Recursion

A *recursive* function is a function that is defined in terms of itself.

Consider this recursive `factorial` function:

```
def factorial(n):
    """Return the factorial of N, a positive integer."""
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Inside of the body of `factorial`, we are able to call `factorial` itself, since the function body is not evaluated until the function is called.

When `n` is 1, we can directly return the factorial of 1, which is 1. This is known as the *base case* of this recursive function, which is the case where we can return from the function call directly without having to first recurse (i.e. call `factorial`) and then returning. The base case is what prevents `factorial` from recursing infinitely.

Since we know that our base case `factorial(1)` will return, we can compute `factorial(2)` in terms of `factorial (1)`, then `factorial(3)` in terms of `factorial(2)`, and so on.

There are three main steps in a recursive definition:

1. **Base case.** You can think of the base case as the case of the simplest function input, or as the stopping condition for the recursion.

   In our example, `factorial(1)` is our base case for the `factorial` function.

2. **Recursive call on a smaller problem.** You can think of this step as calling the function on a smaller problem that our current problem depends on. We assume that a recursive call on this smaller problem will give us the expected result; we call this idea the "recursive leap of faith".

   In our example, `factorial(n)` depends on the smaller problem of `factorial(n-1)`.

3. **Solve the larger problem.** In step 2, we found the result of a smaller problem. We want to now use that result to figure out what the result of our current problem should be, which is what we want to return from our current function call.

   In our example, we can compute `factorial(n)` by multiplying the result of our smaller problem `factorial( n-1)` (which represents `(n-1)!`) by `n` (the reasoning being that `n! = n * (n-1)!`).

Q1: Warm Up: Recursive Multiplication

These exercises are meant to help refresh your memory of the topics covered in lecture.

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)`

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?

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

# You can use more space on the back if you want
```

Q2: Recursion Environment Diagram

Draw an environment diagram for the following code:

> *Note*: If you can't move elements around, make sure you're logged in!

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

rec(3, 2)
```

> *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 for large or complicated inputs, since the large number of frames can be quite unwieldy and confusing. Instead, think in terms of the three steps: base case, recursive call, and solving the full problem.

Q3: Find the Bug

Find the bug in this recursive function.

```
def skip_mul(n):
    """Return the product of n * (n - 2) * (n - 4) * ...

    >>> skip_mul(5) # 5 * 3 * 1
    15
    >>> skip_mul(8) # 8 * 6 * 4 * 2
    384
    """
    if n == 2:
        return 2
    else:
        return n * skip_mul(n - 2)
```

Fix #1:
add another base case:
if n == 1:
return 1

Fix #2:
if n <= 2:
return n

Fix #3:
if n <= 1:
return 1

Q4: Is Prime

Write a function `is_prime` that takes a single argument `n` and returns `True` if `n` is a prime number and `False` otherwise. Assume `n > 1`. We implemented this in Discussion 1 iteratively, now time to do it recursively!

> *Hint*: You will need a helper function! Remember helper functions are nested functions that are useful if you need to keep track of more variables than the given parameters, or if you need to change the value of the input.

```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    "*** YOUR CODE HERE ***"















# You can use more space on the back if you want
```

Q5: Recursive Hailstone

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
    >>> b = hailstone(1)
    1
    >>> b
    1
    """
    "*** YOUR CODE HERE ***"

# You can use more space on the back if you want
```

printed

return value

base case: when does the sequence end?

recursive case: what's the next item in the sequence?

hailstone (x)

1. prints out the sequence starting at x
2. returns length of sequence

hailstone (4) — prints 4 2 1, returns 3

next recursive call: hailstone (2) → prints 2 1, returns 2

+ 1

```
if n == 1:
    print(n)
    return 1      # only 1 item
else:
    if n % 2 == 0:
        next_n = n // 2
    else:
        next_n = n * 3 + 1
    print(n)
    return hailstone(next_n) + 1
```

```
if n % 2 == 0:
    print(n)
    return hailstone(n//2) + 1
```

Q6: Merge Numbers

Write a procedure `merge(n1, n2)`, which takes numbers with digits in decreasing order and returns a single number with all of the digits of the two in decreasing order. Any number merged with 0 will be that number (treat 0 as having no digits). Use recursion.

> *Hint*: If you can figure out which number has the smallest digit out of both, then we know that the resulting number will have that smallest digit, followed by the merge of the two numbers with the smallest digit removed.

```
def merge(n1, n2):
    """Merges two numbers by digit in decreasing order.
    >>> merge(31, 42)
    4321
    >>> merge(21, 0)
    21
    >>> merge (21, 31)
    3211
    """
    "*** YOUR CODE HERE ***"















# You can use more space on the back if you want
```