



deti

universidade de aveiro
departamento de electrónica,
telecomunicações e informática

Fundamentos de Programação

António J. R. Neves

João Rodrigues

Departamento de Electrónica, Telecomunicações e Informática

Universidade de Aveiro

`an@ua.pt / jmr@ua.pt`

`http://elearning.ua.pt/`

- Recursive functions
 - How it works.
 - The program stack.
 - The rules for termination
- Examples
 - Operations on a list
 - Towers of Hanoi
 - A sorting algorithm (kind of quicksort)

A crazy idea



deti

universidade de aveiro
departamento de electrónica,
telecomunicações e informática

- What does this function do?

```
sumsq([1, 2, 3])    #-> 14
```

- Does it work on an empty list?
 - Can you write it with a generator expression? (Homework!)
- Check this weird version!
 - It squares first element;
 - Calls `sumsq` on the rest;
 - And adds.

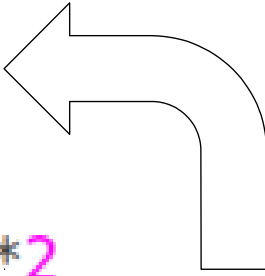
```
def sumsq(lst):  
    s = 0  
    for x in lst:  
        s += x**2  
    return s
```

```
def sumsq2(lst):  
    s = 0  
    if len(lst) > 0:  
        sq0 = lst[0]**2  
        s = sq0 + sumsq(lst[1:])  
    return s
```

- It works, but must call the original `sumsq`. Not very useful.
- But if `sumsq2` works, why not **call itself**?

- This is what would result.

```
def sumsqR(lst):  
    s = 0  
    if len(lst) > 0:  
        sq0 = lst[0]**2  
        s = sq0 + sumsqR(lst[1:])  
    return s
```



- This is a **recursive function**: a function that calls itself.
- Notice that there is no loop instruction, but code gets executed several times, anyway.
- How does it work?

- What happens when we call `sumsqR([1, 2, 3])` ?
 - Watch the [execution in PythonTutor](#).

```
sumsqR([1, 2, 3])  
|    sumsqR([2, 3])  
|    |    sumsqR([3])  
|    |    |    sumsqR([])  
|    |    |    L>0  
|    |    L>9    ( = 3**2 + 0 )  
|    L>13    ( = 2**2 + 9 )  
L>14    ( = 1**2 + 13 )
```

- Notice that at one point, there are 4 frames in memory.
 - 4 variables named `lst`, 4 named `s`, 3 named `sq0`, but all distinct!
- The frames are stored in the **program stack**.

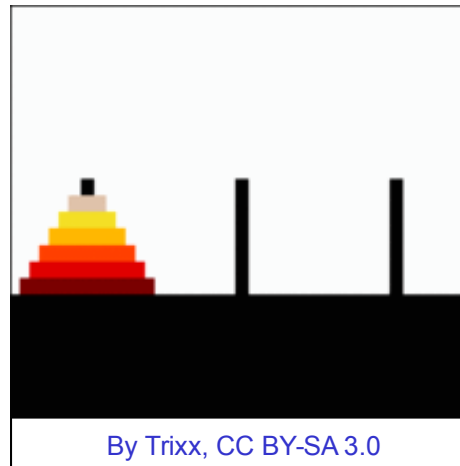
Example: Towers of Hanoi



deti

universidade de aveiro
departamento de electrónica,
telecomunicações e informática

- The Towers of Hanoi (Édouard Lucas, 1883)
- Move tower from A to C, using B temporarily.
 - Move only one disk at a time;
 - No disk may be put on top of a smaller disk.



- Now solve it in 4 lines of code!

Example: quicksort



deti

universidade de aveiro
departamento de electrónica,
telecomunicações e informática

- The quicksort algorithm (C.A.R. Hoare) goes like this:
 - Pick one of the values in the list (generally the first) and store in T.
 - Put values smaller than T into a list L1, the others into a list L2.
 - Sort L1 and L2 (using same algorithm, by the way)
 - Result is L1 + [T] + L2.
- Of course, there's a few more details (the base case).

```
def qsorted(lst):  
    if len(lst) <= 1:      # no need to sort  
        return lst[:]    # just return a copy  
    T = lst[0]  
    L1 = [x for x in lst[1:] if x < T]  
    L2 = [x for x in lst[1:] if x >= T]  
    return qsorted(L1) + [T] + qsorted(L2)
```

- This is simple to understand and quite efficient!

The actual quicksort modifies the list in-place, and is slightly harder.

To guarantee that a recursive function **terminates**, it must obey some rules!

1. There must be **base cases**, which can be solved without recursive calls.
 - In `sumsqR`, the base case is `len(lst) == 0`. In that case, return 0.
2. In the other cases, the *context* passed to recursive calls **must differ** from the context received.
 - In `sumsqR`, the argument `lst[1:] != lst`.
3. The context in successive recursive calls must **converge** towards the base cases.
 - In `sumsqR`, the `lst` is shortened each time, until it's empty.

The **context** is the set of arguments (and global values) that have an impact on the base case / recursive case selection.

- Any problem that can be solved by repetition may be solved by recursion, and vice-versa.
- For certain complex problems, recursive solutions are usually more concise and easier to understand.
- Recursive implementations imply some time and memory cost because of functions calls and stack usage.
- If the problem has a simple iterative solution, that is usually the most efficient, too.

- To develop a recursive function to solve some problem, there are some tricks that help.
 1. Start by defining the **arguments** you need, what they **mean**, and the **result** you **expect**, as *rigorously* as possible.
 2. Now, **assume** the function will work. Describe how the solution to a problem can be obtained from the solutions of **smaller** versions of the problem. This will be the recursive part of the algorithm.
 3. Finally, determine what are the **base cases**: which conditions have a trivial solution? This will be the non-recursive part of the algorithm. (Hint: usually, base cases are conditions outside the domain of the recursive call.)
- While in step 2, you may realize that you need extra arguments. Just add them and go back to step 1.