# Discussion 3: Recursion, Tree Recursion

This is an online worksheet that you can work on during discussions and tutorials. Your work is not graded and you do not need to submit anything.

# Recursion

A *recursive* function is a function that is defined in terms of itself. Consider this recursive `factorial` function:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

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. 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 $n$.

Another way to understand recursion is by separating 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.

Consider this alternative recursive `factorial`:

```
def factorial(n): # WRONG!
    if n == 2:
        return n
    return n * factorial(n-1)
```

It is internally correct, since $2! = 2$ and $n! = n * (n - 1)!$ are both true statements.

However, that `factorial` does not halt on all inputs, since `factorial(1)` results in a call to `factorial(0)`, and then to `factorial(-1)` and so on.

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.

## Q1: Recursion Environment Diagram

Draw an environment diagram for the following code:

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

rec(3, 2)
```

Global frame

Objects

f1: [parent= ]

Return value

f2: [parent= ]

Return value

f3: [parent= ]

Return value

Imagine you were writing the documentation for this function. Come up with a line that describes what the function does:

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 bes quite unwieldy and confusing. Instead, think in terms of the three step process - base case, recursive call, solving the full problem.

## Q2: 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.

```
    >>> merge(21, 31)
    3211
    """
    "*** YOUR CODE HERE ***"

    if n1 == 0 and n2:
        return n2
    elif n2 == 0 and n1:
        return n1
    elif n1 % 10 <= n2 % 10:
        return merge(n1 // 10, n2) * 10 + n1 % 10
    else:
        return merge(n1, n2 // 10) * 10 + n2 % 10
```

## Q3: 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 (/~cs61a/sp21/disc/disc01/) iteratively, now time to do it recursively!

> *Hint*: You will need a helper function! Remember helper functions 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.

```
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    "*** YOUR CODE HERE ***"

    for i in range(2,n):
        if n % i == 0:
            return False
    return True

```

## Q4: (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)`.

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?

> **Challenge:** Try to implement the multiply function tail recursively.

```
            return 0
        elif m == 1:
            return n
        elif n == 1:
            return m
        elif m >= n:
            return multiply(m, n - 1) + m
        else:
            return multiply(m - 1, n) + n
```

## Q5: (Tutorial) 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.

```
    2
    1
    >>> a
    7
    """
    "*** YOUR CODE HERE ***"
    if n == 1:
        print(n)
        return 1
    elif n % 2 == 0:
        print(n)
        return hailstone(n // 2) +1
    else:
        print(n)
        return hailstone(n * 3 + 1)+1

```

# Tree Recursion

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

```
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:

# Visualize a recursive function

**Try one of these functions:** Choose one...

**Or paste the function definition here (starting with `def` ):**

**Type your function call here:**

func(1, 2)

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.

## Q6: Count Stair Ways

Imagine that you want to go up a flight of stairs that has `n` steps, where `n` is a positive integer. You can either take 1 or 2 steps each time. In this question, you'll write a function `count_stair_ways` that solves this problem. Before you code your approach, consider these questions.

How many different ways can you go up this flight of stairs?

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?

Fill in the code for `count_stair_ways`:

```
def count_stair_ways(n):
    """Returns the number of ways to climb up a flight of
    n stairs, moving either 1 step or 2 steps at a time.
    >>> count_stair_ways(4)
    5
    """
    "*** YOUR CODE HERE ***"

    if n == 1:
```

```
return 1
```

## Q7: (Tutorial) Count K

Consider a special version of the `count_stair_ways` 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.

```
def count_k(n, k):
    """Counts the number of paths up a flight of n stairs
    when taking up to and including k steps at a time.
    >>> 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
    """
    "*** YOUR CODE HERE ***"

    if n == 0 or n == 1:
```