# MA1: L01

*Tom Smedsaas*

# Introduction to recursion

# Algorithms

**Definition**: An *algorithm* is a sequence of well defined instructions that solves a  problem or performs a computation with a finite number of steps.

**Components:**

- *sequence* (a set of instructions to be performed in order)
- *selection* (`if`, `elif`, `else`)
- *iteration* (`for`, `while`)
- *abstraction* (functions, methods)

# Example: The factorial function

The factorial function can be defined *iteratively*:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1 \cdot 2 \cdot 3 \cdot \ \dots \ \cdot n & \text{if } n > 0 \end{cases}$$

which can be translated into a program using an iteration:

```
def fac(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```

# Recursive factorial definition

Factorials can also be defined *recursively*:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$

```
def fac(n):
    if n == 0:
        return 1
    else:
        return n*fac(n-1)
```

```
def fac(n):
    result = 1
    if n > 0:
        result = n*fac(n-1)
    return result
```

UPPSALA UNIVERSITET

# Demonstration using the debuggern in Thonny

# Example: Compute $x^n$

The operation $x^n$ where $x$ is real and $n$ is integer:

Iteratively: $$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x \cdot x \cdot \ \dots \ \cdot \mathrm{x} & \text{if } n > 0 \end{cases}$$

Recursively: $$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}$$

# If n is negative?

$$x^n = \begin{cases} \dfrac{1}{x^{-n}} & \text{if } n < 0 \\ 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}$$

```python
def power(x, n):
    if n < 0:
        return 1./power(x, -n)
    elif n == 0:
        return 1
    else:
        return x*power(x, n-1)
```

# Why recursion?

- A powerful method to *find* algorithms for non-trivial problems.
- A powerful method to find *efficient* algorithms.
- *Very natural* in many problems.

But also

- Powerful way to construct hopelessly inefficient algorithms.

# Recursion in general

1. Divide the problem into one or more sub-problems *of the same type*.
2. Solve the sub-problems (recursively)
3. Use the solutions of the sub-problems to construct a solution to the original problem.

There must be at least one recursion-terminating case – *base cases*.

In the factorial calculation $n!$:

- **One** sub-problem: Compute $(n-1)!$.
- **Combine**: Multiply the solution of the sub-problem by $n$.
- **Base case**: $n = 0$.

# How to find sub-problems?

In the examples above the size of the problems were defined by a number *n* which was given as a parameter: $n!$ and $x^n$.

Another example:

```
number_of_digits(x)
```

which is supposed to return the number of (decimal) digits in the integer x.

```
number_of_digits(125) → 3

number_of_digits(2341562) → 7
```

```
def number_of_digits(x):
    if x < 10:
        return 1
    else:
        return 1 + number_of_digits(x//10)
```

# Sometimes there are different ways to choose subproblems

Write the function `reverse(lst)` that returns a new list where the elements come in reverse order.

(We ignore all built-in functions and methods for reversing lists.)

```
def reverse(lst):
    if len(lst) <= 1:
        return lst
    else:
        mid = len(lst)//2
        return reverse(lst[mid:]) + reverse(lst[:mid])
```

**Question**: What do we have to do to make this function work on strings?

# Sometimes the recursion is easily seen

Suppose we have the following *iterative* function:

```
def reverse_list(lst):
    result = []
    for x in lst:
        result.insert(0, x)
    return result
```

In: [1, [2, [3, 4]], [[5, 6], 7], [8, 9]]

Ut: [[8, 9], [[5, 6], 7], [2, [3, 4]], 1]

# Sometimes the recursion is easily seen…

Now assume that the function should also flip on all incoming sublists, ie

In: `[1, [2, [3, 4]], [[5, 6], 7], [8, 9]]`

Out: `[[9, 8], [7, [6, 5]], [[4, 3], 2], 1]`

Easy! We have a function that reverses lists:

```
def reverse_list(lst):
    result = []
    for x in lst:
        result.insert(0, x)
    return result
```

```
def reverse_list(lst):
    result = []
    for x in lst:
        if type(x) is list:
            x = reverse_list(x)
        result.insert(0, x)
    return result
```

# Example with two base cases

A function that receives two lists of numbers and returns a new list where the elements consist of the numbers summed in pairs:

```
summa([1,2,3], [1,2,3,4,5])   →   [2,4,6,4,5]
summa([1,2,3], [5,7])         →   [5,9,3]
```

```python
def summa(x, y):
    if len(x) == 0:
        return y
    elif len(y) == 0:
        return x
    else:
        return [x[0] + y[0]] + summa(x[1:], y[1:])
```

# *The end*