Lab 8: Midterm Review | CS 61A Spring 2020

λ inst.eecs.berkeley.edu//~cs61a/sp20/lab/lab08

Lab 8: Midterm Review
lab08.zip

Due by 11:59pm on Friday, March 20.

Starter Files

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

> 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!

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Required Questions

Linked Lists

Q1: Insert

Implement a function `insert` that takes a `Link`, a `value`, and an `index`, and inserts the `value` into the `Link` at the given `index`. You can assume the linked list already has at least one element. Do not return anything -- `insert` should mutate the linked list.

> **Note**: If the index is out of bounds, you can raise an `IndexError` with:
>
> ```
> raise IndexError
> ```

```
def insert(link, value, index):
    """Insert a value into a Link at the given index.

    >>> link = Link(1, Link(2, Link(3)))
    >>> print(link)
    <1 2 3>
    >>> insert(link, 9001, 0)
    >>> print(link)
    <9001 1 2 3>
    >>> insert(link, 100, 2)
    >>> print(link)
    <9001 1 100 2 3>
    >>> insert(link, 4, 5)
    IndexError
    """
    if ____________________:
        ____________________
        ____________________
        ____________________
    elif ____________________:
        ____________________
    else:
        ____________________
```

Use Ok to test your code:

```
python3 ok -q insert
```

Efficiency WWPD

Q2: Determining Efficiency

> Use Ok to test your knowledge with the following questions:
>
> ```
> python3 ok -q wwpd-efficiency -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
```

Linear.
Explanation: In the worst case, *n* is prime, and we have to execute the loop *n* - 2 times. Each iteration takes constant time (one conditional check and one return statement). Therefore, the total time is (*n* - 2) x constant, or simply linear.

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
```

Quadratic.
Explanation: The body of the while loop in `bar` is executed *n* times. Each iteration, one call to `biz(n)` is made. Note that *n* never changes, so this call takes the same time to run each iteration. Taking a look at `biz`, we see that there is another while loop. Be careful to note that although the term being added to `sum` is cubed (`i**3`), `i` itself is only incremented by 1 in each iteration. This tells us that this while loop also executes *n* times, with each iteration taking constant time , so the total time of `biz(n)` is *n* x constant, or linear. Knowing the runtime of linear, we can conclude that each iteration of the while loop in `bar` is linear. Therefore, the total runtime of `bar(n)` is quadratic.

Recursion and Tree Recursion

Q3: 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]]
    """
    return _____________________________

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([])
    [[]]
    """
    if _________________:
        _________________
    else:
        _________________
        _________________
```

Use Ok to test your code:

```
python3 ok -q subseqs
```

Q4: Increasing Subsequences

In Lab 4, we examined the Subsequences problem. 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. For example, the lists `[]`, `[1, 3]`, `[2]`, and `[1, 3, 2]` are subsequences of `[1, 3, 2]`. Again, we want to write a function that takes a list and returns a list of lists, where each individual list is a subsequence of the original input.

This time we have 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 contains no negative elements.

You may use the provided helper function `insert_into_all`, which takes in an `item` and a list of lists and inserts the `item` to the front of each list.

```
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
```

Generators

Q5: Generate Permutations

Given a sequence of unique elements, a *permutation* of the sequence is a list containing the elements of the sequence in some arbitrary order. For example, `[2, 1, 3]`, `[1, 3, 2]`, and `[3, 2, 1]` are some of the permutations of the sequence `[1, 2, 3]`.

Implement `permutations`, a generator function that takes in a sequence `seq` and returns a generator that yields all permutations of `seq`.

Permutations may be yielded in any order. Note that the doctests test whether you are yielding all possible permutations, but not in any particular order. The built-in `sorted` function takes in an iterable object and returns a list containing the elements of the iterable in non-decreasing order.

Your solution must fit on the lines provided in the skeleton code.

> *Hint:* If you had the permutations of all the elements in `lst` not including the first element, how could you use that to generate the permutations of the full `lst`?

```
def permutations(seq):
    """Generates all permutations of the given sequence. Each permutation is a
    list of the elements in SEQ in a different order. The permutations may be
    yielded in any order.

    >>> perms = permutations([100])
    >>> type(perms)
    <class 'generator'>
    >>> next(perms)
    [100]
    >>> try:
    ...     next(perms)
    ... except StopIteration:
    ...     print('No more permutations!')
    No more permutations!
    >>> sorted(permutations([1, 2, 3])) # Returns a sorted list containing
elements of the generator
    [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
    >>> sorted(permutations((10, 20, 30)))
    [[10, 20, 30], [10, 30, 20], [20, 10, 30], [20, 30, 10], [30, 10, 20], [30,
20, 10]]
    >>> sorted(permutations("ab"))
    [['a', 'b'], ['b', 'a']]
    """
    if ______________________:
        yield ______________________
    else:
        for perm in ________________________:
            for _____ in _________________:
                ____________________________
```

Use Ok to test your code:

```
python3 ok -q permutations
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Suggested Questions

Objects

Q6: 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 Button:
    """
    Represents a single button
    """
    def __init__(self, pos, key):
        """
        Creates a button
        """
        self.pos = pos
        self.key = key
        self.times_pressed = 0

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):
        ________________
        for __________ in ________________:
            ________________

    def press(self, info):
        """Takes in a position of the button pressed, and
        returns that button's output"""
        if ____________________:
            ________________
            ________________
            ________________
            ________________
        ________________

    def typing(self, typing_input):
        """Takes in a list of positions of buttons pressed, and
        returns the total output"""
        ________________
        for _________ in ____________________:
```

```
        _____________
    _____________
```

Use Ok to test your code:

```
python3 ok -q Keyboard
```

Nonlocal

Q7: 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
    """
    _________________
    def _____________(__________):
        _________________
        def _____________(__________):
            _________________
            "*** YOUR CODE HERE ***"
            # as many lines as you want
        _________________
    _________________
```

Use Ok to test your code:

```
python3 ok -q make_advanced_counter_maker
```

Mutable Lists

Q8: 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

    equal_prefix = lambda: ___________________________
    while ____________________________________:
        if ____________________:
            m += 1
        else:
            n += 1

    if equal_prefix():
        first[:m], second[:n] = second[:n], first[:m]
        return 'Deal!'
    else:
        return 'No deal!'
```

Use Ok to test your code:

```
python3 ok -q trade
```

Q9: Shuffle

Define a function `shuffle` that takes a sequence with an even number of elements (cards) and creates a new list that interleaves the elements of the first half with the elements of the second half.

```
def card(n):
    """Return the playing card numeral as a string for a positive n <= 13."""
    assert type(n) == int and n > 0 and n <= 13, "Bad card n"
    specials = {1: 'A', 11: 'J', 12: 'Q', 13: 'K'}
    return specials.get(n, str(n))

def shuffle(cards):
    """Return a shuffled list that interleaves the two halves of cards.

    >>> shuffle(range(6))
    [0, 3, 1, 4, 2, 5]
    >>> suits = ['♡', '♢', '♤', '♧']
    >>> cards = [card(n) + suit for n in range(1,14) for suit in suits]
    >>> cards[:12]
    ['A♡', 'A♢', 'A♤', 'A♧', '2♡', '2♢', '2♤', '2♧', '3♡', '3♢', '3♤',
'3♧']
    >>> cards[26:30]
    ['7♤', '7♧', '8♡', '8♢']
    >>> shuffle(cards)[:12]
    ['A♡', '7♤', 'A♢', '7♧', 'A♤', '8♡', 'A♧', '8♢', '2♡', '8♤', '2♢',
'8♧']
    >>> shuffle(shuffle(cards))[:12]
    ['A♡', '4♢', '7♤', '10♧', 'A♢', '4♤', '7♧', 'J♡', 'A♤', '4♧', '8♡',
'J♢']
    >>> cards[:12]  # Should not be changed
    ['A♡', 'A♢', 'A♤', 'A♧', '2♡', '2♢', '2♤', '2♧', '3♡', '3♢', '3♤',
'3♧']
    """
    assert len(cards) % 2 == 0, 'len(cards) must be even'
    half = _________________
    shuffled = []
    for i in _______________:
        ___________________
        ___________________
    return shuffled
```

Use Ok to test your code:

```
python3 ok -q shuffle
```

Recursive Objects

Q10: 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
    """
    if _______________:
        return 0
    elif _______________:
        return 1
    else:
        return ____________________________
```

Use Ok to test your code:

```
python3 ok -q deep_len
```

Q11: 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)
    '()'
    """
    def printer(lnk):
        if _____________:
            return ____________________________
        else:
            return ____________________________
    return printer
```

Use Ok to test your code:

```
python3 ok -q make_to_string
```

Q12: Prune Small

Complete the function `prune_small` that takes in a `Tree t` and a number `n` and prunes `t` mutatively. If `t` or any of its branches has more than `n` branches, the `n` branches with the smallest labels should be kept and any other branches should be *pruned*, or removed, from the tree.

```
def prune_small(t, n):
    """Prune the tree mutatively, keeping only the n branches
    of each node with the smallest label.

    >>> t1 = Tree(6)
    >>> prune_small(t1, 2)
    >>> t1
    Tree(6)
    >>> t2 = Tree(6, [Tree(3), Tree(4)])
    >>> prune_small(t2, 1)
    >>> t2
    Tree(6, [Tree(3)])
    >>> t3 = Tree(6, [Tree(1), Tree(3, [Tree(1), Tree(2), Tree(3)]), Tree(5,
[Tree(3), Tree(4)])])
    >>> prune_small(t3, 2)
    >>> t3
    Tree(6, [Tree(1), Tree(3, [Tree(1), Tree(2)])])
    """
    while _____________________________:
        largest = max(_________________, key=_____________________)
        ____________________________
    for __ in _____________:
        ____________________
```

Use Ok to test your code:

```
python3 ok -q prune_small
```

Recursion / Tree Recursion

Q13: Number of Trees

How many different possible full binary tree (each node has 2 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):

```
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

    """
    if _____________________:
        return _________________
    return _________________
```

Use Ok to test your code:

```
python3 ok -q num_trees
```