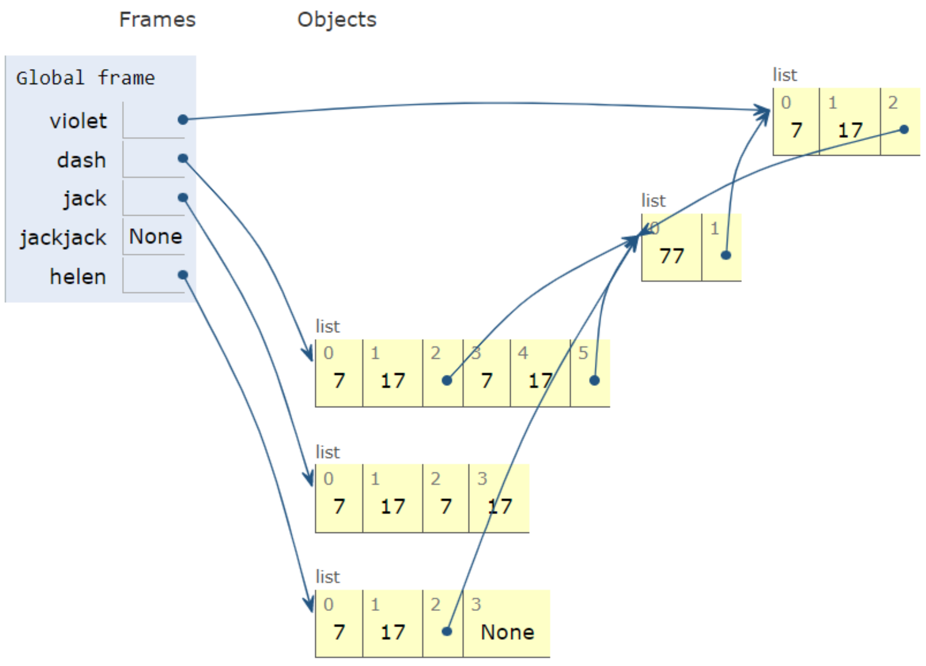
1. Draw the box-and-pointer diagram.

```
>>> 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)
```

https://goo.gl/EAmZBW



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 ______________________________________

    if n == 0:
        return [[]]
    if len(lst) == n:
        return [lst]
    with_first = [[lst[0]] + x for x in subsets(lst[1:], n -
       1)]
    without_first = subsets(lst[1:], n)
    return with_first + without_first
```

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 ___________________________
```

```
def num_elems(lst):
    count = 0
    for elem in lst:
        if isinstance(elem, list):
            for c in num_elems(elem):
                yield c
            count += c
        else:
            count += 1
    yield count
```

count refers to the number of elements in the current list lst (including the number of elements inside any nested list). Determine the value of count by looping through each element of the current list lst. If we have an element elem which is of type **list**, we want to yield the number of elements in each nested list of elem before finally yielding the total number of elements in elem. We can do this with a recursive call to num_elems. Thus, we yield all the values that need to be yielded using the inner for loop. The last number yielded by this inner loop is the total number of elements in elem, which we want to increase count by. Otherwise, if elem is not a list, then we can simply increase count by 1. Finally, yield the total count of the list.

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):
   ```

```python
class PingPongTracker:
    def __init__(self):
        self.current = 0
        self.index = 1
        self.add = True

    def next(self):
        if self.add:
            self.current += 1
        else:
            self.current -= 1
        if has_seven(self.index) or self.index % 7 == 0:
            self.add = not self.add
        self.index += 1
        return self.current
```

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 __________________________________
```

```
def combine_two(lnk, fn):
    if lnk is Link.empty:
        return Link.empty
    elif lnk.rest is Link.empty:
        return Link(lnk.first)
    combined = fn(lnk.first, lnk.rest.first)
    return Link(combined, combine_two(lnk.rest.rest, fn))
```

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 ____________________:

        ____________________________________________

    ________________________________________________
```

```
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))))
    >>> insert_all(original, insert, 3)
    Link(1, Link(2, Link(5, Link(3, Link(4)))))
    """
    if s is Link.empty and x is Link.empty:
        return Link.empty
    if x is not Link.empty and index == 0:
        return Link(x.first, insert_all(s, x.rest, 0))
    return Link(s.first, insert_all(s.rest, x, index - 1))
```

All of our return statements should return a new linked list.

Our base case should be the simplest possible version of the problem: when both `x` and `s` are empty, clearly the result is just the empty list.

We can now think of ways to break down this problem even further. Note that when the index to be inserted at is `0`, the problem is relatively easy: we just have to put all of the elements of `x` followed by all the elements of `s`. So the first element of the new list should `x.first`, and the rest of the new list should be `x.rest` concatenated with `s`, or `insert_all(s, x.rest, 0)`. Since we are using `x.first` and `x.rest`, we must check that `x` is nonempty to ensure that we do not error.

Finally, when the index to be inserted at is nonzero, we know that we're going to have some elements of `s`, then the elements of `x`, and then the rest of the elements from `s`. So the first element of the new list should be `s.first`. Then we can get the rest of the new list by inserting the contents of `x` at index `index - 1` of `s.rest`, reducing the index by 1 to account for the fact that we have removed the first element of `s`.

There's one issue we glossed over here: what if `x` is empty but `s` is not? Then we want to return the contents of `s`. But because the problem requires that we return a new linked list, we must recursively reconstruct `s` instead of simply returning it. You could add another base case to handle this, but as it turns out the second recursive case will handle this just fine since `Link(s.first, insert_all(s.rest, x, index - 1))` is just equivalent to `Link(s.first, s.rest)` when `x` is empty. Since the `x` **`is not`** `Link.empty` condition for the first recursive case will direct all situations where `x` is empty but `s` is not to the second recursive case, it turns out that we do not need to add anything else to this solution.

Convincing yourself that this problem works requires that you eventually reach a base case. Note that in either recursive call, we either reduce `s` or `x` by one element. So the base case will always eventually be reached, and the solution is valid.