Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU22012: Data Structures and Algorithms II

Extra Lecture 1: Recursion

Dr Anthony Ventresque

Complexity

```
function argmax(array):
Input: an array of size n
Output: the index of the maximum value
index ← 0 #assignment, 1 op
foreach i in [1, n-1] do #1 op per loop
   if array[i] > array[index] then #3 ops per loop
       index ← i #1 op per loop, sometimes
   endif
endfor
return index #1 op
```

How many operations if the list has 10 elements? 10,000 elements?

- ***Varies proportional to the size of the input list***: $5n + 2$
- We'll be in the `for` loop longer and longer as the input list grows

Complexity

```
function argmax(array):
Input: an array
Output: the index of the maximum value
index ← 0 #assignment, 1 op
foreach i in [1,array.length[ do #1 op per loop
   if array[i] > array[index] then #3 ops per loop
       index ← i #1 op per loop, sometimes
   endif
endfor
return index #1 op
```

$T(n)=5n+2$

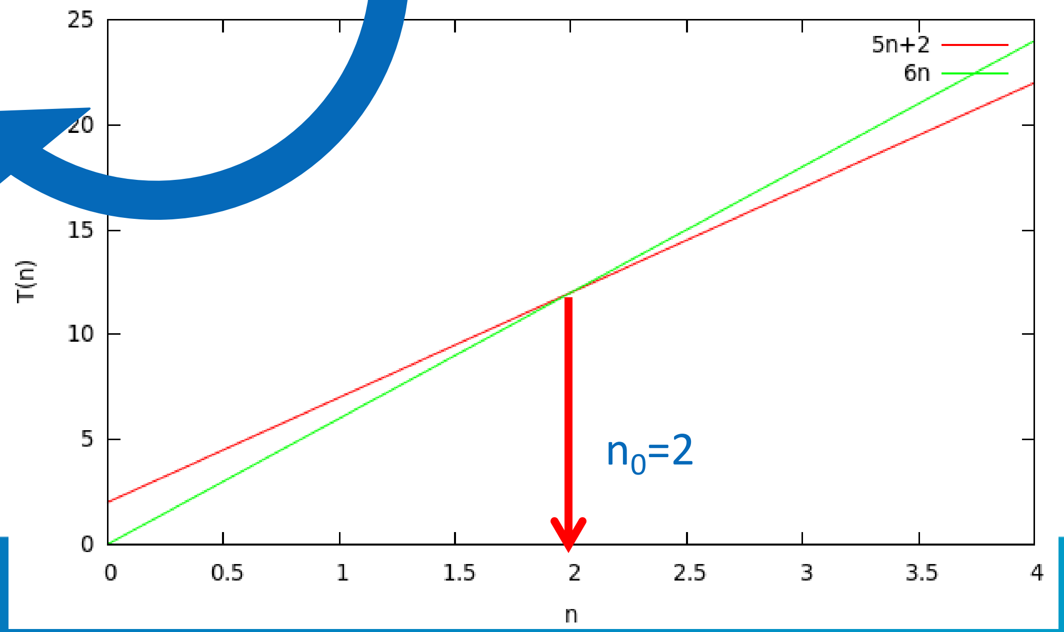
$T(n) \leq 6n$

$g(n)=n$

$c=6$

$n_0=2$

T(n) is O(n)



Outline

Recursion?
Base Case
Call Stack
Recursive vs. Iterative Functions
Tail Recursion
Turning a recursive algorithm into an iterative one
Complexity of recursive functions

Take home message:

Recursion is a method to divide a problem in similar sub-problems

— *Allows a function to call itself*

Recursion?

Recursion is a way of decomposing problems into smaller, simpler sub-tasks that are similar to the original.

Thus, each sub-task can be solved by applying a similar technique.

The whole problem is solved by combining the solutions to the smaller problems.

Requires a ***base case*** (a case simple enough to solve without recursion) to ***end recursion***.

Recursion Example

Factorial: n! = 1 x 2 x … x (n-1) x n
Or: ***n! = n x (n-1)!***, ***1!=1***

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
     return 1
else
     return n * factorial(n-1)
endif
```

Recursion Simulation

```
factorial(3)
        if 3=1 then
                return 1
        else
                return 3 * factorial(2)
        endif
```

```
factorial(2)
        if 2=1 then
                return 1
        else
                return 2 * factorial(1)
        endif
```

```
factorial(1)
        if 1=1 then
                return 1
        endif
```

Recursion Simulation

6

```
factorial(3)
        if 3=1 then
                return 1
        else
                return 3 * factorial(2)
        endif
```

2

```
factorial(2)
        if 2=1 then
                return 1
        else
                return 2 * factorial(1)
        endif
```

1

```
factorial(1)
        if 1=1 then
                return 1
        endif
```

Stopping Case/Base Case

As a recursive function calls itself, it is ***crucial to have a base case/stopping case*** – or the process will never stop!
The basic principle is to ***test first the stopping condition*** and then raise the recursive call if the condition is not met

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
      return 1
else
      return n * factorial(n-1)
endif
```

This is bad! Factorial without a Base Case

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
     return n * factorial(n-1)
endif
```

This is bad! Recursion Simulation: Factorial without a Base Case

```
factorial(3)
        return 3 * factorial(2)
```

```
factorial(2)
        return 2 * factorial(1)
```

```
factorial(1)
        return 1 * factorial(0)
```

```
factorial(0)
        return 0 * factorial(-1)
```

…

Call Stack

The basic idea behind recursion is that every call has a unique context (own memory address, own values for parameters and variables)

The ***call stack*** contains all this information and in the context of recursive function, this keeps track of the recursive calls

Recursion Simulation

Calculate 3 factorial
This is a call to factorial(3), so we put factorial(3) on the call stack

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
      return 1
else
      return n * factorial(n-1)
endif
```

Call Stack

Recursion Simulation

Calculate 3 factorial
This is a call to factorial(3), so we put factorial(3) on the call stack

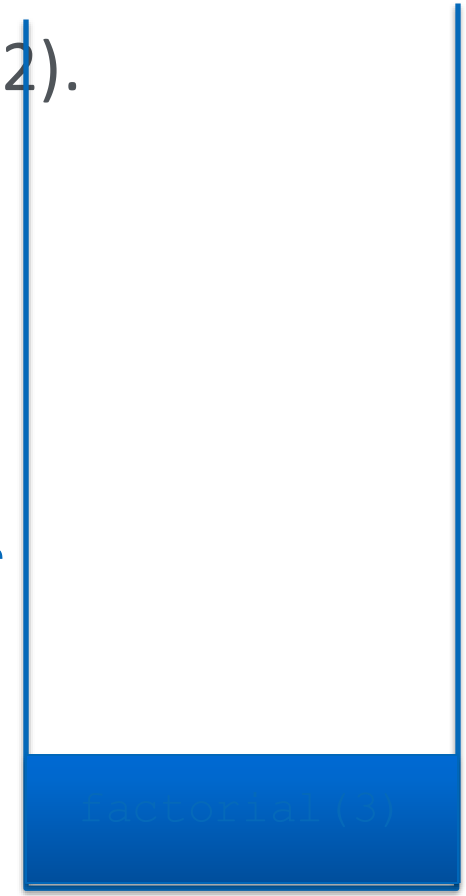
```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
      return 1
else
      return n * factorial(n-1)
endif
```

Call Stack

Recursion Simulation

n != 1, so we return
n*factorial(n-1), which includes a call to factorial(2).
Remember, the call to factorial(3)
has not returned yet, so it is still
on the call stack!

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
      return 1
else
      return n * factorial(n-1)
endif
```

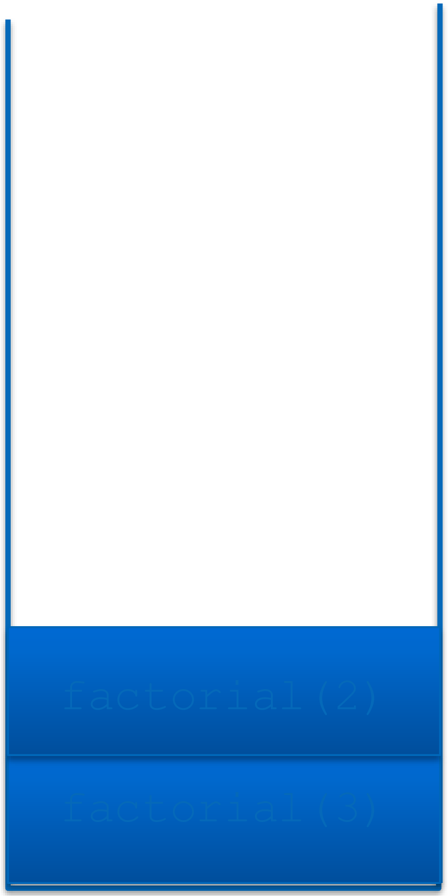


Call Stack

Recursion Simulation

n != 1, so we return n*factorial(n-1), which includes a call to factorial(2).
Remember, the call to factorial(3) has not returned yet, so it is still on the call stack!

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
      return 1
else
      return n * factorial(n-1)
endif
```

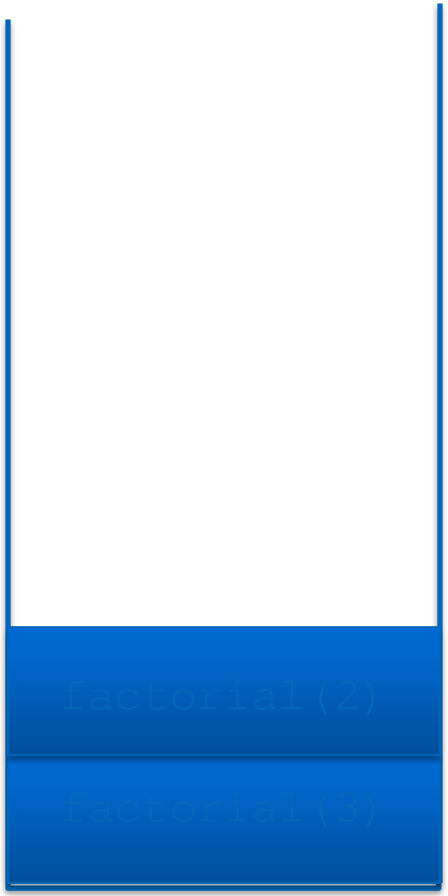


Call Stack

Recursion Simulation

n still != 1, so we return n*factorial(n-1), which includes a call to factorial(1).
Neither factorial(2) nor factorial(3) has returned at this point!

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
      return 1
else
      return n * factorial(n-1)
endif
```

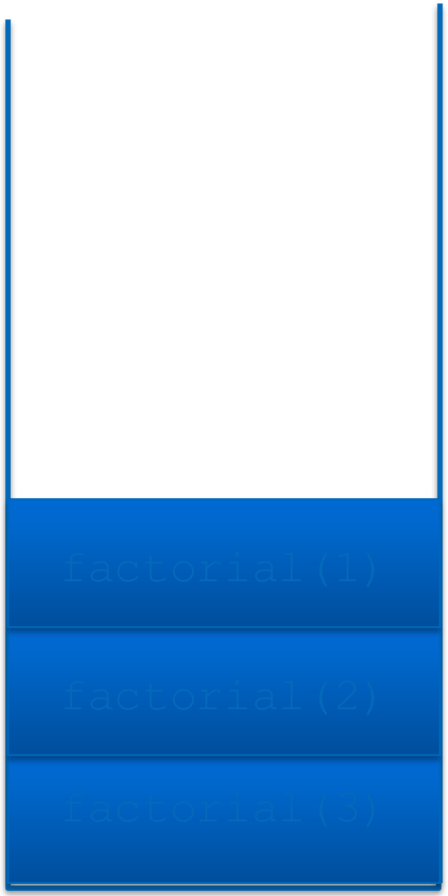


Call Stack

Recursion Simulation

n still != 1, so we return n*factorial(n-1), which includes a call to factorial(1).
Neither factorial(2) nor factorial(3) has returned at this point!

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
        return 1
else
        return n * factorial(n-1)
endif
```

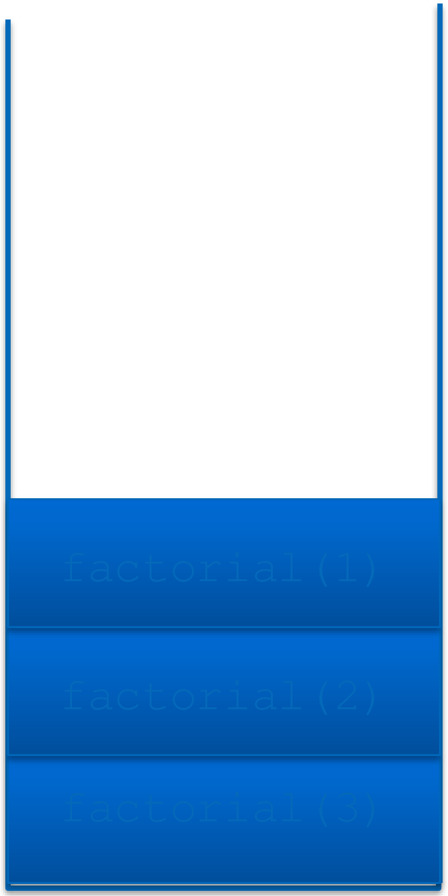


Call Stack

Recursion Simulation

Now n =1, so we return 1!
This is not a recursive call, so factorial(1) returns, and we take it off of the call stack.

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
      return 1
else
      return n * factorial(n-1)
endif
```

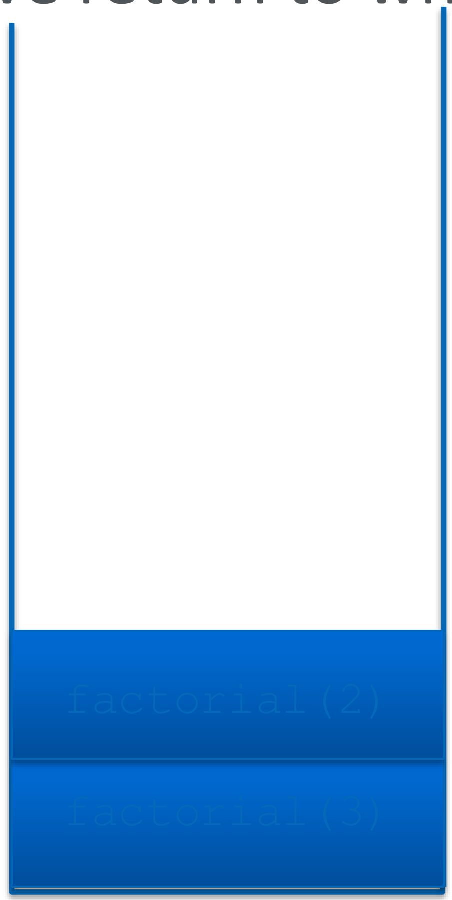


Call Stack

Recursion Simulation

Now factorial(2) is at the top of the call stack, so we return to where we were in factorial(2).
So we return 2*factorial(1), which we now know is 2*1, so factorial(2) returns 2 and is removed from the call stack!

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
        return 1
else
        return n * factorial(n-1)
endif
```



Call Stack

Recursion Simulation

Now factorial(2) is at the top of the call stack, so we return to where we were in factorial(2).
So we return 2*factorial(1), which we now know is 2*1, so factorial(2) returns 2 and is removed from the call stack!

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
        return 1
else
        return n * factorial(n-1)
endif
```

Call Stack

Recursion Simulation

Now factorial(3) is at the top of the call stack, so we're back in factorial(3). Return 3*factorial(2), which we now know is 3*2.

Factorial(3) returns 6 and removes itself from the call stack.

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
        return 1
else
        return n * factorial(n-1)
endif
```

Call Stack

Recursion Simulation

Now factorial(3) is at the top of the call stack, so we're back in factorial(3). Return 3*factorial(2), which we now know is 3*2.
Factorial(3) returns 6 and removes itself from the call stack.

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
        return 1
else
        return n * factorial(n-1)
endif
```

Call Stack

Recursion Simulation

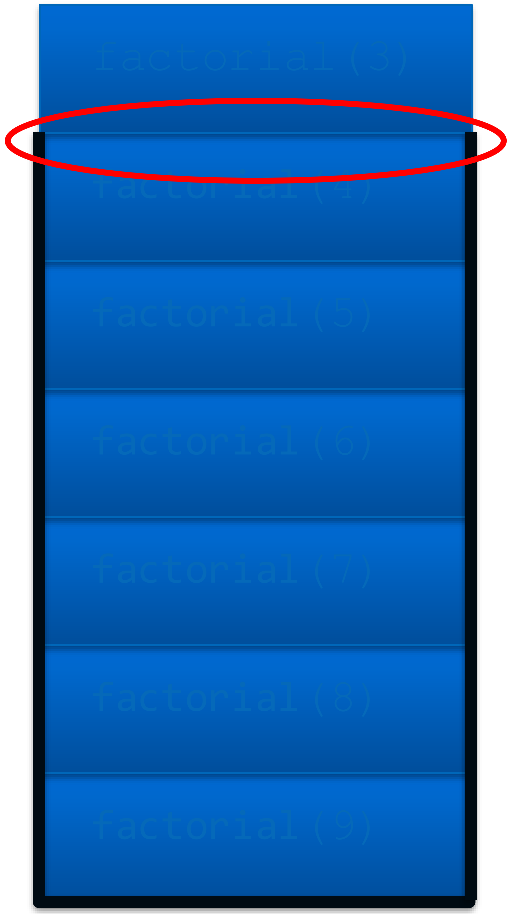
Now our call stack is empty, and we know that factorial(3) is 6!

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then
      return 1
else
      return n * factorial(n-1)
endif
```

Call Stack

Call Stack

It is difficult to predict the number of calls – and the system needs to do dynamic allocation
Which can be a problem: the famous ***stack overflow*** problem being always around the corner in case there are too many calls)
Stack overflow happens when the call stack reaches the stack bound



Call Stack

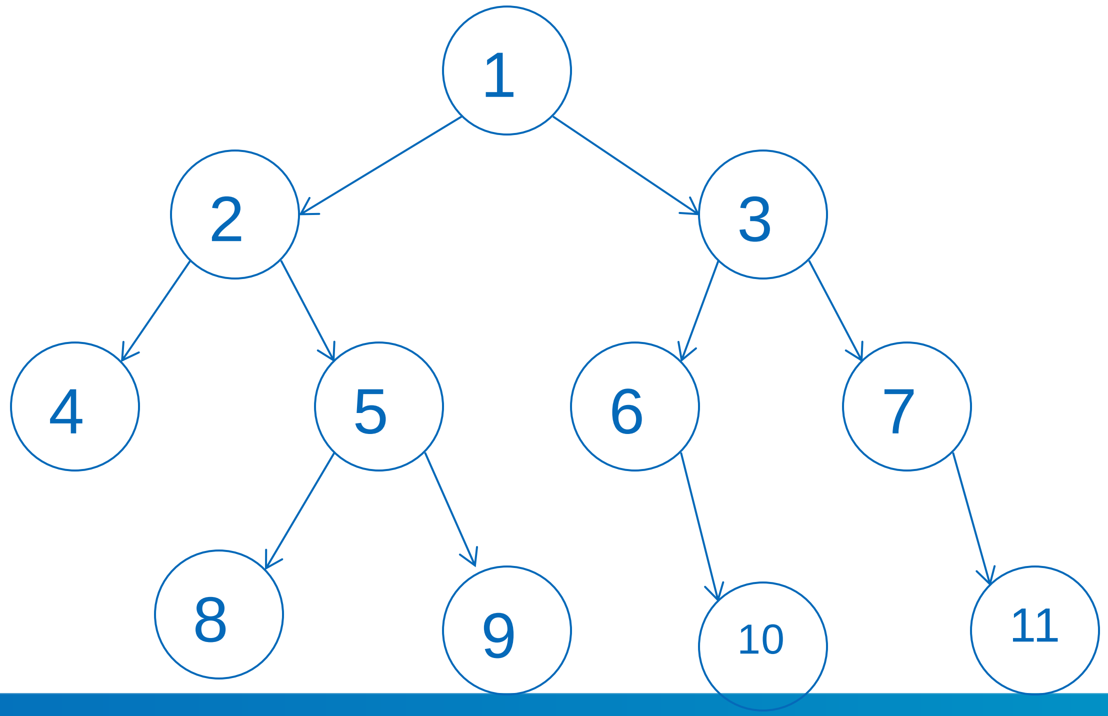
Recursive vs. Iterative Functions

It is often possible to write the same algorithm using recursive or iterative functions

```
function factorial_iterative(n)
Input: n a natural number
Output: the n-th factorial number
fact <- n
while n > 1 do
      n <- n-1
      fact <- fact * n
endwhile
return fact
```

Recursive vs. Iterative Functions

Some data structures are naturally recursive, i.e., it’s much easier to write recursive functions for them than iterative ones



Recursive vs. Iterative Functions

Recursive functions are sometimes slower (calls are expensive in practice)
(Bad) recursive algorithms can generate a large number of calls

Tail Recursion

A function call is said to be ***tail recursive*** if there is nothing to do after the function returns except return its value.
A function is non tail recursive if there is some processing done after the function returns.

Example of non Tail Recursion

Factorial: $n! = 1 \times 2 \times \ldots \times (n-1) \times n$
Or: $\boldsymbol{n! = n \times (n-1)!}$, $\boldsymbol{1!=1}$

```
function factorial_non_tail(n)
Input: n a natural number
Output: the n-th factorial number
if n == 1 then
     return 1
else
     return n * factorial_non_tail(n-1)
endif
```

Recursion Simulation

```
factorial_non_tail(3)
        if 3=1 then
                return 1
        else
                return 3 * factorial_non_tail (2)
        endif
```

```
factorial_non_tail(2)
        if 2=1 then
                return 1
        else
                return 2 * factorial_non_tail (1)
        endif
```

```
factorial_non_tail(1)
        if 1=1 then
                return 1
        endif
```

Recursion Simulation

6

```
factorial_non_tail(3)
        if 3=1 then
                return 1
        else
                return 3 * factorial_non_tail (2)
        endif
```

2

```
factorial_non_tail(2)
        if 2=1 then
                return 1
        else
                return 2 * factorial_non_tail (1)
        endif
```

1

```
factorial_non_tail(1)
        if 1=1 then
                return 1
        endif
```

Example of Tail Recursion

Factorial: n! = 1 x 2 x … x (n-1) x n
Or: ***n! = n x (n-1)!***, ***1!=1***

```
function factorial_tail(n, accumulator)
Input: n and accumulator, two natural numbers
Output: the n-th factorial number
if n = 1 then
     return accumulator
else
     return factorial_tail(n-1, n*accumulator)
endif
```

Recursion Simulation

```
factorial_tail(3,1)
        if 3=1 then
                return 1
        else
                return factorial_tail(2,3*1)
        endif
```

```
factorial_non_tail(2,3)
        if 2=1 then
                return 3
        else
                return factorial_tail(1,2*3)
        endif
```

```
factorial_non_tail(1,6)
        if 1=1 then
                return 6
        endif
```

Why Tail Recursion?

Tail recursion is usually more efficient (although more difficult to write) than non tail recursion

The recursive calls do not need to be added to the call stack: there is only one, the current call, in the stack

It is possible to turn tail recursions into iterative algorithms

Recursion -> Iterative Algorithm

The general form of every ***tail recursion*** is:

- ret is the returned type
- Para is a list of parameters
- cond is the base case
- state0, state1 and state2 are statements
- f is a function transforming the parameters

```
function recursion(Para)
Input: set of parameters Para
Output: ret
state0
if cond then
       state1
else
       state2
       recursion(f(Para))
endif
```

Recursion -> Iterative Algorithm

The ***iterative version of a tail recursion*** is:

- ret is the returned type
- Para is a list of parameters
- cond is the base case
- state0, state1 and state2 are statements
- f is a function transforming the parameters

```
function iterative(Para)
Input: set of parameters Para
Output: ret
state0
while non cond do
       state2
       Para ← f(Para)
       state0
endwhile
state1
endif
```

Factorial Recursive

```
function factorial_tail(n, accumulator)
Input: n and accumulator two natural numbers
Output: the n-th factorial number
if n = 1 then
      return accumulator
else
      return factorial_tail(n-1, n*accumulator)
endif
```

cond

state1

f(Para)

Factorial Iterative

```
function factorial_iter(n, result)
Input: n and accumulator two natural numbers
Output: the n-th factorial number
while n > 1 do
      accumulator← n*accumulator
      n ← n-1
done
return accumulator
endif
```

non cond

f(Para)

state1

Recursion -> Iterative Algorithm

When you want to write iteratively a recursive function, the first technique is to come up with a tail recursion and then use the solution presented in previous slides

Otherwise you need to store context of the calls (in a way, re-doing the call stack in the program)

— use extra structures (e.g., arrays) to store the intermediary results.

This is an exemple of what's called dynamic programming

Recursion -> Iterative Algorithm

```
function factorial_dynamic(n)
Input: n a natural number
Output: the n-th factorial number
array <- array of size n
array[0] <- 1
for i from 2 till n do
     array[i-1] = array[i-2] * i
endfor
return array[n-1]
```

Complexity of Recursive Algorithms

We cannot apply the exact same mechanism we used for iterative algorithms: counting the number of basic operations and loops

Here we need to assess two things:

— the number of basic operations in each activation of the recursion *(this is easy)*

— the number of activations *(this is a little more difficult)*

Recursion Example

Factorial: n! = 1 x 2 x … x (n-1) x n
Or: ***n! = n x (n-1)!***, ***1!=1***

```
function factorial(n)
Input: n a natural number
Output: the n-th factorial number
if n = 1 then #1 operation
     return 1 #1 operation
else
     return n * factorial(n-1) #3 operations
endif
```

Recursion Simulation

3 activations for $n = 3$

6

```
factorial(3)
        if 3=1 then
                return 1
        else
                return 3 * factorial(2)
        endif
```

2

```
factorial(2)
        if 2=1 then
                return 1
        else
                return 2 * factorial(1)
        endif
```

1

```
factorial(1)
        if 1=1 then
                return 1
        endif
```

Complexity of Recursive Algorithms

number of activations: ***O(n)*** (3 for n=3)
number of operations: 2 for base case, 4 otherwise (constant running time anyway) ->
O(4) = O(1)
Total: ***O(n)*** (or T(n) = 4n, O(4n) = O(n))