Estruturas de Dados e Algoritmos II

Vasco Pedro

Departamento de Informática
Universidade de Évora

2021/2022

Pseudo-código

Exemplo

PESQUISA-LINEAR(V, k)

```
n <- |V|                                 // inicialização
i <- 1

while i <= n and V[i] != k do            // pesquisa
    i <- i + 1

if i <= n then                           // resultado:
    return i                             //  - sucesso

return INEXISTENTE                       //  - insucesso
```

`|V|`	nº de elementos de um vector — $O(1)$
`V[1..|V|]`	elementos do vector
`and` e `or`	só é avaliado o segundo operando se necessário
`variável.campo`	acesso a um campo de um "objecto"

(`INEXISTENTE` é uma constante, com valor -1, por exemplo)

Análise da complexidade (1)

Exemplo

Análise da complexidade temporal, no pior caso, da função PESQUISA-LINEAR, por linha de código

1. Obtenção da dimensão de um vector, afectação: operações com complexidade (temporal) constante

$$O(1) + O(1) = O(1)$$

2. Afectação: $O(1)$
3. Acessos a `i`, `n`, `V[i]` e `k`, comparações e saltos condicionais com complexidade constante

$$4\,O(1) + 2\,O(1) + 2\,O(1) = O(1)$$

Executada, no pior caso, $|\mathtt{V}|+1$ vezes

$$(|\mathtt{V}| + 1) \times O(1) = O(|\mathtt{V}|)$$

Análise da complexidade (2)

Exemplo

4. Acesso a `i`, soma e afectação: $O(1) + O(1) + O(1) = O(1)$
   Executada, no pior caso, `|V|` vezes

$$|\mathtt{V}| \times O(1) = O(|\mathtt{V}|)$$

5. Acesso a `i` e `n`, comparação e salto condicional com complexidade constante

$$2\,O(1) + O(1) + O(1) = O(1)$$

6. Saída de função com complexidade constante: $O(1)$
7. Saída de função com complexidade constante: $O(1)$

Análise da complexidade (3)

Exemplo

Juntando tudo

$$\begin{aligned} O(1) + O(1) + O(\texttt{|V|}) + O(\texttt{|V|}) + O(1) + \max\{O(1), O(1)\} &= \\ = 4\,O(1) + 2\,O(\texttt{|V|}) &= \\ = O(\texttt{|V|}) \end{aligned}$$

No pior caso, a função PESQUISA-LINEAR tem complexidade temporal linear na dimensão do vector `V`

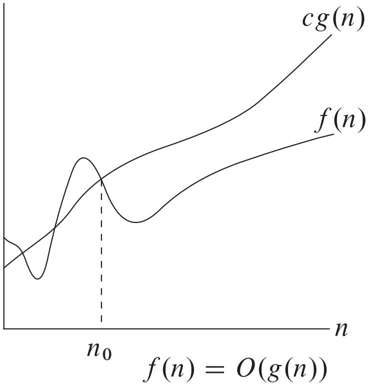
Se n representar a dimensão do vector `V`, o tempo $T(n)$ que a função demora a executar tem complexidade linear em n

$$T(n) = O(n)$$

Isto significa que o tempo que a função demora a executar varia linearmente com a dimensão do *input*

A notação O (1)

$$O(g(n)) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0}\ 0 \leq f(n) \leq c\, g(n)\}$$



A notação O (2)

$$O(g(n)) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n\geq n_0}\ 0 \leq f(n) \leq c\, g(n)\}$$

- $O(n) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n\geq n_0}\ 0 \leq f(n) \leq c\, n\}$

  $n = O(n) \qquad 2n+5 = O(n) \qquad \log n = O(n) \qquad n^2 \neq O(n)$

- $O(n^2) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n\geq n_0}\ 0 \leq f(n) \leq c\, n^2\}$

  $n^2 = O(n^2) \qquad 4n^2+n = O(n^2) \qquad n = O(n^2) \qquad n^3 \neq O(n^2)$

- $O(\log n) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n\geq n_0}\ 0 \leq f(n) \leq c\, \log n\}$

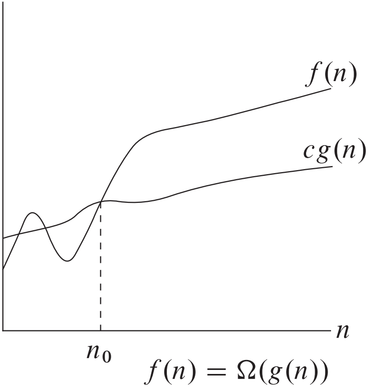
  $1 + \log n = O(\log n) \qquad \log n^2 = O(\log n) \qquad n \neq O(\log n)$

$f(n) = O(g(n))$ significa $f(n) \in O(g(n))$

Lê-se $f(n)$ é O de $g(n)$

A notação O (3)

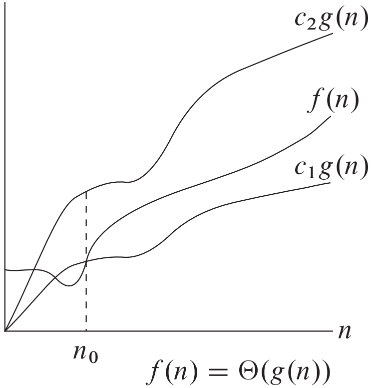
$$\Omega(g(n)) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n\geq n_0}\ 0 \leq c\,g(n) \leq f(n)\}$$



$$n = \Omega(n) \qquad n^2 = \Omega(n) \qquad \log n \neq \Omega(n^2)$$

A notação O (4)

$$\Theta(g(n)) = \{f(n) : \exists_{c_1, c_2, n_0 > 0} \text{ t.q. } \forall_{n \geq n_0}\ 0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)\}$$



$3n^2 + n = \Theta(n^2) \qquad n \neq \Theta(n^2) \qquad n^2 \neq \Theta(n)$

A notação O (5)

$$o(g(n)) = \{f(n) : \forall_{c>0}\ \exists_{n_0>0} \text{ tal que } \forall_{n\geq n_0}\ 0 \leq f(n) < c\,g(n)\}$$

$$n = o(n^2) \qquad n^2 \neq o(n^2) \qquad \log n = o(n)$$

$$\forall_{k>0}\ n^k = o(2^n)$$

$$\forall_{k>0}\ \forall_{b>1}\ n^k = o(b^n)$$

$$\omega(g(n)) = \{f(n) : \forall_{c>0}\ \exists_{n_0>0} \text{ tal que } \forall_{n\geq n_0}\ 0 \leq c\,g(n) < f(n)\}$$

$$n = \omega(\log n) \qquad n^2 = \omega(\log n) \qquad \log n \neq \omega(\log n)$$

$$\forall_{k>0}\ 2^n = \omega(n^k)$$

$$\forall_{b>1}\ \forall_{k>0}\ b^n = \omega(n^k)$$

A notação O (6)

Traduzindo...

$f(n) = O(g(n))$ $f(n)$ não cresce mais depressa que $g(n)$

$f(n) = o(g(n))$ $f(n)$ cresce mais devagar que $g(n)$

$f(n) = \Omega(g(n))$ $f(n)$ não cresce mais devagar que $g(n)$

$f(n) = \omega(g(n))$ $f(n)$ cresce mais depressa que $g(n)$

$f(n) = \Theta(g(n))$ $f(n)$ e $g(n)$ crescem ao mesmo ritmo

Ainda a pesquisa linear

De um valor num vector ordenado

PESQUISA-LINEAR-ORD(V, k)

```
n <- |V|                             // inicialização
i <- 1

while i <= n and V[i] < k do         // pesquisa
    i <- i + 1

if i <= n and V[i] = k then          // resultado:
    return i                         //  - sucesso

return INEXISTENTE                   //  - insucesso
```

Pesquisa dicotómica ou binária

De um valor num vector ordenado

PESQUISA-DICOTÓMICA(V, k)

```
n <- |V|
return PESQUISA-DICOTÓMICA-REC(V, k, 1, n)
```

PESQUISA-DICOTÓMICA-REC(V, k, i, f)

```
if i > f then
    return INEXISTENTE       // intervalo vazio

m <- (i + f) / 2

if k < V[m] then
    return PESQUISA-DICOTÓMICA-REC(V, k, i, m - 1)

if k > V[m] then
    return PESQUISA-DICOTÓMICA-REC(V, k, m + 1, f)

return m                     // V[m] = k
```

Complexidade das pesquisas linear e dicotómica

Pior caso e caso esperado para a complexidade temporal das pesquisas num vector de dimensão n

Pesquisa linear

$$T(n) = O(n)$$

Pesquisa linear num vector ordenado

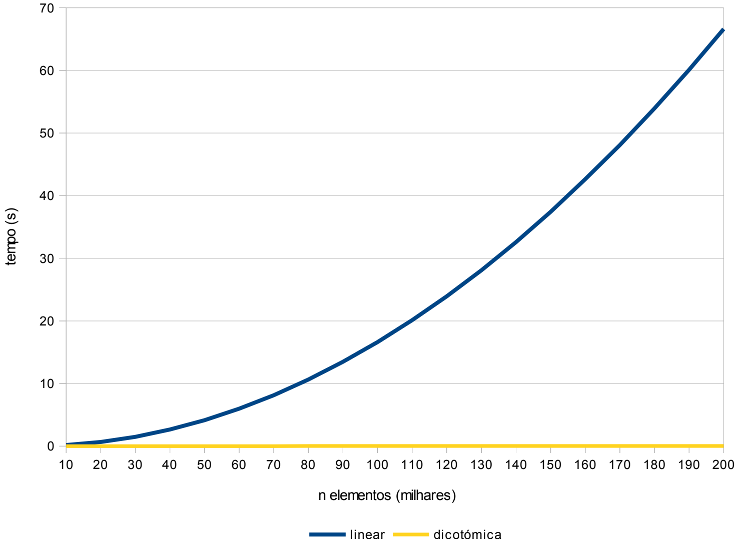
$$T(n) = O(n)$$

Pesquisa dicotómica

$$T(n) = O(\log n)$$

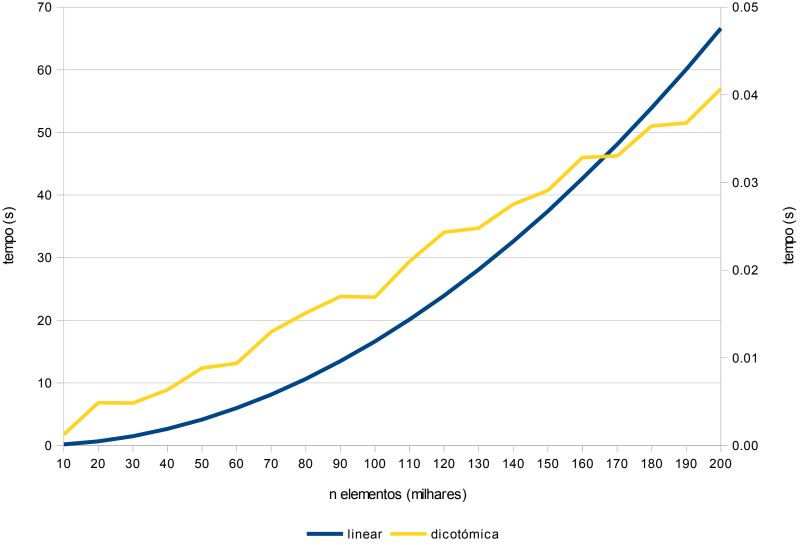
Pesquisas linear e dicotómica

Dos *n* elementos de um vector



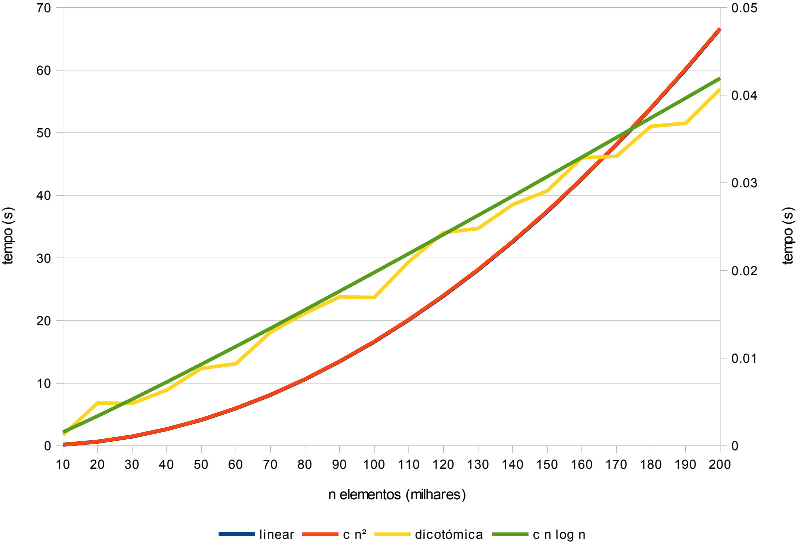
Pesquisas linear e dicotómica (com escalas diferentes)

Dos n elementos de um vector



Pesquisas linear e dicotómica (com escalas diferentes)

Dos *n* elementos de um vector



Números de Fibonacci

Versão recursiva

```
public static int fibonacci(int n)
{
  if (n == 0)
    return 0;

  if (n == 1)
    return 1;

  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Números de Fibonacci

Versão iterativa com tabelação

```
public static int fibonacci(int n)
{
  int[] tabela = new int[n + 1];

  // casos base
  tabela[0] = 0;
  tabela[1] = 1;

  for (int i = 2; i <= n; ++i)
    tabela[i] = tabela[i - 1] + tabela[i - 2];

  return tabela[n];
}
```

Números de Fibonacci

Versão iterativa

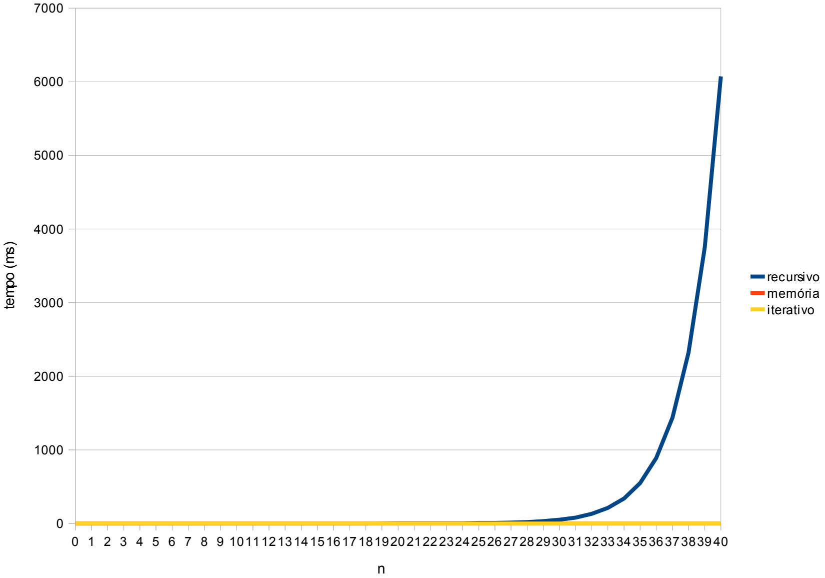
```
public static int fibonacci(int n)
{
  int i = 0;

  int corrente = 0;      // fibonacci(i)
  int anterior = 1;      // fibonacci(i - 1)

  while (i < n)
    {
      // fibonacci(i + 1)
      int proximo = corrente + anterior;

      anterior = corrente;
      corrente = proximo;

      i++;
    }

  return corrente;
}
```

Números de Fibonacci

Versão recursiva com memória

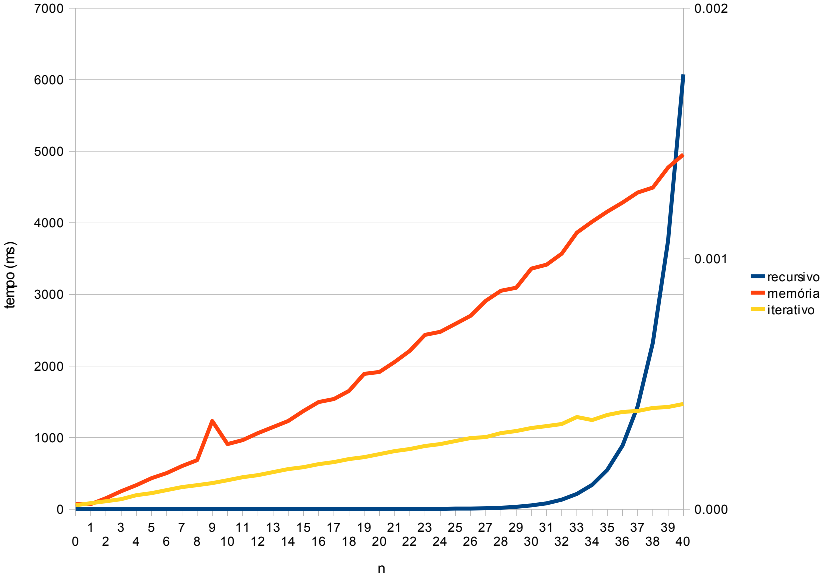
```
private static int CARDINAL_DOMINIO = ...;
private static int[] memoria;
static {
  memoria = new int[CARDINAL_DOMINIO];
  memoria[1] = 1;
}

public static int fibonacci(int n)
{
  if (n > 1 && memoria[n] == 0)
    memoria[n] = fibonacci(n - 1) + fibonacci(n - 2);

  return memoria[n];
}
```

Números de Fibonacci



Números de Fibonacci

Escalas diferentes



Números de Fibonacci

Escalas diferentes