A Mathematical Example: Factorial

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

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

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, …

- 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

def fibonacci(n):
    """Returns: Fibonacci no. \( a_n \) \nPrecondition: \( n \geq 0 \) an int""
    if n <= 1:
        return 1
    return (fibonacci(n-1) +
            fibonacci(n-2))

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

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 cases:
   - 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_ewe(s):
    """Returns: # of 'e's in s""
    # s is empty
    if s == "":
        return 0
    # s has at least one 'w'
    if s[0] == 'w':
        return 1 + num_ewe(s[1:])
    return num_ewe(s[1:]))

- Break problem into parts
- Solve small part directly
Understanding the String Example

- **Step 1:** Have a precise specification
  ```python
def num_es(s):
    """Returns: # of 'e's in s""
    # s is empty
    if s == '':
        return 0
    # return # of 'e's in s[0]+# of 'e's in s[1:]
    if s[0] == 'e':
        return 1+num_es(s[1:])
    return num_es(s[1:])
```

- **Step 2:** Check the base case
  * When s is the empty string, 0 is (correctly) returned.

Exercise: Remove Blanks from a String

1. **Have a precise specification**
   ```python
def deblank(s):
    """Returns: s with blanks removed"
    # s is empty
    if s == '':
        return s
    # s not empty and s[0] not blank
    return (s[0] + s[1:]).strip()
    ```

2. **Base Case:** the smallest String s is "."
   ```python
   if s == ".":
       return s
   ```

3. **Other Cases:** String s has at least 1 character.
   ```python
   return (s[0] + s[1:]).strip()
   ```

Example: Reversing a String

- **Precise Specification:**
  - Returns: reverse of s

- **Solving with recursion**
  - Suppose we can reverse a smaller string (e.g. less one character)
  - Can we use that solution to reverse whole string?
  - Often easy to understand first without Python
  ```python
def reverse(s):
    """Returns: reverse of s"
    if s == '':
       return s
    # s has at least 1 character
    # (reverse of s[1:]) + s[0]
    return reverse(s[1:]) + s[0]
```

Example: Reversing a String

- **Precise Specification:**
  - Returns: reverse of s

- **Solving with recursion**
  - Suppose we can reverse a smaller string (e.g. less one character)
  ```python
def reverse(s):
    """Returns: reverse of s"
    if s == '':
        return s
    # s has at least one character
    # (reverse of s[1:]) + s[0]
    return reverse(s[1:]) + s[0]
```