CS 61A Spring 2020

Final Review

Discussion 12: April 22, 2020 Solutions

1 Recursion

1.1 (Adapted from Fall 2013) Fill in the blanks in the implementation of `paths`, which takes as input two positive integers `x` and `y`. It returns a list of paths, where each path is a list containing steps to reach `y` from `x` by repeated incrementing or doubling. For instance, we can reach 9 from 3 by incrementing to 4, doubling to 8, then incrementing again to 9, so one path is `[3, 4, 8, 9]`

```
def paths(x, y):
    """Return a list of ways to reach y from x by repeated
    incrementing or doubling.
    >>> paths(3, 5)
    [[3, 4, 5]]
    >>> sorted(paths(3, 6))
    [[3, 4, 5, 6], [3, 6]]
    >>> sorted(paths(3, 9))
    [[3, 4, 5, 6, 7, 8, 9], [3, 4, 8, 9], [3, 6, 7, 8, 9]]
    >>> paths(3, 3) # No calls is a valid path
    [[3]]
    """
    if ____________________________:

        return ____________________________________________

    elif ________________________:

        return ____________________________________________

    else:

        a = ____________________________________________

        b = ____________________________________________

        return ____________________________________________
```

```
def paths(x, y):
    if x > y:
        return []
    elif x == y:
        return [[x]]
    else:
```

```
    a = paths(x + 1, y)
    b = paths(x * 2, y)
    return [[x] + subpath for subpath in a + b]
```

1.2 We will now write one of the faster sorting algorithms commonly used, known as *merge sort*. Merge sort works like this:

1. If there is only one (or zero) item(s) in the sequence, it is already sorted!
2. If there are more than one item, then we can split the sequence in half, sort each half recursively, then merge the results, using the `merge` procedure described below. The result will be a sorted sequence.

Using the algorithm described, write a function `mergesort(seq)` that takes an unsorted sequence and sorts it.

Recall the `merge` procedure is as follows:

```
def merge(s1, s2):
    """ Merges two sorted lists """
    if len(s1) == 0:
        return s2
    elif len(s2) == 0:
        return s1
    elif s1[0] < s2[0]:
        return [s1[0]] + merge(s1[1:], s2)
    else:
        return [s2[0]] + merge(s1, s2[1:])

def mergesort(seq):


    if len(seq) <= 1:
        return seq
    else:
        mid = len(seq) // 2
        return merge(mergesort(seq[:mid]),
            mergesort(seq[mid:]))
```

2 Trees

2.1 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)))))]
    """


    paths = []
    if n <= 0 and tree.is_leaf():
        paths.append(Link(tree.label))
    for b in tree.branches:
        for path in long_paths(b, n - 1):
            paths.append(Link(tree.label, path))
    return paths
```

2.2 Write a function that takes a `Tree` object and returns the elements at the depth with the most elements.

In this problem, you may find it helpful to use the second optional argument to `sum`, which provides a starting value. All items in the sequence to be summed will be concatenated to the starting value. By default, start will default to 0, which allows you to sum a sequence of numbers. We provide an example of sum starting with a list, which allows you to concatenate items in a list.

```
def widest_level(t):
    """
    >>> sum([[1], [2]], [])
    [1, 2]
    >>> t = Tree(3, [Tree(1, [Tree(1), Tree(5)]),
    ...              Tree(4, [Tree(9, [Tree(2)])])])
    >>> widest_level(t)
    [1, 5, 9]
    """
    levels = []
    x = [t]

    while ____________________________________________________:

        ____________________________________________________

        __________ = sum(_________________________________, [])

    return max(levels, key=_____________________________________)
```

```
def widest_level(t):
    levels = []
    x = [t]
    while x:
        levels.append([t.label for t in x])
        x = sum([t.branches for t in x], [])
    return max(levels, key=len)
```

Main idea: we'll traverse each level of the tree and keep track of the elements of the levels. After we're done, we return the level with the most items.

Here, `x` keeps track of the trees in the current level. To get the next level of trees, we take all the branches from all the trees in the current level. The special `sum` call is needed to make sure we get a list of trees, instead of a list of branches (since branches are a list of trees themselves).

Finally, we use `max` with a key to select the list with the longest length from our list of levels.

3 Mutation

3.1 For each row below, fill in the blanks in the output displayed by the interactive Python interpreter when the expression is evaluated. Expressions are evaluated in order, and expressions may affect later expressions.

```
>>> cats = [1, 2]
>>> dogs = [cats, cats.append(23), list(cats)]
>>> cats
```

```
[1, 2, 23]
```

```
>>> dogs[1] = list(dogs)
>>> dogs[1]
```

```
[[1, 2, 23], None, [1, 2, 23]]
```

```
>>> dogs[0].append(2)
>>> cats
```

```
[1, 2, 23, 2]
```

```
>>> cats[1::2]
```

```
[2, 2]
```

```
>>> cats[:3]
```

```
[1, 2, 23]
```

```
>>> dogs[2].extend([list(cats).pop(0), 3])
>>> dogs[3]
```

```
Index Error
```

```
>>> dogs
```

```
[[1, 2, 23, 2], [[1, 2, 23, 2], None, [1, 2, 23, 1, 3]], [1, 2, 23, 1, 3]]
```

4 OOP

4.1 (Summer 2015 Final) The TAs are building a social networking website called CS61A+. The TAs plan to represent the network in a class called `Network` that supports the following method:

- `add_friend(user1, user2)` adds `user1` and `user2` to each other's friends lists. If `user1` or `user2` are not in the `Network`, add them to the dictionary of friends.

Help the TAs implement these two methods to make their social networking website popular!

```
class Network:
    """
    >>> cs61a_plus = Network()
    >>> cs61a_plus.add_friend('Robert', 'Jeffrey')
    >>> cs61a_plus.friends['Robert']
    ['Jeffrey']
    >>> cs61a_plus.friends['Jeffrey']
    ['Robert']
    >>> cs61a_plus.add_friend('Jessica', 'Robert')
    >>> cs61a_plus.friends['Robert']
    ['Jeffrey', 'Jessica']
    """
    def __init__(self):
        self.friends = {}       # Maps users to a list of their friends

    def add_friend(self, user1, user2):


        if ____________________________________________________:


            ____________________________________________________


        if ____________________________________________________:


            ____________________________________________________


        ____________________________________________________


        ____________________________________________________


    def add_friend(self, user1, user2):
        if user1 not in self.friends:
```

```
            self.friends[user1] = []
        if user2 not in self.friends:
            self.friends[user2] = []
        self.friends[user1].append(user2)
        self.friends[user2].append(user1)
```

CS61A+ turns out to be unpopular. To attract more users, the TAs want to implement a feature that checks if two users have at most `n` degrees of separation. Consider the following CS61A+ `Network`:

```
self.friends = {
    'Robert':  ['Jeffrey', 'Jessica'],
    'Jeffrey': ['Robert', 'Jessica', 'Yulin'],
    'Jessica': ['Robert', 'Jeffrey', 'Yulin'],
    'Yulin':   ['Jeffrey', 'Jessica'],
    'Albert':  []
}
```

- There is 1 degree of separation between Robert and Jeffrey, because they are direct friends.
- There are 2 degrees of separation between Robert and Yulin (Robert → Jessica → Yulin)
- The degree of separation between Albert and anyone else is undefined, since Albert has no friends.

```
class Network:
    # Code from previous question

    def degrees(self, user1, user2, n):
        """In these doctests, assume cs61a_plus is a Network with the
        dictionary of friends described in the example.

        >>> cs61a_plus.degrees('Robert', 'Yulin', 2)    # Exactly 2 degrees
        True
        >>> cs61a_plus.degrees('Robert', 'Jessica', 2)  # Less than 2 degrees
        True
        >>> cs61a_plus.degrees('Yulin', 'Robert', 1)    # More than 1 degree
        False
        >>> cs61a_plus.degrees('Albert', 'Jessica', 10) # No friends!
        False
        """
        if ______________________________________:
            return True

        elif ______________________________________:
            return False

        for friend in _____________________________:

            if ________________________________________:
                return True

        return ____________________________________
```

```
class Network:
    # Code from previous question

    def degrees(self, user1, user2, n):
        """
        >>> cs61a_plus = Network()
        >>> cs61a_plus.friends = {
        ...     'Robert': ['Jeffrey', 'Jessica'],
        ...     'Jeffrey': ['Robert', 'Jessica', 'Yulin'],
        ...     'Jessica': ['Robert', 'Jeffrey', 'Yulin'],
        ...     'Yulin': ['Jeffrey', 'Jessica'],
        ...     'Albert': []
        ... }
        >>> cs61a_plus.degrees('Robert', 'Yulin', 2)    # Exactly 2 degrees
        True
        >>> cs61a_plus.degrees('Robert', 'Jessica', 2)  # Less than 2 degrees
        True
        >>> cs61a_plus.degrees('Yulin', 'Robert', 1)    # More than 1 degree
        False
        >>> cs61a_plus.degrees('Albert', 'Jessica', 10) # No friends!
        False
        """

        if user1 == user2:

            return True

        elif n <= 0:

            return False

        for friend in self.friends[user1]:

            if self.degrees(friend, user2, n - 1):

                return True

        return False
```

5 Mutable Linked Lists and Trees

5.1 Write a recursive function `flip_two` that takes as input a linked list `lnk` and mutates `lnk` so that every pair is flipped.

```
def flip_two(lnk):
    """
    >>> one_lnk = Link(1)
    >>> flip_two(one_lnk)
    >>> one_lnk
    Link(1)
    >>> lnk = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> flip_two(lnk)
    >>> lnk
    Link(2, Link(1, Link(4, Link(3, Link(5)))))
    """
```

Recursive solution:

```
    if lnk is Link.empty or lnk.rest is Link.empty:
        return
    lnk.first, lnk.rest.first = lnk.rest.first, lnk.first
    flip_two(lnk.rest.rest)
```

If there's only a single item (or no item) to flip, then we're done.

Otherwise, we swap the contents of the first and second items in the list. Since we've handled the first two items, we then need to recurse on

Although the question explicitly asks for a recursive solution, there is also a fairly similar iterative solution:

```
    while lnk is not Link.empty and lnk.rest is not Link.empty:
        lnk.first, lnk.rest.first = lnk.rest.first, lnk.first
        lnk = lnk.rest.rest
```

We will advance `lnk` until we see there are no more items or there is only one more Link object to process. Processing each `Link` involves swapping the contents of the first and second items in the list (same as the recursive solution).

Notice that the code is remarkably similar to the recursive implementation of `flip_two`.
Video walkthrough

6 Generators

6.1 Write a generator function that yields functions that are repeated applications of a one-argument function `f`. The first function yielded should apply `f` 0 times (the identity function), the second function yielded should apply `f` once, etc.

```
def repeated(f):
    """
    >>> double = lambda x: 2 * x
    >>> funcs = repeated(double)
    >>> identity = next(funcs)
    >>> double = next(funcs)
    >>> quad = next(funcs)
    >>> oct = next(funcs)
    >>> quad(1)
    4
    >>> oct(1)
    8
    >>> [g(1) for _, g in
    ...  zip(range(5), repeated(lambda x: 2 * x))]
    [1, 2, 4, 8, 16]
    """


    g = ____________________________________________________________________

    while True:


        ____________________________________________________________________


        ____________________________________________________________________
```

```
def repeated(f):
    g = lambda x: x
    while True:
        yield g
        g = (lambda g: lambda x: f(g(x)))(g)
```

Video walkthrough

6.2 Ben Bitdiddle proposes the following alternate solution. Does it work?

```
def ben_repeated(f):
    g = lambda x: x
    while True:
        yield g
        g = lambda x: f(g(x))
```

This solution does not work. The value g changes with each iteration so the bodies of the lambdas yielded change as well.

6.3 Implement `accumulate`, which takes in an `iterable` and a function `f` and yields each accumulated value from applying `f` to the running total and the next element.

```
from operator import add, mul

def accumulate(iterable, f):
    """
    >>> list(accumulate([1, 2, 3, 4, 5], add))
    [1, 3, 6, 10, 15]
    >>> list(accumulate([1, 2, 3, 4, 5], mul))
    [1, 2, 6, 24, 120]
    """
    it = iter(iterable)


    ____________________________________________________________________________


    ____________________________________________________________________________


    for ____________________________________________________________________:


        ________________________________________________________________________


        ________________________________________________________________________


    total = next(it)
    yield total
    for element in it:
        total = f(total, element)
        yield total
```

7 Scheme

7.1 Write a function that takes a procedure and applies to every element in a given nested list.

The result should be a nested list with the same structure as the input list, but with each element replaced by the result of applying the procedure to that element.

Use the built-in `list?` procedure to detect whether a value is a list.

```
(define (deep-map fn lst)


  (cond ((null? lst) lst)
        ((list? (car lst)) (cons (deep-map fn (car lst)) (deep-map fn (cdr lst))))
        (else (cons (fn (car lst)) (deep-map fn (cdr lst))))
        )
   )
```

```
scm> (deep-map (lambda (x) (* x x)) '(1 2 3))
(1 4 9)
scm> (deep-map (lambda (x) (* x x)) '(1 ((4) 5) 9))
(1 ((16) 25) 81)
```