A Mathematical Example: Factorial

- Non-recursive definition:

  n! = n × n-1 × … × 2 × 1

  = n (n-1 × … × 2 × 1)

- Recursive definition:

  n! = n (n-1)!   for n ≥ 0   **Recursive case**

  0! = 1   **Base case**

What happens if there is no base case?

Example: Fibonnaci 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

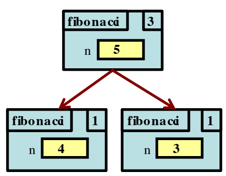
  - Get the next number by adding previous two
  - What is a_8?
- Recursive definition:
  - $a_n = a_{n-1} + a_{n-2}$ **Recursive Case**
  - $a_0 = 1$ **Base Case**
  - $a_1 = 1$ **(another) Base Case**

Why did we need two base cases this time?

Fibonacci as a Recursive Function

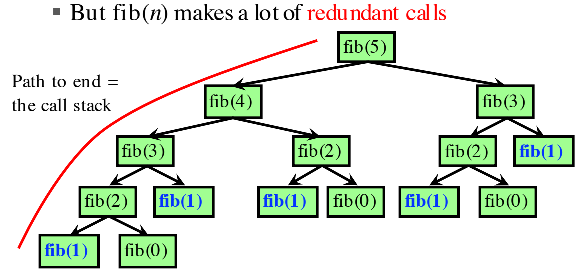
```
def fibonacci(n):
    """Returns: Fibonacci no. an
    Precondition: n ≥ 0 an int"""
    if n <= 1:
        return 1

    return (fibonacci(n-1)+
            fibonacci(n-2))
```

- Function that calls itself
  - Each call is new frame
  - Frames require memory
  - ∞ calls = ∞ memory



Fibonacci: # of Frames vs. # of Calls

- Fibonacci is very inefficient.
  - fib(n) has a stack that is always ≤ n
  - But fib(n) makes a lot of redundant calls

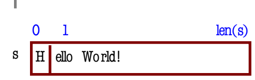


How to Think About Recursive Functions

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 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 == '':              Base case
        return 0

    # s has at least one 'e'
    if s[0] == 'e':          Recursive case
        return 1+num_es(s[1:])

    return 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 =
  number of e's in s[1:]
  (+1 if s[0] is an 'e')
  (+0 is s[0] not an 'e')

Understanding the String Example

- **Step 1:** Have a precise specification

```
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:]))
```

Base case

Recursive case

"Write" your return statement using the **specification**

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

Understanding the String Example

- **Step 3:** Recursive calls make progress toward termination

```
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:]))
```

parameter s

argument s[1:]

argument s[1:] is smaller than parameter s, so there is progress toward reaching base case 0

- **Step 4:** Check the recursive case
  - Does it match the specification?

Exercise: Remove Blanks from a String

1. Have a precise specification

```
def deblank(s):
    """Returns: s but with its blanks removed"""
```

2. **Base Case:** the smallest String s is ''.

```
if s == '':
    return s
```

3. **Other Cases:** String s has at least 1 character.

```
return (s[0] with blanks removed) + (s[1:] with blanks removed)
```

Exercise: Remove Blanks from a String

```
def deblank(s):
    """Returns: s with blanks removed"""
    if s == '':
        return s

    # s is not empty
    if s[0] is a blank:
        return s[1:] with blanks removed

    # s not empty and s[0] not blank
    return (s[0] +
            s[1:] with blanks removed)
```

- 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:])

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
  - Then sit down and code

H	e	l	l	o	!

!	o	l	l	e	H

H	e	l	l	o	!

!	o	l	l	e

Example: Reversing a String

```
def reverse(s):
    """Returns: reverse of s

    Precondition: s a string"""
    # s is empty
    if s == '':
        return s

    # s has at least one char
    # (reverse of s[1:])+s[0]
    return reverse(s[1:])+s[0]
```

H	e	l	l	o	!

!	o	l	l	e

✔ 1. Precise specification?
✔ 2. Base case: correct?
✔ 3. Recursive case: progress to termination?
✔ 4. Recursive case: correct?