Funções puras e impuras

Definição. Uma função é **pura** se

1. retorna o mesmo valor para os mesmos argumentos,
2. não altera o estado do programa, e
3. não interage com o ambiente (não realiza I/O)

Em outras palavras, uma função pura é uma função no mesmo sentido da matemática.

Em geral,

- funções puras são convenientes quando estamos usando o paradigma funcional ou concorrente/paralelo
- funções não puras são adequadas para o paradigma imperativo
  - Dependem de *aliasing*

Funções recursivas não puras (com efeitos colaterais)

Todos os exemplos sobre recursão vistos até agora tem sido exemplos de funções puras.

Vejamos agora exemplos de funções recursivas que realizam efeitos colaterais.

1. A metáfora do amigo também funciona para funções não puras.
2. Os mesmos cuidados anteriores são necessários (chamada recursiva com tamanho menor e garantir que os casos base são alcançados)
3. Precisamos observar a ordem correta de execução

Exemplo: inverter uma lista

A lista passada como argumento é invertida.

Delimitamos a porção da chamada recursiva com dois índices `i` e `j`, para o início e fim, respectivamente.

$$[\ldots, a_1, a_2, a_3, \ldots, a_{k-2}, a_{k-1}, a_k, \ldots]$$
$$\uparrow \qquad\qquad\qquad\qquad\qquad \uparrow$$

trocamos o primeiro e último da porção e avançamos as setas

$$[\ldots, a_k, a_2, a_3, \ldots, a_{k-2}, a_{k-1}, a_1, \ldots]$$
$$\uparrow \qquad\qquad\qquad\qquad\qquad \uparrow$$

O amigo inverte a porção marcada pelas setas

$$[\ldots, a_k, a_{k-1}, a_{k-2}, \ldots, a_3, a_2, a_1, \ldots]$$

In [1]:

```
def inverter(xs):
    inverterDeAte(0, len(xs)-1, xs)

def inverterDeAte(i, j, xs):
    if i < j:
        xs[i], xs[j] = xs[j], xs[i]
        inverterDeAte(i+1, j-1, xs)

xs = [1,2,3]
inverter(xs)

xs
```

Out[1]:

```
[3, 2, 1]
```

Exemplo: Ordenação por inserção (versão recursiva)

A lista passada como argumento é ordenada.

Como o algoritmo atua alterando a lista, uma chamada recursiva deverá atuar sobre uma porção menor da lista. Portanto, precisamos delimitar a porção sobre a qual a chamada recursiva opera. A forma natural de especificar esta porção é através de um índice de início e outro de fim.

Particularmente, nesta ordenação teremos duas funções recursivas, ambas operando na recursão numa porção da lista que começa no índice zero e termina num índice `j`.

`ordenarIns(j, xs)` : ordena a porção relativa a `xs[:j+1]`

$$[2, 0, 4, 8, 1, 2, 5, 3]$$
$$\uparrow$$

O *bom amigo* (chamada recursiva) ordena a porção à esquerda de ↑

$$[0, 1, 2, 2, 4, 5, 8, 3]$$
$$\uparrow$$

Chamamos `InserirOrd` para inserir o elemento apontado por ↑ na porção já ordenada

$$[0, 1, 2, 2, 3, 4, 5, 8]$$

In [2]:

```
def ordenar(xs):
    ordenarIns(len(xs)-1, xs)

def ordenarIns(j, xs):
# ordena a porção [:j+1] de xs
    if j > 0:
        ordenarIns(j-1, xs)
        inserirOrd(j, xs)
```

`inserirOrd(j, xs)` : supondo que a porção relativa a `xs[:j]` já está ordenada, insere/acrescenta `xs[j]` a esta porção mantendo a ordem.

$$[0, 1, 2, 4, 8, 1, 2, 5, 3]$$
$$\uparrow$$

troca 8 com 1

$$[0, 1, 2, 4, 1, 8, 2, 5, 3]$$
$$\uparrow$$

O *bom amigo* (chamada recursiva) insere 1 na porção à esquerda de $\uparrow$, mantendo a ordem.

$$[0, 1, 1, 2, 4, 8, 2, 5, 3]$$

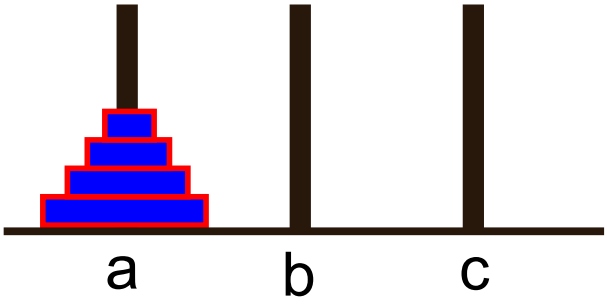
In [3]:

```
1
2  def inserirOrd(j, xs):
3      if j > 0 and xs[j] < xs[j-1]:
4          xs[j], xs[j-1] = xs[j-1], xs[j]
5          inserirOrd(j-1, xs)
6
7  xs = [3,2,1]
8  ordenar(xs)
9  xs
```

Out[3]:

```
[1, 2, 3]
```

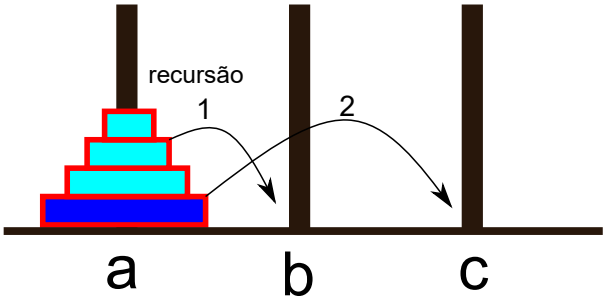
Exemplo: O problema das torres de hanoi

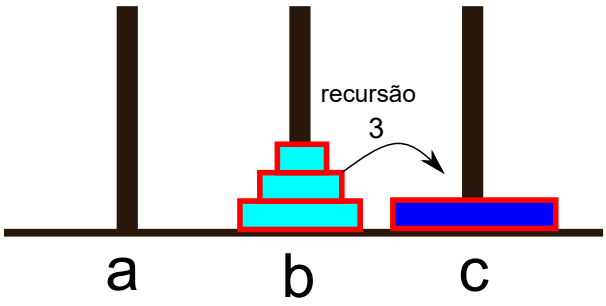


O quebra-cabeça chamado Torres de Hanoi possui três torres onde são empilhadas discos. Inicialmente a torre **a** tem empilhada `n` discos de diferentes tamanhos, sendo que não há um disco maior acima de um menor. O objetivo do jogo é passar os `n` discos da torre **a** para a torre **c** usando a torre **b** como intermediária e seguindo as regras seguintes

- Só é possível mover um disco de cada vez
- Cada movimento consiste em pegar o disco no topo de uma das torres e coloca-lo no topo de outra torre
- Nunca um disco maior pode ficar acima de um menor.

A solução natural a este problema é recursiva e a podemos ilustrar assim





In [4]:

```
def hanoi(a, b, c, n):
# move n torres desde a até c usando b como intermediária
    if n == 1:
        print("Mova ", 1, " de", a, " para", c)
    else:
        hanoi(a, c, b, n-1)
        print("Mova ", n, " de", a, " para", c)
        hanoi(b, a, c, n-1)

hanoi('a', 'b', 'c', 3)
```

```
Mova  1  de a  para c
Mova  2  de a  para b
Mova  1  de c  para b
Mova  3  de a  para c
Mova  1  de b  para a
Mova  2  de b  para c
Mova  1  de a  para c
```