recursion

by regina and add

slides: links.cs61a.org/recursion-review

3 Steps for Recursion

1. Base Case: What's the simplest possible input to a function?
2. Recursive Call: Calling your function on a "simpler" input
3. Using the output of the recursive call to solve our entire problem

Recursion Tips

- Generally, recursive solutions will have some kind of if/elif/else structure
- When writing the body of your function, pretend that your function is already complete and perfect!
  - the “recursive leap of faith”
- Make sure each `return` statement is returning the same data type
- If you already have a base case, make sure each recursive call’s arguments help you get “closer” to the base case

factorial

Write factorial as a function.

Recall: n! = n * (n-1) * … * 1

```
def factorial(n):
    """
    >>> factorial(3) # 3 * 2 * 1
    6
    >>> factorial(4) # 4 * 3 * 2 * 1
    24
    """
    return ____
```

factorial

Write factorial as a function.

Recall: n! = n * (n-1) * … * 1

```
def factorial(n):
    """
    >>> factorial(3) # 3 * 2 * 1
    6
    >>> factorial(4) # 4 * 3 * 2 * 1
    24
    """
    return n * factorial(n-1)
```

list prod

Write `list_prod`, which takes a list of numbers and returns the product of all the numbers in the list. Do not use any for loops or built in functions.

```
def list_prod(lst):
      _____________
            __________
      _____________
```

list prod

Write `list_prod`, which takes a list of numbers and returns the product of all the numbers in the list. Do not use any for loops or built in functions.

```
def list_prod(lst):
    if len(lst) == 0:
        return 1
    return lst[0] * list_prod(lst[1:])
```

won't you be my neighbor?

Implement `repeat_digits`, which takes in a positive integer N, and returns a number with each digit repeated.

```
def repeat_digits(n):
    """
    >>> repeat_digits(1234)
    11223344
    """
    last, rest = _____, _____
    if _____:
        return _____
    return _____(_____) * _____ + _____
```

won't you be my neighbor?

Implement `repeat_digits`, which takes in a positive integer N, and returns a number with each digit repeated.

```
def repeat_digits(n):
    """
    >>> repeat_digits(1234)
    11223344
    """
    last, rest = n % 10, n // 10
    if rest == 0:
        return last * 11 # or last * 10 + last
    return repeat_digits(rest) * 100 + last * 11
```

repeated

In Homework 2 you encountered the `repeated` function, which takes arguments `f` and `n` and returns a function equivalent to the nth repeated application of `f`.

This time, we want to write `repeated` recursively. You'll want to use `compose1`, given for your convenience.

```
def compose1(f, g):
    """Return a function h, such that h(x) = f(g(x))."""
    def h(x):
        return f(g(x))
    return h

def repeated(f, n):
    """
    >>> add_three = repeated(increment, 3)
    >>> add_three(5)
    8
    >>> repeated(triple, 5)(1) # 3 * 3 * 3 * 3 * 3 * 1
    243
    """
    if n == 0:
        return _____
    return _____
```

repeated

In Homework 2 you encountered the `repeated` function, which takes arguments `f` and `n` and returns a function equivalent to the nth repeated application of `f`.

This time, we want to write `repeated` recursively. You'll want to use `compose1`, given for your convenience.

```
def compose1(f, g):
    """Return a function h, such that h(x) = f(g(x))."""
    def h(x):
        return f(g(x))
    return h

def repeated(f, n):
    """
    >>> add_three = repeated(increment, 3)
    >>> add_three(5)
    8
    >>> repeated(triple, 5)(1) # 3 * 3 * 3 * 3 * 3 * 1
    243
    """
    if n == 0:
        return lambda x: x # Identity function
    return compose1(f, repeated(f, n-1))
```

group link

[fa15] Implement `group_link`, which takes a one-argument function `f` and a `Link` instance `s`. It returns a linked list of groups. Each group is a `Link` instance containing all the elements `x` in `s` that return equal values for `f(x)`.

```
def group_link(f, s):
    """
    >>> five = Link(3, Link(4, Link(5, Link(2, Link(1)))))
    >>> group_link(lambda x: x % 2, five)
    Link(Link(3, Link(5, Link(1))), Link(Link(4, Link(2))))
    >>> group_link(lambda x: x % 3, five)
    Link(Link(3), Link(Link(4, Link(1)), Link(Link(5,
    Link(2)))))
    """
    if s is Link.empty:
        return s
    else:
        a = filter_link(lambda x : ____, ____)
        b = filter_link(lambda x : ____, ____)
        return ________
```

group link

[fa15] Implement `group_link`, which takes a one-argument function `f` and a `Link` instance `s`. It returns a linked list of groups. Each group is a `Link` instance containing all the elements `x` in `s` that return equal values for `f(x)`.

```
def group_link(f, s):
    """
    >>> five = Link(3, Link(4, Link(5, Link(2, Link(1)))))
    >>> group_link(lambda x: x % 2, five)
    Link(Link(3, Link(5, Link(1))), Link(Link(4, Link(2))))
    >>> group_link(lambda x: x % 3, five)
    Link(Link(3), Link(Link(4, Link(1)), Link(Link(5,
    Link(2)))))
    """
    if s is Link.empty:
        return s
    else:
        a = filter_link(lambda x : f(x) == f(s.first), s)
        b = filter_link(lambda x : f(x) != f(s.first), s)
        return Link(a, group_link(f, b))
```

summer camp

Implement `sums`, which takes two positive integers n and k. It returns a list of lists containing all the ways that a list of k positive integers can sum to n. Results can appear in any order.

```
def sums(n, k):
    """
    >>> sums(2, 2)
    [[1, 1]]
    >>> sums(2, 3)
    []
    >>> sums(4, 2)
    [[3, 1], [2, 2], [1, 3]]
    """
    if ____:
        return ____
    y = []
    for x in ____:
        y.extend([____ for s in sums(____)])
    return y
```

summer camp

Implement `sums`, which takes two positive integers n and k. It returns a list of lists containing all the ways that a list of k positive integers can sum to n. Results can appear in any order.

```
def sums(n, k):
    """
    >>> sums(2, 2)
    [[1, 1]]
    >>> sums(2, 3)
    []
    >>> sums(4, 2)
    [[3, 1], [2, 2], [1, 3]]
    """
    if k == 1:
        return [[n]]
    y = []
    for x in range(1, n):
        y.extend([s + [x] for s in sums(n-x, k-1)])
    return y
```

summer camp

Implement `sums`, which takes two positive integers n and k. It returns a list of lists containing all the ways that a list of k positive integers can sum to n. Results can appear in any order.

CHALLENGE EDITION

```
f = lambda x, y: (x and [_____ for z in y] + f(_____,
_____)) or []

def sums(n, k):
    g = lambda w: (w and f(_____)) or [[]]
    return [v for v in g(k) if sum(v) == n]
```

summer camp

Implement `sums`, which takes two positive integers n and k. It returns a list of lists containing all the ways that a list of k positive integers can sum to n. Results can appear in any order.

CHALLENGE EDITION

```
f = lambda x, y: (x and [[x] + z for z in y] + f(x-1, y))
or []

def sums(n, k):
    g = lambda w: (w and f(n, g(w-1))) or [[]]
    return [v for v in g(k) if sum(v) == n]
```

thanos

[su18] A *messenger function* is a function that takes a single word and returns another messenger function, until a period is provided as input, in which case a sentence containing the words provided is returned. At least one word must be provided before the period. We have provided `simple_messenger` as such a function.

Write `thanos_messenger`, which is a messenger function that discards every other word that's provided. The first word should be included in the final sentence, the second word should be discarded, and so on.

```
>>> simple_messenger("Avengers")("assemble")(".")
'Avengers assemble.'
>>> simple_messenger("Get")("this")("man")("a")("shield")(".")
'Get this man a shield.'

def thanos_messenger(word):
    """A messenger function that discards every other word.
    >>> thanos_messenger("I")("don't")("feel")("so")("good")(".")
    'I feel good.'
    >>>thanos_messenger("Thanos")("always")("kills")("half")(".")
    'Thanos kills.'
    """
    assert word != '.', 'No words provided!'
    def make_new_messenger(message, skip_next):
        def new_messenger(word):
            if word == '.':
                return _____
            if _____:
                return _____
            return _____
        return _____
    return _____
```

thanos

[su18] A *messenger function* is a function that takes a single word and returns another messenger function, until a period is provided as input, in which case a sentence containing the words provided is returned. At least one word must be provided before the period. We have provided `simple_messenger` as such a function.

Write `thanos_messenger`, which is a messenger function that discards every other word that's provided. The first word should be included in the final sentence, the second word should be discarded, and so on.

```
>>> simple_messenger("Avengers")("assemble")(".")
'Avengers assemble.'
>>> simple_messenger("Get")("this")("man")("a")("shield")(".")
'Get this man a shield.'

def thanos_messenger(word):
    """A messenger function that discards every other word.
    >>> thanos_messenger("I")("don't")("feel")("so")("good")(".")
    'I feel good.'
    >>>thanos_messenger("Thanos")("always")("kills")("half")(".")
    'Thanos kills.'
    """
    assert word != '.', 'No words provided!'
    def make_new_messenger(message, skip_next):
        def new_messenger(word):
            if word == '.':
                return message + '.'
            if skip_next:
                return make_new_messenger(message, False)
            return make_new_messenger(message + " " + word, True)
        return new_messenger
    return make_new_messenger(word, True)
```

zombies

[fa15] In this question, assume that all of f, g, and h are functions that take one non-negative integer argument and return a non-negative integer. You do not need to consider negative or fractional numbers.

Implement the higher-order function `decompose1`, which takes two functions f and h as arguments. It returns a function g that relates f to h in the following way: For any non-negative integer x, `h(x)` equals `f(g(x))`. Assume that `decompose1` will be called only on arguments for which such a function g exists. Furthermore, assume that there is no recursion depth limit in Python.

```
def decompose1(f, h):
    """
    >>> add_one = lambda x: x + 1
    >>> square_then_add_one = lambda x: x * x + 1
    >>> g = decompose1(add_one, square_then_add_one)
    >>> g(5)
    25
    >>> g(10)
    100
    """
    def g(x):
        def r(y):
            if _____:
                return _____
            else:
                return _____
        return r(0)
    return _____
```

zombies

[fa15] In this question, assume that all of f, g, and h are functions that take one non-negative integer argument and return a non-negative integer. You do not need to consider negative or fractional numbers.

Implement the higher-order function `decompose1`, which takes two functions f and h as arguments. It returns a function g that relates f to h in the following way: For any non-negative integer x, `h(x)` equals `f(g(x))`. Assume that `decompose1` will be called only on arguments for which such a function g exists. Furthermore, assume that there is no recursion depth limit in Python.

```
def decompose1(f, h):
    """
    >>> add_one = lambda x: x + 1
    >>> square_then_add_one = lambda x: x * x + 1
    >>> g = decompose1(add_one, square_then_add_one)
    >>> g(5)
    25
    >>> g(10)
    100
    """
    def g(x):
        def r(y):
            if h(x) == f(y):
                return y
            else:
                return r(y+1)
        return r(0)
    return g
```

good luck!