Lab 9 Solutions **lab09.zip (lab09.zip)**

Solution Files

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.

All Questions Are Optional

The questions in this assignment are not graded, but they are highly recommended to help you prepare for the upcoming exam. You will receive credit for this lab even if you do not complete these questions.

Recursion and Tree Recursion

Q1: Subsequences

A subsequence of a sequence `S` is a subset of elements from `S`, in the same order they appear in `S`. Consider the list `[1, 2, 3]`. Here are a few of it's subsequences `[]`, `[1, 3]`, `[2]`, and `[1, 2, 3]`.

Write a function that takes in a list and returns all possible subsequences of that list. The subsequences should be returned as a list of lists, where each nested 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 each nested list, and returns the resulting list.

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

    >>> nl = [[], [1, 2], [3]]
    >>> insert_into_all(0, nl)
    [[0], [0, 1, 2], [0, 3]]
    """
    return [[item] + lst for lst in nested_list]

def subseqs(s):
    """Return a nested list (a list of lists) of all subsequences of S.
    The subsequences can appear in any order. You can assume S is a list.

    >>> seqs = subseqs([1, 2, 3])
    >>> sorted(seqs)
    [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
    >>> subseqs([])
    [[]]
    """
    if not s:
        return [[]]
    else:
        subset = subseqs(s[1:])
        return insert_into_all(s[0], subset) + subset
```

Use Ok to test your code:

```
python3 ok -q subseqs
```

Q2: Non-Decreasing Subsequences

Just like the last question, 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 `non_decrease_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 non_decrease_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 = non_decrease_subseqs([1, 3, 2])
    >>> sorted(seqs)
    [[], [1], [1, 2], [1, 3], [2], [3]]
    >>> non_decrease_subseqs([])
    [[]]
    >>> seqs2 = non_decrease_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 subseq_helper(s[1:], prev)
        else:
            a = subseq_helper(s[1:], s[0])
            b = subseq_helper(s[1:], prev)
            return insert_into_all(s[0], a) + b
    return subseq_helper(s, 0)
```

Use Ok to test your code:

```
python3 ok -q non_decrease_subseqs
```

Q3: Number of Trees

A **full binary tree** is a tree where each node has either 2 branches or 0 branches, but never 1 branch.

Write a function which returns the number of unique full binary tree structures 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):
    """Returns the number of unique full binary trees with 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 n == 1:
        return 1
    return sum(num_trees(k) * num_trees(n-k) for k in range(1, n))
```

Use Ok to test your code:

```
python3 ok -q num_trees
```

Generators

Q4: Merge

Implement `merge(incr_a, incr_b)`, which takes two iterables `incr_a` and `incr_b` whose elements are ordered. `merge` yields elements from `incr_a` and `incr_b` in sorted order, eliminating repetition. You may assume `incr_a` and `incr_b` themselves do not contain repeats, and that none of the elements of either are `None`. You may **not** assume that the iterables are finite; either may produce an infinite stream of results.

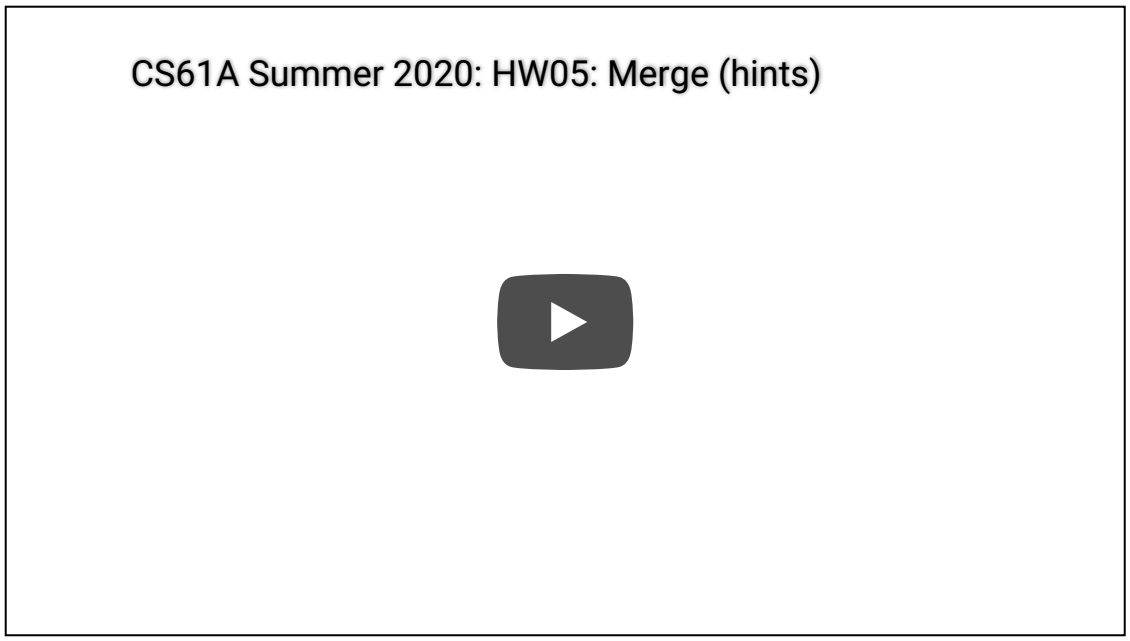
You will probably find it helpful to use the two-argument version of the built-in `next` function: `next(incr, v)` is the same as `next(incr)`, except that instead of raising `StopIteration` when `incr` runs out of elements, it returns `v`.

See the doctest for examples of behavior.

```
def merge(incr_a, incr_b):
    """Yield the elements of strictly increasing iterables incr_a and incr_b, removing
    repeats. Assume that incr_a and incr_b have no repeats. incr_a or incr_b may or may nc
    be infinite sequences.

    >>> m = merge([0, 2, 4, 6, 8, 10, 12, 14], [0, 3, 6, 9, 12, 15])
    >>> type(m)
    <class 'generator'>
    >>> list(m)
    [0, 2, 3, 4, 6, 8, 9, 10, 12, 14, 15]
    >>> def big(n):
    ...     k = 0
    ...     while True: yield k; k += n
    >>> m = merge(big(2), big(3))
    >>> [next(m) for _ in range(11)]
    [0, 2, 3, 4, 6, 8, 9, 10, 12, 14, 15]
    """
    iter_a, iter_b = iter(incr_a), iter(incr_b)
    next_a, next_b = next(iter_a, None), next(iter_b, None)
    while next_a is not None or next_b is not None:
        if next_a is None or next_b is not None and next_b < next_a:
            yield next_b
            next_b = next(iter_b, None)
        elif next_b is None or next_a is not None and next_a < next_b:
            yield next_a
            next_a = next(iter_a, None)
        else:
            yield next_a
            next_a, next_b = next(iter_a, None), next(iter_b, None)
```

Watch the hints video below for somewhere to start:



YouTube link (https://youtu.be/wBwcnB5CNZg)

Use Ok to test your code:

```
python3 ok -q merge
```

Objects

Q5: Bank Account

Implement the class `Account`, which acts as a a Bank Account. `Account` should allow the account holder to deposit money into the account, withdraw money from the account, and view their transaction history. The Bank Account should also prevents a user from withdrawing more than the current balance.

Transaction history should be stored as a list of tuples, where each tuple contains the type of transaction and the transaction amount. For example a withdrawal of 500 should be stored as ('withdraw', 500)

> **Hint:** You can call the `str` function on an integer to get a string representation of the integer. You might find this function useful when implementing the `__repr__` and `__str__` methods.
>
> **Hint:** You can alternatively use fstrings (https://realpython.com/python-f-strings/) to implement the `__repr__` and `__str__` methods cleanly.

```
class Account:
    """A bank account that allows deposits and withdrawals.
    It tracks the current account balance and a transaction
    history of deposits and withdrawals.

    >>> eric_account = Account('Eric')
    >>> eric_account.deposit(1000000)   # depositing paycheck for the week
    1000000
    >>> eric_account.transactions
    [('deposit', 1000000)]
    >>> eric_account.withdraw(100)      # make a withdrawal to buy dinner
    999900
    >>> eric_account.transactions
    [('deposit', 1000000), ('withdraw', 100)]
    >>> print(eric_account) #call to __str__
    Eric's Balance: $999900
    >>> eric_account.deposit(10)
    999910
    >>> eric_account #call to __repr__
    Accountholder: Eric, Deposits: 2, Withdraws: 1
    """

    interest = 0.02

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
        "*** YOUR CODE HERE ***"
        self.transactions = []

    def deposit(self, amount):
        """Increase the account balance by amount, add the deposit
        to the transaction history, and return the new balance.
        """
        "*** YOUR CODE HERE ***"
        self.transactions.append(('deposit', amount))
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        """Decrease the account balance by amount, add the withdraw
        to the transaction history, and return the new balance.
        """
        "*** YOUR CODE HERE ***"
```

```
        self.transactions.append(('withdraw', amount))
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance

    def __str__(self):
        return f"{self.holder}'s Balance: ${self.balance}"

    def __repr__(self):
        num_deposits, num_withdraws = 0, 0
        for transaction in self.transactions:
            if transaction[0] == "withdraw":
                num_withdraws += 1
            else:
                num_deposits += 1
        return f"Accountholder: {self.holder}, Deposits: {num_deposits}, Withdraws: {num_v
```

Use Ok to test your code:

```
python3 ok -q Account
```

Mutable Lists

Q6: 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]
    >>> d = [1, 1]
    >>> e = [2]
    >>> trade(d, e)
    'Deal!'
    >>> d
    [2]
    >>> e
    [1, 1]
    """
    m, n = 1, 1

    equal_prefix = lambda: sum(first[:m]) == sum(second[:n])
    while m <= len(first) and n <= len(second) and not equal_prefix():
        if sum(first[:m]) < sum(second[:n]):
            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
```

Q7: 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.

To interleave two sequences `s0` and `s1` is to create a new sequence such that the new sequence contains (in this order) the first element of `s0`, the first element of `s1`, the second element of `s0`, the second element of `s1`, and so on. If the two lists are not the same length, then the leftover elements of the longer list should still appear at the end.

> **Note:** If you're running into an issue where the special heart / diamond / spades / clubs symbols are erroring in the doctests, feel free to copy paste the below doctests into your file as these don't use the special characters and should not give an "illegal multibyte sequence" error.

```
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 = ['H', 'D', 'S', 'C']
    >>> cards = [card(n) + suit for n in range(1,14) for suit in suits]
    >>> cards[:12]
    ['AH', 'AD', 'AS', 'AC', '2H', '2D', '2S', '2C', '3H', '3D', '3S', '3C']
    >>> cards[26:30]
    ['7S', '7C', '8H', '8D']
    >>> shuffle(cards)[:12]
    ['AH', '7S', 'AD', '7C', 'AS', '8H', 'AC', '8D', '2H', '8S', '2D', '8C']
    >>> shuffle(shuffle(cards))[:12]
    ['AH', '4D', '7S', '10C', 'AD', '4S', '7C', 'JH', 'AS', '4C', '8H', 'JD']
    >>> cards[:12]  # Should not be changed
    ['AH', 'AD', 'AS', 'AC', '2H', '2D', '2S', '2C', '3H', '3D', '3S', '3C']
    """
    assert len(cards) % 2 == 0, 'len(cards) must be even'
    half = len(cards) // 2
    shuffled = []
    for i in range(half):
        shuffled.append(cards[i])
        shuffled.append(cards[half+i])
    return shuffled
```

Use Ok to test your code:

```
python3 ok -q shuffle
```

Linked Lists

Q8: 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 should raise an `IndexError` with:
>
> ```
> raise IndexError('Out of bounds!')
> ```

```
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>
    >>> other_link = link
    >>> insert(link, 9001, 0)
    >>> print(link)
    <9001 1 2 3>
    >>> link is other_link # Make sure you are using mutation! Don't create a new linked l
    True
    >>> insert(link, 100, 2)
    >>> print(link)
    <9001 1 100 2 3>
    >>> insert(link, 4, 5)
    Traceback (most recent call last):
        ...
    IndexError: Out of bounds!
    """
    if index == 0:
        link.rest = Link(link.first, link.rest)
        link.first = value
    elif link.rest is Link.empty:
        raise IndexError("Out of bounds!")
    else:
        insert(link.rest, value, index - 1)

# iterative solution
def insert_iter(link, value, index):
    while index > 0 and link.rest is not Link.empty:
        link = link.rest
        index -= 1
    if index == 0:
        link.rest = Link(link.first, link.rest)
        link.first = value
    else:
        raise IndexError
```

Use Ok to test your code:

```
python3 ok -q insert
```

Q9: 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 lnk is Link.empty:
        return 0
    elif not isinstance(lnk, Link):
        return 1
    else:
        return deep_len(lnk.first) + deep_len(lnk.rest)
```

Video Walkthrough:
YouTube link (https://youtu.be/pbMeCRUU7yw?t=2m28s)

Use Ok to test your code:

```
python3 ok -q deep_len
```

Q10: 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 lnk is Link.empty:
            return empty_repr
        else:
            return front + str(lnk.first) + mid + printer(lnk.rest) + back
    return printer
```

Video Walkthrough:
YouTube link (https://youtu.be/pbMeCRUU7yw?t=14m22s)

Use Ok to test your code:

```
python3 ok -q make_to_string
```

Trees

Q11: 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 list of node labels that starts with the root and ends at a leaf. Each subsequent element must be from a label of a branch 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 ordered in the output list from left to right in the tree. See the doctests for some examples.

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

    >>> long_paths(Tree(1), 0)
    [[1]]
    >>> long_paths(Tree(1), 1)
    []
    >>> 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])
    >>> print(whole)
    0
      1
        2
        3
          4
          4
          5
      13
      6
        7
          8
        9
      11
        12
          13
            14
    >>> 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)
    [[0, 11, 12, 13, 14]]
    """
    if n <= 0 and t.is_leaf():
      return [[t.label]]
    paths = []
    for b in t.branches:
      for path in long_paths(b, n - 1):
          paths.append([t.label] + path)
    return paths
```

Use Ok to test your code:

```
python3 ok -q long_paths
```

Efficiency

Q12: Growth: Is Palindrome

> This question was reformatted from question 6(d) on fall 2019's final (https://cs61a.org/exam/fa19/final/61a-fa19-final.pdf#page=7).

Choose the term that fills in the blank: the `is_palindrome` function defined below runs in `____` time in the length of its input.

- Constant
- Logarithmic
- Linear
- Quadratic
- Exponential
- None of these

```
def is_palindrome(s):
    """Return whether a list of numbers s is a palindrome."""
    return all([s[i] == s[len(s) - i - 1] for i in range(len(s))])
```

Assume that `len` runs in constant time and `all` runs in linear time in the length of its input. Selecting an element of a list by its index requires constant time. Constructing a range requires constant time.

Use Ok to test your understanding:

```
python3 ok -q is_palindrome -u
```

> **Solution: Linear.**

We're interested in seeing how the function runs in relation to the length of its input, which is `len(s)`.

In `is_palindrome`, it takes constant time to calculate `len(s)` and then to construct a range from 0 to `len(s)` (exclusive).

For each element in this range, we will select two elements from `s` (`s[i]` and `s[len(s)-i-1]`) and compare them, which takes some constant time for each element. Once we've done this for all the elements, we will have built up the input list to `all` in linear time in relation to the length of `is_palindrome`'s input.

We assume that `all` runs in linear time in the length of *its* input, which is the length of the list we've just built and the same as the length of `is_palindrome`'s input.

Overall, `is_palindrome` will therefore take linear time in relation to the length of its input.