1 Recursion

A recursive function is a function that is defined in terms of itself. A good example is the factorial function. Although we haven’t finished defining factorial, we are still able to call it since the function body is not evaluated until the function is called. Note that when n is 0 or 1, we just return 1. This is known as the base case, and it prevents the function from infinitely recursing. Now we can compute factorial(2) in terms of factorial(1), and factorial(3) in terms of factorial(2), and factorial(4) – well, you get the idea.

There are three common steps in a recursive definition:

1. Figure out your base case: The base case is usually the simplest input possible to the function. For example, factorial(0) is 1 by definition. You can also think of a base case as a stopping condition for the recursion. If you can’t figure this out right away, move on to the recursive case and try to figure out the point at which we can’t reduce the problem any further.

2. Make a recursive call with a simpler argument: Simplify your problem, and assume that a recursive call for this new problem will simply work. This is called the “leap of faith”. For factorial, we reduce the problem by calling factorial(n-1).

3. Use your recursive call to solve the full problem: Remember that we are assuming the recursive call works. With the result of the recursive call, how can you solve the original problem you were asked? For factorial, we just multiply (n-1)! by n.

Note: One way to go understand recursion is to separate out two things: “internal correctness” and not running forever (known as “halting”).

A recursive function is internally correct if it is always does the right thing assuming that every recursive call does the right thing. For example, the same factorial function from above but with no base case is internally correct, but does not halt.

A recursive function is correct if and only if it is both internally correct and halts; but you can check each property separately. The “recursive leap of faith” is temporarily placing yourself in a mindset where you only check internal correctness.
Questions

1.1 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:** \( 5\times3 = 5 + 5\times2 = 5 + 5 + 5\times1 \).

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

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?

```python
def multiply(m, n):
    """
    >>> multiply(5, 3)
    15
    """
```
1.2 Draw an environment diagram for the following code:

```python
def rec(x, y):
    if y > 0:
        return x * rec(x, y - 1)
    return 1
rec(3, 2)
```

Bonus question: what does this function do?

*Note: This problem is meant to help you understand what really goes on when we make the "recursive leap of faith". However, when approaching or debugging recursive functions, you should avoid visualizing them in this way.*
In discussion 1, we implemented the function `is_prime`, which takes in a positive integer and returns whether or not that integer is prime, iteratively.

Now, let’s implement it recursively! As a reminder, an integer is considered prime if it has exactly two unique factors: 1 and itself.

```python
def is_prime(n):
    """
    >>> is_prime(7)
    True
    >>> is_prime(10)
    False
    >>> is_prime(1)
    False
    """

def prime_helper(____________________):
    if  ________________:
        ________________

    elif  ________________:
        ________________

    else:
        ________________

    return  ________________
```
1.4 (Optional)

Define a function `make_fn_repeater` which takes in a one-argument function `f` and an integer `x`. It should return another function which takes in one argument, another integer. This function returns the result of applying `f` to `x` this number of times.

Make sure to use recursion in your solution.

```python
def make_func_repeater(f, x):
    """
    >>> incr_1 = make_func_repeater(lambda x: x + 1, 1)
    >>> incr_1(2) #same as f(f(x))
    3
    >>> incr_1(5)
    6
    """

    def repeat(___________________):
        if _____________________:
            return __________________

        else:
            return __________________

    return _____________________
```
Consider a function that requires more than one recursive call. A simple example is the recursive fibonacci function:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

This type of recursion is called tree recursion, because it makes more than one recursive call in its recursive case. If we draw out the recursive calls, we see the recursive calls in the shape of an upside-down tree:

```
fib(4)
  /    /
fib(3)  fib(2)
  /    /    /
fib(2) fib(1) fib(1) fib(0)
```

We could, in theory, use loops to write the same procedure. However, problems that are naturally solved using tree recursive procedures are generally difficult to write iteratively. It is sometimes the case that a tree recursive problem also involves iteration: for example, you might use a while loop to add together multiple recursive calls.

As a general rule of thumb, whenever you need to try multiple possibilities at the same time, you should consider using tree recursion.
Questions

2.1 You want to go up a flight of stairs that has \( n \) steps. You can either take 1 or 2 steps each time. How many different ways can you go up this flight of stairs? Write a function `count_stair_ways` that solves this problem. Assume \( n \) is positive.

Before we start, what’s the base case for this question? What is the simplest input?

What do `count_stair_ways(n - 1)` and `count_stair_ways(n - 2)` represent?

Use those two recursive calls to write the recursive case:

```python
def count_stair_ways(n):
```
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.

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