

Departamento de Engenharia de Sistemas e Computação

Recursão

Funções que chamam a si mesmas

João Araujo

Algoritmos Computacionais I

Aula C1 (Eng. de Computação)

Aula extra para Engenharia de Computação

Onde se encaixa

Esta aula e a seguinte (C2: Ordenação) correspondem aos capítulos **7** e **8** do livro e **não** fazem parte do curso genérico para as demais engenharias. Sugestão: após a Aula 23 (Arquivos) ou como Bloco 3B, antes da Prova 2.

Roteiro

- A fila de programadores míopes (Cap. 7.1)
- Cálculo recursivo do tamanho de uma lista (7.2)
- Modelo de execução: a pilha (7.3)
- Coelhos de Fibonacci (7.4)
- Eficiência da recursão e *memoização* (7.5)

A fila de programadores míopes

Você está numa fila enorme e quer saber sua **posição**, mas não enxerga ninguém.

1. Você pergunta à pessoa da frente: "qual a sua posição?"
2. Ela pergunta à da frente, e assim por diante...
3. O primeiro da fila responde: "sou o número 1"
4. Cada um volta somando 1 à resposta recebida
5. Você recebe a resposta e soma 1

A estrutura da solução

Se você souber a resposta, responda. Senão, faça *a mesma pergunta* à pessoa da frente, some 1 e responda.

Poucas pessoas sabiam sua posição, mas o **problema se reduziu** a uma versão menor do mesmo problema.

Anatomia de uma função recursiva

1. Caso base

A situação *trivial*, que se resolve sem nova chamada (o primeiro da fila).

2. Passo recursivo

Reduz o problema a uma versão *menor* do mesmo e usa a resposta dela.

Sem caso base (ou sem avançar para ele)...

a função chama a si mesma indefinidamente até estourar a pilha: `RecursionError: maximum recursion depth exceeded`.

Receita: (1) qual é o caso mais simples? (2) como reduzir o problema? (3) confie que a chamada recursiva já resolve o problema menor.

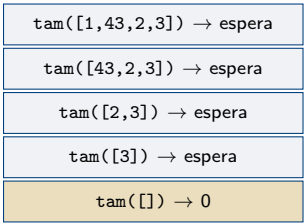
Tamanho de uma lista, recursivamente

```
def tam(f):
    """Calcula tamanho da lista f."""
    if f == []:
        return 0              # caso base: lista vazia
    else:
        return 1 + tam(f[1:]) # 1 + tamanho do resto

def main():
    fila = [1, 43, 2, 3]
    tamanho = tam(fila)
    print('O tamanho da fila é', tamanho)

main()
```

- $\text{tam}([1, 43, 2, 3]) = 1 + \text{tam}([43, 2, 3]) = 1 + 1 + \text{tam}([2, 3]) = \cdots = 4$
- `f[1:]` (fatia, Aula 17) é a "fila sem o primeiro"

Fonte: `code/Cap7/prog.7.01_len_recursivo.py`

Modelo de execução: a pilha de chamadas



- A **pilha** guarda, para cada chamada: os *parâmetros* e o *endereço de retorno*
- Como uma pilha de pratos: só se retira o do **topo**
- Ao atingir o caso base, as chamadas *retornam* uma a uma: $0 \to 1 \to 2 \to 3 \to 4$
- Cada chamada tem suas **próprias variáveis locais**

Python limita a profundidade (cerca de 1000 chamadas): `sys.getrecursionlimit()`.

Acompanhando a recursão com `print`

```
def tam(f, nivel=0):
    print('  ' * nivel + 'tam(', f, ')')
    if f == []:
        resultado = 0
    else:
        resultado = 1 + tam(f[1:], nivel + 1)
    print('  ' * nivel + 'devolve', resultado)
    return resultado

tam([7, 8, 9])
```

A indentação mostra a profundidade da pilha: o "vai" (chamadas) e o "volta" (retornos). Faça o mesmo *teste de mesa* no papel antes de rodar.

Fatorial recursivo (Exercício 7.4)

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

```python
def fatorial(n):
    if n == 0:              # caso base
        return 1
    return n * fatorial(n - 1)

print(fatorial(5))          # 120
```

- A definição matemática *já é* recursiva: a tradução para código é direta
- Compare com a versão iterativa da Aula 8: mesmo resultado; a iterativa não usa a pilha

Mais exemplos: soma e string invertida (Exercícios 7.1 e 7.6)

```python
def soma(n):
    """Soma dos n primeiros números: 1 + 2 + ... + n."""
    if n == 0:
        return 0
    return n + soma(n - 1)

def inverte(s):
    """Devolve a string s invertida."""
    if s == '':
        return ''
    return inverte(s[1:]) + s[0]

print(soma(10))         # 55
print(inverte('abc'))   # cba
```

Pense: qual o caso base? Como o problema *diminui* a cada chamada?

Palíndromo recursivo (Exercícios 7.2 e 7.3)

```python
def palindromo(s):
    """Diz se a palavra s é palíndromo."""
    if len(s) < 2:                  # 0 ou 1 letra: é palíndromo
        return True
    if s[0] != s[-1]:               # extremos diferentes: não é
        return False
    return palindromo(s[1:-1])      # compara o "miolo"

print(palindromo('arara'))   # True
print(palindromo('ab'))      # False
```

- Dois casos base; o problema encolhe de dois em dois caracteres
- Exercício 7.3: para frases, ignore espaços e maiúsculas *sem* usar laços

Coelhos de Fibonacci

Um casal de coelhos jovens leva 1 mês para amadurecer e depois gera 1 casal por mês. Quantos casais após n meses?

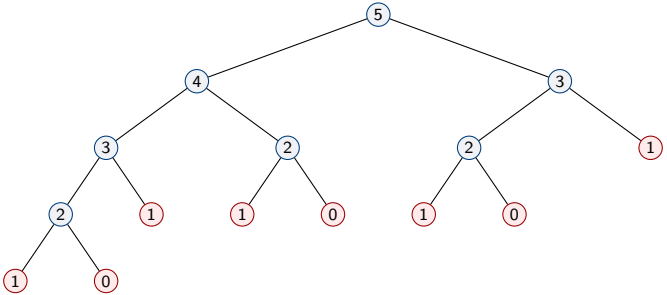
$$F_n = F_{n-1} + F_{n-2}, \quad F_0 = 0,\ F_1 = 1$$

n	0	1	2	3	4	5	6	7	8	9
F_n	0	1	1	2	3	5	8	13	21	34

```python
def fib(n):
    """n-ésimo termo de Fibonacci."""
    if n == 0 or n == 1:      # casos base
        return n
    else:
        return fib(n - 1) + fib(n - 2)

for i in range(10):
    print(fib(i), end=' ')
```

- Duas chamadas recursivas por vez
- Aparece na natureza, na arquitetura e em análise de algoritmos

Árvore de chamadas de `fib(5)`



`fib(3)` é calculado 2 vezes, `fib(2)` 3 vezes... O *mesmo* subproblema é refeito muitas vezes.

Eficiência: o custo da recursão

- Número de chamadas de `fib(n)`: chamadas$(n) = 2F_{n+1} - 1$
- Cresce exponencialmente

n	Chamadas
10	177
20	21 891
25	242 785
35	29 860 703
50	$\approx 4 \times 10^{10}$

Custo = tempo + memória

Recursão consome memória de pilha e pode repetir cálculos. Para n grande, é *impraticável* sem melhorias.

Experimente

Rode o Programa 7.5 do livro com `range(100)` e veja a partir de que termo ele "trava".

Memoização: guardar o que já foi calculado

```python
fibonacci = {0: 0, 1: 1}      # tabela de valores já calculados

def fib(n):
    """Calcula o n-ésimo termo da série de Fibonacci."""
    if n not in fibonacci:
        fibonacci[n] = fib(n - 2) + fib(n - 1)
    return fibonacci[n]

for i in range(100):
    print('fib' + str(i) + '):' + str(fib(i)))
```

- Cada termo é calculado uma única vez: de exponencial para **linear**
- Continua recursivo, agora com a ajuda de um **dicionário** (Aula 20)
- Exercício 7.5: reescreva usando uma *lista* no lugar do dicionário (justifique a escolha!)

Fonte: `code/Cap7/prog.7.06_calcula_fibonacci4.py`

Versão iterativa e `functools.cache`

```python
def fib_iterativo(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

from functools import cache

@cache                          # memoização automática
def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)

print(fib(90), fib_iterativo(90))
```

- A versão iterativa usa memória constante e não sofre limite de pilha
- `@cache` (decorador) faz a memoização por você

Recursão ou iteração?

Recursão é natural quando...	**Prefira iteração quando...**
a definição do problema é recursiva (fatorial, Fibonacci, árvores, diretórios)	o laço é simples (somar, contar, percorrer)
o problema se divide em subproblemas *iguais* (divisão e conquista: *mergesort*)	a profundidade pode ser grande (limite de pilha)
é preciso *voltar atrás* (*backtracking*, labirintos)	o desempenho é crítico

Toda recursão pode virar iteração (e vice-versa): a escolha é de **clareza** e **eficiência**.

Atividades

1. Escreva `potencia(base, n)` recursiva ($base^n = base \cdot base^{n-1}$). Depois uma versão rápida: $base^n = (base^{n/2})^2$ para n par.
2. Escreva `soma_digitos(n)`: `soma_digitos(1234)` $= 10$.
3. Escreva `conta(lista, x)`: quantas vezes x aparece na lista (recursivamente).
4. Escreva `binario(n)` que devolve a string binária de n ($n//2$ e $n\%2$). Conecte com a Aula 2!
5. *Torre de Hanói*: escreva `hanoi(n, origem, destino, aux)` que imprime os movimentos.

Para praticar: exercícios do livro

Site do livro – Capítulo 7

- **7.1** (string na ordem inversa), **7.2** (palíndromo recursivo), **7.3** (palíndromo em frases)
- **7.4** (fatorial recursivo), **7.5** (Fibonacci com lista), **7.6** (soma recursiva)

Enunciados e soluções comentadas em
`https://www.araujo.eng.uerj.br/livro/exercicios/cap7/`

Exercícios adicionais (sem solução): `https://www.araujo.eng.uerj.br/livro/extras/`

Resumo da aula

- Recursão: uma função que chama a si mesma para resolver versões menores do problema
- Todo algoritmo recursivo precisa de um **caso base** e de um passo que **se aproxima** dele
- As chamadas empilham-se na **pilha**; cada uma tem suas variáveis
- Chamadas repetidas custam caro: use **memoização** (dicionário) ou iteração
- Próxima aula: ordenação de listas e o poder de *dividir e conquistar*