COMP20230: Data Structures & Algorithms
Lecture 5: Recursion

Dr Andrew Hines

Office: E3.13 Science East
School of Computer Science
University College Dublin

andrew.hines@ucd.ie

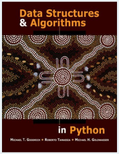
Factorial: 5!

5!

5x4x3x2x1

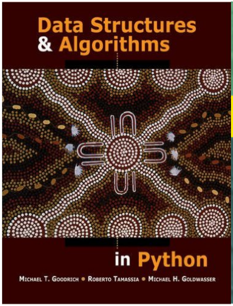
Books

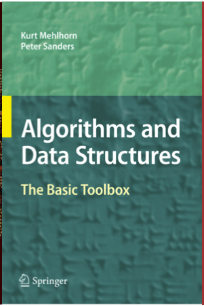
Data Structures and Algorithms in Python

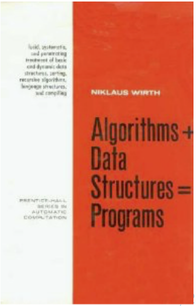


Authors: Goodrich, Tamassia and Goldwasser

Some examples:







Recursion

Last Week

- Running time and theoretical analysis
- Big-$\mathcal{O}$ notation, Big-Ω (omega) and Big-Θ (theta)

This week: Recursion

Today

- Recursion
- Base case
- Call stack

Tomorrow

- Recursive and Iterative Functions
- Tail Recursion
- Complexity of recursive functions

Take home message

Recursion is a method to divide a problem in similar sub-problems.
Simplify repetition using a function calling itself

Recursion

Today

- Recursion
- Base case
- Call stack

Take home message

Recursion is a method to divide a problem in similar sub-problems.
Simplify repetition using a function calling itself

Definition

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*).
- Requires a **stop condition** to end the recursion.

Recursion Example

Factorial

$n! = 1 \times 2 \times ... \times (n-1) \times n$

or:

$n! = n \times (n-1)!, \qquad 1! = 1$

Algorithm *factorial*(n)

Input: n, a natural number

Output: the nth factorial number

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

Factorial Example

```
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
```

Factorial Example

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!

Basic principle

1. First test the stopping condition
2. Then raise the recursive call if the **stopping condition** is not met

Algorithm *factorial*(n)

Input: n, a natural number

Output: the nth factorial number

```
if n = 1 then        # this is my stopping condition
    return 1
else
    return n * factorial(n − 1)
endif
```

Example: No Base Case

Without a base case?

What happens is bad!

Algorithm *factorial*(n)

Input: n, a natural number
Output: the nth factorial number

1: **return** $n * factorial(n-1)$
2: endif

Example: No Base Case

Without a base case?

Will it ever stop?

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

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

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

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

...

Recursion Limit: Will it ever stop?

Python has a backstop!

Exceeding this value will raise a exception `RecursionError`.

Examine size with

```
>>> import sys
>>> sys.getrecursionlimit()
1000
```

Python Recursion Limit: http://docs.python.org/3/library/sys.html#sys.setrecursionlimit
Image Source: https://www.irishtimes.com/news/politics/still-don-t-know-what-the-brexit-backstop-is-we-explain-it-through-cricket-1.3778683

Call Stack

The call stack

Is a stack data structure
It stores information about the active subroutines of a computer program
We will look at stacks in detail in a few weeks time.
For now, treat it as a stack of pancakes/pringles (you can only add or remove items from the top)

- 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: Calculating 3 factorial

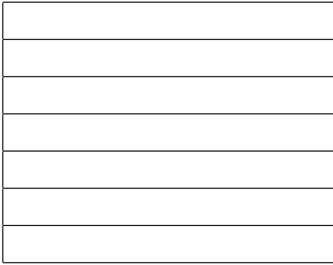
Call to `factorial(3)`

So we put `factorial(3)` on the call stack

Algorithm *factorial*(n)

Input: n, a natural number
Output: the nth factorial number

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



Call Stack

Recursion Simulation: Calculating 3 factorial

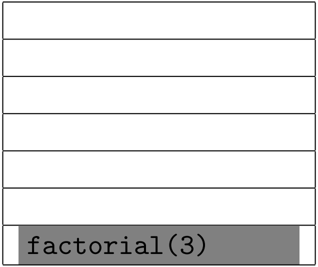
Call to `factorial(3)`

So we put `factorial(3)` on the call stack

Algorithm *factorial*(n)

Input: n, a natural number
Output: the nth factorial number

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



Call Stack

Recursion Simulation: Calculating 3 factorial

Check `if` $n = 1$

$n! = 1$: Return is `n*factorial(n-1)`, including `factorial(2)` call.

Remember

Call to `factorial(3)` has not yet returned so it is still on the call stack

Algorithm *factorial*(n)

Input: n, a natural number
Output: the nth factorial number

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

`factorial(3)`

Call Stack

Recursion Simulation: Calculating 3 factorial

Check if $n = 1$

$n! \neq 1$: Return is `n*factorial(n-1)`, including `factorial(2)` call.

Remember

Call to `factorial(3)` has not yet returned so it is still on the call stack

---

Algorithm *factorial*(n)

Input: n, a natural number
Output: the nth factorial number

if $n = 1$ then
    return 1
else
    return $n * factorial(n - 1)$
endif

`factorial(2)`
`factorial(3)`

Call Stack

Recursion Simulation: Calculating 3 factorial

Check `if` $n = 1$

$n! = 1$: Return is `n*factorial(n-1)`, including `factorial(1)` call.

Remember

Neither `factorial(2)` or `factorial(3)` has returned at this point

Algorithm *factorial*(n)

Input: n, a natural number
Output: the nth factorial number

if $n = 1$ then
  return 1
else
  return $n * factorial(n - 1)$
endif

`factorial(2)`
`factorial(3)`

Call Stack

Recursion Simulation: Calculating 3 factorial

Check again if $n = 1$

$n! = 1$: Return is `n*factorial(n-1)`, including `factorial(1)` call.

Note

Neither `factorial(2)` or `factorial(3)` has returned at this point

Algorithm *factorial*(n)

Input: n, a natural number
Output: the nth factorial number

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

`factorial(1)`
`factorial(2)`
`factorial(3)`

Call Stack

Recursion Simulation

Check `if` $n = 1$

$n = 1$: So return 1

Not a recursive call in return

So `factorial(1)` returns 1
We take it off the call stack

`factorial(1)`
`factorial(2)`
`factorial(3)`

Call Stack

Algorithm *factorial*(n)

Input: n, a natural number
Output: the nth factorial number

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

Recursion Simulation

Top of stack: `factorial(2)`

So we return to where we were in `factorial(2)`.

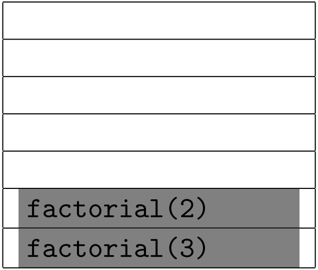
return 2*`factorial(1)`

We know this is 2*1 so `factorial(2)` returns 2.
And remove it from the stack.

Algorithm *factorial*(n)

Input: n, a natural number
Output: the nth factorial number

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



Call Stack

Recursion Simulation

Top of stack: `factorial(3)`

So we return to where we were in `factorial(3)`.

return 3*`factorial(2)`

We now know this is 3*2 so `factorial(3)` returns 6.
And remove it from the stack.

Algorithm *factorial*(n)

Input: n, a natural number
Output: the nth factorial number

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

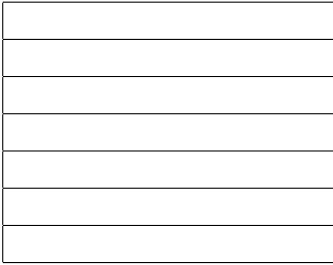
`factorial(3)`

Call Stack

Recursion Simulation

Call stack is empty

We know now that `factorial(3)` is 6.



Call Stack

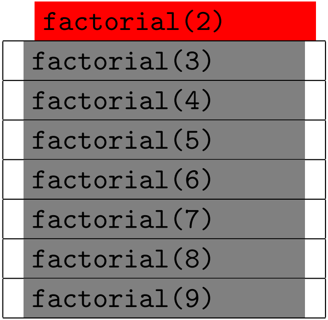
Algorithm $factorial(n)$

Input: n, a natural number
Output: the nth factorial number

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

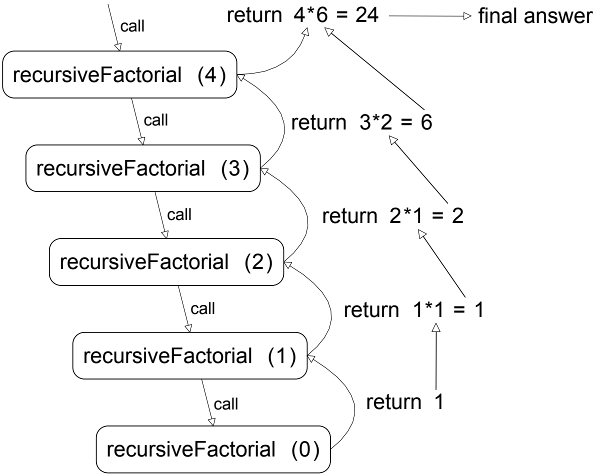
Call Stack

- 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
- A **stack overflow** occurs when the call stack reaches the stack bound, i.e. no more room on the stack for another call to be allocated



Call Stack

Drawing a recursion trace: factorial example



Note: You may spot that this goes to 0, not 1 as in our example. The value of 0! is 1, according to the convention for an empty product. See `https://en.wikipedia.org/wiki/Empty_product` for more details. So technically for correctness we should use n==0 as the base case.

Conclusion

Recursion can make for efficient and elegant code

A method to divide a problem in similar sub-problems.
Simplify repetition using a function calling itself.

Beware of stack overflows (or in Python `RecursionError` exceptions – set up your base case!

Housekeeping

Tutorial

Recursive functions worksheet.
Esri will be walking through some solutions to last week's lab and tutorial.

Tomorrow

- Why/when using recursion
- Tail recursion
- Turning a recursive algorithm into an iterative one
- Complexity analysis of recursive functions