Fundamentos da Programação

Recursão em Árvore

Aula 23

José Monteiro

(slides adaptados do Prof. Alberto Abad)

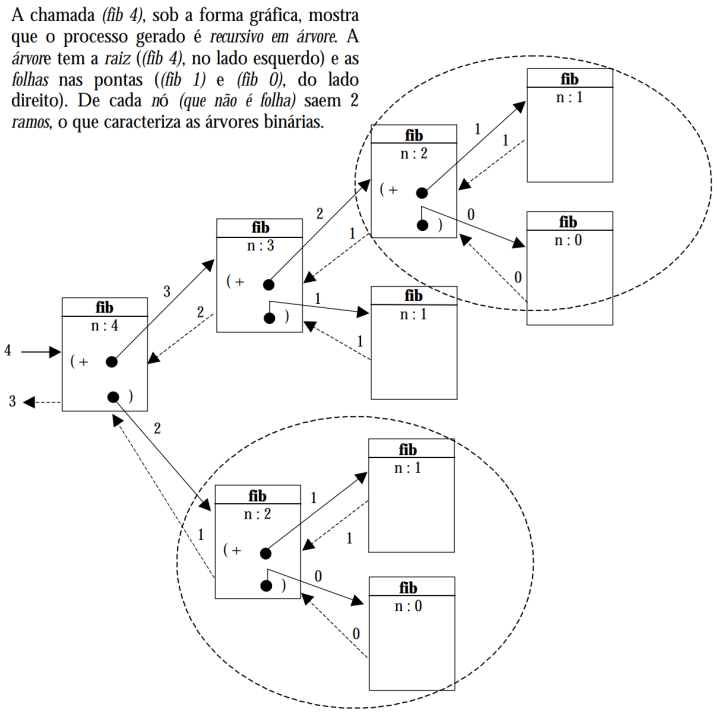
Recursão Múltipla: Recursão em Árvore

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

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

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

Recursão em Árvore



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 nas múltiplas fases de expansão e contração.

Recursão em Árvore

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

Optimização Fibonacci

- A implementação anterior tem dois problemas:
  - Cria muitos ambientes locais
  - Existem muitos cálculos repetidos
- E se experimentarmos a recursão de cauda?

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)
| | |  | fib_aux(3, 5, 0)
| | |  | return 3
| | | return 3
| | return 3
| return 3
return 3
return 3
```

In [3]:
```
def fib_rc(n): # versão com recursão de cauda...
    def fib_aux(primeiro, segundo, n):
        if n == 0:
            return primeiro
        else:
            return fib_aux(segundo, primeiro+segundo, n-1)

    return fib_aux(0, 1, n)
```

Recursão em Árvore

In [21]:
```
def fib(n):
        if n == 0:
            return 0
        elif n == 1:
            return 1
        else:
            return fib(n - 1) + fib(n - 2)
```

Optimização Fibonacci

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

In [22]:
```
def fib_rc(n): # versão com recursão de cauda...
    def fib_aux(primeiro, segundo, n):
        if n == 0:
            return primeiro
        else:
            return fib_aux(segundo, primeiro+segundo, n-1)

    return fib_aux(0, 1, n)
```

In [23]:
```
def fib_il(n): ## Conversão recursão de cauda em iteração linear
    primeiro, segundo = 0, 1
    while True:
        if n == 0:
            return primeiro
        else:
            primeiro, segundo, n = segundo, primeiro+segundo, n-1
```

In [24]:
```
%timeit -n 10 fib(25)
print(fib(25))
%timeit -n 10 fib_rc(25)
print(fib_rc(25))
%timeit -n 10 fib_il(25)
print(fib_il(25))
```

```
29.1 ms ± 441 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
75025
3.76 µs ± 144 ns per loop (mean ± std. dev. of 7 runs, 10 loops each)
75025
2.06 µs ± 47.6 ns per loop (mean ± std. dev. of 7 runs, 10 loops each)
75025
```

Recursão em Árvore - Optimização Fibonacci

- Versão optimizada 2 (utilizar memória/dicionário):

In [1]:

```
def fib_mem(n):
    mem = {0:0, 1:1}
    def fib_aux(n):
        if n in mem:
            return mem[n]
        else:
            mem[n] = fib_aux(n - 1) + fib_aux(n - 2)
            return mem[n]
    return fib_aux(n)

print(fib_mem(25))
```

```
75025
```

In [20]:

```
%timeit -n 10 fib(25)
%timeit -n 10 fib_rc(25)
%timeit -n 10 fib_il(25)
%timeit -n 10 fib_mem(25)
```

```
25 ms ± 756 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
3.23 µs ± 83.1 ns per loop (mean ± std. dev. of 7 runs, 10 loops each)
1.8 µs ± 35.3 ns per loop (mean ± std. dev. of 7 runs, 10 loops each)
7.41 µs ± 244 ns per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Recursão e Iteração: Considerações sobre Eficiência

Sumário

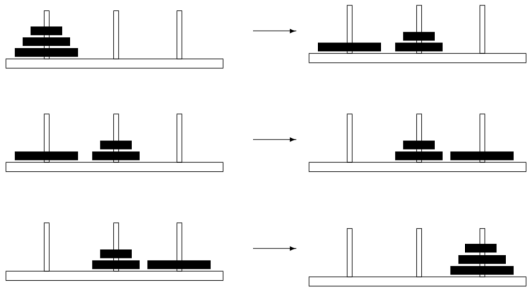
- A minimização dos recursos computacionais consumidos por um programa é um dos aspetos que nos preocupa quando escrevemos programas.
- Diferenças na evolução dos processos levam a diferenças nos recursos computacionais consumidos:
  - **Tempo** que um programa demora a executar (número de passos atómicos 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	$O(2^n)$	O(n)

Recursão em Árvore

Exemplo 1: Torres de Hanoi

- A movimentação de n discos pode ser definida em função da moviementação de n-1 discos



In [27]:
```
def mover(n, ori, dest, aux):
    def mover_disco(ori, dest):
        print(ori, "->", dest)

    if n == 1:
        mover_disco(ori, dest)
    else:
        mover(n-1, ori, aux, dest)
        mover_disco(ori, dest)
        mover(n-1, aux, dest, ori)

mover(3, 'E', 'D', 'C')
```

```
E -> D
E -> C
D -> C
E -> D
C -> E
C -> D
E -> D
```

Otimização de Operações

Exemplo 2: Potência rápida recursiva

$$x^n = \begin{cases} x & \text{se } n = 1 \\ x.(x^{n-1}) & \text{se } n \text{ for ímpar} \\ (x^{n/2})^2 & \text{se } n \text{ for par} \end{cases}$$

- Esta função gera um processo computacional com um padrão de crescimento em tempo e espaço `O(log(n))`
- Visualizar no Python Tutor

In [13]:

```
def potencia(x, n):
    pot = 1
    while n > 0:
        pot = pot * x
        n = n - 1

    return pot


def potencia_rapida(x, n):
    if n == 1:
        return x
    else:
        if n % 2 != 0:
            return x * potencia_rapida(x, n-1)
        else:
            return potencia_rapida(x, n//2)**2

print(100)
%timeit -n 100 potencia(2,100)
%timeit -n 100 potencia_rapida(2,100)
print(400)
%timeit -n 100 potencia(2,400)
%timeit -n 100 potencia_rapida(2,400)
```

```
100
9.13 µs ± 560 ns per loop (mean ± std. dev. of 7 runs, 100 loops each)
2.95 µs ± 101 ns per loop (mean ± std. dev. of 7 runs, 100 loops each)
400
41.2 µs ± 1.53 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
3.89 µs ± 131 ns per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Recursão em Árvore

Exemplo 3: Soma elementos atómicos

5. Escreva a função recursiva **soma_els_atomicos** que recebe como argumento um tuplo, cujos elementos podem ser outros tuplos, e que devolve a soma dos elementos correspondentes a tipos elementares de dados que existem no tuplo original. Não é necessário verificar os dados de entrada. Por exemplo,

```
>>> soma_els_atomicos((3, ((((((6, (7, ))), ), ), ), ), 2, 1))
19
>>> soma_els_atomicos(((((),),),))
0
```

In [17]:
```python
# versão recursão múltipla
def soma_els_atomicos_rm(t):
    if not t:
        return 0
    else:
        if type(t[0]) == tuple:
            return soma_els_atomicos_rm(t[0]) + soma_els_atomicos_rm(t[1:])
        else:
            return t[0] + soma_els_atomicos_rm(t[1:])
```

In [123…
```python
# versão recursão linear
def soma_els_atomicos_rl(t):
    if not t:
        return 0
    else:
        if type(t[0]) == tuple:
            return soma_els_atomicos_rl(t[0]+t[1:])
        else:
            return t[0] + soma_els_atomicos_rl(t[1:])
```

Tarefas Próxima Semana

- Estudar matéria e completar exemplos

- A **Ficha 5** da próxima semana é sobre **recursão**
  - primeira aula prática da semana

- **ATENÇÃO: Hoje** prazo de entrega do 1º projeto, **17h00**!!

- Teóricas da próxima semana: Funções de ordem superior
  - matéria para a Ficha 6, na segunda aula prática