QUICK CONTENT REVIEW

- Tree Recursion
  - a function that requires more than one recursive call in its recursive call

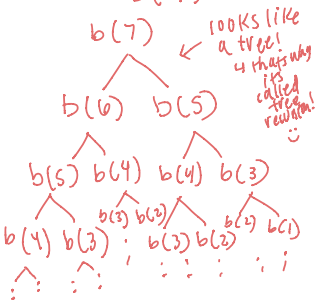
ie.)
```
def a(x):
    if x == 1:
        return 1
    elif x == 5:
        return (1 + a(x-2))
    else:
        return (1 + a(x-1))
```

there are 2 recursive calls but this is not tree recursion because at any given call to the function a, we will only evaluate at most one of the recursive calls

```
def b(x):
    if x == 1 or x == 0:
        return 1
    else:
        return (b(x-1) + b(x-2))
```

→ this is tree recursion because within one call to the function b, we are making 2 different recursive calls

To evaluate b(7):



  - whenever you see a problem in which you need to try multiple possibilities at the same time consider using tree recursion!

- Recursion
  - First, think of **base case**

    What is the simplest case of this problem? (should be able to solve this directly)

    ex.) simplest case for factorial: factorial(1) = 1 ← we know this, don't need to compute. in other words, if the argument passed into our factorial function is 1, we know that our output should always be 1

  - Next, make **recursive call** with simpler argument

    How can I express this problem in terms of itself with a smaller input?

    ex.) factorial(5) = 5 x factorial(4) ← can rewrite factorial(n) as n x factorial(n-1) (calling myself, just with smaller input)

  - Use Recursive call to solve whole problem → **Recursive Leap of Faith**

    Assume your recursive call returns the correct answer & solve from there.

    ex.) If we assume factorial(n-1) returns the correct value, then we know factorial(n) is just n * this value.

Q3: Recursive Multiplication (warmup)

Q3: (Tutorial) Warm Up: Recursive Multiplication

These exercises are meant to help refresh your memory of topics covered in lecture and/or lab this week before tackling more challenging problems.

Write a function that takes two numbers `m` and `n` and returns their product. Assume `m` and `n` are positive integers. Use **recursion**, not `mul` or `*`!

Hint: `5 * 3 = 5 + (5 * 2) = 5 + 5 + (5 * 1)`.

5x1 or 1x5

(m, 1) or (1, n)

For the base case, what is the simplest possible input for multiply?

Your Answer:

if m == 1: return n if n == 1: return m

Log in to save your work!

reduce by n

reduce by m

For the recursive case, what does calling `multiply(m - 1, n)` do? What does calling `multiply(m, n - 1)` do? Do we prefer one over the other?

5x3 = 5 + 5 + 5

5 + (5x2)

m x n = m + (m x n-1)

n x m = n + (n x m-1)

Your Answer:

No, we do not prefer one over the other

Log in to save your work!

Challenge: Try to implement the multiply function tail recursively.

mxn

```
def multiply(m, n):
    """ Takes two positive integers and returns their product using recursion.
    >>> multiply(5, 3)
    15
    """
    "*** YOUR CODE HERE ***"
```

Run in 61A Code

Log in to save your work!

```
if m == 1:
    return n
if n == 1:
    return m
return (n + multiply(m-1, n))
```

m x n = n + (m-1 x n)

PRACTICE PROBLEMS

Q4: Recursive Hailstone

1.3 **Tutorial:** Recall the `hailstone` function from Homework 1. First, pick a positive integer `n` as the start. If `n` is even, divide it by 2. If `n` is odd, multiply it by 3 and add 1. Repeat this process until `n` is 1. Write a recursive version of hailstone that prints out the values of the sequence and returns the number of steps.

Hint: When taking the recursive leap of faith, consider both the return value and side effect of this function.

```
def hailstone(n):
    """Print out the hailstone sequence starting at n, and return the
    number of elements in the sequence.

    >>> a = hailstone(10)
    10
    5
    16
    8
    4
    2
    1
    >>> a
    7
    """
```

Hints to get started:

- What is the base case? (can you see it in the instructions?)
  - base case is always in terms of the input
    - in our case, input is n (# hailstone sequence starts at)
      - so, do we know a single value that should always be returned for a given 'n'?
        - yes! we know from question, n=1 is where we stop so at n=1, we know we just want to print 1 and return 1
- If n is some number other than 1 (base case), what should we do? Are there different cases? what do we do in each case? (TRUST recursive leap of faith!!)
  - there are 2 different cases → n is even & n is odd
    - what to do in each?

n is even	n is odd
- print this #	- print this #
- now divide n by 2	- now multiply n by 3 and add 1
- what next? ↳ well, we are at identical situation to beginning but only n is diff => recursive call!	- what next? ↳ well, we are at identical situation to beginning but only n is diff => recursive call!

remember for the 'return' in each case we have to express the return value in terms of our recursive call. This is why the recursive leap of faith is so powerful! It enables us to proceed/code with the assumption that the recursive call returns the correct value.

SOLUTION:

```
def hailstone(n):
    print(n)
    if n == 1:
        return 1
    elif n % 2 == 0:
        return 1 + hailstone(n//2)
    else:
        return 1 + hailstone(3 * n + 1)
```

→ we want to print n at each step of hailstone sequence (putting print statement here ensures we are printing every time regardless of which condition we fall under)

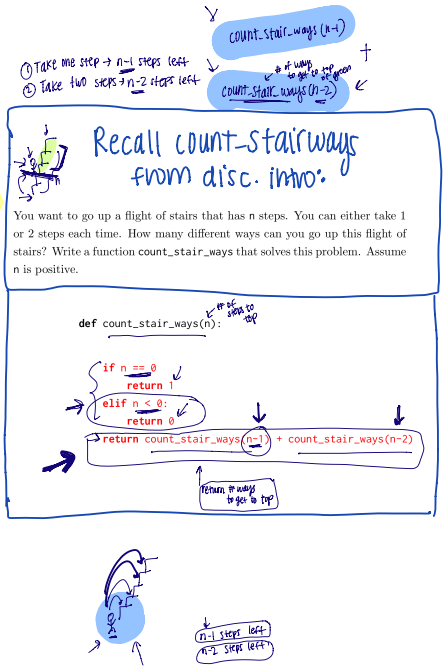
base case: smthg that we know to be true (if n=1, we always return 1)

hailstone completes is one step (printing 1)

2 recursive calls split into cases based on whether n is even or odd

if n is even,
we have already printed n (in the first line)
all we have left to do is call hailstone on the new "updated" n (n//2)
now to make sure we end up with the correct final return value, we assume hailstone(n//2) correctly returns the # of elements in the hailstone sequence starting at n//2. If we know that, then what is the total # of elements in the hailstone sequence starting at n? just hailstone(n//2) + 1 ! Again this is the power of recursive leap of faith!! ☺

Q6: count K



1.2 **Tutorial:** Consider a special version of the `count_stairways` problem, where instead of taking 1 or 2 steps, we are able to take **up to and including** `k` steps at a time.

Write a function `count_k` that figures out the number of paths for this scenario. Assume `n` and `k` are positive. → lowest possible vals for n & k is 0 does this help you come up with base cases?

```
def count_k(n, k): # of steps in staircase
    """
    >>> count_k(3, 3) # 3, 2 + 1, 1 + 2, 1 + 1 + 1
    4
    >>> count_k(4, 4)
    8
    >>> count_k(10, 3)
    274
    >>> count_k(300, 1) # Only one step at a time
    1
    """
```

Towers of Hanoi

HINTS / THINGS TO THINK ABOUT:

- Recognize that this is very similar to count_stair_ways(n) except that now instead of only being able to take 1 or 2 steps we can take up to & including k steps
  - ↳ can you generalize count_stair_ways() solution to apply here?
- Remember when there are choices for how we can do things (in this case choices for how many steps to take in any given path) we can use tree recursion
  - ↳ we can also use iteration (loops) in conjunction with tree recursion!

```
def count_k(n, k):
```

- What are base cases? Are they different than count_stair_ways?
- How can we generalize what we already did in count_stair_ways?

(If time permits, maybe try coding together on collabedit)

SOLUTION:

```
def count_K(n, K):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    else:
        total = 0
        i = 1
        while i <= K:
            total = total + count_K(n-i, K)
            i = i + 1
        return total
```

same base cases as for count_stair_ways (same stopping condition)

helps us generalize & add up paths from all K instead of just 0 & 1

use recursive leap of faith! For each $i \in [1, K]$ we want to count # of paths we can have [generalized version of recursive calls in count_stair_ways!)

$$count\_K(n-1, K) + count\_K(n-2, K) + count\_K(n-3, K) + \ldots + count\_K(n-K, K)$$