Lab 8: Midterm Review **lab08.zip (lab08.zip)**

Due at 11:59pm on Friday, 03/22/2019.

Starter Files

Download lab08.zip (lab08.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Submission

By the end of this lab, you should have submitted the lab with `python3 ok --submit`. You may submit more than once before the deadline; only the final submission will be graded. Check that you have successfully submitted your code on okpy.org (https://okpy.org/).

- In order to facilitate midterm studying, solutions to this lab were released with the lab. We encourage you to try out the problems and struggle for a while before looking at the solutions!

Linked Lists

Q1: Deep Linked List Length

A linked list that contains one or more linked lists as elements is called a *deep* linked list. Write a function `deep_len` that takes in a (possibly deep) linked list and returns the *deep length* of that linked list. The deep length of a linked list is the total number of non-link elements in the list, as well as the total number of elements contained in all contained lists. See the function's doctests for examples of the deep length of linked lists.

Hint: Use `isinstance` to check if something is an instance of an object.

```
def deep_len(lnk):
    """ Returns the deep length of a possibly deep linked list.

    >>> deep_len(Link(1, Link(2, Link(3))))
    3
    >>> deep_len(Link(Link(1, Link(2)), Link(3, Link(4))))
    4
    >>> levels = Link(Link(Link(1, Link(2)), \
            Link(3)), Link(Link(4), Link(5)))
    >>> print(levels)
    <<<1 2> 3> <4> 5>
    >>> deep_len(levels)
    5
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q deep_len
```

Q2: Linked Lists as Strings

Kevin and Jerry like different ways of displaying the linked list structure in Python. While Kevin likes box and pointer diagrams, Jerry prefers a more futuristic way. Write a function `make_to_string` that returns a function that converts the linked list to a string in their preferred style.

Hint: You can convert numbers to strings using the `str` function, and you can combine strings together using `+`.

```
>>> str(4)
'4'
>>> 'cs ' + str(61) + 'a'
'cs 61a'
```

```
def make_to_string(front, mid, back, empty_repr):
    """ Returns a function that turns linked lists to strings.

    >>> kevins_to_string = make_to_string("[", "|-]-->", "", "[]")
    >>> jerrys_to_string = make_to_string("(", " . ", ")", "()")
    >>> lst = Link(1, Link(2, Link(3, Link(4))))
    >>> kevins_to_string(lst)
    '[1|-]-->[2|-]-->[3|-]-->[4|-]-->[]'
    >>> kevins_to_string(Link.empty)
    '[]'
    >>> jerrys_to_string(lst)
    '(1 . (2 . (3 . (4 . ()))))'
    >>> jerrys_to_string(Link.empty)
    '()'
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q make_to_string
```

Trees

Q3: Tree Map

Define the function `tree_map`, which takes in a tree and a one-argument function as arguments and returns a new tree which is the result of mapping the function over the entries of the input tree.

```
def tree_map(fn, t):
    """Maps the function fn over the entries of t and returns the
    result in a new tree.

    >>> numbers = Tree(1,
    ...                [Tree(2,
    ...                      [Tree(3),
    ...                       Tree(4)]),
    ...                 Tree(5,
    ...                      [Tree(6,
    ...                            [Tree(7)]),
    ...                       Tree(8)])])
    >>> print(tree_map(lambda x: 2**x, numbers))
    2
      4
        8
        16
      32
        64
          128
        256
    >>> print(numbers)
    1
      2
        3
        4
      5
        6
          7
        8
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q tree_map
```

Q4: Long Paths

Implement `long_paths`, which returns a list of all *paths* in a tree with length at least `n`. A path in a tree is a linked list of node values that starts with the root and ends at a leaf. Each subsequent element must be from a child of the previous value's node. The *length* of a path is the number of edges in the path (i.e. one less than the number of nodes in the path). Paths are listed in order from left to right. See the doctests for some examples.

```
def long_paths(tree, n):
    """Return a list of all paths in tree with length at least n.

    >>> t = Tree(3, [Tree(4), Tree(4), Tree(5)])
    >>> left = Tree(1, [Tree(2), t])
    >>> mid = Tree(6, [Tree(7, [Tree(8)]), Tree(9)])
    >>> right = Tree(11, [Tree(12, [Tree(13, [Tree(14)])])])
    >>> whole = Tree(0, [left, Tree(13), mid, right])
    >>> for path in long_paths(whole, 2):
    ...     print(path)
    ...
    <0 1 2>
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 6 9>
    <0 11 12 13 14>
    >>> for path in long_paths(whole, 3):
    ...     print(path)
    ...
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 11 12 13 14>
    >>> long_paths(whole, 4)
    [Link(0, Link(11, Link(12, Link(13, Link(14)))))]
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q long_paths
```

Orders of Growth

Q5: Finding Orders of Growth

Use Ok to test your knowledge with the following questions:

```
python3 ok -q growth -u
```

Be sure to ask a lab assistant or TA if you don't understand the correct answer!

What is the order of growth of `is_prime` in terms of `n`?

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

What is the order of growth of `bar` in terms of `n`?

```
def bar(n):
    i, sum = 1, 0
    while i <= n:
        sum += biz(n)
        i += 1
    return sum

def biz(n):
    i, sum = 1, 0
    while i <= n:
        sum += i**3
        i += 1
    return sum
```

What is the order of growth of `foo` in terms of `n`, where `n` is the length of `lst`? Assume that slicing a list and calling `len` on a list can both be done in constant time.

```
def foo(lst, i):
    mid = len(lst) // 2
    if mid == 0:
        return lst
    elif i > 0:
        return foo(lst[mid:], -1)
    else:
        return foo(lst[:mid], 1)
```

Recursion & Tree Recursion

Q6: Subsequences

A subsequence of a sequence `S` is a sequence of elements from `S`, in the same order they appear in `S`, but possibly with elements missing. Thus, the lists `[]`, `[1, 3]`, `[2]`, and `[1, 2, 3]` are some (but not all) of the subsequences of `[1, 2, 3]`. Write a function that takes a list and returns a list of lists, for which each individual list is a subsequence of the original input.

In order to accomplish this, you might first want to write a function `insert_into_all` that takes an item and a list of lists, adds the item to the beginning of nested list, and returns the resulting list.

```
def insert_into_all(item, nested_list):
    """Assuming that nested_list is a list of lists, return a new list
    consisting of all the lists in nested_list, but with item added to
    the front of each.

    >>> nl = [[], [1, 2], [3]]
    >>> insert_into_all(0, nl)
    [[0], [0, 1, 2], [0, 3]]
    """
    "*** YOUR CODE HERE ***"

def subseqs(s):
    """Assuming that S is a list, return a nested list of all subsequences
    of S (a list of lists). The subsequences can appear in any order.

    >>> seqs = subseqs([1, 2, 3])
    >>> sorted(seqs)
    [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
    >>> subseqs([])
    [[]]
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q subseqs
```

Q7: Increasing Subsequences

Now, we wish to find subsequences subject to another condition: we only want the subsequences for which consecutive elements are *nondecreasing*. For example, `[1, 3, 2]` is a subsequence of `[1, 3, 2, 4]`, but since 2 < 3, this subsequence would *not* be included in our result.

Fill in the blanks to complete the implementation of the `inc_subseqs` function. You may assume that the input list only contains positive elements.

You may use the helper function `insert_into_all` you defined in the previous part.

```
def inc_subseqs(s):
    """Assuming that S is a list, return a nested list of all subsequences
    of S (a list of lists) for which the elements of the subsequence
    are strictly nondecreasing. The subsequences can appear in any order.

    >>> seqs = inc_subseqs([1, 3, 2])
    >>> sorted(seqs)
    [[], [1], [1, 2], [1, 3], [2], [3]]
    >>> inc_subseqs([])
    [[]]
    >>> seqs2 = inc_subseqs([1, 1, 2])
    >>> sorted(seqs2)
    [[], [1], [1], [1, 1], [1, 1, 2], [1, 2], [1, 2], [2]]
    """
    def subseq_helper(s, prev):
        if not s:
            return ____________________
        elif s[0] < prev:
            return ____________________
        else:
            a = ______________________
            b = ______________________
            return insert_into_all(________, ______________) + ________________
    return subseq_helper(____, ____)
```

Use Ok to test your code:

```
python3 ok -q inc_subseqs
```

Q8: Number of Trees

How many different possible full binary tree (each node has two branches or 0, but never 1) structures exist that have exactly n leaves?

For those interested in combinatorics, this problem does have a closed form solution (http://en.wikipedia.org/wiki/Catalan_number)):

```
def num_trees(n):
    """How many full binary trees have exactly n leaves? E.g.,

    1   2        3       3    ...
    *   *        *       *
       / \      / \     / \
      *   *    *   *   *   *
              / \         / \
             *   *       *   *

    >>> num_trees(1)
    1
    >>> num_trees(2)
    1
    >>> num_trees(3)
    2
    >>> num_trees(8)
    429

    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q num_trees
```

Objects

Q9: Keyboard

We'd like to create a `Keyboard` class that takes in an arbitrary number of `Button`s and stores these `Button`s in a dictionary. The keys in the dictionary will be ints that represent the postition on the `Keyboard`, and the values will be the respective `Button`. Fill out the methods in the `Keyboard` class according to each description, using the doctests as a reference for the behavior of a `Keyboard`.

```
class Keyboard:
    """A Keyboard takes in an arbitrary amount of buttons, and has a
    dictionary of positions as keys, and values as Buttons.

    >>> b1 = Button(0, "H")
    >>> b2 = Button(1, "I")
    >>> k = Keyboard(b1, b2)
    >>> k.buttons[0].key
    'H'
    >>> k.press(1)
    'I'
    >>> k.press(2) #No button at this position
    ''
    >>> k.typing([0, 1])
    'HI'
    >>> k.typing([1, 0])
    'IH'
    >>> b1.times_pressed
    2
    >>> b2.times_pressed
    3
    """

    def __init__(self, *args):
        "*** YOUR CODE HERE ***"

    def press(self, info):
        """Takes in a position of the button pressed, and
        returns that button's output"""
        "*** YOUR CODE HERE ***"

    def typing(self, typing_input):
        """Takes in a list of positions of buttons pressed, and
        returns the total output"""
        "*** YOUR CODE HERE ***"

class Button:
    """
    Represents a single button
    """
    def __init__(self, pos, key):
        """
        Creates a button
        """
        self.pos = pos
        self.key = key
        self.times_pressed = 0
```

Use Ok to test your code:

```
python3 ok -q Keyboard
```

Nonlocal

Q10: Advanced Counter

Complete the definition of `make_advanced_counter_maker`, which creates a function that creates counters. These counters can not only update their personal count, but also a shared count for all counters. They can also reset either count.

```
def make_advanced_counter_maker():
    """Makes a function that makes counters that understands the
    messages "count", "global-count", "reset", and "global-reset".
    See the examples below:

    >>> make_counter = make_advanced_counter_maker()
    >>> tom_counter = make_counter()
    >>> tom_counter('count')
    1
    >>> tom_counter('count')
    2
    >>> tom_counter('global-count')
    1
    >>> jon_counter = make_counter()
    >>> jon_counter('global-count')
    2
    >>> jon_counter('count')
    1
    >>> jon_counter('reset')
    >>> jon_counter('count')
    1
    >>> tom_counter('count')
    3
    >>> jon_counter('global-count')
    3
    >>> jon_counter('global-reset')
    >>> tom_counter('global-count')
    1
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q make_advanced_counter_maker
```

Mutable Lists

Q11: Environment Diagram

Draw an environment diagram for the following program.

> Some things to remember:
>
> - When you mutate a list, you are changing the original list.
> - When you concatenate two lists, you are creating a new list.
> - When you assign a name to an existing object, you are creating another reference to that object rather than creating a copy of that object.

```
def got(lst, el, f):
    welcome = []
    for e in lst:
        if e == el:
            el = f(lst[1:], 2, welcome)
    return lst[3:] + welcome

def avocadis(lst, i, lst0):
    lst0.append(lst.pop(i))
    return len(lst0)

bananis = [1, 6, 1, 6]
n = bananis[3]
we = got(bananis, n, avocadis)
```

You can check your solution here (https://goo.gl/iRnSUx). If you get stuck, ask a Lab Assistant or TA for help before checking the solution! There is nothing to submit for this problem.

Q12: Trade

In the integer market, each participant has a list of positive integers to trade. When two participants meet, they trade the smallest non-empty prefix of their list of integers. A prefix is a slice that starts at index 0.

Write a function `trade` that exchanges the first `m` elements of list `first` with the first `n` elements of list `second`, such that the sums of those elements are equal, and the sum is as small as possible. If no such prefix exists, return the string `'No deal!'` and do not change either list. Otherwise change both lists and return `'Deal!'`. A partial implementation is provided.

Hint: You can mutate a slice of a list using *slice assignment*. To do so, specify a slice of the list `[i:j]` on the left-hand side of an assignment statement and another list on the right-hand side of the assignment statement. The operation will replace the entire given slice of the list from `i` inclusive to `j` exclusive with the elements from the given list. The slice and the given list need not be the same length.

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> b = a
>>> a[2:5] = [10, 11, 12, 13]
>>> a
[1, 2, 10, 11, 12, 13, 6]
>>> b
[1, 2, 10, 11, 12, 13, 6]
```

Additionally, recall that the starting and ending indices for a slice can be left out and Python will use a default value. `lst[i:]` is the same as `lst[i:len(lst)]`, and `lst[:j]` is the same as `lst[0:j]`.

```
def trade(first, second):
    """Exchange the smallest prefixes of first and second that have equal sum.

    >>> a = [1, 1, 3, 2, 1, 1, 4]
    >>> b = [4, 3, 2, 7]
    >>> trade(a, b) # Trades 1+1+3+2=7 for 4+3=7
    'Deal!'
    >>> a
    [4, 3, 1, 1, 4]
    >>> b
    [1, 1, 3, 2, 2, 7]
    >>> c = [3, 3, 2, 4, 1]
    >>> trade(b, c)
    'No deal!'
    >>> b
    [1, 1, 3, 2, 2, 7]
    >>> c
    [3, 3, 2, 4, 1]
    >>> trade(a, c)
    'Deal!'
    >>> a
    [3, 3, 2, 1, 4]
    >>> b
    [1, 1, 3, 2, 2, 7]
    >>> c
    [4, 3, 1, 4, 1]
    """
    m, n = 1, 1

    "*** YOUR CODE HERE ***"

    if False: # change this line!
        first[:m], second[:n] = second[:n], first[:m]
        return 'Deal!'
    else:
        return 'No deal!'
```

Use Ok to test your code:

```
python3 ok -q trade
```

More Orders of Growth

Q13: Boom

What is the order of growth in time for the following function boom? Use big-θ notation.

```
def boom(n):
    sum = 0
    a, b = 1, 1
    while a <= n*n:
        while b <= n*n:
            sum += (a*b)
            b += 1
        b = 0
        a += 1
    return sum
```

Use ok to test your understanding:

```
python3 ok -q boom -u
```

Q14: Zap

What is the order of growth in time for the following function `zap`? Use big-θ notation.

```
def zap(n):
    i, count = 1, 0
    while i <= n:
        while i <= 5 * n:
            count += i
            print(i / 6)
            i *= 3
    return count
```

Use ok to test your understanding:

```
python3 ok -q zap -u
```

CS 61A (/)

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)