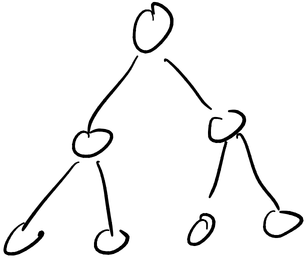
Recursion

- Recursion is a programming technique involving a function (or procedure) calling itself (a recursive call)
- Each recursive call solves a smaller instance of the problem
- The smallest instance must be solvable without a recursive call (the base case)
- Common uses
  - Divide and conquer
  - Iterating over trees
  - Implementing recurrences in code

    (though this can be less efficient than looping in many cases)



Recurrence

- A description of a function in terms of its value on smaller inputs

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = \prod_{x=1}^{n} x$$

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

recurrence

```
def factorial(n):
    product = 1
    for x in range(1, n+1):
        product *= x
    return product
```

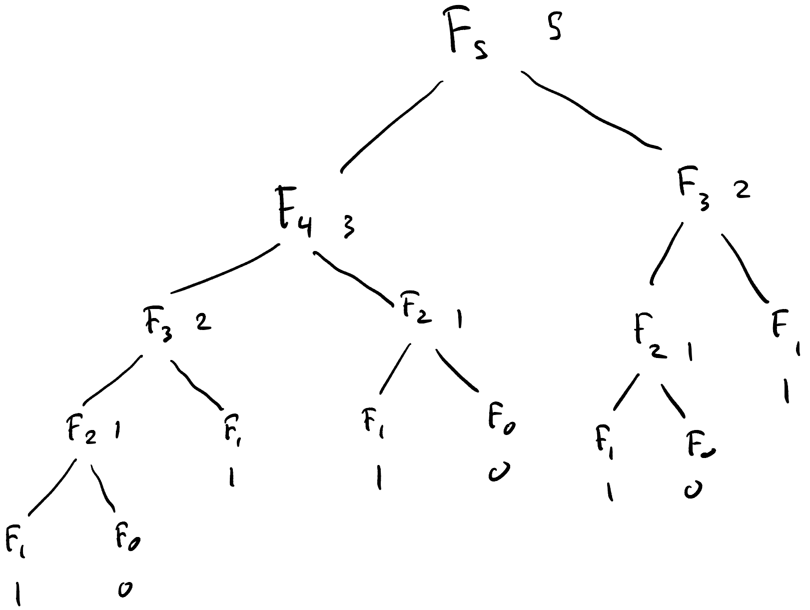
```
def recursive_factorial(n):
    if n == 1:
        return 1
    else:
        return n * recursive_factorial(n-1)
```

Recursion vs Iteration

- What can be done with recursion can also be done with a loop
- Loops can sometimes perform better
    - Avoids the overhead of function calls
    - If the recursion will be deep there is a risk of overflowing the call stack
- Loops might be more difficult to implement
    - Recursion keeps track of data on the call stack
    - Looping might require using your own stack
- Functional programming languages encourage recursion, and can optimize so that it actually uses a loop behind the scenes in the compiled version

Fibonacci Numbers

F_n is the Fth Fibonacci number



$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

Recursive fibonacci is not feasible with doing some extra book keeping