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

What happens if there is no base case?

**Factorial as a Recursive Function**

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

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

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 \(a_0\)?

- 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**

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

Path to end = the call stack

**Recursion is best for Divide and Conquer**

Goal: Solve problem P on a piece of data

Idea: Split data into two parts and solve problem

Combine Answer!
Divide and Conquer Example

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

\[
\begin{align*}
\text{p e n n e} & \quad \text{Two 'e's} \\
\text{p e} & \quad \text{One 'e'} \\
\text{n n e} & \quad \text{One 'e'}
\end{align*}
\]

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

Divide and Conquer Example

```python
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:
                return 0
    # 2. Break into two parts
    left = num_es(s[0])
    right = num_es(s[1:])
    # 3. Combine the result
    return left + right
```

Exercise: Remove Blanks from a String

```python
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
```

Minor Optimization

```python
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
```

Following the Recursion