

SME0827 - Estruturas de Dados

Análise de Algoritmos Aula 06 – Parte 2

Professor: André C. P. L. F. de Carvalho, ICMC-USP
PAE: Moisés Rocha dos Santos
Monitor:

© André de Carvalho - ICMC/USP

1

1

Aula de hoje

- Introdução
- Análise de Algoritmos
- Notação Assintótica
- Análise Assintótica
- Indução
- Conclusão

© André de Carvalho - ICMC/USP

2

2

Análise assintótica

- Complexidade computacional da ordenação por seleção

Tempo de execução $\cong \frac{N^2 + N}{2} \rightarrow O(\frac{N^2 + N}{2})$ Expressão complicada

$$\frac{N^2 + N}{2} = \frac{N^2}{2} + \frac{N}{2} \rightarrow \frac{N^2}{2} \rightarrow N^2$$

Constante
Insignificante quando N se torna grande

Complexidade: $O(n^2)$

© André de Carvalho - ICMC/USP

3

3

Taxas de crescimento

$$(n^2 + n)/2$$

n	$n^2/2$	$n/2$	$(n^2 + n)/2$
10	50	5	55
100	5000	50	5050
1000	500.000	500	500.500
10.000	50.000.000	5000	50.005.000
100.000	5.000.000.000	50.000	5.000.050.000

© André de Carvalho - ICMC/USP

4

4

Classes de complexidade

Classe	Notação	Exemplo
constante	$O(1)$	Retornar o primeiro elemento de um vetor
logarítmica	$O(\log n)$	Busca binária em vetor ordenado
linear	$O(n)$	Busca linear em um vetor
$N \log N$	$O(n \log n)$	Ordenação por combinação
quadrático	$O(n^2)$	Ordenação por seleção
cúbico	$O(n^3)$	Algoritmos convencionais para multiplicação de vetores
exponencial	$O(2^n)$	Torre de Hanói

$$O(1) < O(\log n) < O(n) < O(n * \log n) < O(n^2) < O(n^3) < O(2^n)$$

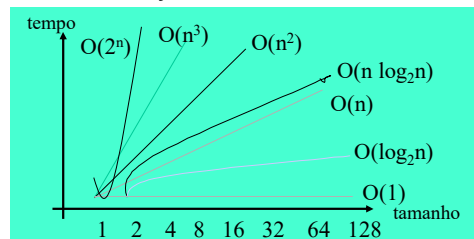
© André de Carvalho - ICMC/USP

5

5

Classes de complexidade

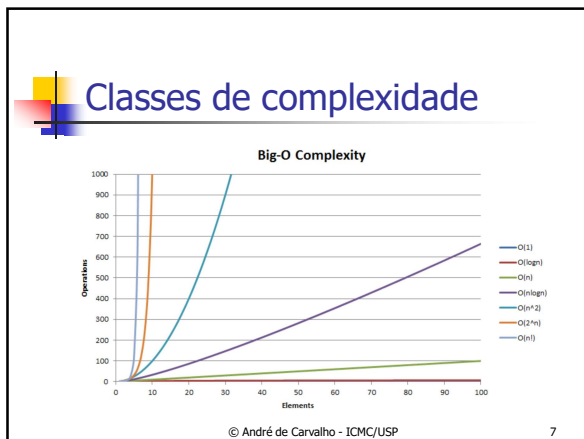
Relação entre as ordens



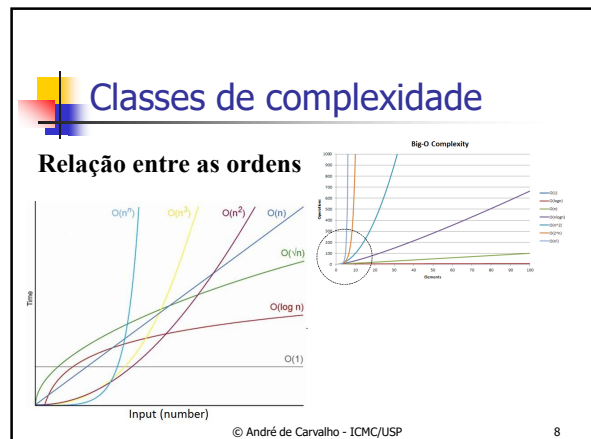
© André de Carvalho - ICMC/USP

6

6



7



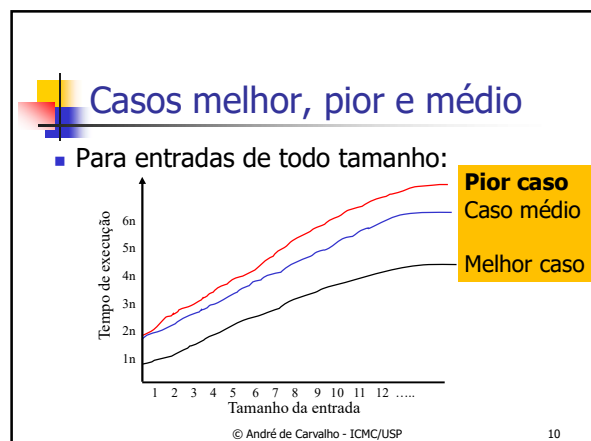
8

Taxas de crescimento

	Valor de n					
	n=1	n=2	n=4	n=8	n=16	n=32
1	1	1	1	1	1	1
log n	0	1	2	3	4	5
n	1	2	4	8	16	32
n log n	0	2	8	24	64	160
n ²	1	4	16	64	256	1024
n ³	1	8	64	512	4096	32768
2 ⁿ	2	4	16	256	65536	4294967296
n!	1	2	24	40320	Nem queira saber	

© André de Carvalho - ICMC/USP

9



10

Atenção!!!!

- Cuidado com as grandes constantes
 - Um algoritmo rodando no tempo 1.000.000n ainda é $O(n)$
 - Mas pode ser menos eficiente para um conjunto de dados do que um algoritmo executando no tempo $2n^2$, que é $O(n^2)$

© André de Carvalho - ICMC/USP

11

Notação assintótica

- Regra simples: remover termos de ordem mais baixa e constantes
 - $50n \log n$ é ...
 - $7n - 3$ é ...
 - $8n^2 \log n + 5n^2 + n$ é ...

© André de Carvalho - ICMC/USP

12

Notação assintótica

- Regra simples: remover termos de ordem mais baixa e constantes
 - $50n \log n$ é $O(n \log n)$
 - $7n - 3$ é $O(n)$
 - $8n^2 \log n + 5n^2 + n$ é $O(n^2 \log n)$

© André de Carvalho - ICMC/USP 13

13

Pausa



© André de Carvalho - ICMC/USP 14

14

Não existe apenas Big O

- Big O**
 - Define limite superior (upper bound) rígido
- Little o
 - Define limite superior (upper bound) flexível
- Big Ω (ômega)
- Little ω (ômega)
- Big Θ (theta)
- Little θ (theta)

© André de Carvalho - ICMC/USP 15

15

Não existe apenas Big O

- Vale para $O, o, \Omega, \omega, \Theta, \theta$
- Definido para funções de números naturais
 - E.g.: $f(n) = O(n^2)$
 - Descreve como $f(n)$ cresce em comparação com n^2
- Define um conjunto de funções
 - Usado para comparar limite(s) de 2 funções
- Descreve diferentes relações de taxa de crescimento entre
 - Uma função definidora e um conjunto definido de funções

© André de Carvalho - ICMC/USP 16

16

Além do Big O

- Big-Omega: limite inferior (lower bound)
 - Dadas duas funções $f(n)$ e $g(n)$, dizemos que $f(n)$ é $\Omega(g(n))$ se $g(n)$ é $O(f(n))$
 - Existe uma constante real $c > 0$ e um inteiro constante $n_0 \geq 1$, tal que $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
 - Tradução: quando o tamanho dos dados n é grande o suficiente, haverá uma função $c \cdot g(n)$ que é menor do que $f(n)$

© André de Carvalho - ICMC/USP 17

17

Além de Big O

- Definição de Big-Omega permite afirmar assintoticamente que:
 - Uma função é maior ou igual que outra até um fator constante
 - A maior função que é um limite inferior é definida por $g(n)$

Verdade	Melhor
$3n + 3 = \Omega(1)$	$\Omega(n)$
$n^2 + n^3 = \Omega(n)$	$\Omega(n^3)$

© André de Carvalho - ICMC/USP 18

18

Além do Big O

- Big-Theta
 - Dadas duas funções $f(n)$ e $g(n)$, $f(n)$ é $\Theta(g(n))$ se $f(n)$ é $O(g(n))$ e $f(n)$ é $\Omega(g(n))$
 - Existem constantes reais $c_1, c_2 > 0$ e uma constante inteira $n_0 \geq 1$, tal que $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for $n \geq n_0$
 - Tradução: duas funções são assintoticamente iguais, até um fator constante

© André de Carvalho - ICMC/USP 19

19

Além do Big O

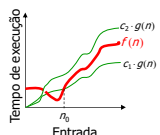
- Big-Theta
 - Quando o limite superior e inferior são os mesmos, o Big Θ pode ser usado para descrever a classe de complexidade
 - É usado porque é mais preciso

© André de Carvalho - ICMC/USP 20

20

Notação assintótica

Dada uma função $g(n)$, define-se $\Theta(g(n))$, como o conjunto:

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, \text{ e } n_0 > 0 \mid \forall n \geq n_0, \text{ tem-se } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$


Intuitivamente: Conjunto de todas as funções que têm a mesma taxa de crescimento que $g(n)$.

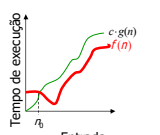
$g(n)$ é um limite asymptoticamente rígido para $f(n)$

© André de Carvalho - ICMC/USP 21

21

Notação assintótica

Dada uma função $g(n)$, define-se $O(g(n))$, como o conjunto:

$$O(g(n)) = \{f(n) : \exists c, \text{ e } n_0 > 0 \mid \forall n \geq n_0, \text{ tem-se } 0 \leq f(n) \leq c g(n)\}$$


Intuitivamente: Conjunto de todas as funções cuja taxa de crescimento seja igual ou inferior à de $g(n)$.

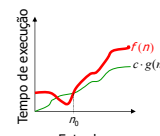
$g(n)$ é um limite superior asymptoticamente rígido para $f(n)$

© André de Carvalho - ICMC/USP 22

22

Notação assintótica

Dada uma função $g(n)$, define-se $\Omega(g(n))$, como o conjunto:

$$\Omega(g(n)) = \{f(n) : \exists c, \text{ e } n_0 > 0 \mid \forall n \geq n_0, \text{ tem-se } 0 \leq c g(n) \leq f(n)\}$$


Intuitivamente: Conjunto de todas as funções cuja taxa de crescimento seja igual ou superior à de $g(n)$.

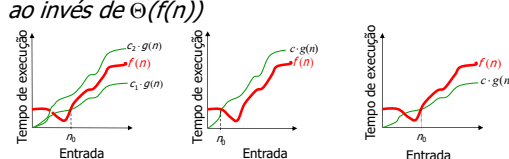
$g(n)$ é um limite inferior asymptoticamente rígido para $f(n)$

© André de Carvalho - ICMC/USP 23

23

Relação entre Θ , O , Ω

- $f(n) = \Theta(g(n))$ se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$
- $O(f(n))$ é frequentemente usado incorretamente ao invés de $\Theta(f(n))$



$f(n) = \Theta(g(n))$ $f(n) = O(g(n))$ $f(n) = \Omega(g(n))$

© André de Carvalho - ICMC/USP 24

24

E os little ones?

- Big O

$$O(g(n)) = \{f(n) : \exists c \in \mathbb{R}, \exists n_0 > 0 \mid \forall n \geq n_0, \text{ tem-se } 0 \leq f(n) \leq cg(n)\}$$

- Little O

$$o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \mid \forall n \geq n_0, \text{ tem-se } 0 \leq f(n) \leq cg(n)\}$$

© André de Carvalho - ICMC/USP 25

25

E os little ones?

- Big O

$$O(g(n)) = \{f(n) : \exists c \in \mathbb{R}, \exists n_0 > 0 \mid \forall n \geq n_0, \text{ tem-se } 0 \leq f(n) \leq cg(n)\}$$


- Little O

$$o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \mid \forall n \geq n_0, \text{ tem-se } 0 \leq f(n) \leq cg(n)\}$$

© André de Carvalho - ICMC/USP 26

26

Pausa



© André de Carvalho - ICMC/USP 27

27

Complexidade computacional

- Ordenação com recursão
 - Ordenação por seleção não é eficiente para vetores grandes
 - Complexidade: $O(N^2)$
 - O mesmo vale para outros algoritmos de ordenação que processam os elementos em ordem linear
 - Mas sabemos que: $N^2 > (\frac{N}{2})^2 + (\frac{N}{2})^2$
 - Solução: aplicar estratégias de divisão e conquista

© André de Carvalho - ICMC/USP 28

28

Ordenação por combinação

Vetor

56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

Vetor 1

56	25	37	58
0	1	2	3

Vetor 2

95	19	73	30
0	1	2	3

Vetor 1

25	37	56	58
0	1	2	3

Vetor 2

19	30	73	95
0	1	2	3

© André de Carvalho - ICMC/USP 29

29

Ordenação por combinação

Vetor 1

25	37	56	58
0	1	2	3

Vetor 2

19	30	73	95
0	1	2	3

vetor

19							
0	1	2	3	4	5	6	7

Vetor 1

25	37	56	58
0	1	2	3

Vetor 2

19	30	73	95
0	1	2	3

vetor

19	25						
0	1	2	3	4	5	6	7

© André de Carvalho - ICMC/USP 30

30

Ordenação por combinação

- Nome original: *merge sort*
- Algoritmo
 - Checa se a lista está vazia ou tem apenas um elemento (caso simples da recursão)
 - Senão, divide a lista ao meio em duas sub-listas
 - Ordena cada sub-lista recursivamente
 - Combina os dois sub-listas de volta à lista original

© André de Carvalho - ICMC/USP 31

31

Ordenação por combinação

- Caso simples
 - Lista com 0 ou 1 elementos
- Parte recursiva
 - Divide lista ao meio em duas sub-listas
 - Re-arranja os elementos de cada sub-lista
 - Combina as sub-listas

© André de Carvalho - ICMC/USP 32

32

Ordenação por combinação

```
def ordena_combinacao(lista1):
    comb_rec(lista1, 0, len(lista1)-1)

def comb_rec(lista1, ini, fim):
    if (ini < fim):
        centro = (ini + fim) / 2
        comb_rec(lista1, ini, centro)
        comb_rec(lista1, centro+1, fim)
        combina(lista1, ini, fim, centro)
```

© André de Carvalho - ICMC/USP 33

33

Ordenação por combinação

```
def combina(lista1, ini, fim, centro):
    lista_aux = []
    i = ini
    j = centro + 1
    while ini <= centro and j <= fim:
        if (lista1[i] <= lista1[j]):
            lista_aux.append(lista1[i])
            i += 1
        else:
            lista_aux.append(lista1[j])
            j += 1
    while (ini <= centro):
        lista_aux.append(lista1[i])
        i += 1
    while (j <= fim):
        lista_aux.append(lista1[j])
        j += 1
    for k in range(0, fim-ini+1):
        lista1[ini+k] = lista_aux[k]
```

© André de Carvalho - ICMC/USP 34

34

Ordenação por combinação

```
def combina(lista1, ini, fim, centro):
    lista_aux = []
    i = ini
    j = centro + 1
    while ini <= centro and j <= fim:
        if (lista1[i] <= lista1[j]):
            lista_aux.append(lista1[i])
            i += 1
        else:
            lista_aux.append(lista1[j])
            j += 1
    while (ini <= centro):
        lista_aux.append(lista1[i])
        i += 1
    while (j <= fim):
        lista_aux.append(lista1[j])
        j += 1
    for k in range(0, fim-ini+1):
        lista1[ini+k] = lista_aux[k]
```

© André de Carvalho - ICMC/USP 35

35

Ordenação por combinação

- Complexidade computacional
 - Tempo necessário para as operações do nível atual da decomposição recursiva
 - Função *combina()* preenche n posições de um vetor
 - $O(n)$
 - Tempo necessário para as chamadas recursivas
 - Definir

© André de Carvalho - ICMC/USP 36

36

Ordenação por combinação

Ordenar um vetor de tamanho n

Requer que sejam ordenados dois vetores de tamanho $n/2$

Que requer que sejam ordenados quatro vetores de tamanho $n/4$

Que requer que sejam ordenados oito vetores de tamanho $n/8$

E assim por diante

n
$2 \times n/2$
$4 \times n/4$
$8 \times n/8$

© André de Carvalho - ICMC/USP 37

37

Ordenação por combinação

- Complexidade computacional
 - Número de níveis
 - A cada nível da hierarquia, n é dividido por 2, até chegar a 1
 - Ex.: $n=8$ ($8-4-2$): 3 níveis, $n=16$ ($16-8-4-2$): 4 níveis
 - Para encontrar número de níveis k , basta encontrar o valor de k para o qual $n = 2^k$ ($k = \log_2 n$)
 - Trabalho recursivo é proporcional a $n \log_2 n$
 - Complexidade: $O(n + n \log_2 n) = O(n \log_2 n)$

© André de Carvalho - ICMC/USP 38

38

Ordenação por combinação

Comparação dos desempenhos n e n^2 e $\log n$

n	n^2	$n \log n$
10	100	33
100	10.000	664
1.000	1.000.000	9.965
10.000	100.000.000	132.877

© André de Carvalho - ICMC/USP 39

39


Ordenação por combinação

n	Ord. Seleção	Ord. Combinação
10	0.12 mseg.	0.54 mseg.
20	0.39 mseg.	1.17 mseg.
40	1.46 mseg.	2.54 mseg.
100	8.72 mseg.	6.90 mseg.
200	33.33 mseg.	14.84 mseg.
400	135.42 mseg.	31.25 mseg.
1000	841.67 mseg.	84.38 mseg.
2000	3.35 seg.	179.17 mseg.
4000	13.42 seg.	383.33 mseg.
10000	83.90 seg.	997.67 mseg.

© André de Carvalho - ICMC/USP 40

40

Pausa



© André de Carvalho - ICMC/USP 41

41

Indução

- Usado por cientistas da computação para provar propostas
 - Provar o caso base
 - Provar que a proposição se mantém verdadeira quando n tem o valor base, geralmente 0 ou 1
 - Provar o caso indutivo
 - Demonstrar que, se você assumir a proposição para ser verdade para n , também deve ser verdade para $n + 1$

© André de Carvalho - ICMC/USP 42

42

Exemplo de Indução

- Provar usando indução matemática que:

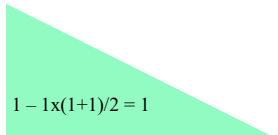
$$n + n-1 + n-2 + \dots + 3 + 2 + 1 = \frac{n(n+1)}{2}$$

© André de Carvalho - ICMC/USP 43

43

Indução

- Caso base: $n = 1$



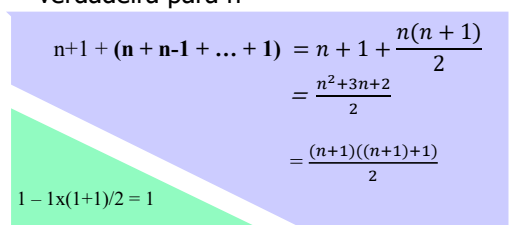
$$1 - 1 \times (1+1)/2 = 1$$

© André de Carvalho - ICMC/USP 44

44

Indução matemática

- Caso indutivo: assumir que igualdade é verdadeira para n



$$\begin{aligned} n+1 + (n + n-1 + \dots + 1) &= n + 1 + \frac{n(n+1)}{2} \\ &= \frac{n^2 + 3n + 2}{2} \\ &= \frac{(n+1)((n+1)+1)}{2} \end{aligned}$$

$$1 - 1 \times (1+1)/2 = 1$$

© André de Carvalho - ICMC/USP 45

45

Conclusão

- Análise de algoritmos
- Medidas de complexidade
- Algoritmos de Ordenação
 - Seleção, Combinação e Inserção
 - Complexidade

© André de Carvalho - ICMC/USP 46

46

Classes de complexidade

- Quanto maior o crescimento do tempo, mais complexo é o algoritmo
- Algoritmos cujo tempo independe do número de exemplos tem complexidade constante
 - Algoritmo constante tem complexidade $O(1)$
- Quando o tempo aumenta linearmente com o número de exemplos, a complexidade é linear
 - Algoritmos linear tem complexidade $O(n)$
 - Quando o tempo aumenta mais rapidamente, complexidade é não linear

© André de Carvalho - ICMC/USP 47

47

Classes de complexidade

- Algoritmos polinomiais
 - São executados em um tempo n^k para alguma constante K
 - Algoritmos de complexidade constante, linear, quadrática, cúbica, etc.
- Os piores algoritmos são os de complexidade exponencial
 - Ex.: complexidade $O(2^n)$

© André de Carvalho - ICMC/USP 48

48



Complexidade de espaço

- O espaço necessário por um programa é a soma de:
 - Requisitos de espaço fixo que não dependem do tamanho de entradas ou saídas
 - Requisitos de espaço variável que dependem do número, tamanho e valores de entradas e saídas
 - Nem todo o espaço necessário pode ser usado a qualquer momento

© André de Carvalho - ICMC/USP

49

49



Questions



© André de Carvalho - ICMC/USP

50

50