Recursion

- **Recursive Definition:**
  A definition that is defined in terms of itself

- **Recursive Function:**
  A function that calls itself (directly or indirectly)

- **Recursion:**
  If you get the point, stop; otherwise, see Recursion

- **Infinite Recursion:**
  See Infinite Recursion

---

**A Mathematical Example: Factorial**

- Non-recursive definition:
  \[ n! = n \times (n-1) \times \ldots \times 2 \times 1 \]

- Recursive definition:
  \[ n! = n \times (n-1)! \quad \text{for } n \geq 0 \]

  *Recursive case*

  *Base case*

  What happens if there is no base case?

---

**Example: Fibonacci Sequence**

- Sequence of numbers: 1, 1, 2, 3, 5, 8, 13, ...
  \[ a_0, a_1, a_2, a_3, a_4, a_5, a_6 \]
  * Get the next number by adding previous two
  * What is \( a_n \)?

- Recursive definition:
  \[ a_n = a_{n-1} + a_{n-2} \quad \text{Recursive Case} \]
  \[ a_0 = 1 \quad \text{Base Case} \]
  \[ a_1 = 1 \quad \text{(another) Base Case} \]

  Why did we need two base cases this time?

---

**Fibonacci as a Recursive Function**

- Function that calls itself
  * Each call is new frame
  * Frames require memory
  * \( \infty \) calls = \( \infty \) memory

---

**Fibonacci: # of Frames vs. # of Calls**

- Fibonacci is very inefficient.
  * \( \text{fib}(n) \) has a stack that is always \( \leq n \)
  * But \( \text{fib}(n) \) makes a lot of redundant calls

---

**String: Two Recursive Examples**

- `length(s)`
  **Returns:** # characters in `s`
  * If `s` is empty
    \[ \text{return 0} \]
  * Else
    \[ \text{return 1 + length(s[1:])} \]

- `num_es(s)`
  **Returns:** # of `'e'`s in `s`
  * If `s` is empty
    \[ \text{return 0} \]
  * Else
    \[ \text{return 1 + num_es(s[1:])} \]

Imagine `len(s)` does not exist
How to Think About Recursive Functions

1. Have a precise function specification.
2. Base case(s):
   - When the parameter values are as small as possible
   - When the answer is determined with little calculation.
3. Recursive case(s):
   - Recursive calls are used.
   - Verify recursive cases with the specification
4. Termination:
   - Arguments of calls must somehow get "smaller"
   - Each recursive call must get closer to a base case

Understanding the String Example

```
def num_es(s):
    """Returns: # of 'e's in s""
    # {s is empty}
    if s == '':
        return 0
    # { s at least one char }
    return (1 if s[0] == 'e' else 0) + num_es(s[1:])
```

• Break problem into parts
  - number of e's in s = number of e's in s[0] + number of e's in s[1:]
• Solve small part directly
  - number of e's in s = (1 if s[0] = 'e' else 0) + number of e's in s[1:]

Exercise: Remove Blanks from a String

```
def deblank(s):
    """Returns: s but with its blanks removed""
    if s == '':
        return s
    if s[0] is a blank:
        return s[1:] with blanks removed
    return (s[0] with blanks removed) + deblank(s[1:]
```

• Sometimes easier to break up the recursive case
  - Particularly on small part
• Write recursive case as a sequence of if-statements
• Write code in pseudocode
  - Mixture of English and code
  - Similar to top-down design
• Stuff in red looks like the function specification!
  - But on a smaller string
  - Replace with deblank(s[1:])