

# Introdução à Complexidade de Algoritmos

Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados I (AE22CP)  
Engenharia de Computação  
Departamento Acadêmico de Informática (Dainf)  
Universidade Tecnológica Federal do Paraná (UTFPR)  
Campus Pato Branco

- Análise de algoritmos
- Cálculo do Tempo de Execução
- Notação  $O$  (*Big Oh*)
- Taxas de Crescimento

- Por que estudar a complexidade de algoritmos?
- Você sabe diferenciar programa de algoritmo?

- Programa vs. Algoritmo

<b>Programa</b>	<b>Algoritmo</b>
Linguagem concreta (C, Java, Python)	Linguagem abstrata (pseudo-código)
Dependente de sistema operacional	Independente de sistema operacional
Dependente de hardware	Independente de hardware
Avaliação em tempo real (empírica)	Avaliação por estimativa (assintótica)

## Análise de algoritmos

- Após a implementação, é importante determinar os recursos necessários para a sua execução:
  - Tempo
  - Espaço
- Um algoritmo que soluciona um determinado problema, mas requer o processamento de um ano, não deve ser usado

- O que dizer de uma afirmação como a abaixo?  
"Desenvolvi um novo algoritmo chamado TripleX que leva 14,2 segundos para processar 1.000 números, enquanto o método SimpleX leva 42,1 segundos."
- Você trocaria o SimpleX que roda em sua empresa pelo TripleX?

- A afirmação tem que ser examinada, pois há diversos fatores envolvidos:
  - Características da máquina
  - Linguagem de programação
  - Implementação pouco cuidadosa do algoritmo SimpleX vs. "super" implementação do algoritmo TripleX
  - Quantidade de dados processados: Se o TripleX é mais rápido para processar 1.000 números, ele também é mais rápido para processar quantidades maiores de números?



- A comunidade de computação começou a pesquisar formas de comparar algoritmos de forma independente de
  - Hardware
  - Sistema operacional
  - Linguagem de programação
  - Habilidade do programador
- É desejável a comparação de algoritmos e não programas
- Área: análise/complexidade de algoritmos
  - Comparação de algoritmos
  - Determinar se um algoritmo é ótimo

- Sabe-se que:
  - Processar 100.000 números leva mais tempo do que 10.000 números
  - Cadastrar 20 itens em um sistema de vendas leva mais tempo do que cadastrar 10
  - Etc
- Pode ser uma boa ideia estimar a eficiência de um algoritmo em função do tamanho do problema (número de elementos processados):
  - Geralmente, é assumido que  $n$  é o tamanho do problema
  - É calculado o número de operações realizadas sobre os  $n$  elementos

- O melhor algoritmo é aquele que requer menos operações sobre a entrada
- Que operações?
- Toda operação leva o mesmo tempo?

- Exemplo: TripleX vs. SimpleX
  - TripleX: para uma entrada de tamanho  $n$ , o algoritmo realiza  $n^2 + n$  operações:
    - $f(n) = n^2 + n$
  - SimpleX: para uma entrada de tamanho  $n$ , o algoritmo realiza  $1.000n$  operações:
    - $g(n) = 1.000n$

- Exemplo: TripleX vs. SimpleX

<b>Tamanho da Entrada</b>	<b>1</b>	<b>10</b>	<b>100</b>	<b>1.000</b>	<b>10.000</b>
$f(n) = n^2 + n$					
$g(n) = 1.000n$					

- Exemplo: TripleX vs. SimpleX

Tamanho da Entrada	1	10	100	1.000	10.000
$f(n) = n^2 + n$	2	110	10.100	<b>1.001.000</b>	<b>100.010.000</b>
$g(n) = 1.000n$	<b>1.000</b>	<b>10.000</b>	<b>100.000</b>	1.000.000	10.000.000

- partir de  $n = 1.000$ ,  $f(n)$  mantém-se maior e cada vez mais distante de  $g(n)$ 
  - Diz-se que  $f(n)$  cresce mais rápido do que  $g(n)$

## Cálculo do Tempo de Execução

# Cálculo do Tempo de Execução

- Existem basicamente 2 formas de estimar o tempo de execução de programas e decidir quais são os melhores:
  - Empiricamente
  - Teoricamente
- É desejável e possível estimar qual o melhor algoritmo sem ter que executá-los: função da análise de algoritmos



# Cálculo do Tempo de Execução

- Para proceder a uma análise de algoritmos e determinar as taxas de crescimento, necessitamos de um modelo de computador e das operações que executa
- Assume-se o uso de um computador tradicional, em que as instruções de um algoritmo são executadas sequencialmente

- Repertório de instruções simples: soma, multiplicação, comparação, atribuição, etc
  - Por simplicidade e viabilidade da análise, assume-se que cada instrução demora exatamente uma unidade de tempo para ser executada
  - Operações complexas, como inversão de matrizes e ordenação de valores, não são realizadas em uma única unidade de tempo
  - Operações complexas devem ser analisadas em partes

# Cálculo do Tempo de Execução

- Regras para o cálculo de execução

- Repetições:

```
para  $i \leftarrow 0$  até  $n$  faça  
   $x \ += \ 1$ ;
```

# Cálculo do Tempo de Execução

- Regras para o cálculo de execução
  - Repetições:
    - No exemplo abaixo são realizadas  $3n + 2$  operações (uma unidade para iniciar  $i \leftarrow 0$  \* (incremento na variável  $i$  + uma comparação + atribuição na variável  $x$ ) + uma última comparação, que é o momento em que a variável  $i$  atinge o valor de  $n$ )
    - Por mais que o operador  $+=$  seja equivalente a duas operações (uma atribuição e uma soma), o mesmo é contado como uma unidade

```
para  $i \leftarrow 0$  até  $n$  faça  
     $x += 1$ ;
```

- Regras para o cálculo de execução
  - Se... então... senão:  
se  $i < j$   
então  $i \leftarrow i + 1$   
senão para  $k \leftarrow 0$  até  $n$  faça  
     $i \leftarrow i * k;$

# Cálculo do Tempo de Execução

- Regras para o cálculo de execução
  - Se... então... senão:
    - Para uma cláusula condicional, o tempo de execução nunca é maior do que o tempo do teste (então) mais o tempo do senão
    - No cálculo do tempo de execução, é considerado o comando que leva mais tempo
    - O exemplo abaixo pode executar até  $4n + 3$  instruções

```
se  $i < j$ 
  então  $i \leftarrow i + 1$ 
  senão para  $k \leftarrow 0$  até  $n$  faça
     $i \leftarrow i * k;$ 
```

- Regras para o cálculo de execução

- Repetições aninhadas

```
para  $i \leftarrow 0$  até  $n$  faça  
  para  $j \leftarrow 0$  até  $n$  faça  
     $k \leftarrow k + 1$ ;
```

- Regras para o cálculo de execução
  - Repetições aninhadas
    - A análise é feita de dentro para fora
    - Tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições

```
para  $i \leftarrow 0$  até  $n$  faça  
  para  $j \leftarrow 0$  até  $n$  faça  
     $k \leftarrow k + 1$ ;
```

- Nas duas últimas linhas do código acima (laço interno), são realizadas  $4n + 2$  operações: uma atribuição para a variável  $j + n * ($  uma atualização de  $j +$  mais uma comparação entre  $i$  e  $n +$  uma atribuição na variável  $k +$  uma soma na variável  $k)$
- A operação acima é realizada  $n$  vezes, ou seja, o total de operações no fragmento de código acima é  $n * (4n + 2 + 2) + 2 = 4n^2 + 4n + 2$



- Regras para o cálculo de execução
  - Chamadas de sub-rotinas: uma sub-rotina deve ser analisada primeiro e depois ter suas unidades de tempo incorporadas ao programa/sub-rotina que a chamou

# Cálculo do Tempo de Execução

- Supondo que as operações simples demoram uma unidade de tempo para executar, considere o programa abaixo para calcular o resultado de

$$\sum_{i=1}^n i^3$$

- 1 Início
- 2 declare soma\_parcial numérico;
- 3 soma\_parcial  $\leftarrow$  0;
- 4 para  $i \leftarrow 0$  até  $n$  faça
- 5    soma\_parcial  $\leftarrow$  soma\_parcial +  $i*i*i$ ;
- 6    escreva(soma\_parcial);
- 7 Fim

$$\sum_{i=1}^n i^3$$

- 3 1 unidade de tempo
- 4 1 unidade para iniciação de  $i$ ,  $n + 1$  unidades para testar se  $i = n$  e  $n$  unidades para incrementar  $i = 2n + 2$
- 5 4 unidades (1 da soma, 2 das multiplicações e 1 da atribuição) executada  $n$  vezes (pelo comando "para") =  $4n$  unidades
- 6 1 unidade para escrita
- **Custo total:**  $6n + 4$

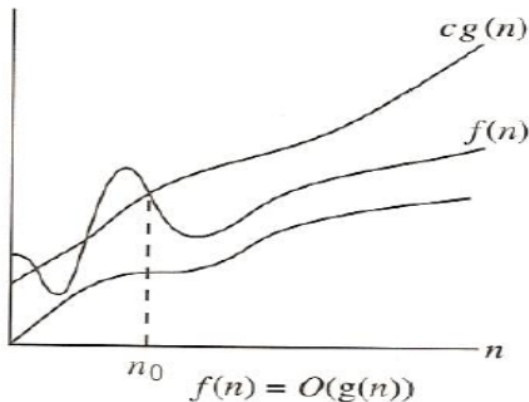
## Notação O (*Big Oh*)

# Notação $O$ (*Big Oh*)

- Na análise de algoritmos, devemos nos preocupar com a eficiência quando o tamanho da entrada ( $n$ ) for grande
- A comparação de algoritmos é feita por meio de análise assintótica
- Notação Big-Oh ( $O$ )

# Notação $O$ (*Big Oh*)

- Dada duas funções,  $f(n)$  e  $g(n)$ 
  - Uma função  $f(n)$  é da ordem de (*big-oh*)  $g(n)$  ou função  $f(n)$  é  $O(g(n))$  se existirem duas constantes positivas  $c$  e  $n_0$  tais que  $f(n) \leq cg(n)$ , para todo  $n \geq n_0$



# Notação $O$ (*Big Oh*)

- Exemplos:

- Seja  $f(n) = (n + 1)^2$ . Logo,  $f(n)$  é  $O(n^2)$

- Seja  $f(n) = 2n^3 + n^2 + 3n$ . Logo,  $f(n)$  é  $O(n^3)$

- Exemplos:
  - Seja  $f(n) = (n + 1)^2$ . Logo,  $f(n)$  é  $O(n^2)$ , quando  $n_0 = 1$  e  $c = 4$ , pois  $(n + 1)^2 \leq 4n^2$  para  $n \geq 1$
  - Seja  $f(n) = 2n^3 + n^2 + 3n$ . Logo, para provar que  $f(n)$  é  $O(n^3)$ , basta provar que  $2n^3 + n^2 + 3n \leq 6n^3$  para  $n \geq 0$
- Notação  $O$  é a mais utilizada para a análise de complexidade de algoritmos
- Ao dizer que  $f(n) = O(g(n))$ , tem-se que  $g(n)$  é o limite superior em comparação com  $f(n)$



# Notação $O$ (*Big Oh*)

- Quando dizemos que um algoritmo possui um custo de  $O(n^2)$ , isso significa que, no pior caso, o algoritmo nunca terá um custo maior que  $n^2$
- Se  $T(x)$  é um polinômio de grau  $n$ , então  $T(x) = O(x^n)$ 
  - Em uma expressão polinomial (e.g.  $2n^3 + n^2 + n + 1$ ), o custo na notação *Big-Oh* sempre será denotada em função do maior expoente, sem considerar constantes
  - Uma função que realiza  $2n^3 + n^2 + n + 1$  operações no pior caso tem a complexidade na ordem de  $O(n^3)$

## Notação $O$ (*Big Oh*)

- Se  $T_1(n) = O(f(n))$  e  $T_2(n) = O(g(n))$ , então:
  - $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$
  - $T_1(n) * T_2(n) = O(f(n) * g(n))$
- Se  $T(n)$  for igual a um valor constante, então a sua complexidade é  $O(1)$

## Taxas de Crescimento

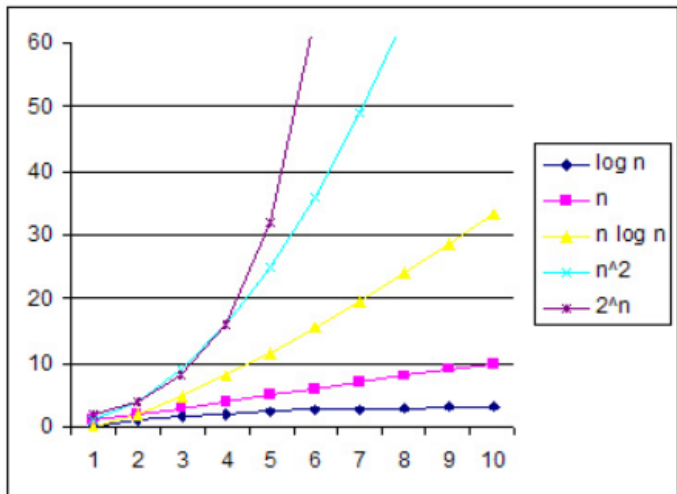
# Taxas de Crescimento

- Funções e taxas de crescimento mais comuns:

$c$	constante
$\log n$	logarítmica
$n$	linear
$n \log n$	linear
$n^2$	quadrática
$n^3$	cúbica
$2^n$	exponencial
$a^n$	exponencial

# Taxas de Crescimento

- Crescimentos de algumas funções





Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.  
*Introduction to Algorithms*.  
Third edition, The MIT Press, 2009.



Horowitz, E., Sahni, S. Rajasekaran, S.  
*Computer Algorithms*.  
Computer Science Press, 1998.



Marin, L. O.  
Complexidade de Algoritmos – parte 2. AE23CP-3CP  
Algoritmos e Estrutura de Dados II.  
*Slides*. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2017.



Marin, L. O.

Complexidade de Algoritmos – parte 1. AE23CP-3CP  
Algoritmos e Estrutura de Dados II.

*Slides.* Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2018.



Rosa, J. L. G.

Análise de Algoritmos. SCE-181 – Introdução à Ciência da Computação II.

*Slides.* Ciência de Computação. ICMC/USP, 2008.



Rosa, J. L. G.

Análise de Algoritmos - parte 1. SCC-201 – Introdução à Ciência da Computação II.

*Slides.* Ciência de Computação. ICMC/USP, 2016.



Ziviani, N.

*Projeto de Algoritmos - com implementações em Java e C++.*

Thomson, 2007.