

A Mathematical Example: Factorial

- Non-recursive definition:

$$n! = n \times n-1 \times \dots \times 2 \times 1$$

$$= n (n-1 \times \dots \times 2 \times 1)$$
- Recursive definition:

$$n! = n (n-1)! \quad \text{for } n \geq 0 \quad \text{Recursive case}$$

$$0! = 1 \quad \text{Base case}$$

What happens if there is no base case?

1

Factorial as a Recursive Function

```
def factorial(n):
    """Returns: factorial of n.
    Pre: n ≥ 0 an int"""
    if n == 0:
        return 1
    return n*factorial(n-1)
```

- $n! = n (n-1)!$
- $0! = 1$

Base case(s)

Recursive case

What happens if there is no base case?

2

Example: Fibonacci Sequence

- Sequence of numbers: 1, 1, 2, 3, 5, 8, 13, ...

$a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6$

 - 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?

3

Fibonacci as a Recursive Function

```
def fibonacci(n):
    """Returns: Fibonacci no. a_n
    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

4

Fibonacci: # of Frames vs. # of Calls

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

Path to end = the call stack

5

Recursion is best for Divide and Conquer

Goal: Solve problem P on a piece of data

data

data 1

data 2

Solve Problem P

Solve Problem P

Combine Answer!

Idea: Split data into two parts and solve problem

6

Divide and Conquer Example

Count the number of 'e's in a string:

p	e	n	n	e
---	---	---	---	---

Two 'e's

p	e
---	---

+

n	n	e
---	---	---

One 'e'

One 'e'

7

Three Steps for Divide and Conquer

1. **Decide what to do on "small" data**
 - Some data cannot be broken up
 - Have to compute this answer directly
2. **Decide how to break up your data**
 - Both "halves" should be smaller than whole
 - Often no wrong way to do this (next lecture)
3. **Decide how to combine your answers**
 - Assume the smaller answers are correct
 - Combining them should give bigger answer

8

Divide and Conquer Example

```
def num_es(s):
    """Returns: # of 'e's in s"""
    # 1. Handle small data
    if s == '':
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0

    # 2. Break into two parts
    left = num_es(s[0])
    right = num_es(s[1:])

    # 3. Combine the result
    return left+right
```

"Short-cut" for

```
if s[0] == 'e':
    return 1
else:
    return 0
```

s[0]	s[1:]
p	e n n e
0	+ 2

9

Exercise: Remove Blanks from a String

```
def deblank(s):
    """Returns: s w/o blanks"""
    if s == '':
        return s
    elif len(s) == 1:
        return s if s[0] != ' ' else s

    left = deblank(s[0])
    right = deblank(s[1:])

    return left+right
```

Handle small data

Break up the data

Combine answers

10

Minor Optimization

```
def deblank(s):
    """Returns: s w/o blanks"""
    if s == '':
        return s

    left = s[0]
    if s[0] == ' ':
        left = ""
    right = deblank(s[1:])

    return left+right
```

Eliminate the second base by combining

Less recursive calls

11

Following the Recursion

deblank

a	b	c
---	---	---

→

a	b	c
---	---	---

✗
deblank

a	b	c
---	---	---

→

a	b	c
---	---	---

a
deblank

b	c
---	---

→

a	b	c
---	---	---

✗
deblank

b	c
---	---

→

b	c
---	---

b
deblank

c

→

b	c
---	---

✗
deblank

c

→

c

c
deblank

c

→

c

12