Exam Prep 6: Object-Oriented Programming, Trees, Linked Lists

Students from past semesters wanted more content and structured time to prepare for exams. Exam Prep sections are a way to solidify your understanding of the week's materials. The problems are typically designed to be a bridge between discussion/lab/homework difficulty and exam difficulty.

Reminder: There is nothing to turn in and there is no credit given for attending Exam Prep Sections.

We try to make these problems **exam level** , so you are not expected to be able to solve them coming straight from lecture without additional practice. To get the most out of Exam Prep, we recommend you **try these problems first on your own** before coming to the Exam Prep section, where we will explain how to solve these problems while giving tips and advice for the exam. Do not worry if you struggle with these problems, **it is okay to struggle while learning**.

You can work with anyone you want, including sharing solutions. We just ask you don't spoil the problems for anyone else in the class. Thanks!

You may only put code where there are underscores for the codewriting questions.

You can test your functions on their doctests by clicking the red test tube in the top right corner after clicking Run in 61A Code. Passing the doctests is not necessarily enough to get the problem fully correct. You must fully solve the stated problem.

Q1: Iterator Tree Link Tree Iterator

Difficulty: ⭐⭐

Part A: Fill out the function `funcs`, which is a generator that takes in a linked list `link` and yields functions.

The linked list `link` defines a path from the root of the tree to one of its nodes, with each element of link specifying which branch to take by index. Applying all functions sequentially to a Tree instance will evaluate to the label of the node at the end of the specified path.

For example, using the Tree `t` defined in the code, `funcs(Link(2))` yields 2 functions. The first gets the third branch from t -- the branch at index 2 -- and the second function gets the label of this branch.

```
>>> func_generator = funcs(Link(2)) # get label of third branch
>>> f1 = next(func_generator)
>>> f2 = next(func_generator)
>>> f2(f1(t))
4
```

Part B: Using `funcs` from above, fill out the definition for `apply`, which applies `g` to the element in `t` who's position is at the end of the path defined by `link`.

```
def funcs(link):
    """
    >>> t = Tree(1, [Tree(2,
    ...                   [Tree(5),
    ...                    Tree(6, [Tree(8)])]),
    ...              Tree(3),
    ...              Tree(4, [Tree(7)])])
    >>> print_tree(t)
    1
      2
```

```
        5
        6
          8
      3
      4
        7
    >>> func_generator = funcs(Link.empty) # get root label
    >>> f1 = next(func_generator)
    >>> f1(t)
    1
    >>> func_generator = funcs(Link(2)) # get label of third branch
    >>> f1 = next(func_generator)
    >>> f2 = next(func_generator)
    >>> f2(f1(t))
    4
    >>> # This just puts the 4 values from the iterable into f1, f2, f3, f4
    >>> f1, f2, f3, f4 = funcs(Link(0, Link(1, Link(0))))
    >>> f4(f3(f2(f1(t))))
    8
    """
    if link is Link.empty:
        yield lambda t: t.label
    else:
        yield lambda t: t.branches[link.first]
        yield from funcs(link.rest)

def apply(g, t, link):
    """
    >>> t = Tree(1, [Tree(2,
    ...                   [Tree(5),
    ...                    Tree(6, [Tree(8)])]),
    ...               Tree(3),
    ...               Tree(4, [Tree(7)])])
    >>> print_tree(t)
    1
      2
        5
        6
```

```
          8
      3
      4
        7
    >>> apply(lambda x: x, t, Link.empty) # root label
    1
    >>> apply(lambda x: x, t, Link(0))    # label at first branch
    2
    >>> apply(lambda x: x * x, t, Link(0, Link(1, Link(0))))
    64
    """
    for f in funcs(link):
        t = f(t)
    return g(t)

t = Tree(1, [Tree(2,
                [Tree(5),
                 Tree(6, [Tree(8)])]),
             Tree(3),
             Tree(4, [Tree(7)])])

def print_tree(t, indent=0):
    """Print a representation of this tree in which each node is
    indented by two spaces times its depth from the root.
    """
    print('  ' * indent + str(t.label))
```

Q2: Cucumber - Fall 2015 Final Q7

Difficulty: ⭐⭐

Cucumber is a card game. Cards are positive integers (no suits). Players are numbered from 0 up to `players` (0, 1, 2, 3 in a 4-player game).

In each `Round`, the players each `play` one card, starting with the `starter` and in ascending order (player 0 follows player 3 in a 4-player game). If the `card` played is as high or higher than the `highest` card played so far, that player takes control. The winner is the last player who took control after every player has played once.

Implement `Round` so that `Game` behaves as described in the doctests below.

EDIT: The first two lines in the play function should be:

```
assert ______________________________, f'The round is over, player {who}'
assert _____________________________, f'It is not your turn, player {who}'
```

```
    Each (who, card) pair in cards indicates who plays and what card they play.
    >>> g = Game()
    >>> g.play_round(3, [(3, 4), (0, 8), (1, 8), (2, 5)])
    >>> g.winners
    [1]
    >>> g.play_round(1, [(3, 5), (1, 4), (2, 5), (0, 8), (3, 7), (0, 6), (1, 7)])
    It is not your turn, player 3
    It is not your turn, player 0
    The round is over, player 1
    >>> g.winners
    [1, 3]
    >>> g.play_round(3, [(3, 7), (2, 5), (0, 9)]) # Round is never completed
    It is not your turn, player 2
    >>> g.winners
    [1, 3]
    """
```

```
    def __init__(self):
        self.winners = []

    def play_round(self, starter, cards):
        r = Round(starter)
        for who, card in cards:
            try:
                r.play(who, card)
            except AssertionError as e:
                print(e)
        if r.winner != None:
            self.winners.append(r.winner)

class Round:
    players = 4

    def __init__(self, starter):
        self.starter = starter
        self.next_player = starter
        self.highest = -1
        self.winner = None

    def play(self, who, card):
        assert not self.is_complete(), f'The round is over, player {who}'
        assert who == self.next_player, f'It is not your turn, player {who}'
        self.next_player = (who + 1) % Round.players
        if card >= self.highest:
            self.highest = card
            self.control = who
        if self.is_complete():
            self.winner = self.control

    def is_complete(self):
        """ Checks if a game could end. """
        return self.next_player == self.starter and self.highest > -1
```

Q3: Count Coins Tree

IMPORTANT: For this problem, you will be given time during the Exam Prep section to solve on your own before we go over it.

Difficulty: ⭐⭐⭐

You want to help your friend learn tree recursion. They don't quite understand all the recursive calls and how they work, so you decide to make a tree of recursive calls to showcase the tree in tree in tree recursion. You pick the `count_coins` problem.

Implement `count_coins_tree`, which takes in a non-negative integer n and returns a tree representing the recursive calls of count change.

Since you don't want your trees to get too big, you decide to only include the recursive calls that eventually lead to a valid way of making change.

See the code for an implementation of `count_coins`.

For the times when either recursive call returns `None`, you don't want to include that in your branches when making the tree. If both recursive calls return `None`, then you want to return `None`.

Each leaf for the `count_coins_tree(15, [1, 5, 10, 25])` tree is one of these groupings.

- 15 1-cent coins
- 10 1-cent, 1 5-cent coins
- 5 1-cent, 2 5-cent coins
- 5 1-cent, 1 10-cent coins
- 3 5-cent coins
- 1 5-cent, 1 10-cent coin

```
def count_coins_tree(change, denominations):
    """
    >>> count_coins_tree(1, []) # Return None since no ways to make change wuth no denominations
    >>> t = count_coins_tree(3, [1, 2])
```

```
>>> print_tree(t) # 2 ways to make change for 3 cents
3, [1, 2]
  2, [1, 2]
    2, [2]
      1
    1, [1, 2]
      1
>>> # The tree that shows the recursive calls that result
>>> # in the 6 ways to make change for 15 cents
>>> t = count_coins_tree(15, [1, 5, 10, 25])
>>> print_tree(t)
15, [1, 5, 10, 25]
  15, [5, 10, 25]
    10, [5, 10, 25]
      10, [10, 25]
        1
      5, [5, 10, 25]
        1
  14, [1, 5, 10, 25]
    13, [1, 5, 10, 25]
      12, [1, 5, 10, 25]
        11, [1, 5, 10, 25]
          10, [1, 5, 10, 25]
            10, [5, 10, 25]
              10, [10, 25]
                1
              5, [5, 10, 25]
                1
            9, [1, 5, 10, 25]
              8, [1, 5, 10, 25]
                7, [1, 5, 10, 25]
                  6, [1, 5, 10, 25]
                    5, [1, 5, 10, 25]
                      5, [5, 10, 25]
                        1
                      4, [1, 5, 10, 25]
                        3, [1, 5, 10, 25]
                          2, [1, 5, 10, 25]
```

```
                                1, [1, 5, 10, 25]
                                  1
    """
    "*** YOUR CODE HERE ***"
    if change == 0:
        return Tree(1)
    if change < 0:
        return None
    if len(denominations) == 0:
        return None
    branches = []
    without_current = count_coins_tree(change, denominations[1:])
    with_current = count_coins_tree(change - denominations[0], denominations)
    if without_current:
        branches.append(without_current)
    if with_current:
        branches.append(with_current)
    if branches:
        return Tree(f'{change}, {denominations}', branches)
def count_coins(change, denominations):
    """
    Given a positive integer change, and a list of integers denominations,
    a group of coins makes change for change if the sum of the values of
    the coins is change and each coin is an element in denominations.
    count_coins returns the number of such groups.
    """
    if change == 0:
```

Q4: Extra Practice

Difficulty: >⭐⭐⭐

Fall 2020 Midterm 2 Review Exam Prep
(https://drive.google.com/file/d/1FYHQVESZ_jBiHCubKp4O-mrim3x3eFEP/view?usp=sharing)