PANC: Projeto e Análise de Algoritmos

Aula 05: Notação Assintótica e Crescimento de Funções

Breno Lisi Romano

http://sites.google.com/site/blromano

Instituto Federal de São Paulo – IFSP São João da Boa Vista
Bacharelado em Ciência da Computação – 3º Semestre

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Sumário

- Revisão de Conteúdo
- Funções Tipicamente Utilizadas
- Comparação da Taxa de Crescimento de Funções
- Análise Assintótica
- Notações para Análise Assintótica
- Propriedades das Classes de Notações
- Classes de Comportamentos Assintóticos
- Resolução de Exercícios

Recapitulando... (1)

- A função de **complexidade de tempo T(n)** mede o tempo necessário para executar um algoritmo **(número de instruções)** → **Quantidade de Operações Executadas**
- **Ordenação por Inserção (*Insertion Sort*):**
  - Caracterizada pelo **princípio** no qual se **divide** o **array** em **dois segmentos**: um já **ordenado** e o outro **não ordenado**
  - **Análise da Complexidade – T(n):**
    - **Melhor Caso:** T(n) é Linear – O(n)
    - **Pior Caso:** T(n) é Quadrático – $O(n^2)$
- **Ordenação por Intercalação (*Merge Sort*):**
  - Utiliza o Paradigma de Divisão e Conquista para ordenar Arrays
    - **Divide** o Array; **Conquista** ordenando os Sub(Arrays); **Combina** Intercalando os Sub(Arrays) ordenados
  - **Análise da Complexidade – T(n): O(n.lg n)**
- Precisamos definir **terminologias apropriadas** para **representar** a complexidade **T(n)** dos algoritmos → **AULA DE HOJE**

Recapitulando... (2)

- *"Algorithm analysis usually means 'give a big-O figure for the running time of an algorithm (Of course, a big-Θ would be even better). This can be done by getting a big-O figure for parts of the algorithm and then combining these figures using the sum and product rules for big-O."*

- *"It is useful for the analysis of algorithms since it captures the asymptotic growth pattern of functions and ignores the constant multiple (which is out of our control anyway when algorithms are translated into programs)."*

- **- Ian Parberry, *Problems on Algorithms***

Introdução (1)

- Vamos expressar **complexidade** através de **funções em variáveis** que descrevam o **tamanho de instâncias do problema**

- Exemplos:
  - **Problemas de aritmética** de precisão arbitrária: **número de bits** (ou bytes) dos inteiros
  - **Problemas em grafos**: número de **vértices e/ou arestas**
  - **Problemas de ordenação de arrays**: tamanho do array – **quantidade de elementos**
  - **Busca em textos**: **número de caracteres** do texto ou padrão de busca

Introdução (2)

- **Ordem de crescimento** do tempo de execução de um algoritmo → **Eficiência** do algoritmo e **Comparação de Desempenho** com algoritmos alternativos

- Embora, às vezes, seja possível **determinar o tempo exato** de execução de um algoritmo, **o que se ganha em precisão em geral não vale o esforço**

- **Queremos encontrar um T(n) aproximado!!!**

Introdução (3)

- Para **entradas suficientemente grandes**, as **constantes multiplicativas** e os **termos de ordem mais baixa** são dominados pelos efeitos do próprio tamanho da entrada
- Observar **tamanhos de entrada** suficientemente **grandes** → **Eficiência assintótica** dos algoritmos
  - **Preocupação**: modo como o **tempo de execução** de um algoritmo **aumenta** com **o tamanho da entrada no limite**, à medida que o tamanho da entrada aumenta sem limitação
- Vamos **supor** que **funções** que expressam **complexidade** são sempre **positivas**, já que estamos medindo **número de operações**

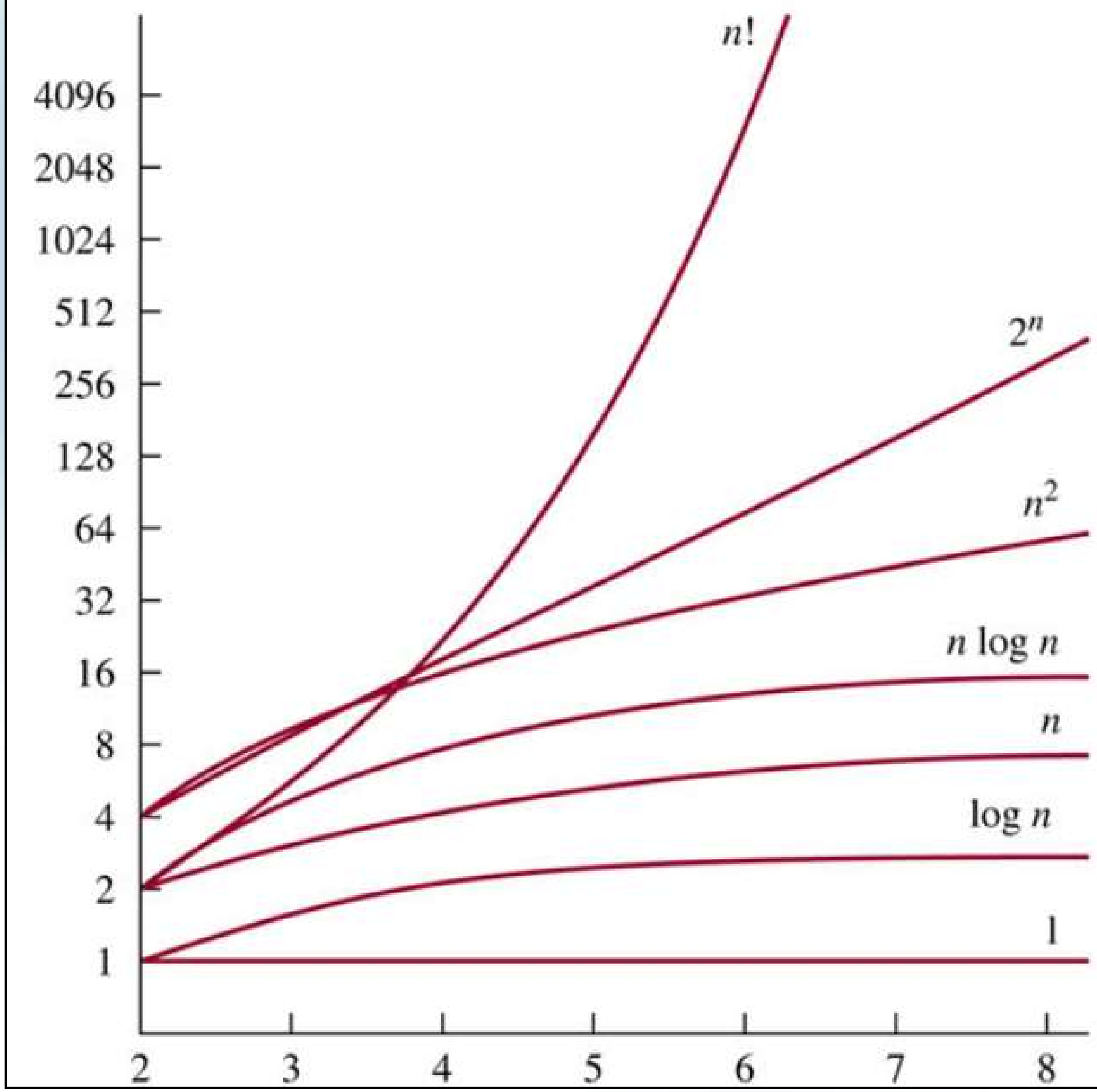
Funções Tipicamente Utilizadas – T(n) (1)

- **k:** constante em k, geralmente denotado como 1
  - Independe do tamanho de n.
- **lg n**: logarítmico (geralmente a base é 2)
  - Menor do que uma constante grande
  - n = 1000 → lg n = 3;
  - n = 1.000.000 → lg n = 6
- **n:** linear
  - Aumenta no mesmo ritmo que n
- **n.lg n:** linear logarítmico, ou apenas n.lg n
  - n = 1000 → n.log n = 3000;
  - n = 1.000.000 → n.log n = 6.000.000.

Funções Tipicamente Utilizadas – T(n) (2)

- **n^2:** quadrático
  - n=1000 → n^2 = 1.000.000.
- **n^k:** polinomial, proporcional à n elevado a k (constante)
- **2^n:** exponencial, proporcional à 2 elevado a n
  - n = 20 → 2^n >1.000.000
  - Quando n dobra, 2^n se eleva ao quadrado
- **n!**: fatorial
  - n=10 → n!= 3.628.800
- **n^n:** sem nome formal, apenas n^n

Comparação da Taxa de Crescimento de Funções (1)



Comparação da Taxa de Crescimento de Funções (2)

- **Comparação de funções assintoticamente** → para valores grandes, desprezando constantes multiplicativas e termos de menor ordem

	$n = 100$	$n = 1000$	$n = 10^4$	$n = 10^6$	$n = 10^9$
$\log n$	2	3	4	6	9
n	100	1000	10^4	10^6	10^9
$n \log n$	200	3000	$4 \cdot 10^4$	$6 \cdot 10^6$	$9 \cdot 10^9$
n^2	10^4	10^6	10^8	10^{12}	10^{18}
$100n^2 + 15n$	$1,0015 \cdot 10^6$	$1,00015 \cdot 10^8$	$\approx 10^{10}$	$\approx 10^{14}$	$\approx 10^{20}$
2^n	$\approx 1,26 \cdot 10^{30}$	$\approx 1,07 \cdot 10^{301}$	?	?	?

Comparação da Taxa de Crescimento de Funções (3)

- **Não afetam a taxa de crescimento**:
  - Fatores Constantes
  - Fatores de ordem mais baixa

- **Exemplos:**
  - $10^2n + 10^5$: é uma função linear
  - $10^5n^2 + 10^8n$: é uma função quadrática
  - $10^{-9}n^3 + 10^{20}n^2$: é uma função cúbica
  - E por ai vai....

Análise Assintótica (1)

- Aplicado para representar o T(n) → Exemplo: T(n) ∈ **O(n2)**
- Descreve como o **tempo de execução** cresce à medida em que a entrada **aumenta indefinidamente**, o comportamento de **limite**
- É uma **análise teórica**, independente de hardware e que utiliza funções cujos **domínios** são o **conjunto dos números naturais**
- Os **termos** de **mais baixa ordem** e **constantes** são ignorados
- Utiliza **cinco notações**: O, Θ, Ω, o, ω
  - **Observação**: Assumimos que as funções utilizadas são assintoticamente positivas, ou seja, são positivas para todo n suficientemente grande

Análise Assintótica (2)

- Precisamos de uma **ferramenta matemática** para **comparar funções**

- Para a análise de algoritmo será feita uma **análise assintótica**:
  - **Matematicamente**: estudando o comportamento de limites ($n \rightarrow \infty$)
  - **Computacionalmente**: estudando o comportamento para entrada arbitrariamente grande ou descrevendo taxa de crescimento

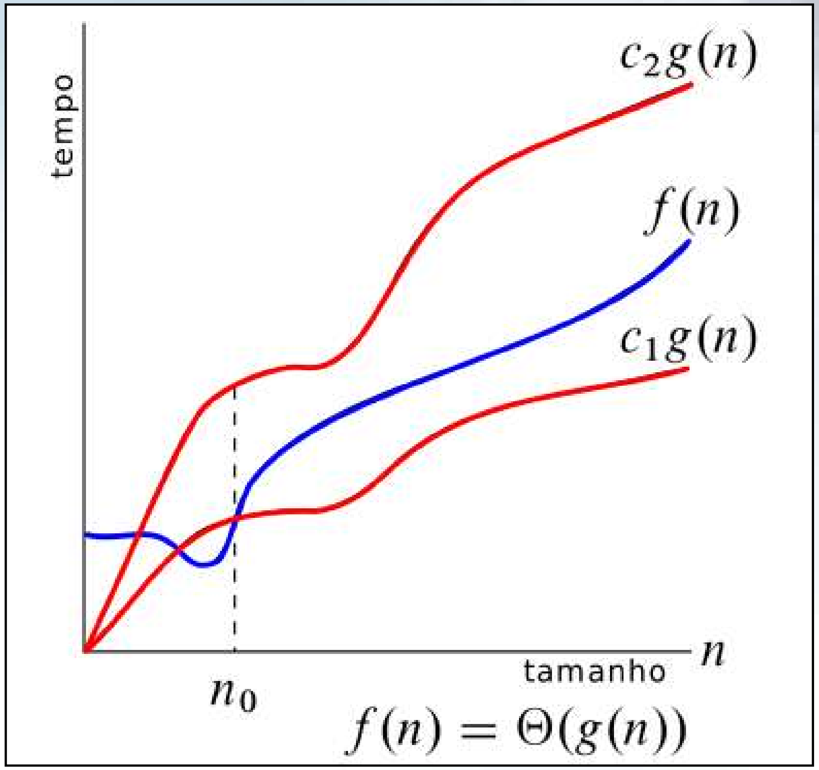
- O foco está nas **ordens de crescimento**

Notações para Análise Assintótica (1)

- **Classe Θ (Big- Θ)**:
  - **Θ(g(n))** = *{f(n): existe constantes positivas c_1, c_2 e n_0 tais que* $\mathbf{0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)}$, *para todo* $n \geq n_0$*}*
  - **Informalmente:** dizemos que, se **f(n) ∈ Θ(g(n))**, então **f(n) cresce tão rapidamente quanto g(n)**
  - **Significado**:
    - Significa dizer que uma função **f(n) pertence a g(n)** caso existam constantes c_1 e c_2 tal que ela **seja "imprensada" entre c_1g(n) e c_2g(n)**
    - **Abuso de notação**: **f(n) = Θ(g(n))**
    - **f(n)** é a função que expressa a **complexidade de tempo**, **T(n)**, e **g(n)** geralmente é uma das funções tipicamente utilizadas (slides anteriores)

Notações para Análise Assintótica (2)

- **Classe Θ (Big- Θ)**:
  - A função **g(n)** denota **o limite assintoticamente restrito** para **f(n)**
  - Para n ≥ n_0, **f(n) estará "presa"** entre **duas inclinações de g(n)**, ou seja, **não será maior nem menor**: a taxa de crescimento é **igual**

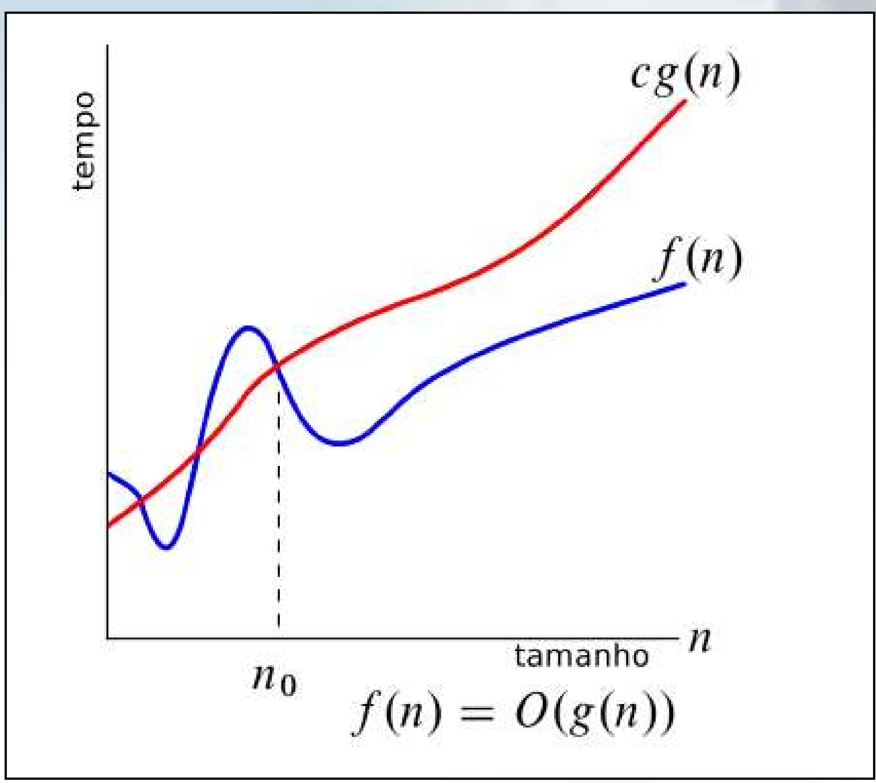


Notações para Análise Assintótica (3)

- **Classe O (Big-O)**:
  - **Lê-se** "Ó grande de g de n" ou, às vezes, apenas "ó de g de n"
  - **O(g(n))** = *{f(n): existe constantes positivas c e n_0 tais que* ***0 ≤ f(n) ≤ c.g(n)****, para todo n ≥ n_0}*
  - **Informalmente:** dizemos que, se **f(n)** ∈ **O(g(n))**, então **f(n) cresce no máximo tão rapidamente quanto g(n)**
  - **Significado**:
    - Significa dizer que uma função f(n) pertence a g(n) caso exista uma constante c tal que ela **seja limitada superiormente** por c.g(n)
    - **Abuso de notação: f(n) =O(g(n))**
    - **f(n)** é a função que expressa **a complexidade de tempo**, **T(n)**, e **g(n)** geralmente é uma das funções tipicamente utilizadas (slides anteriores)

Notações para Análise Assintótica (4)

- **Classe O (Big-O)**:
  - A função **g(n)** denota o **limite assintoticamente superior** para **f(n)**
  - Para n ≥ n0, f(n) estará limitada por um fator constante de g(n), ou seja, não será maior: **a taxa de crescimento é no máximo igual**



Notações para Análise Assintótica (5)

- Classe O (Big-O):
  - **Relação entre O e Θ:** Note que se $T(n) = \Theta(n)$ → Não é incorreto dizer que $T(n) = O(n^2)$
    - De fato, qualquer função linear é $O(n^2)$, porém é impreciso
  - **Utilização**: Geralmente, a **notação O** é utilizada para descrever o **pior caso**
    - Para maior precisão, combina-se notações e perspectivas da execução do algoritmo
  - **Para pensar**: $T(n) = 6n^3 + 2n^2 + 72n + 1$ possui qual complexidade utilizando-se a notação O?
    - **$T(n) = O(n^3)$**

Notações para Análise Assintótica (6)

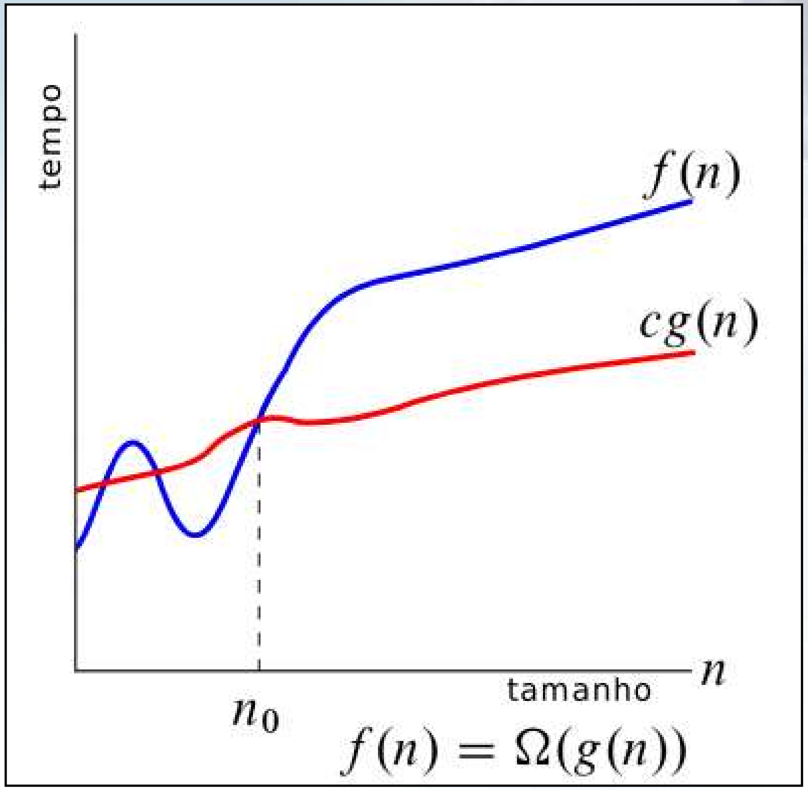
- **Classe O (Big-O):**
  - **Operações Básicas:**
    - $f(n) = O(f(n))$
    - $c.f(n) = O(f(n))$ (c constante)
    - $O(f(n)) + O(f(n)) = O(f(n))$
    - $O(O(f(n))) = O(f(n))$
    - $O(f(n)) + O(g(n)) = O(max(f(n); g(n)))$
    - $O(f(n)).O(g(n)) = O(f(n)g(n))$
    - $f(n).O(g(n)) = O(f(n)g(n))$

Notações para Análise Assintótica (7)

- **Classe Ω (Big-Ω):**
  - **Lê-se** "ômega grande de g de n" ou, às vezes, "ômega de g de n"
  - **Ω(g(n))** = *{f(n): existe constantes positivas c e n_0 tais que* ***0 ≤ c.g(n) ≤ f(n)****, para todo n ≥ n_0}*
  - **Informalmente:** dizemos que, se **f(n)** ∈ **Ω(g(n))**, então **f(n) cresce no mínimo tão lentamente quanto g(n)**
  - **Significado**:
    - Significa dizer que uma função f(n) pertence a g(n) caso exista uma constante c e n_0 tal que ela **seja limitada inferiormente** por c.g(n) para n ≥ n0
    - **Abuso de notação: f(n) =Ω (g(n))**

Notações para Análise Assintótica (8)

- **Classe Ω (Big-Ω):**
  - A função **g(n)** denota o **limite assintoticamente inferior** para **f(n)**
  - Para n ≥ n0, f(n) estará limitada por um fator constante de g(n), ou seja, não será menor: **a taxa de crescimento é no mínimo igual**



Notações para Análise Assintótica (9)

- **Classe o:**
  - **Lê-se** "ó pequeno de g de n"
  - **o(g(n)) =** *{f(n): para toda constante positiva c, existe uma constante $n_0 > 0$ tal que* $\mathbf{0 \le f(n) < c.g(n)}$*, para todo $n \ge n_0$}*
  - **Informalmente:** dizemos que, se **f(n)** ∈ **o(g(n))**, então **f(n) cresce mais lentamente que g(n)**
  - **Formalmente:** $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$
  - **Exemplo:**
    - $1000.n^2 \in o(n^3)$

Notações para Análise Assintótica (10)

- Classe ω:
  - **Lê-se** "ômega pequeno de g de n"
  - **ω(g(n)) =** *{f(n): para toda constante positiva c, existe uma constante $n_0 > 0$ tal que* $\mathbf{0 \leq c.g(n) < f(n)}$*, para todo* $n \geq n_0$*}*
  - **Informalmente:** dizemos que, se **f(n)** ∈ **ω(g(n))**, então **f(n) cresce mais rapidamente que g(n)**
  - **Formalmente:** $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$
  - **Exemplo:**
    - $10^{-3}.n^2 \in \omega(n)$

Relação entre as Classes O, Θ e Ω

- Para um dado T(n), se $\mathbf{T(n) = \Omega(n^2)}$ **e** $\mathbf{T(n) = O(n^2) \rightarrow T(n) = \Theta(n^2)}$
  - **Justificativa**: T(n) é limitado inferiormente e superiormente por n^2

Notações Assintóticas - Resumo

- **f(n) = O(g(n))** se houver constantes positivas n_0 e c tal que **f(n) ≤ c g(n)** para todo n ≥ n_0
- **f(n) = Ω(g(n))** se houver constantes positivas n_0 e c tal que **f(n) ≥ c g(n)** para todo n ≥ n_0
- **f(n) = Θ(g(n))** se houver constantes positivas n_0, c_1 e c_2 tal que **$c_1 g(n) \leq f(n) \leq c_2 g(n)$** para todo n ≥ n_0
- **f(n) = o(g(n))** se, para qualquer constante positiva c, existe n_0 tal que **f(n) < c g(n)** para todo n ≥ n_0
- **f(n) = ω(g(n))** se, para qualquer constante positiva c, existe n_0 tal que **f(n) > c g(n)** para todo n ≥ n_0

Notações Assintóticas - Analogia

- **Analogia** entre duas funções **f e g** e dois números **a e b**:
  - **$f(n) = O(g(n))$ equivalente $a \leq b$**
  - **$f(n) = \Omega(g(n))$ equivalente $a \geq b$**
  - **$f(n) = \Theta(g(n))$ equivalente $a = b$**
  - **$f(n) = o(g(n))$ equivalente $a < b$**
  - **$f(n) = \omega(g(n))$ equivalente $a > b$**

Propriedades das Classes (1)

Transitividade:

Se $f(n) \in O(g(n))$ e $g(n) \in O(h(n))$, então $f(n) \in O(h(n))$.

Se $f(n) \in \Omega(g(n))$ e $g(n) \in \Omega(h(n))$, então $f(n) \in \Omega(h(n))$.

Se $f(n) \in \Theta(g(n))$ e $g(n) \in \Theta(h(n))$, então $f(n) \in \Theta(h(n))$.

Se $f(n) \in o(g(n))$ e $g(n) \in o(h(n))$, então $f(n) \in o(h(n))$.

Se $f(n) \in \omega(g(n))$ e $g(n) \in \omega(h(n))$, então $f(n) \in \omega(h(n))$.

Propriedades das Classes (2)

Reflexividade:

$f(n) \in O(f(n))$.

$f(n) \in \Omega(f(n))$.

$f(n) \in \Theta(f(n))$.

Simetria:

$f(n) \in \Theta(g(n))$ se, e somente se, $g(n) \in \Theta(f(n))$.

Simetria Transposta:

$f(n) \in O(g(n))$ se, e somente se, $g(n) \in \Omega(f(n))$.

$f(n) \in o(g(n))$ se, e somente se, $g(n) \in \omega(f(n))$.

Notação Assintótica – Algumas Regras Práticas

- **Multiplicação por uma constante:**

$$\Theta(c\, f(n)) = \Theta(f(n))$$
$$99\, n^2 = \Theta(n^2)$$

- **Mais alto expoente de um polinômio:**

$$a_x n^x + a_{x-1} n^{x-1} + \cdots + a_2 n^2 + a_1 n + a_0:$$
$$3\mathbf{n^3} - 5n^2 + 100 = \Theta(n^3)$$
$$6\mathbf{n^4} - 20n^2 = \Theta(n^4)$$
$$0.8\mathbf{n} + 224 = \Theta(n)$$

- **Termo dominante:**

$$\mathbf{2^n} + 6n^3 = \Theta(2^n)$$
$$\mathbf{n}! - 3n^2 = \Theta(n!)$$
$$n \log n + 3\mathbf{n^2} = \Theta(n^2)$$

Notação Assintótica – Dominância

- Quando uma função é **melhor** que outra?
  - Se queremos reduzir o tempo, funções "menores" são melhores
  - Uma função **domina** sobre outra se, a medida que n cresce, a função continua "maior"
  - **Matematicamente**: f(n) >> g(n) se: $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Relação de Dominância:
$n! \;>>\; 2^n \;>>\; n^3 \;>>\; n^2 \;>>\; n.\log n \;>>\; n \;>>\; \log n \;>>\; 1$

Notação Assintótica – Mais uma Visão Prática

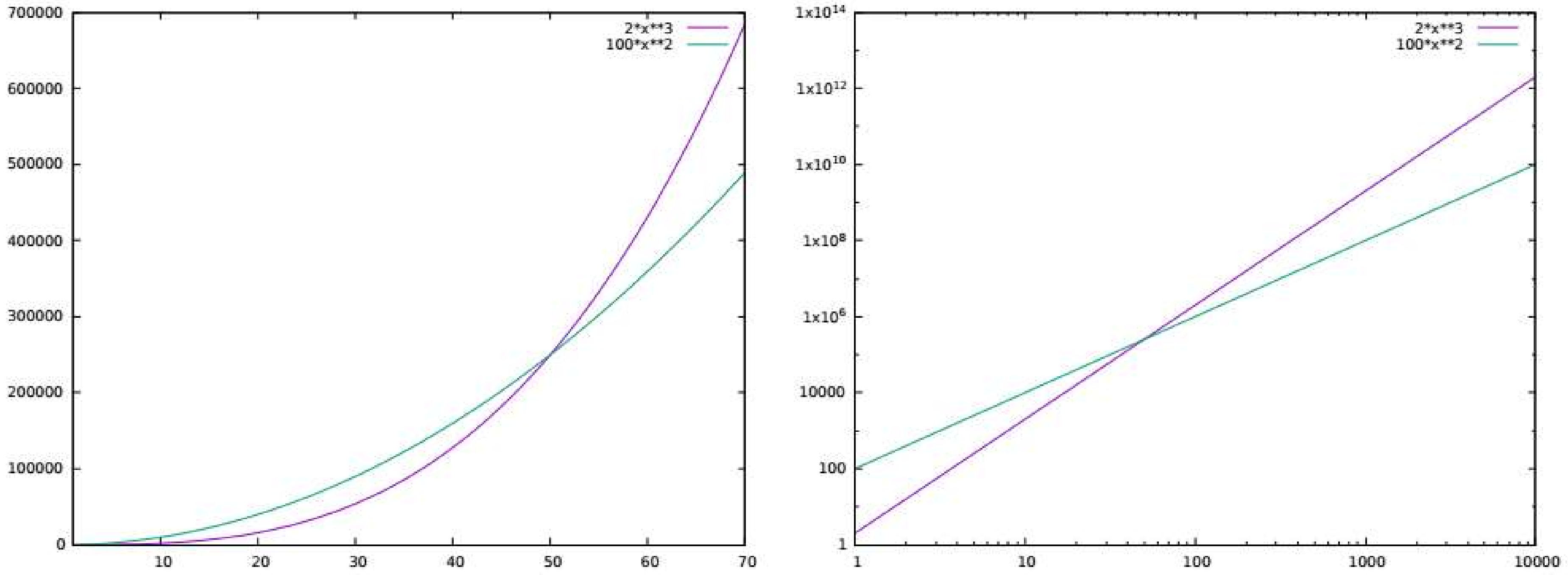
- **Premissa:** 01 operação leva 10^{-9} segundos

	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$
20	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	77 anos
30	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$1.07s$	
40	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	18.3 min	
50	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	13 dias	
100	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	10^{13} anos	
10^3	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$1s$		
10^4	$< 0.01s$	$< 0.01s$	$< 0.01s$	$0.1s$	16.7 min		
10^5	$< 0.01s$	$< 0.01s$	$< 0.01s$	$10s$	11 dias		
10^6	$< 0.01s$	$< 0.01s$	$0.02s$	16.7 min	31 anos		
10^7	$< 0.01s$	$0.01s$	$0.23s$	1.16 dias			
10^8	$< 0.01s$	$0.1s$	$2.66s$	115 dias			
10^9	$< 0.01s$	$1s$	$29.9s$	31 anos			

Notação Assintótica – Desenhando Funções (1)

- **Comparando: $2n^3$** com **$100n^2$** usando o gnuplot

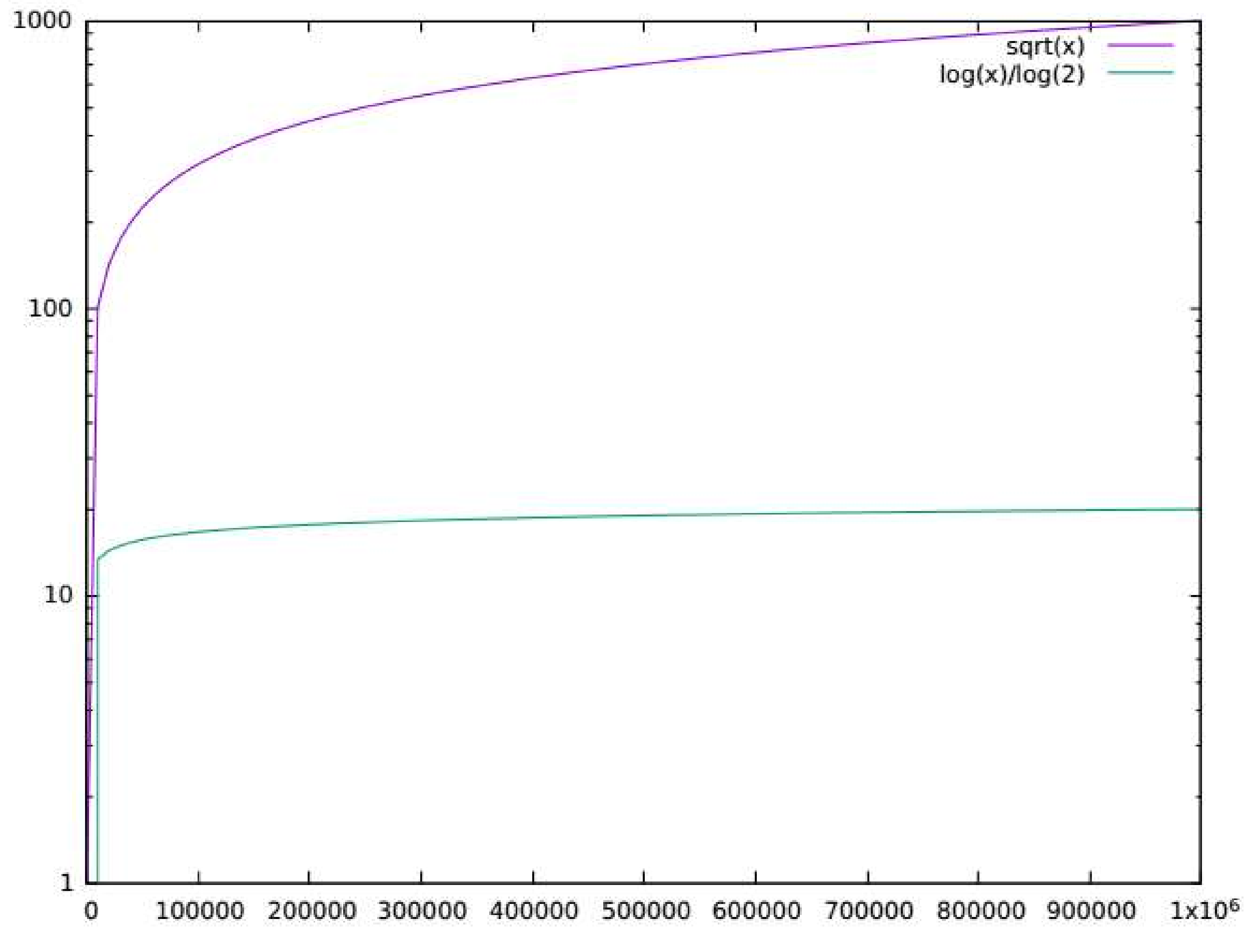
```
gnuplot> plot [1:70] 2*x**3, 100*x**2
gnuplot> set logscale xy 10
gnuplot> plot [1:10000] 2*x**3, 100*x**2
```



Notação Assintótica – Desenhando Funções (2)

- **Comparando:** $\sqrt{\mathbf{n}}$ com $\mathbf{log_2}\, \boldsymbol{n}$ usando o gnuplot

```
gnuplot> set logscale y 10
gnuplot> plot [1:1000000] sqrt(x), log(x)/log(2)
```



Classes de Comportamentos Assintóticos (1)

- **T(n) = O(1) ou Complexidade Constante:**
  - A complexidade do algoritmo independe do tamanho da entrada, ou seja, de n
  - Ocorre quando as instruções do algoritmo são executadas um número constante de vezes
- **Exemplo:**
  - Laços de repetição com condição de parada fixa

Classes de Comportamentos Assintóticos (2)

- **T(n) = O(log n) ou Complexidade Logarítmica:**
  - Ocorre tipicamente em algoritmos que dividem o problema original em subproblemas e algoritmos que usam árvores.
  - O tempo de execução pode ser considerado menor do que uma constante grande. Supondo logaritmo na base 2:
    - $n = 1.000$, $\log_2 n \approx 10$;
    - $n = 1.000.000$, $\log_2 n \approx 20$
- **Exemplo:**
  - Busca Binária

Classes de Comportamentos Assintóticos (3)

- **T(n) = O(n) ou Complexidade Linear:**
  - Normalmente, os algoritmos de complexidade linear realizam alguma operação leve sobre cada elemento da entrada.
  - É a melhor situação possível para um algoritmo que precisa processar n elementos de entrada ou produzir n elementos de saída
- **Exemplo:**
  - Pesquisa Sequencial

Classes de Comportamentos Assintóticos (4)

- **T(n) = O(n. log n) ou Complexidade Linear Logarítmica:**
  - Ocorre tipicamente em algoritmos que dividem o problema original em subproblemas, resolvem cada um deles e depois agrupam os resultados em um só
  - Muito associado ao paradigma Divisão e Conquista
  - Supondo logaritmo na base 2:
    - $n = 1.000.000$, $n \log_2 n \approx 20.000.000$
    - $n = 2.000.000$, $n \log_2 n \approx 42.000.000$
- **Exemplo:**
  - *Merge Sort*

Classes de Comportamentos Assintóticos (5)

- **$T(n) = O(n^2)$ ou Complexidade Quadrática:**
  - Ocorre quando os elementos da entrada são processados aos pares, como em laços aninhados
  - Sempre que n dobra, a complexidade é multiplicada por 4
  - Embora pareçam ruins, são úteis para resolver instâncias relativamente pequenas
- **Exemplo:**
  - *Insertion Sort, Bubble Sort, Quick Sort* (Pior Caso)

Classes de Comportamentos Assintóticos (6)

- **$T(n) = O(n^3)$ ou Complexidade Cúbica:**
  - Geralmente são úteis apenas para resolver problemas relativamente pequenos, ou quando comprovadamente o pior caso não é frequente
  - Sempre que n dobra, a complexidade é multiplicada por 8
  - Para alguns problemas, os melhores algoritmos conhecidos são cúbicos
- **Exemplo:**
  - Multiplicação de Matrizes, Algoritmos para Problemas NP-Difíceis

Classes de Comportamentos Assintóticos (7)

- **$T(n) = O(k^n)$ ou Complexidade Exponencial:**
  - Não são úteis do ponto de vista prático caso esta complexidade seja uma perspectiva frequente
  - São associados **a força bruta** (ou **busca exaustiva**).
  - Sempre que n dobra, a complexidade se eleva ao quadrado
- **Exemplo:**
  - Algoritmos para o Problema do Caixeiro Viajante

Classes de Comportamentos Assintóticos (8)

- **T(n) = O(n!) ou Complexidade Fatorial:**
  - Também é considerado como complexidade exponencial, embora apresente um comportamento muito pior do que $O(k^n)$
  - Também associado ao paradigma de força bruta e à geração de permutações
  - Cresce muito rapidamente:
    - n = 20, n! resulta em um número com 19 dígitos
    - n = 40, n! resulta em um número com 48 dígitos
- **Exemplo:**
  - Geração de permutações

Conclusões

- Na **análise de complexidade** de um algoritmo estamos **mais interessados** no seu **comportamento geral** do que em outros detalhes que podem depender da máquina, do sistema operacional, da linguagem, dos compiladores, etc

- Procura-se **medir a complexidade** de um algoritmo em **função** de um **parâmetro** do problema → o **tamanho da entrada**
  - Alguma **operação** (ou conjunto de operações) devem **balizar a análise de complexidade** de um algoritmo

- Considera-se a **eficiência assintótica** dos algoritmos executados em máquinas **que operam em um determinado modelo computacional**
  - **Dois algoritmos** para o **mesmo problema** com **análises assintóticas iguais** precisam **ser comparados em experimentos computacionais**

Resoluções de Exercícios em Sala (1)

- **Provar que $T(n) = 100n = O(n^2)$?**
  - **$O(g(n))$ =** *{f(n): existe constantes positivas c e n_0 tais que* ***$0 \leq f(n) \leq c.g(n)$****, para todo $n \geq n_0$}*
  - Precisamos definir constante c e n_0 tal que $100n \leq cn^2$
  - Para simplificar, dividimos os dois lados por n.
    - $100 \leq c.n$
  - A afirmação acima é verdade para $c = 1$ e $n \geq 100$
  - Portanto, **$T(n) = 100n$ é $O(n^2)$!**

Resoluções de Exercícios em Sala (2)

- **Provar que $T(n) = 6n^3 \neq \Theta(n^2)$?**
  - **$\Theta(g(n))$ =** *{f(n): existe constantes positivas c_1, c_2 e n_0 tais que* ***$0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)$****, para todo $n \geq n_0$}*
  - A título de **contradição**, suponha que existam c_2 e n_0 tais que $6n^3 \leq c_2n^2$ para todo $n \geq n_0$
  - Para simplificar, dividimos os dois lados por n^2:
    - $6.n \leq c_2$
  - O que não pode ser válido para um valor de n arbitrariamente grande, já que c_2 é constante
  - Portanto, **$T(n) = 6n^3 \neq \Theta(n^2)$!**

PANC: Projeto e Análise de Algoritmos

Aula 05: Notação Assintótica e Crescimento de Funções

Breno Lisi Romano

Dúvidas???

http://sites.google.com/site/blromano

Instituto Federal de São Paulo – IFSP São João da Boa Vista
Bacharelado em Ciência da Computação – 3º Semestre

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Definições Equivalentes

$$f(n) \in o(g(n)) \text{ se } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

$$f(n) \in O(g(n)) \text{ se } \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

$$f(n) \in \Theta(g(n)) \text{ se } 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

$$f(n) \in \Omega(g(n)) \text{ se } \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0.$$

$$f(n) \in \omega(g(n)) \text{ se } \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$

Resoluções de Exercícios em Sala (3)

- **Provar que n.lg n está em Ω(n)?**
  - **Ω(g(n))** = *{f(n): existe constantes positivas c e n_0 tais que* ***0 ≤ c.g(n) ≤ f(n)****, para todo n ≥ n_0}*
  - Precisamos definir constante c e n_0 tal que c.n ≤ n.lg n
  - Para simplificar, dividimos os dois lados por n.
    - c ≤ lg.n
  - A afirmação acima é verdade para c = 1 e n ≥ 2
  - Portanto, **T(n) = n.lgn é Ω(n)!**

Resoluções de Exercícios em Sala (4)

- **Provar que 100n não está em $\Omega(n^2)$?**
  - **$\Omega(g(n))$ =** *{f(n): existe constantes positivas c e n_0 tais que **$0 \leq c.g(n) \leq f(n)$**, para todo $n \geq n_0$}*
  - Precisamos definir constante c e n_0 tal que $c.n^2 \leq 100n$
    - Vamos assumir que $c \in N$
  - Resolvendo: $c . n . n \leq 100n$
    - $n . n \leq 100 n/c$
    - $n \leq 100/c$
    - Mas $n \in N$
    - Então temos valores de $n \geq 100/c \rightarrow$ Para c=1, temos $n \geq 100$
  - **Portanto, 100 não é $\Omega(n^2)$!**

Resoluções de Exercícios em Sala (5)

- Provar que $T(n) = \frac{1}{2}n^2 - 3n = \Theta(n^2)$?
  - **Definição: Θ(g(n)) =** *{f(n): existe constantes positivas c_1, c_2 e n_0 tais que* ***$0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)$****, para todo $n \geq n_0$}*

Resoluções de Exercícios em Sala (6)

- **Provar que $n^{1.5}$ está em $O(n^2)$?**
  - **O(g(n))** = *{f(n): existe constantes positivas c e n_0 tais que* ***$0 \leq f(n) \leq c.g(n)$****, para todo $n \geq n_0$}*
  - Precisamos definir constante c e n_0 tal que $n^{1.5} \leq cn^2$
  - Para simplificar, dividimos os dois lados por $n^{1.5}$:
    - $1 \leq c.n^{0.5}$
  - A afirmação acima é verdade para c = 1 e n ≥ 1
  - Portanto, **$T(n) = n^{1.5}$ é $O(n^2)$!**

Resoluções de Exercícios em Sala (7)

- Provar que $T(n) = n^3 + 2n^2 = \Omega(n^3)$?

Resoluções de Exercícios em Sala (8)

- **Provar que, considerando T(n) = n para n ímpar (n≥1) e T(n) = $\frac{n^2}{10}$ para n par (n≥0), então T(n)=Ω(n²) ?**