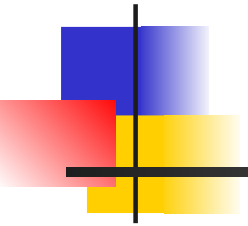


Recursion

James Brucker

What is Recursion?

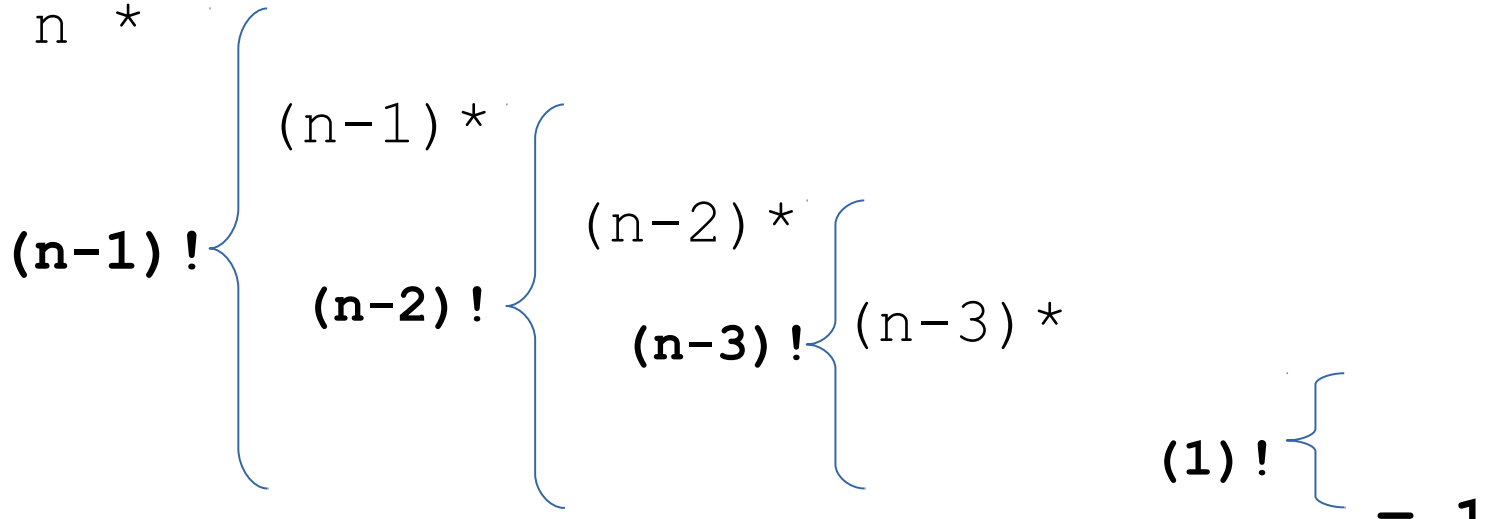
Recursion means for a function or method to call itself.

A typical example is computing factorials:

```
n! = n * (n-1)!
```

n factorial is (recursively) defined using n-1 factorial.

n! =



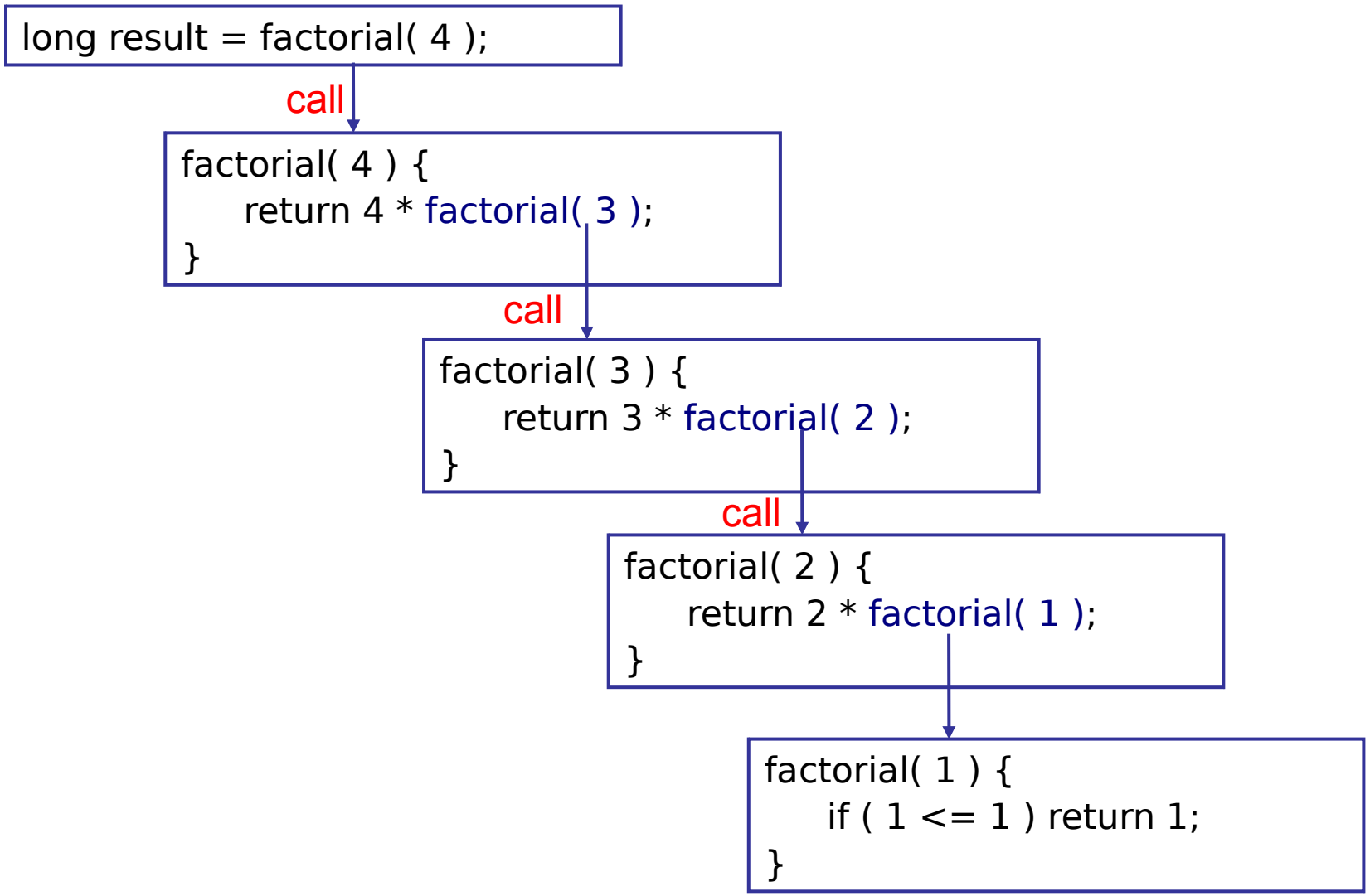
Recursive factorial(n)

We can write a function that computes factorials by calling itself to compute factorial of a smaller number:

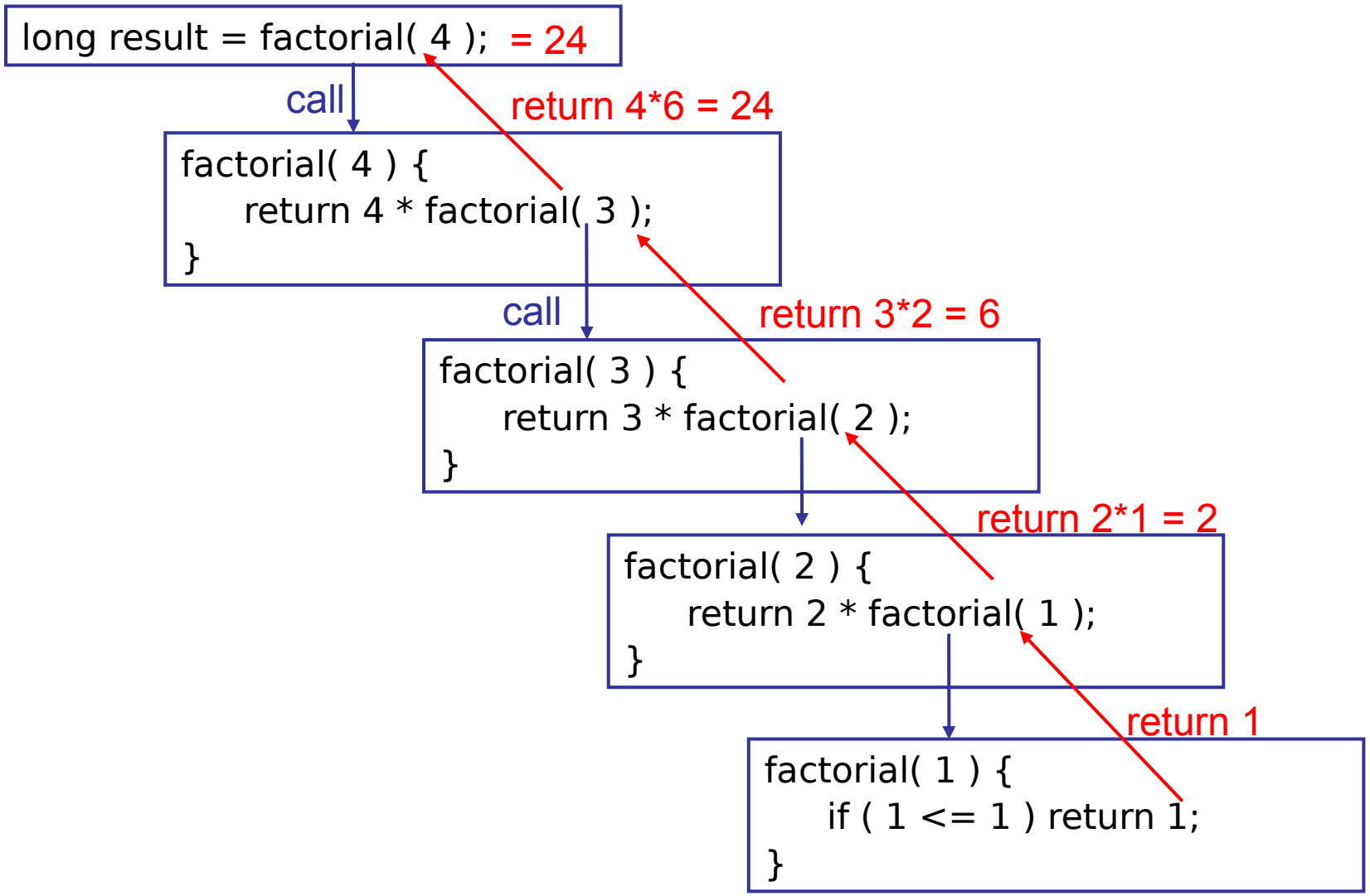
```
def factorial(n: int):
    if n <= 1:
        return 1
    return n * factorial(n-1)
```

Suppose we call this function to compute factorial(4). What statements will be executed?

factorial(n) execution trace



factorial(n) return trace



Recursion Must Eventually Stop

Recursion must guarantee to stop eventually (no infinite calls)

Recursion should not change any state variable that other levels of recursion will use, except by design.

```
def factorial( n ):
    if n <= 1:   return 1
    return n * factorial(n-1)
```

This test (n <= 1) guarantees that factorial( ) will eventual stop using recursion.

Wrong:

```
def factorial( n ):
    if n == 1:   return 1
    return n * factorial(n-1)
```

What happens if factorial(0) is called?

Base Case

The case where recursion stops is called the **base case**.

factorial(n): base case is n == 1

but you should also test for n < 1

Recursive Sum

`long sum(int n)` - compute sum of 1 to n

Recursion: n + { sum of 1 to n-1 }

Code for recursive sum

Complete this code

```
def sum(n):
    """Return sum of 1 + ... + n."""
    # what is the base case?
    if ____________________:
    # what is the recursive step?
    return ____________________
```

Designing Recursion

1) Discover a **pattern** for recursion:
   - solve a small problem by hand
   - observe how you break down the problem
2) Determine the **base case** when recursion stops.
3) Termination criteria: what can you test to **guarantee** recursion will stop?
4) Construct an expression for the recursive step.

Designing Recursion Example

`sum(n) = 1 + 2 + 3 + ... + n`

1) Discover a pattern for recursion:
   - sum(n) = (1 + 2 + ...+ n-1) + n = sum(n-1) + n

2) **base case**: sum(n) = 0 for any n <= 0.
   Note: to *guarantee* recursion will always stop we need to consider case n < 0, too! Not just n == 0.
   If n < 0 either throw exception or return 0.

3) Guarantee Termination? Yes - each time we reduce the value of the parameter (n) by 1,
   so eventually we must have n <= 0.

Does Recursion Provide Insight?

For some problems, recursion makes the solution **easier** to **understand** and **implement**.

Recursion provides *insight* into the solution.

For other problems, it provides no insight.

Only use recursion when it makes the problem easier to understand or solve.

Famous examples: Quicksort algorithm. Knight's tour.

Does Recursive Sum Offer Insight?

```
sum(n) = n + sum(n-1)   if n > 0
```

My opinion: No.

1 + 2 + ... + n ==> looks like iteration (a loop)

Sum Elements in a List

What is wrong with this code?

```
def sum( lst ):
    if not lst:  # empty list
        return 0
    last_element = lst.pop()
    return last_element + sum(lst)
```

It computes the correct result, but is a poor design.

Helper Function

Sometimes you need a "helper function" with extra parameters in order to perform recursion.

To sum a list, without modifying the list, use a helper function that has a param for the last element to sum.

```
def sum(lst):  return sumTo(lst, len(lst)-1)
```

```
def sumTo(lst, last_index):
  """sum elements 0 up to last_index."""
  if last_index < 0:
      return 0      # nothing to sum
  return lst[last_index] \
                   + sumTo(lst, last_index-1)
```

Learn more about Helper Functions

Big Java, Chapter 13 (*Recursion*) has a section on helper methods.

Recursion uses more memory

- We can easily sum 1 to 10,000,000 using a loop.
  but recursive sum will fail with "out of memory" error.

- Why?
  - each function call creates a ***stack frame*** to store information about the invocation (parameters, local vars, saved register values) and a return value.
  - The stack frames consume memory.
  - Eventually, recursive calls may fill all the stack space.

- For the curious: read about "tail recursion"
  - avoids creating stack frames in special cases

Backtracking

- In some problems, an attempt to find a solution using recursion fails.
- You have to "undo" or "backtrack" some recursive steps and try a different solution.

References

Big Java, Chapter 13 *Recursion.*

Recursion in Python on RealPython.com

`https://realpython.com/python-recursion/`

`http://codingbat.com` - programming problems using recursion. Recursion-1 set is easy, Recursion-2 is more challenging & use backtracking. Only available for Java.