

Fundamentos de Programação

António J. R. Neves
João Rodrigues

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

Summary

- Recursive functions
 - How recursion works
 - The rules for termination
 - Recursion vs. repetition
 - How to write recursive functions
- Examples
 - Operations on a list
 - Towers of Hanoi
 - A sorting algorithm (kind of quicksort)

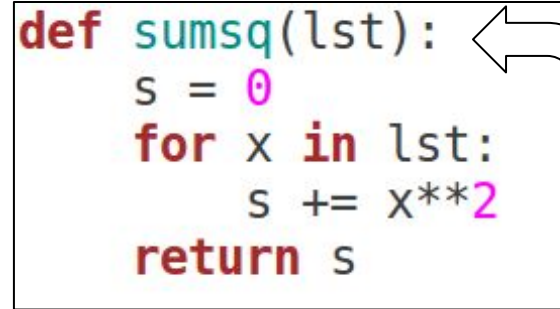
A crazy idea

- 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!)

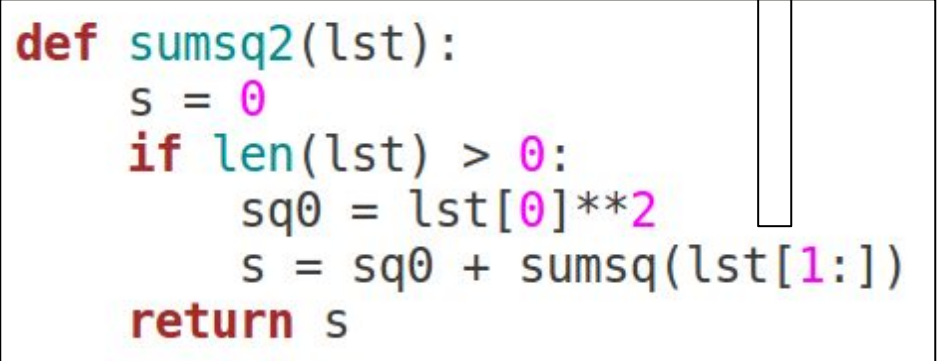
```
def sumsq(lst):  
    s = 0  
    for x in lst:  
        s += x**2  
    return s
```



- Check out this weird version!

- It squares first element;
- Calls `sumsq` on the rest;
- And adds.
- For empty `lst`, just return zero.

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

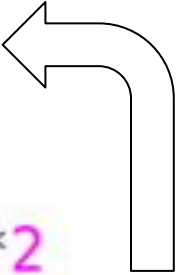


- It is equivalent to `sumsq` in every case, but still calls the original `sumsq`. Not very useful.
- But if `sumsq2 <=> sumsq`, why not **call itself**?

Recursive functions

- 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?

How recursion works

- What happens when we call `sumsqR([1, 2, 3])` ?

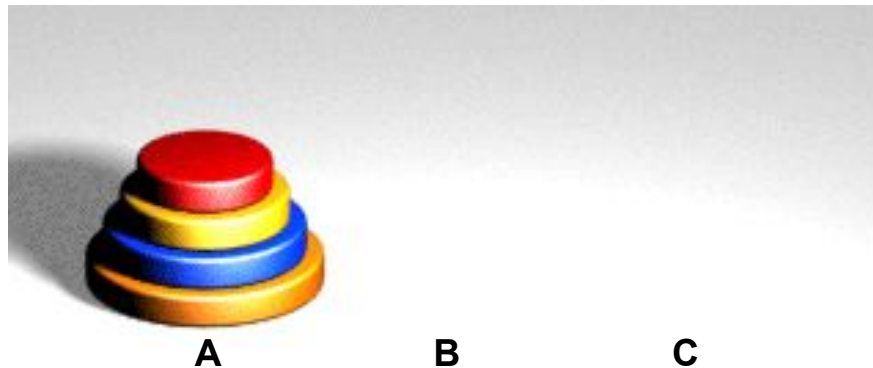
```
sumsqR([1, 2, 3])
├── sumsqR([2, 3])
│   ├── sumsqR([3])
│   │   ├── sumsqR([])
│   │   │   └── L>0
│   │   └── L>9    ( = 3**2 + 0 )
│   └── L>13    ( = 2**2 + 9 )
└── L>14    ( = 1**2 + 13 )
```



- Notice that at one point, there are 4 frames in memory.
 - 4 variables named `lst`, 4 named `s`, 3 named `sq0`, but all distinct!
- Each frame stores the *local context* of a single function call.
- The frames are stored in the **program stack**.

Example: Towers of Hanoi

- The Towers of Hanoi puzzle (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.



[André Karwath aka Aka, CC BY-SA 2.5](#), via Wikimedia Commons

- Now solve it in 4 lines of code!

[Play ▶](#)

Example: quicksort

- The quicksort algorithm (C.A.R. Hoare) goes like this:
 1. Pick one of the values in the list (generally the first) and store in T.
 2. Put values smaller than T into a list L1, the others into a list L2.
 3. Sort L1 and L2 (using same algorithm, by the way)
 4. 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)
```

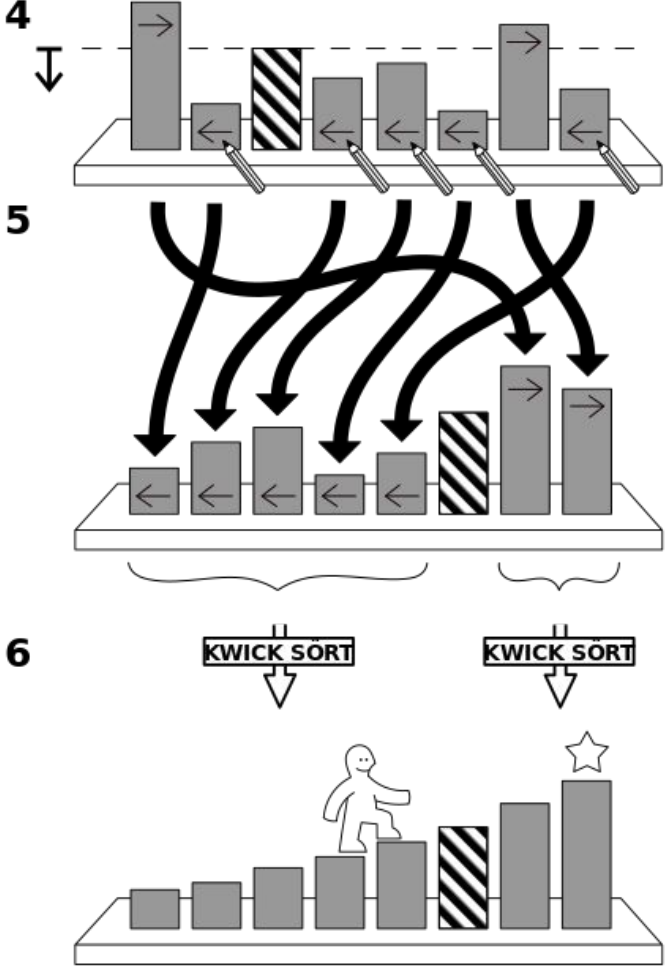
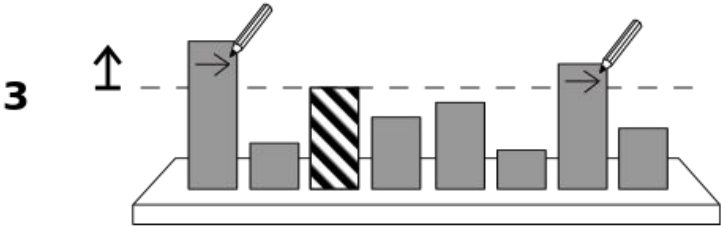
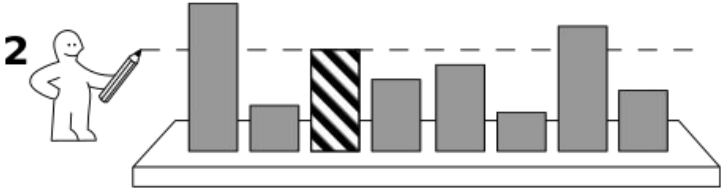
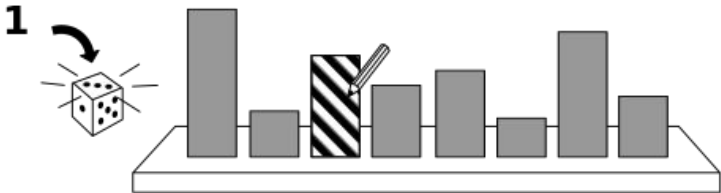
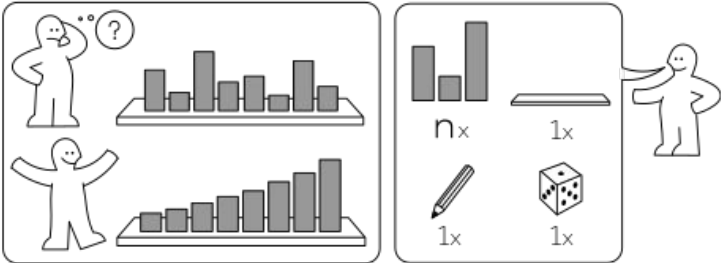
- This is simple to understand and quite efficient!

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

Example: quicksort

KWICK SÖRT

idea-instructions.com/quick-sort/
v1.0, CC by-nc-sa 4.0



The rules for recursion termination

To guarantee that a recursive function **terminates**, it must respect some **rules**:

1. There must be cases that can be solved *without* recursive calls. These are called the **base cases**.
 - 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 always differ** from the *context received*.
 - In `sumsqR`, the argument `lst[1:] != lst` (always)
3. In successive recursive calls, the context must **converge** towards the base cases.
 - In `sumsqR`, the `lst` is shortened on each call, 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.

Recursion vs repetition

- 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 may incur extra time and memory cost because of function calls and stack usage.
- If the problem has a simple iterative solution, that is usually the most efficient, too.

Writing recursive functions

- To develop a recursive function to solve some problem, follow these guidelines:
 1. First, **define** 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 by **modifying** the solutions to **smaller** versions of the problem. This will be the recursive part of the algorithm.
 3. Finally, **determine** the **base cases**: which conditions have a trivial solution? This will be the non-recursive part of the algorithm. (Hint: base cases are usually *outside the domain* of the recursive part.)
- While in step 2, you may realize that you need *extra arguments*. Just add them and go back to step 1.