

Análise e Complexidade de Algoritmos

UFFS

Ciência da Computação
Estrutura de Dados

Prof. Denio Duarte

duarte@uffs.edu.br

Prof.

Algoritmos

- Informalmente, um algoritmo é um procedimento computacional bem definido que:
 - Recebe um conjunto de valores como entrada.
 - Produz um conjunto de valores como saída
- Assim, um algoritmo é uma ferramenta para resolver um problema computacional.

Algoritmos

- Problema
 - Primalidade (determinar se um número é primo):
 - Entrada: 9411461
 - Saída: é primo
 - Entrada: 8411461
 - Saída: Não é primo

Algoritmos

- Problema:
 - Ordenação: $A[1..n]$ é crescente se $A[1] \leq \dots \leq A[n]$ ¹
 - Rearranjar o vetor A de modo que fique ordenado.

• Entrada:

1								n
42	10	56	78	64	25	33	18	40

• Saída:

1								n
10	18	25	33	40	42	56	64	78

¹ Por simplificação, a primeira posição do vetor será considerada a 1

Algoritmos

- Instância de um problema: conjunto de valores que serve de entrada para um determinado problema:
 - Os números *9411461* e *8411461* são instâncias para o problema de primalidade.
 - O vetor

	1								<i>n</i>
42	10	56	78	64	25	33	18	40	

 é uma instância para o problema de ordenação.

Algoritmos

- Descrição de algoritmos:
 - Linguagem de programação (C, Pascal, Java, etc)
 - Implementado como um *hardware*
 - Pseudo-Código

Algoritmos

- Razões para o estudo:
 - Evitar reinventar a roda:
 - Existem bons algoritmos que solucionam problemas importantes.
 - Ajudar no desenvolvimento de seus algoritmos:
 - Nem sempre existe um algoritmo de prateleira que sirva para resolver o seu problema.

Complexidade

- Muitas vezes, não é suficiente saber que um determinado algoritmo produz uma saída correta.
 - Um algoritmo extremamente lento em geral não tem muita utilidade.
- Queremos projetar/desenvolver algoritmos eficientes (=rápidos).

Complexidade

- Mas o que seria uma boa medida da eficiência de um algoritmo?
 - Uma possibilidade: estimar através de uma análise matemática o tempo que o algoritmo gasta em função do **tamanho da entrada**.

Complexidade

- Exemplo

- O tamanho do problema de ordenação corresponde ao tamanho do vetor a ser ordenado.
- Dado um vetor de n posições, o tamanho do problema será n .

Complexidade

- O comportamento limite de um algoritmo (em tempo ou espaço) conforme o crescimento da entrada é chamado *comportamento assintótico*.
- Ou seja, o comportamento de um algoritmo para uma entrada GRANDE.
- Podemos dizer que o *comportamento assintótico* representa o “pior caso” para um algoritmo.

Complexidade

- A complexidade é dada por uma fórmula matemática:
 - $3+n+4n$, onde n é o tamanho da entrada.
 - Como estamos interessados em n s GRANDES, na fórmula acima o termo dominante é $4n$, assim podemos simplificar a complexidade para $4n$.
 - Chamamos de **notação assintótica**

entrada	complexidade real	notação assintótica
n	$3+n+4n$	$4n$
10	53	40
50	253	200
100	503	400
5000	25003	20000

Complexidade

- Se um algoritmo A processa uma entrada n no tempo kn^2 (onde k é uma constante), dizemos que a complexidade de A é n^2 .
- Formalmente escrevemos $A \in O(n^2)$, ou seja, A pertence a classe dos algoritmos quadráticos
- Dizemos também que A é da **Ordem de n^2** (daí o símbolo O)

Complexidade

- Exemplo de cálculo:

```
procedimento teste (inteiro n)
variaveis
    inteiros b, c, i;
inicio
    b:=n*2;
    c:=0;
    para i de 1 ate n faça
        c:=b+n;
    fim para;
fim;
```

Complexidade

procedimento teste (inteiro n)	custo	vezes
variaveis		
inteiros b, c, i;		
inicio		
b:=n*2;	c_1	1
c:=0;	c_2	1
para i de 1 ate n faça	c_3	n
c:=b+n;	c_4	n
fim para;		
fim;		

A constante c_k representa o custo (tempo) de cada instrução.

Complexidade

procedimento teste (inteiro n)	custo	vezes
variaveis		
inteiros b, c, i;		
inicio		
b:=n*2;	c_1	1
c:=0;	c_2	1
para i de 1 ate n faça	c_3	n
c:=b+n;	c_4	n
fim para;		
fim;		

$T(n)$, tempo de execução com entrada de tamanho n , é de:

$$T(n) = 1*c_1 + 1*c_2 + n*c_3 + n*c_4, \text{ ou } c_1 + c_2 + nc_3 + nc_4.$$

Complexidade

procedimento teste (inteiro n)	custo	vezes
variaveis		
inteiros b, c, i;		
inicio		
b:=n*2;	c_1	1
c:=0;	c_2	1
para i de 1 ate n faça	c_3	n
c:=b+n;	c_4	n
fim para;		
fim;		

$T(n)$, tempo de execução com entrada de tamanho n , é de:

$$T(n) = 1*c_1 + 1*c_2 + n*c_3 + n*c_4, \text{ ou } c_1 + c_2 + nc_3 + nc_4.$$

Para o pior caso (*comportamento assintótico*) podemos resumir a complexidade de tempo em $O(n)$.

Complexidade

- A importância de medir a complexidade nasce da necessidade de comparar algoritmos.

- Exemplo:

Algoritmo	Complexidade Tempo
A_1	n
A_2	$n \log n$
A_3	n^2
A_4	n^3
A_5	2^n

Complexidade

Suponhamos que o **custo da instrução é 1 milissegundo**,
o tamanho do problema que pode ser resolvido por
cada algoritmo é:

Algoritmo	O	1seg	1min	1h
A_1	n	1000	$6 \cdot 10^4$	$3,6 \cdot 10^6$
A_2	$n \log n$	140	4893	$2,0 \cdot 10^5$
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Qual é o melhor?

Complexidade

- Por que se importar com a complexidade de tempo se os computadores aumentam seus poderes de processamento?
 - Por exemplo: GPU's e multicores

Complexidade

- Por que se importar com a complexidade de tempo se os computadores aumentam seus poderes de processamento?

Algoritmo		Máquina Atual	10 vezes mais rápida
A_1	n	v_1	$10v_1$
A_2	$n \log n$	v_2	$\cong 10v_2$
A_3	n^2	v_3	$3,16v_3$
A_4	n^3	v_4	$2,15v_4$
A_5	2^n	v_5	$v_5+3,3$

Complexidade

- Armadilhas da notação assintótica:
 - Se temos um algoritmo A cujo $T(n)=100n$ podemos dizer que A é $O(n)$.
 - Se temos um algoritmo B cujo $T(n)=n \log_{10} n$ podemos dizer que B é $O(n \log_{10} n)$.
 - Assim A é assintoticamente mais eficiente que B , certo?

Complexidade

- Armadilhas da notação assintótica:
 - Se temos um algoritmo A cujo $T(n)=100n$ podemos dizer que A é $O(n)$.
 - Se temos um algoritmo B cujo $T(n)=n \log_{10} n$ podemos dizer que B é $O(n \log_{10} n)$.
 - Assim A é assintoticamente mais eficiente que B , certo?
 - **Errado**, A só é mais eficiente que B quando $n=10^{100}$

Aviso: cuidado com as constantes no momento da análise assintótica

Cálculo

- Valor dominante da direita VDD:

- $L = (10; 9; 5; 13; 2; 7; 1; 8; 4; 6; 3)$

- $VDD = (13, 8, 6, 3)$

//Entrada: Vetor $L[1..n]$

//Retorno: Vetor com os elementos dominantes à direita

```
DominanteADireita(L) {
```

```
    D = vetor vazio
```

```
    for (i = 1 to n) {
```

```
        isDominante = true
```

```
        for (j = i+1 to n)
```

```
            if ( $L[i] \leq L[j]$ ) isDominante = false
```

```
            if (isDominante) acrescenta  $L[i]$  em D
```

```
        }
```

```
    return L
```

```
}
```


Cálculo

```
// Entrada: Vetor L[1..n]
// Retorno: Vetor com os elementos dominantes à direita
DominanteADireita(L) {
    D = vetor vazio
    for (i = 1 to n) { //n
        isDominante = true
        for (j = i+1 to n) //
            if (L[i] <= L[j]) isDominante = false
        if (isDominante) acrescenta L[i] em D
    }
    return L
}
```

- Temos o bloco do `for` (variável i) executado n vezes
- O bloco do `for` (variável j) é executado de $i+1$ até n vezes, totalizando $n-(i+1)+1$ vezes que é igual à $n-i$ (pois a constante 1 não tem grande influência no resultado).

Cálculo

- Temos o bloco do `for` (variável i) executado n vezes
- O bloco do `for` (variável j) é executado de $i+1$ até n vezes, totalizando $n-(i+1)+1$ vezes que é igual à $n-i$.
- Assim temos:

$$\sum_{i=1}^n (n-i) = (n-1) + (n-2) + \dots + 1 + 0 = \frac{(n-1) * (n+1)}{2} = \frac{n^2 - 1}{2}$$

Assintoticamente o algoritmo *DominanteADireita* é $O(n^2)$.

Ordens de Complexidade

- Melhor caso, notação Ω (limite assintótico inferior)
- Caso médio, notação Θ (limite assintótico médio)
- Pior caso, notação O (limite assintótico superior)

Ordens de Complexidade

- Cola:

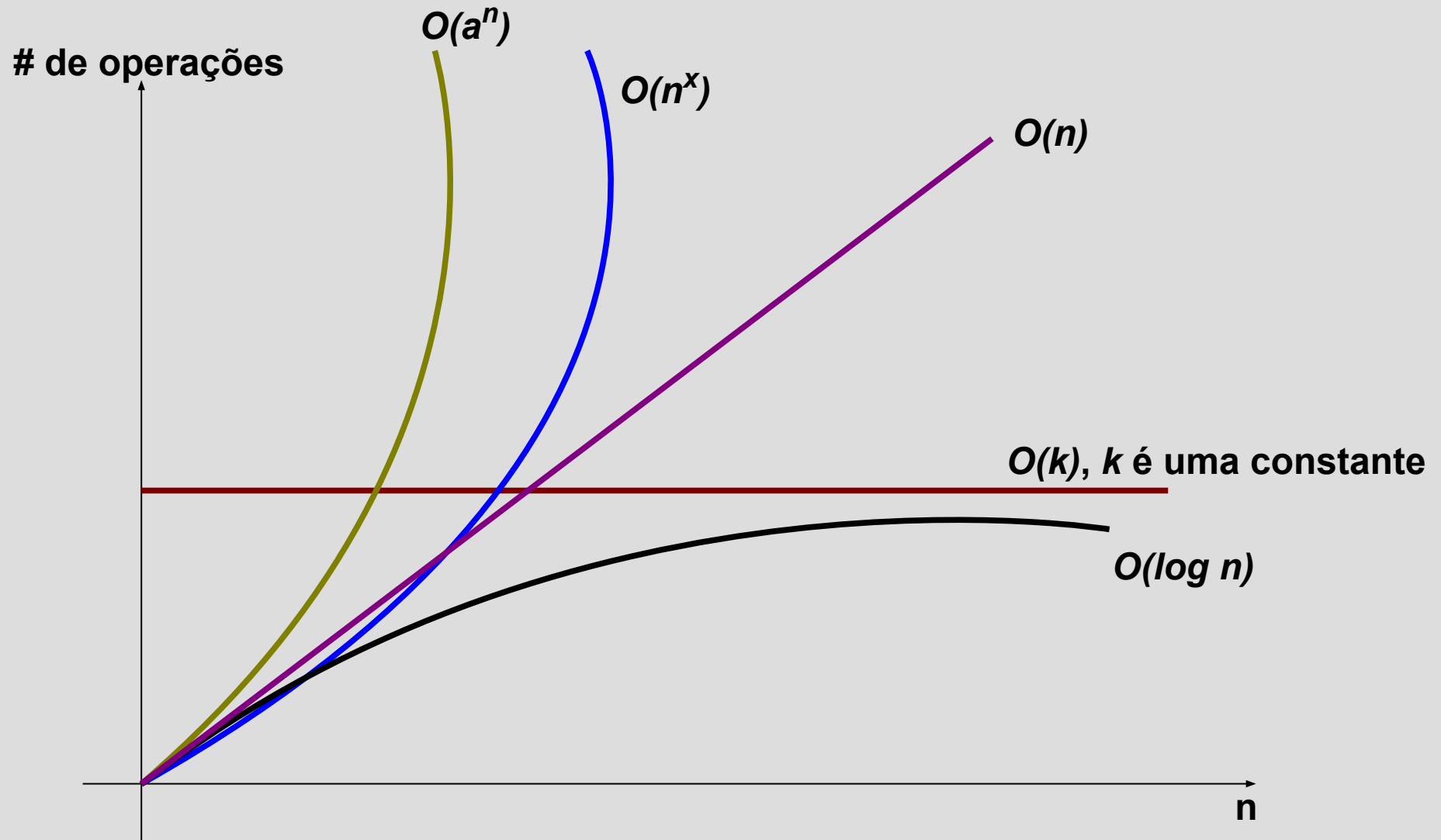
f é ...	significa que f cresce	e se escreve ¹
... O de g	não mais depressa que g	$f=O(g)$
... <i>theta</i> de g	aproximadamente como g	$f=\Theta(g)$
... <i>ômega</i> de g	não mais devagar	$f=\Omega(g)$

¹ Lembrem-se o correto é utilizar o sinal \in pois as ordens são classes e f é um elemento da classe

Classes do Comportamento Assintótico

CLASSE	NOME
$O(1)$	constante
$O(\log n)$	logarítmica
$O(n)$	linear
$O(n \log n)$	$n \log n$
$O(n^2)$	quadrática
$O(n^3)$	cúbica
$O(n^k)$ com $k \geq 1$	polinomial
$O(2^n)$	exponencial
$O(a^n)$ com $a > 1$	exponencial

Crescimento das Classes



Exemplos

- Primo – Versão 1

```
int IsPrimeV1(int n)
{
    int i;
    for (i=2; i<n; i++)
        if (n%i==0)
            return 0;
    return 1;
}
```

$\in O(n)$

Exemplos

- Primo – Versão 2

```
int IsPrimeV2(int n)
{
    int i;
    for (i=2; i<n/2; i++)
        if (n%i==0)
            return 0;
    return 1;
}
```

$\in O(n/2)$

Exemplos

- Primo – Versão 2

```
int IsPrimeV2(int n)
{
    int i;
    for (i=2; i<sqrt(n); i++)
        if (n%i==0)
            return 0;
    return 1;
}
```

$\in O(\sqrt{n})$

Testar: 524.287 9.369.319 21.47.483.647

Exemplos

- Valor dominante da direita VDD:
 - Versão anterior $O(n^2)$
 - É possível fazer uma versão $O(n)$?

Exemplos

- Valor dominante da direita VDD:
 - Versão anterior $O(n^2)$
 - É possível fazer uma versão $O(n)$?

//Entrada: Vetor $L[1..n]$

//Retorno: Vetor com os elementos dominantes à direita

```
DominanteADireita(L) {  
    D,bigger = L[n]  
    for (i = n-1 to 1) {  
        if (L[i]>bigger)  
        {  
            acrescenta L[i] em D  
            bigger=L[i]  
        }  
    }  
    return L  
}
```