Recursion: Basis and Progress

A recursive method may call itself on a smaller problem.

Always divided into basis and progress:
- **Basis:** Solve a small problem directly
- **Progress:** Break off a smaller problem to solve with a recursive call

Factorial

\[ n! = 1 \times 2 \times 3 \times 4 \times \ldots \times n \]

**Recursive definition:**
- \( n! = 1 \) if \( n < 2 \)
- \( n! = n \times (n-1)! \) if \( n \geq 2 \)

From definition to function ...

**Recursive definition:**
\[
\begin{align*}
n! &= 1 \quad \text{if } n < 2 \\
n! &= n \times (n-1)! \quad \text{if } n \geq 2
\end{align*}
\]

**Python code:**

```python
def fact(n):
    if n < 2:
        return 1
    else:
        return n * fact(n - 1)
```
fact(5):
return 5 * fact(5 - 1)

fact(4):
return 4 * fact(4 - 1)

fact(3):
return 3 * fact(3 - 1)

fact(2):
return 2 * fact(2 - 1)

fact(1):
return 1

fact(5):
return 5 * fact(5 - 1)

fact(4):
return 4 * fact(4 - 1)

fact(3):
return 3 * fact(3 - 1)

fact(2):
return 2 * fact(2 - 1)

fact(1):
return 1
Recursion vs Loop

def fact(n):
    if n < 2:
        return 1
    else:
        return n * fact(n - 1)
def factLP(n):
    prod = 1
    for i in range(2, n+1):
        prod = prod * i
    return prod

Recall Inductive Loop Design

while (not basis case):
    make the problem a little smaller

solve the basis case

Almost the same

def foo(problem):
    if (basis case):
        return the solution
    else:
        foo (smaller problem)

Facts for computing gcd(a,b)

gcd(n,0) = n
    because n × 0 = 0, for all n

gcd(a,b) = gcd(b,a)
gcd(a, b) = gcd(b, a mod b) if a > b
    progress case for loop or recursion
Recursive Function

def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a % b)

Choose the loop

def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a % b)

def gcdLp(a, b):
    while b > 0:
        tmp = b
        b = a % b
        a = tmp
    return a

Recall the Palindrome Test

The same forward as backward
We defined basis and progress cases for a loop

Same basis and progress work for recursive solution

Base cases: A palindrome

```
LEVEL
```

Base case: Not a palindrome

```
LEVEL
```
Progress: So far so good

Let’s write it …

def palindrome(s):

A solution (not the only one)

def palindrome(s):
    if len(s) < 2:
        return True
    if s[0] != s[-1]:
        return False
    return palindrome(s[1:-1])

When is recursion needed?

A loop can always be directly expressed as recursion
Recursion can often be directly* expressed as a loop … but not always

(*There is always an indirect expression using a stack data structure)

Progress case is one sub-problem, or more?
One? Use a loop
More? Recursion will be simpler
Depth First Search

A classic algorithm, natural for recursion:

if (basis case) :
    return the answer

while (some progress case possible) :
    if (any progress case succeeds): return true

return false;