Fundamentos de Programação
@ LEIC/LETI

Semana 8

Recursão e Iteração
Recursão e Iteração linear. Recursão de cauda. Recursão em árvore.

Alberto Abad, Tagus Park, IST, 2018

Recursão e Iteração

- No desenvolvimento de programas é importante ter em conta como é que um programa é executado e, em particular, como é que o processo computacional inerente à execução do programa evolui.

- Hoje, vamos a analisar alguns padrões típicos de evolução de programas e funções, em paricular:

    - Recursão linear
    - Iteração linear
    - Recursão de cauda
    - Recursão em árvore

Recursão Linear

- A recursão linear é a forma mais comum de recursão.
- Vários ambientes locais são gerados por causa da chamada repetida da pópria função: **expansão** de memória.
- Em cada ambiente ficamos com uma operação adiada até atingir o caso terminal, em que os ambientes vão sendo libertados e ocurre uma **contracção**.

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

Recursão Linear - Padrão de execução

```
factorial(4)
| factorial(3)
| | factorial(2)
| | | factorial(1)
| | | | factorial(0)
| | | | return 1
| | | return 1
| | return 2
| return 6
return 24
```

- O número de abientes cresce **linearmente** em função de um determinado valor da entrada → *Processo Recursivo Linear*
- Considerações sobre eficiência:
    - Tempo: linear, *O(n)*
    - Espaço: linear, *O(n)*

Interação linear

- Na iteração linear, gera-se um processo iterativo caraterizado por:
  - Um conjunto de variáveis de estado,
  - Regras que especificam como actualizá-las.

```
>>> def factorial(n):
        res = 1
        for i in range(1, n+1):
            res = res * i
        return res
```

- O número de operações sobre as variáveis de estado cresce linearmente com um valor associado à função → *Processo Iterativo Linear*
- Considerações sobre eficiência:
  - Tempo: linear, *O(n)*
  - Espaço: constante, *O(1)*

Recursão de cauda

- É possível definir processos recursivos que utilizem variáveis de estado de forma semelhante aos processos iterativos → *Recursão de cauda*
- Na recursão de cauda:
  - Primeiro o cálculo é realizado e, logo, é feita a chamada recursiva.
  - Na chamada são passados os resultados da etapa atual para a próxima etapa recursiva.
  - A chamada recursiva é a última operação realizada pela função, não existindo operações adiadas.

```
>>> def factorial(n, acc):
        if n == 0:
            return acc
        else:
            return factorial(n - 1, n * acc)
```

Recursão de cauda

- Podemos organhizar um pouco melhor a função como vimos em semanas anteriores:
  - Definimos uma função auxiliar.
  - O acumulador na função auxiliar é inicializado com o valor a retornar no caso base da recursão linear.

```
def factorial(n):
    def factorial_aux(n, acc):
        if n == 0:
            return acc
        else:
            return factorial_aux(n - 1, n * acc)
    return factorial_aux(n, 1)
```

Recursão de cauda - Padrão de execução

```
factorial(4)
| factorial_aux(4, 1)
| | factorial_aux(3, 4)
| | | factorial_aux(2, 12)
| | | | factorial_aux(1, 24)
| | | | | factorial_aux(0, 24)
| | | | | return 24
| | | | return 24
| | | return 24
| | return 24
| return 24
return 24
```

- Como a recursão linear, o número de abientes cresce **linearmente** em função de um determinado valor da entrada.
- Considerações sobre eficiência:
  - Tempo: linear, *O(n)*
  - Espaço: linear, *O(n)*

Recursão de cauda e Iteração linear - Vantagems da recursão de cauda

- A recursão de cauda pode facilmente ser convertida/optimizada em uma iteração linear:

In [ ]:
```
def factorial(n):
    def factorial_aux(n, acc): ## <--- CHANGE HERE
        if n == 0:
            return acc
        else:
            return factorial_aux(n - 1, n * acc) ## <---CHANGE HERE
    return factorial_aux(n, 1) ## <--- CHANGE HERE
```

- Algumas linguagens fazem a optimização da recursão de cauda para um processo iterativo automáticamente.
- O Python **não optimiza** as recusões de cauda.

Recursão e Iteração - Exercício 1, *potencia*

In [8]:

```
def potencia_il(x, n):
    res = 1
    for i in range(1, n+1):
        res = res*x
    return res

def potencia_rl(x,n):
    if n == 0:
        return 1
    else:
        return x * potencia_rl(x, n-1)

def potencia_rc(x, n):
    pass

potencia_rc(2,4)
```

Out[8]: 16

Recursão e Iteração - Exercício 2, *soma*

In [13]:
```
def soma_il(lst):
    res = 0
    for e in lst:
        res = res + e
    return res

def soma_rl(lst):
    if lst == []:
        return 0
    else:
        return lst[0] + soma_rl(lst[1:])

def soma_rc(lst):
    pass

soma_rc([2,4])
```

Out[13]:
```
6
```

Recursão múltipla: Recursão em árvore

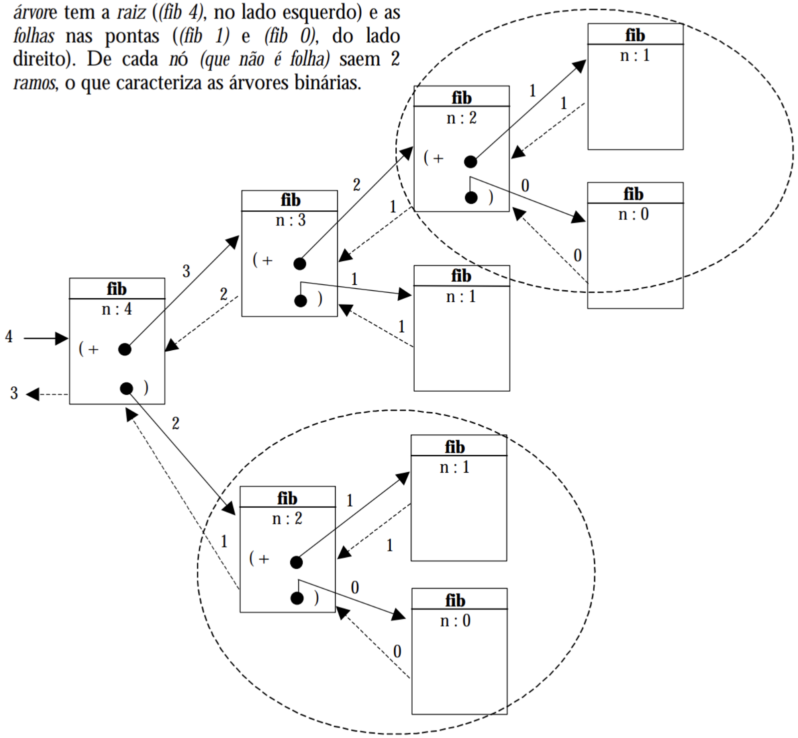
- Para além da recursão linear (1 chamada recursiva), existem outros padrões bastante comuns como é a recursão múltipla (múltiplas chamadas recursivas).
- Un exemplo de recursão múltipla é a recursão em árvore ou binária.
- Exemplo, números de Fibonacci:

$$fib(n) = \begin{cases} 0 & \text{se } n = 0, \\ 1 & \text{se } n = 1, \\ fib(n-1) + fib(n-2) & \text{se } n > 1 \end{cases}$$

```
def fib(n):
        if n == 0:
            return 0
        elif n == 1:
            return 1
        else return fib(n - 1) + fib(n - 2)
```

Recursão em árvore

A chamada *(fib 4)*, sob a forma gráfica, mostra que o processo gerado é *recursivo em árvore*. A *árvor*e tem a *raiz* (*(fib 4)*, no lado esquerdo) e as *folhas* nas pontas (*(fib 1)* e *(fib 0)*, do lado direito). De cada *nó* *(que não é folha)* saem 2 *ramos*, o que caracteriza as árvores binárias.



Recursão em árvore - Padrão de execução

- Se avaliarmos `fib(4)`, o processo computacional gerado pela função *fib* apresenta a seguinte evolução:

```
fib(4)
| fib(3)
| | fib(2)
| | | fib(1)
| | | return 1
| | | fib(0)
| | | return 0
| | return 1
| | fib(1)
| | return 1
| return 2
| fib(2)
| | fib(1)
| | return 1
| | fib(0)
| | return 0
| return 1
return 3
```

- Reparar múltiples fases de expansão e contracção.

Recursão em árvore - Optimização Fibonacci

- A anterior implementação têm dois problemas:
  - Cria muitos ambientes locais
  - Existem muitos cómputos repetidos
- Versão optimizada (recursão de cauda):

In [22]:
```
def fib(n):
    def fib_aux(s, t, n):
        if n == 0:
            return s
        elif n == 1: #Esta condição é desnecessária, é só para poupar uma chamada
 recursiva
            return t
        else:
            return fib_aux(t, s+t, n - 1)

    return fib_aux(0, 1, n)

fib(4)
```

Out[22]: 3

Recursão em árvore - Optimização Fibonacci

Padrão de execução

```
fib(4)
fib_aux(0, 1, 4)
| fib_aux(1, 1, 3)
| | fib_aux(1, 2, 2)
| | | fib_aux(2, 3, 1)
| | | return 3
| | return 3
| return 3
return 3
return 3
```

Considerações sobre eficiência - Sumário

- A minimização dos recursos computacionais consumidos por um programa é um dos aspectos que nos preocupa quando escrevemos programas.
- Diferéncias na evolução dos processos, levam a diferenças nos recursos computacionais consumidos:
  - **Tempo** que um programa demora a executar (número de passos atómicos realizados).
  - **Espaço** de memória que um programa utiliza durante a sua execução (em geral queremos saber o máximo necessário, não a soma).

Padrão	Tempo	Espaço
Recursão Linear	O(n)	O(n)
Iteração Linear	O(n)	O(1)
Recursão Binária	$O(k^n)$	O(n)