

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 x == y:
        return return [[x]]

    elif x > y:
        return return []

    else:
        a = paths(x + 1, y)
        b = paths(x * 2, y)

        return [[x] + path for path in a + b]
```

2 Final Review

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:  
        return merge(mergesort(seq[:len(seq)//2]), mergesort(seq[len(seq)//2:]))
```

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

- 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 ____X_____:
        levels.append([t.label for t in x])
        ____X____ = sum([t.branches for t in x], [])

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

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]
```

Error list index out of range

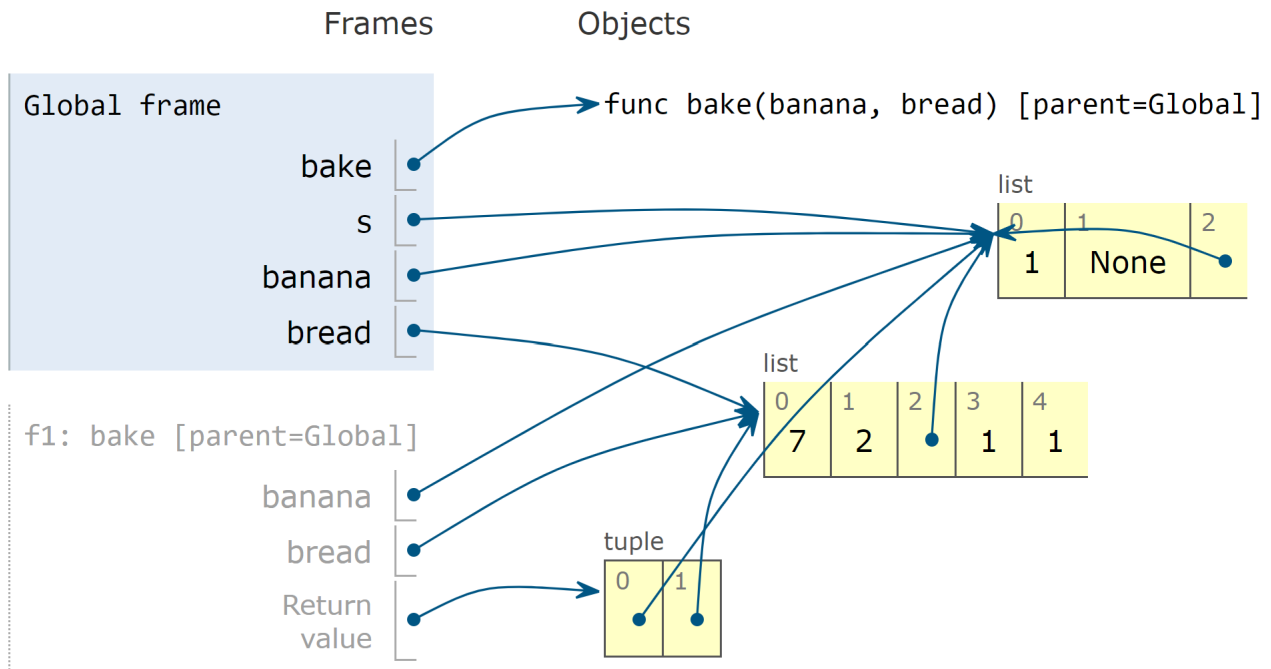
```
>>> dogs
```

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

- 3.2 Fill in the following lines so that the code creates the environment diagram shown below. **You may only use append, extend, 1, banana, and bread in your solution.**

```
def bake(banana, bread):
    ____A____(_____(____)) # This line is Multiple Choice
    # Select all correct answers for the blank above
    # A. banana.append(bread.append(1))
    # B. bread.append(banana.append(1))
    # C. banana.extend([bread.extend([1])])
    # D. bread.extend([banana.extend([1])])
    bread += banana[: (len(banana) - 1)]
    banana.append(bread[bread - 1])
    return banana, bread
```

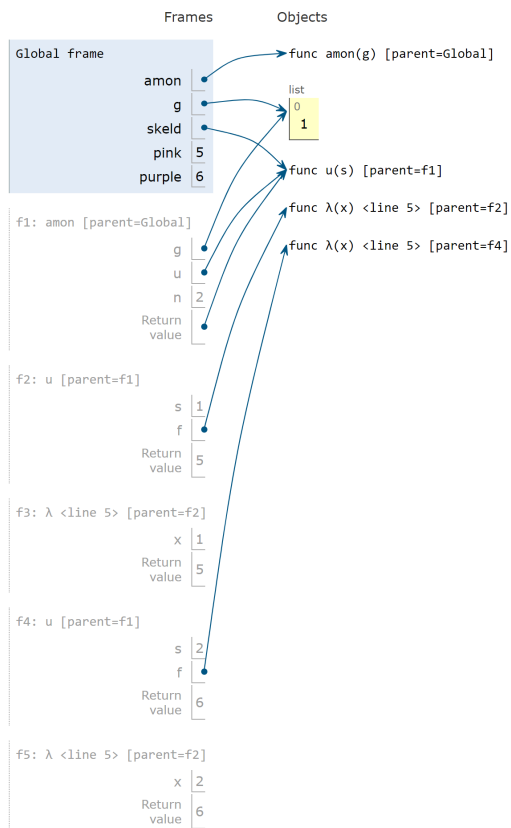
```
s = [1]
banana, bread = bake(s, [7, 2, s])
```



3.3 Fill in the following lines so that the code creates the environment diagram shown below.

```
def amon(g):
        n = 0    
    (a)
def u(s):
        nonlocal n    
    (b)
    f = lambda x: x + g.pop() + n
    (c)
        n += 1    
    (d)
    return f(s)
return u
```

```
g = [1, 2, 3]
skeld = amon(g)
pink = skeld(1)
purple = skeld(2)
```



4 OOP

- 4.1 Fill in the classes Emotion, Joy, and Sadness below so that you get the following output from the Python interpreter.

```
>>> Emotion.num
0
>>> joy = Joy()
>>> sadness = Sadness()
>>> Emotion.num # number of Emotion instances created
2
>>> joy.power
5
>>> joy.catchphrase() # Print Joy's catchphrase
Think positive thoughts
>>> sadness.catchphrase() #Print Sad's catchphrase
I'm positive you will get lost
>>> sadness.power
5
>>> joy.feeling(sadness) # When both Emotions have same power value, print "Together"
Together
>>> sadness.feeling(joy)
Together
>>> joy.power = 7
>>> joy.feeling(sadness) # Print the catchphrase of the more powerful feeling before the less
powerful feeling
Think positive thoughts
I'm positive you will get lost
>>> sadness.feeling(joy)
Think positive thoughts
I'm positive you will get lost
```



```

class Emotion(_____):

    num = 0

    def __init__(self):
        Emotion.num += 1

    def feeling(self, other):
        if self.power == other.power:
            print('Together')
        elif self.power < other.power:
            other.catchphrase()
            self.catchphrase()
        else:
            self.catchphrase()
            other.catchphrase()

class Joy(Emotion):
    power = 5
    def catchphrase(self):
        print('Think positive thoughts')

class Sadness(Emotion):
    power = 5
    def catchphrase(self):
        print('I\'m positive you will get lost')

```

5 Mutable Linked Lists and Trees

- 5.1 Write a function that takes a sorted linked list of integers and mutates it so that all duplicates are removed.

```
def remove_duplicates(lnk):
    """
    >>> lnk = Link(1, Link(1, Link(1, Link(1, Link(5)))))
    >>> remove_duplicates(lnk)
    >>> lnk
    Link(1, Link(5))
    """

    if lnk == Link.empty or lnk.rest == Link.empty:
        return
    if lnk.first == lnk.rest.first:
        lnk.rest = lnk.rest.rest
        remove_duplicates(lnk)
    else:
        remove_duplicates(lnk.rest)
```

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]
    """
```

$g = \text{lambda } x: x$

while True:

yield g

$g = (\text{lambda } y: \text{lambda } x: f(y(x)))(g)$

- 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))
```

no

- 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)
```

```
    total = next(it)
```

```
    yield total
```

```
    for elemnet in it:
```

```
        total = f(total, elemnet)
```

```
        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)
```

8 SQL

(Adapted from Fall 2019) The scoring table has three columns, a player column of strings, a points column of integers, and a quarter column of integers. The players table has two columns, a name column of strings and a team column of strings. Complete the SQL statements below so that they would compute the correct result even if the rows in these tables were different than those shown. Important: You may write anything in the blanks including keywords such as WHERE or ORDER BY. Use the following tables for the questions below:

```
CREATE TABLE scoring AS
  SELECT "Donald Stewart" AS player, 7 AS points, 1 AS quarter UNION
  SELECT "Christopher Brown Jr.", 7, 1 UNION
  SELECT "Ryan Sanborn", 3, 2 UNION
  SELECT "Greg Thomas", 3, 2 UNION
  SELECT "Cameron Scarlett", 7, 3 UNION
  SELECT "Nikko Remigio", 7, 4 UNION
  SELECT "Ryan Sanborn", 3, 4 UNION
  SELECT "Chase Garbers", 7, 4;
```

```
CREATE TABLE players AS
  SELECT "Ryan Sanborn" AS name, "Stanford" AS team UNION
  SELECT "Donald Stewart", "Stanford" UNION
  SELECT "Cameron Scarlett", "Stanford" UNION
  SELECT "Christopher Brown Jr.", "Cal" UNION
  SELECT "Greg Thomas", "Cal" UNION
  SELECT "Nikko Remigio", "Cal" UNION
  SELECT "Chase Garbers", "Cal";
```

- 8.1 Write a SQL statement to select a one-column table of quarters in which more than 10 total points were scored.

SELECT s.quarter FROM scoring AS s GROUP BY s.quarter HAVING SUM(s.points) > 10;

- 8.2 Write a SQL statement to select a two-column table of the points scored by each team. Assume that no two players have the same name.

**SELECT p.team, SUM(s.points) FROM scoring AS s, players AS p
WHERE s.player = p.name GROUP BY p.team;**

