

CES-11

o Noções de complexidade de algoritmos

- Complexidade de algoritmos
- Avaliação do tempo de execução
- Razão de crescimento desse tempo
- Notação O
- Exercícios

COMPLEXIDADE DE ALGORITMOS

o Importância de análise de algoritmos

- Um **mesmo problema** pode, em muitos casos, ser resolvido por **vários algoritmos**.
- Nesse caso, qual algoritmo deve ser o escolhido?

o Critério 1: fácil compreensão, codificação e correção

- Geralmente, são algoritmos ineficientes
- Este critério seria adequado se o algoritmo fosse executado poucas vezes:  
Custo de programação (Cp) > Custo de execução (Ce)

COMPLEXIDADE DE ALGORITMOS

o Critério 2: eficiência no uso dos recursos computacionais e rapidez na execução

- Geralmente, são algoritmos mais complicados
- Critério mais adequado para o caso de algoritmos muito utilizados ( Cp < Ce )

o Medição de eficiência: uso de memória e tempo de execução

- Nesta disciplina, a ênfase é dada para algoritmos eficientes no **tempo de execução**.
- Há muitos casos em que algoritmos simples **não** são executados em **tempo viável**

COMPLEXIDADE DE ALGORITMOS

o Exemplo: Regra de Cramer para a solução de sistemas de equações lineares

$$\begin{matrix} A_{11} x_1 + A_{12} x_2 + \dots + A_{1n} x_n = B_1 \\ A_{21} x_1 + A_{22} x_2 + \dots + A_{2n} x_n = B_2 \\ \vdots \\ A_{n1} x_1 + A_{n2} x_2 + \dots + A_{nn} x_n = B_n \end{matrix}$$

$$X_i = \frac{\begin{vmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,i-1} & B_i & A_{1,i+1} & \dots & A_{1,n} \\ A_{2,1} & A_{2,2} & \dots & A_{2,i-1} & B_i & A_{2,i+1} & \dots & A_{2,n} \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ A_{n,1} & A_{n,2} & \dots & A_{n,i-1} & B_i & A_{n,i+1} & \dots & A_{n,n} \end{vmatrix}}{\begin{vmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,i-1} & A_{1,i} & A_{1,i+1} & \dots & A_{1,n} \\ A_{2,1} & A_{2,2} & \dots & A_{2,i-1} & A_{2,i} & A_{2,i+1} & \dots & A_{2,n} \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ A_{n,1} & A_{n,2} & \dots & A_{n,i-1} & A_{n,i} & A_{n,i+1} & \dots & A_{n,n} \end{vmatrix}}$$

COMPLEXIDADE DE ALGORITMOS

o Considerando n = 20, quantos determinantes seriam calculados?

- 20 para os numeradores e 1 para o denominador

$$X_i = \frac{\begin{vmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,i-1} & B_i & A_{1,i+1} & \dots & A_{1,n} \\ A_{2,1} & A_{2,2} & \dots & A_{2,i-1} & B_i & A_{2,i+1} & \dots & A_{2,n} \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ A_{n,1} & A_{n,2} & \dots & A_{n,i-1} & B_i & A_{n,i+1} & \dots & A_{n,n} \end{vmatrix}}{\begin{vmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,i-1} & A_{1,i} & A_{1,i+1} & \dots & A_{1,n} \\ A_{2,1} & A_{2,2} & \dots & A_{2,i-1} & A_{2,i} & A_{2,i+1} & \dots & A_{2,n} \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ A_{n,1} & A_{n,2} & \dots & A_{n,i-1} & A_{n,i} & A_{n,i+1} & \dots & A_{n,n} \end{vmatrix}}$$

COMPLEXIDADE DE ALGORITMOS

o Cálculo do número m de multiplicações:

$$\begin{aligned} 21 \det_{20} &= 21 ( 20m + 20 \det_{19} ) \\ &= 21 ( 20m + 20 ( 19m + 19 \det_{18} ) ) \\ &= 21 ( 20m + 20 ( 19m + 19 ( 18m + 18 \det_{17} ) ) ) \\ &= 21 ( 20m + 20 ( 19m + 19 ( 18m + 18 ( 17m + 17 ( \dots ( 3m + 3 ( 2m ) \dots ) ) ) ) ) ) \end{aligned}$$

Lembrando do cálculo recursivo de determinante:

$$\det \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,1} & a_{n-1,2} & \dots & a_{n-1,n} \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = \sum_{j=1}^n a_{ij} (-1)^{i+j} \cdot \det A_{-i,-j}$$

### COMPLEXIDADE DE ALGORITMOS

$$= 21 ( 20m + 20 ( 19m + 19 ( 18m + 18 ( 17m + 17 ( \dots ( 3m + 3 ( 2m ) \dots ) ) ) ) ) ) )$$

$$= m ( 2 \times 3 \times 4 \times 5 \times \dots \times 20 \times 21 + 3 \times 4 \times 5 \times \dots \times 20 \times 21 + 4 \times 5 \times \dots \times 20 \times 21 + 5 \times \dots \times 20 \times 21 + \dots + 19 \times 20 \times 21 + 20 \times 21 )$$

$$21! \left( 1 + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{19!} \right) \cong 8.778 \times 10^{19} \text{ multiplicações}$$

### COMPLEXIDADE DE ALGORITMOS

$$21! \left( 1 + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{19!} \right) \cong 8.778 \times 10^{19} \text{ multiplicações}$$

- Utilizando um supercomputador atual:
  - 10<sup>11</sup> multiplicações por segundo
  - Tempo gasto: 8,778 x 10<sup>8</sup> s = 27,8 anos!
- Um algoritmo mais eficiente é o Método da Eliminação de Gauss
  - 209 divisões + 2850 multiplicações

### CES-11

- Noções de complexidade de algoritmos
  - Complexidade de algoritmos
  - Avaliação do tempo de execução
  - Razão de crescimento desse tempo
  - Notação O
  - Exercícios

### AVALIAÇÃO DO TEMPO DE EXECUÇÃO

- Ex: método de ordenação *Bubble-Sort*

```
void BubbleSort (int n, vetor A) {
  int i, p, trocou; float aux;
  p = n - 1; trocou = TRUE;
  while (p >= 1 && trocou) {
    trocou = FALSE; i = 1;
    while (i <= p) {
      if (A[i] > A[i+1]) {
        aux = A[i];
        A[i] = A[i+1];
        A[i+1] = aux;
        trocou = TRUE;
      }
      i = i + 1;
    }
    p = p - 1; } }
```

### Avaliação do tempo de execução

- Ex: método de ordenação *Bubble-Sort*

```
void BubbleSort (int n, vetor A) {
  int i, p, trocou; float aux;
  p = n - 1; trocou = TRUE;
  while (p >= 1 && trocou) {
    trocou = FALSE; i = 1;
    while (i <= p) {
      if (A[i] > A[i+1]) {
        aux = A[i];
        A[i] = A[i+1];
        A[i+1] = aux;
        trocou = TRUE;
      }
      i = i + 1;
    }
    p = p - 1; } }
```

### Avaliação do tempo de execução

Operação	Tempo(ns)	Operação	Tempo(ns)	Operação	Tempo(ns)
Atrib int	1	+-<< <= int	2	+-<< <= float	15
Atrib float	2	&&	1,5	[]	8
* int	5	* float	20		
/ int	8	/ float	30		

```
void BubbleSort (int n, vetor A) {
  int i, p, trocou; float aux;
  p = n - 1; trocou = TRUE;
  while (p >= 1 && trocou) {
    trocou = FALSE; i = 1;
    while (i <= p) {
      if (A[i] > A[i+1]) {
        aux = A[i]; A[i] = A[i+1];
        A[i+1] = aux; trocou = TRUE;
      }
      i = i + 1;
    }
    p = p - 1; } }
```

Obs: supomos que vetor A contém números do tipo float

Mas quantas vezes cada trecho será executado?

## Avaliação do tempo de execução

- Análise do pior caso
  - Ocorrerá quando o teste do `if` for sempre verdadeiro
  - O que isso significa isso? **Vetor em ordem decrescente**

```
void BubbleSort (int n, vetor A)
int i, p, trocou; float aux;
p = n - 1; trocou = TRUE;
while (p >= 1 && trocou) {
  trocou = FALSE; i = 1;
  while (i <= p) {
    if (A[i] > A[i+1]) {
      aux = A[i]; A[i] = A[i+1];
      A[i+1] = aux; trocou = TRUE;
    }
    i = i + 1;
  }
  p = p - 1; } }
```

4 ns executado 1 vez  
 3,5 ns executado n vezes  
 2 ns executado n-1 vezes  
 2 ns executado p+1 vezes a cada iteração do while externo  
 79 ns executado p vezes a cada iteração do while externo  
 3 ns executado n-1 vezes

## Avaliação do tempo de execução

- Análise do pior caso
  - Ocorrerá quando o teste do `if` for sempre verdadeiro
  - O que isso significa isso? **Vetor em ordem decrescente**

```
void BubbleSort (int n, vetor A)
int i, p, trocou; float aux;
p = n - 1; trocou = TRUE;
while (p >= 1 && trocou) {
  trocou = FALSE; i = 1;
  while (i <= p) {
    if (A[i] > A[i+1]) {
      aux = A[i]; A[i] = A[i+1];
      A[i+1] = aux; trocou = TRUE;
    }
    i = i + 1;
  }
  p = p - 1; } }
```

4 ns executado 1 vez  
 3,5 ns executado n vezes  
 2 ns executado n-1 vezes  
 2 ns Total = n + (n-1) + ... + 3 + 2  
 79 ns Total = (n-1) + ... + 3 + 2 + 1  
 3 ns executado n-1 vezes

## Avaliação do tempo de execução

$$T(n) = 4 + 3,5n + 2(n-1) + 2(n + (n-1) + \dots + 3 + 2) + 79((n-1) + (n-2) + \dots + 2 + 1) + 3(n-1)$$

```
void BubbleSort (int n, vetor A)
int i, p, trocou; float aux;
p = n - 1; trocou = TRUE;
while (p >= 1 && trocou) {
  trocou = FALSE; i = 1;
  while (i <= p) {
    if (A[i] > A[i+1]) {
      aux = A[i]; A[i] = A[i+1];
      A[i+1] = aux; trocou = TRUE;
    }
    i = i + 1;
  }
  p = p - 1; } }
```

4 ns executado 1 vez  
 3,5 ns executado n vezes  
 2 ns executado n-1 vezes  
 2 ns Total = n + (n-1) + ... + 3 + 2  
 79 ns Total = (n-1) + ... + 3 + 2 + 1  
 3 ns executado n-1 vezes

## Avaliação do tempo de execução

- Pior caso do *Bubble-Sort*:

$$T(n) = 4 + 3,5n + 2(n-1) + 2(n + (n-1) + \dots + 3 + 2) + 79((n-1) + (n-2) + \dots + 2 + 1) + 3(n-1)$$

$$T(n) = 40,5n^2 - 30n - 3$$

- Quando  $n$  aumenta indefinidamente, o termo com  $n^2$  predomina sobre os demais
- $T(n)$  é proporcional a  $n^2$
- Também há casos melhores: nem todos os testes do comando `if` são sempre verdadeiros

## Avaliação do tempo de execução

- Casos mais estudados:
  - Pior caso
  - Caso médio
- Razões para estudar o pior caso:
  - O tempo de execução do pior caso de um algoritmo é o *limite superior* do tempo de execução para uma entrada qualquer.
  - Para alguns algoritmos, o pior caso ocorre com bastante frequência.
  - Geralmente, o caso médio não é fácil de ser calculado. Várias vezes, ele é quase tão ruim quanto o pior caso.

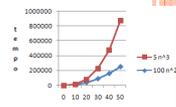
## CES-11

- Noções de complexidade de algoritmos
  - Complexidade de algoritmos
  - Avaliação do tempo de execução
  - Razão de crescimento desse tempo
  - Notação O
  - Exercícios

RAZÃO DE CRESCIMENTO DESSE TEMPO

- Ex1: Sejam A1 e A2 dois algoritmos que resolvem o mesmo problema, e com os respectivos tempos de execução:

- $T_1(n) = 100n^2$
  - $T_2(n) = 5n^3$



- Qual desses algoritmos é o melhor?

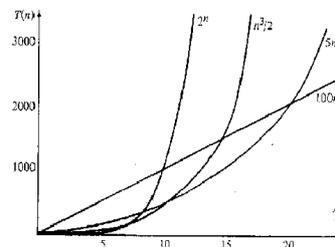
- Dependerá do valor de n
  - Empate:  $100n^2 = 5n^3 \Rightarrow n = 20$
  - Para  $n < 20$ , A2 é melhor
  - Para  $n > 20$ , A1 é melhor

A1 é considerado o melhor

RAZÃO DE CRESCIMENTO DESSE TEMPO

- Ex2: Considere 4 algoritmos que resolvem o mesmo problema de tamanho n.
- Abaixo, seus respectivos tempos de execução:

- $T_1(n) = 100n$
  - $T_2(n) = 5n^2$
  - $T_3(n) = n^3/2$
  - $T_4(n) = 2^n$



RAZÃO DE CRESCIMENTO DESSE TEMPO

- Suponha que esse problema precise ser resolvido em no máximo 1.000 segundos.

T(n)	n para 10 <sup>3</sup> seg
100n	10
5n <sup>2</sup>	14,14
n <sup>3</sup> /2	12,60
2 <sup>n</sup>	9,97

- Os problemas solucionáveis pelos 4 algoritmos têm tamanho da mesma ordem de magnitude (em torno de 10).

RAZÃO DE CRESCIMENTO DESSE TEMPO

- Considerando uma máquina mais rápida, elevemos esse tempo para 10.000 segundos.

T(n)	n para 10 <sup>3</sup> seg	n para 10 <sup>4</sup> seg	Ganho
100n	10	100	10
5n <sup>2</sup>	14,14	44,72	3,16
n <sup>3</sup> /2	12,60	27,14	2,15
2 <sup>n</sup>	9,97	13,28	1,33

- Repare que o algoritmo 4 só poderá resolver um problema **1,33 vezes** maior...

CES-11

o Noções de complexidade de algoritmos

- Complexidade de algoritmos
- Avaliação do tempo de execução
- Razão de crescimento desse tempo
- Notação O
- Exercícios

NOTAÇÃO O

- Seja o tempo de execução de um algoritmo igual a uma somatória de termos (funções do tamanho da entrada):

- $T_1(n) = c_1n^3 + c_2n^2 + c_3n + c_4$
  - $T_2(n) = 2^n + n^3/2 + 5n^2 + 100n$

$T_1(n) = O(n^3)$

$T_2(n) = O(2^n)$

- À medida que n aumenta indefinidamente, um dos termos passa a ter domínio sobre os demais:

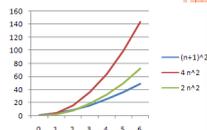
- $T_1(n)$  é proporcional a  $n^3$ : é da ordem de  $n^3$
  - $T_2(n)$  é proporcional a  $2^n$ : é da ordem de  $2^n$

### NOTAÇÃO O

▪ **Definição:**  $T(n) = O(f(n))$ , ou seja,  $T(n)$  é da ordem de  $f(n)$  se e somente se existirem **constantes positivas  $c$  e  $n_0$**  tais que, para qualquer  $n \geq n_0$ ,  $T(n) \leq c \cdot f(n)$

- Ex1:  $T(n) = (n+1)^2 = O(n^2)$ 
  - $(n+1)^2 \leq 4n^2, n \geq 1$  (basta escolher  $n_0 = 1$  e  $c = 4$ )
  - Poderia ser  $c = 2$ ? Sim, mas  $n_0 = 3$

n	0	1	2	3	4	5
$(n+1)^2$	1	4	9	16	25	36
$4n^2$	0	4	16	36	64	100
$2n^2$	0	2	8	18	32	50



### NOTAÇÃO O

▪ Ex2: Pior caso do *Bubble-Sort*  
 $T(n) = 40,5n^2 - 30n - 3 = O(n^2)$

- $T(n) \leq 40,5n^2$  (sendo  $n_0 = 1$  e  $c = 40,5$ )

▪ Ex3: Seja  $T(n)$  o polinômio de grau  $p \in \mathbb{N}^+$ , onde  $a_0 \geq 0$ :

$$\begin{aligned}
 T(n) &= a_0 \cdot n^p + a_1 \cdot n^{p-1} + \dots + a_{p-1} \cdot n + a_p \\
 &\leq a_0 \cdot n^p + |a_1| \cdot n^{p-1} + \dots + |a_{p-1}| \cdot n + |a_p| \\
 &\leq a_0 \cdot n^p + |a_1| \cdot n^p + \dots + |a_{p-1}| \cdot n^p + |a_p| \cdot n^p \\
 &\leq (a_0 + |a_1| + \dots + |a_{p-1}| + |a_p|) \cdot n^p \\
 &\leq c \cdot n^p, \text{ para } n \geq 1
 \end{aligned}$$

$T(n) = O(n^p)$

### NOTAÇÃO O

▪  $T(n) = O(f(n))$

- $f(n)$  é um limite superior para a taxa de crescimento de  $T(n)$
- Dado  $T(n)$ , temos uma única  $f(n)$ ?
  - Não, pois várias funções podem satisfazer a definição.
  - Exemplo:  $T(n) = 4n^2 + 3n + 1$
  - $T(n)$  é  $O(n^2)$ ,  $O(n^3)$ ,  $O(n^{10})$
  - No entanto,  $T(n)$  não é  $O(n)$
- Na análise de algoritmos, as taxas mais usadas são  $n^2$ ,  $n^3$ ,  $n^1$ ,  $\log n$ ,  $n \cdot \log n$ ,  $2^n$ ,  $3^n$ , etc.

Dentre todas as possíveis funções  $f(n)$ , o objetivo é encontrar a que tenha o menor crescimento possível

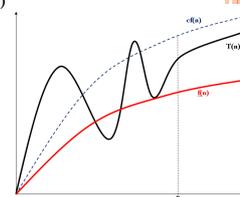
### NOTAÇÃO O

▪ Notações similares:

- $T(n) = \Omega(f(n))$ 
  - $T(n) \geq c \cdot f(n)$  para  $n \geq n_0$
  - $f(n)$  é um limite inferior para  $T(n)$
- $T(n) = \Theta(f(n))$ 
  - $T(n) \leq c_1 \cdot f(n)$  para  $n \geq n_0$
  - $T(n) \geq c_2 \cdot f(n)$  para  $n \geq n_0$
  - $f(n)$  é um limite inferior e superior de  $T(n)$

Big-Omega

Big-Teta



### CES-11

o Noções de complexidade de algoritmos

- Complexidade de algoritmos
- Avaliação do tempo de execução
- Razão de crescimento desse tempo
- Notação O
- Exercícios

### EXERCÍCIOS

o 1. Provar que  $n^3 \neq O(n^2)$

- Por absurdo, suponhamos que  $n^3 = O(n^2)$
- Pela definição, existem constantes positivas  $c$  e  $n_0$  tais que, para  $n \geq n_0$ ,  $n^3 \leq c \cdot n^2$
- Logo,  $c \geq n$ . Portanto, à medida que  $n$  cresce indefinidamente,  $c$  também crescerá
- Isso contraria a definição de  $c$  ser constante

EXERCÍCIOS

2. Provar que  $3^n \neq O(2^n)$

- Por absurdo, suponhamos que  $3^n = O(2^n)$
- Pela definição, existem constantes positivas  $c$  e  $n_0$  tais que, para  $n \geq n_0$ ,  $3^n \leq c \cdot 2^n$
- Logo,  $c \geq (3/2)^n$ . Portanto, à medida que  $n$  cresce indefinidamente,  $(3/2)^n$  também crescerá
- Assim, para qualquer valor de  $n$ , não existe uma constante que exceda  $(3/2)^n$

EXERCÍCIOS

3. Analisar o pior caso de algoritmo ao lado, que calcula o valor de  $n^n$

```
int func(int n) {
    int i, r;
    r = 1;
    for (i=1; i<=n; i++)
        r = r*n;
    return r;
}
```

```
int func(int n) {
    int i, r;
    r = 1;
    i = 1;
    while (i <= n) {
        r = r*n;
        i++;
    }
    return r;
}
```

$$T(n) = t_1 + t_2(n+1) + t_3 \cdot n$$

$$= t_1 + t_2 + t_2 \cdot n + t_3 \cdot n$$

$$= c_1 + c_2 \cdot n$$

$T(n) = O(n)$

EXERCÍCIOS

4. Analisar o pior caso do algoritmo abaixo:

```
int func (int n) {
    int a, b, c, r;
    b = n; c = n; r = 1;
    while (b >= 1) {
        a = b % 2;
        if (a == 1)
            r = r * c;
        c = c * c * c;
        b = b / 2;
    }
    return r;
}
```

n	b	iterações
2	2,1,0	2
3	3,1,0	2
4	4,2,1,0	3
16	16,8,4,2,1,0	5
17	17,8,4,2,1,0	5
31	31,15,7,3,1,0	5
$2^{x-1} \leq n < 2^x$	...	x

$x = ?$        $x = \lfloor \log_2 n \rfloor + 1$

EXERCÍCIOS

4. Analisar o pior caso do algoritmo abaixo:

```
int func (int n) {
    int a, b, c, r;
    b = n; c = n; r = 1;
    while (b >= 1) {
        a = b % 2;
        if (a == 1)
            r = r * c;
        c = c * c * c;
        b = b / 2;
    }
    return r;
}
```

$$T(n) = t_1 + t_2 \cdot (x+1) + t_3 \cdot x$$

$$= t_1 + t_2 + (t_2 + t_3) \cdot x$$

$$= c_1 + c_2 \cdot x$$

$$= c_1 + c_2 \cdot (\lfloor \log_2 n \rfloor + 1)$$

$$= c_1 + c_2 + c_2 \cdot \lfloor \log_2 n \rfloor$$

$$= c_3 + c_2 \cdot \lfloor \log_2 n \rfloor$$

$$\leq c_3 + c_2 \cdot \log_2 n$$

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

$x = \lfloor \log_2 n \rfloor + 1$

EXERCÍCIOS

5. Analisar o pior caso do cálculo recursivo de fatoriais

```
int fat(int n) {
    if (n <= 1) fat = 1;
    else fat = fat(n-1) * n;
}
```

$T(n) = d$ , se  $n \leq 1$   
 $T(n) = c + T(n-1)$ , se  $n > 1$

Para  $n > 1$ :

$$T(n) = c + T(n-1)$$

$$= c + c + T(n-2)$$

$$= 2c + T(n-2)$$

$$= 3c + T(n-3)$$

$$\dots$$

$$= (n-1)c + T(1)$$

$$= (n-1)c + d$$

$T(n) = O(n)$