



CENTRO DE CIÊNCIA E TECNOLOGIA  
LABORATÓRIO DE CIÊNCIAS MATEMÁTICAS  
UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE

# Introdução à Eficiência de Algoritmos

*Disciplina: Estruturas de Dados I*

**Prof. Fermín Alfredo Tang Montané**

**Curso: Ciência da Computação**

# Eficiência de Algoritmos

## Discussão

---

- Raramente existe um único algoritmo para resolver qualquer problema.
- Quando comparamos dois algoritmos diferentes para resolver o mesmo problema, com frequência podemos encontrar que um algoritmo é uma ordem de magnitude mais eficiente que o outro.
- Neste caso, faz sentido tentar reconhecer e escolher o algoritmo mais eficiente.
- Embora os cientistas de computação tenham estudado os algoritmos e a sua eficiência de maneira bastante ampla, esta área não possui um nome oficial.
- Brassard e Bratley, cunharam o termo **algorithmics**, ou algorítmica, que seria o estudo sistemático de técnicas fundamentais para projetar e analisar algoritmos eficientes.

# Eficiência de Algoritmos

## Discussão

---

- Quando um algoritmo é sequencial, ou seja ele não possui loops ou usa recursividade, a sua eficiência depende do número de instruções que ele contêm.
- Neste caso, independente do tamanho dos dados processados,  $n$ , o algoritmo executara em tempo constante  $C$ .
- Considerando que o tempo de execução depende do número de operações executadas pelo algoritmo poderíamos representar este número de operações como uma função do número de dados  $n$ . Temos assim:

$$f(n) = C$$

- Neste caso, a eficiência do algoritmo depende da velocidade do computador, entre outras coisas. Mas isso não muda o fato de que o esforço realizado pelo algoritmo é constante, independente do tamanho da entrada.
- Por outro lado, algoritmos que utilizam loops ou recursividade variam consideravelmente de eficiência.
- Faremos uma discussão sobre eficiência de algoritmos baseada em loops.

# Eficiência de Algoritmos

## Discussão

---

- Ao discutir a eficiência de algoritmos consideramos uma função  $f(n)$  do número de elementos processados,  $n$ , pelo algoritmo (tamanho da entrada de dados) que pode representar o número de operações realizadas pelo algoritmo ou de maneira mais simplificada, o número de iterações realizadas pelo algoritmo.
- A forma desta função seria:

$$f(n) = \text{equação}$$

# Eficiência de Algoritmos

## Loops Lineares

- O tipo de loop mais simples é o loop linear. Neste caso o número de iterações é diretamente proporcional ao parâmetro de controle,  $n$ , do loop. Quanto maior o valor do  $n$ , maior será proporcionalmente o número de repetições.

```
for (i = 0; i < 1000; i++)
    application code
```

- Queremos saber quantas vezes o corpo do loop é repetido?
- Neste caso, a resposta é exatamente 1000.
- Vale observar que o número exato de operações realizadas neste loop depende de quantas operações estão sendo realizadas no corpo do loop. p.e. se forem realizadas 5 operações no corpo do loop, teremos 5.000 operações em total.
- Independente do número de operações no corpo loop, teremos que o número de operações será múltiplo de 1000 (múltiplo de  $n$ ).
- Assim, temos que a eficiência é linear (diretamente proporcional ao número de iterações  $n$ ), com isso:

$$f(n) = n$$

# Eficiência de Algoritmos

## Loops Lineares

- O primeiro exemplo, foi bastante direto. No entanto podemos ter eficiência linear em outros casos como o mostrado no seguinte exemplo:

```
for (i = 0; i < 1000; i += 2)
    application code
```

- Queremos saber quantas vezes o corpo do loop é repetido?
- Como o índice  $i$  incrementa de dois em dois, a resposta é 500.
- Neste exemplo, o número de iterações é metade do parâmetro de controle,  $n$ , do loop.
- O número de iterações continua sendo diretamente proporcional ao valor de  $n$ , mais precisamente ao valor de  $n/2$ . Quanto maior o valor de  $n$ , maior o número de iterações.
- Assim, temos que a eficiência é diretamente proporcional a metade do número de iterações  $n$ , com isso:

$$f(n) = n / 2$$

- Se fossemos desenhar qualquer uma destas funções  $f(n)$  teríamos uma reta, por isso são chamadas de **funções de desempenho linear**.

# Eficiência de Algoritmos

## Loops Lineares

- Para ajudar a visualizar a resposta mostra-se uma tabela com os valores da variável de controle  $i$  até atingir o valor de  $n = 1.000$  e finalizar o loop.

Incremento de 1		Incremento de 2		Incremento de 3	
Iteração	Valor de i	Iteração	Valor de i	Iteração	Valor de i
1	0	1	0	1	0
2	1	2	2	2	3
3	2	3	4	3	6
4	3	4	6	4	9
5	4	5	8	...	...
...	...	...	...	334	999
...	...	500	998	(exit)	1002
1000	999	(exit)	1000		
(exit)	1000				

- Observe que o número de iterações é proporcional ao valor de  $n = 1.000$ .

# Eficiência de Algoritmos

## Loops Logarítmicos

- Em um loop linear, o contador de iterações varia seja por incrementos ou decrementos, ou seja, a variável de controle  $i$  é adicionada ou subtraída em uma constante em cada iteração.
- Em um loop logarítmico, o contador de iterações varia seja por múltiplos ou fatores, ou seja, a variável de controle  $i$  é multiplicada ou dividida por uma constante, em cada iteração.
- Considere os seguintes exemplos de loops logarítmicos:

Multiply Loops

```
for (i = 1 ; i < 1000; i *= 2)  
    application code
```

Divide Loops

```
for (i = 1000 ; i >= 1 ; i /= 2)  
    application code
```

- Novamente queremos saber quantas vezes o corpo do loop é repetido?
- Para ajudar a visualizar a resposta mostra-se uma tabela com os valores da variável de controle de  $i$  até superar o valor de 1000 e finalizar o loop.

# Eficiência de Algoritmos

## Loops Logarítmicos

- Para ajudar a visualizar a resposta mostra-se uma tabela com os valores da variável de controle  $i$  até superar o valor de  $n = 1.000$  e finalizar o loop.

Multiply		Divide	
Iteration	Value of i	Iteration	Value of i
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
(exit)	1024	(exit)	0

- Observe que apesar do valor de  $n = 1.000$  o número de iterações é 10 em ambos casos.

# Eficiência de Algoritmos

## Loops Logarítmicos

- Observe que apesar do valor de  $n = 1.000$  o número de iterações é 10 em ambos casos. O motivo disso é que o valor da variável de controle  $i$  dobra no loop multiplicativo e cai pela metade no loop de divisão.
- O número de iterações é uma função do fator de multiplicação ou de divisão, que neste caso é 2. O loop continua enquanto a condição mostrada embaixo é verdadeira:

```
multiply 2Iterations < 1000  
divide    1000 / 2Iterations >= 1
```

- Neste caso, o número de iterações é proporcional ao logaritmo em base 2 de 1000.

$$\log_2 1000$$

- Em geral, podemos dizer que este tipo de loops tem comportamento logarítmico. A base do logaritmo pode variar dependendo do fator de multiplicação ou divisão. O mais comum é 2.
- Vale ressaltar que a taxa de crescimento de uma função logarítmica é muito inferior ao de uma linear.
- Em geral, temos que a eficiência é logarítmica:  $f(n) = \log n$

# Eficiência de Algoritmos

## Loops Logarítmicos

- É importante observar, que quando não explicitada a base do logaritmo, na área de computação, refere-se a base 2. Assim:  $\log n \equiv \log_2 n$ .
- Por outro lado, vale ressaltar que  $\log n$  em outros contextos considera base 10.
- $\log n \equiv \log_{10} n$ .
- Temos ainda, o logaritmo neperiano denotado como  $\ln n \equiv \log_e n$  que considera a base e.
- Para realizar o calculo de um logaritmo com ajuda de uma calculadora científica devemos considerar base 10 ou e. A seguinte formula de conversão de bases de

$$\frac{\log n}{\log 2} = \log_2 n$$

$$\frac{\ln n}{\ln 2} = \log_2 n$$

# Eficiência de Algoritmos

## Loops Aninhados

- Os loops aninhados são loops que contêm outros loops.
- Quando analisamos loops aninhados, primeiro devemos determinar quantas iterações realiza cada loop. Com isso, o número total de iterações do loop aninhado é igual ao produto do número de iterações no loop interior vezes o número de iterações do loop exterior.

```
Iterations = outer loop iterations x inner loop iterations
```

- Mostraremos três tipos de loops aninhados mais representativos: linear logarítmico, quadrático, e dependente quadrático.

# Eficiência de Algoritmos

## Loops Aninhados: Linear Logarítmico

- Em um loop linear logarítmico, temos dois loop aninhados. Um loop linear com controle por incrementos ou decrementos e outro loop logarítmico com controle por multiplicação ou divisão de um fator. Considere o seguinte exemplo:

```
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j *= 2)
        application code
```

- No exemplo, o loop exterior é linear, executa 10 vezes. Enquanto o loop interior é logarítmico, executa  $\lceil \log_2 10 \rceil$ . Temos assim, que o número total de iterações do loop aninhado é:

$$10 \log_2 10$$

- Em geral, considerando que  $n$  é o número de iterações do loop linear e que  $n$  é o limite do loop logarítmico, temos que a eficiência do loop aninhado é dada por uma função linear logarítmica.

$$f(n) = n \log n$$

# Eficiência de Algoritmos

## Loops Aninhados: Quadrático

- Em um loop quadrático, o número de vezes que o loop interior executa é igual ao loop exterior. Considere o seguinte exemplo:

```
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        application code
```

- O loop exterior (for  $i$ ) executa 10 vezes. Para cada uma de suas iterações, o loop interior (for  $j$ ) também executa 10 vezes. O número total de iterações é portanto  $10 \times 10 = 100$  iterações.
- Em geral, considerando que  $n$  é o número de iterações de cada loop temos que a eficiência do loop é dada por uma função quadrática.

$$f(n) = n^2$$

# Eficiência de Algoritmos

## Loops Aninhados: Quadrático dependente

- Em um loop quadrático dependente, o número de iterações do loop interior depende do loop exterior. Considere o seguinte exemplo:

```
for (i = 0; i < 10; i++)
    for (j = 0; j <= i; j++)
        application code
```

- O loop exterior é um loop linear que executa 10 vezes. Já o loop interior depende do exterior para definir o número de iterações em cada uma das 10 repetições.
- O loop interior é executado apenas uma vez na primeira iteração, duas vezes na segunda iteração, três vezes na terceira iteração e assim por diante.
- O número total de iterações do loop aninhado seria:

$$1 + 2 + 3 + \dots + 9 + 10 = 55$$

- Trata-se de uma série aritmética representada pela seguinte fórmula:

$$n\left(\frac{n+1}{2}\right)$$

- Em geral, eficiência do loop aninhado é dada por:

$$f(n) = n\left(\frac{n+1}{2}\right)$$

# Análise do Tempo de Execução

## Contando o número de Operações

---

- A análise do tempo de execução de um algoritmo permite medir o tempo de execução de um algoritmo sem necessidade de executá-lo propriamente em um computador.
- Uma primeira abordagem para obter uma medida do tempo de execução é contar o número operações (instruções simples) realizadas pelo algoritmo.
- O tempo de execução será **proporcional** ao número de operações realizadas que pela sua vez depende do **tamanho do problema** (tamanho da entrada).

# Análise do Tempo de Execução

## Contando o número de execuções de cada comando

- Exemplo I:

**Algoritmo 1 (n):**

**inicio**

**se** n=0 **então**

    p  $\leftarrow$  a[0]

{ 1 }

{ 1 }

**se-não**

    p  $\leftarrow$  a[0]

{ 1 }

    y  $\leftarrow$  x

{ 1 }

**para** i  $\leftarrow$  1 **até** n **faça** { n+1 }

        p  $\leftarrow$  p + a[i] \* y

{ n }

        y  $\leftarrow$  y \* x

{ n }

**fim-para**

**fim-se**

**fim**

**Tempo de Execução por Comando**

**se** n=0 : T(n) = 2

**se** n>0 : T(n) = 3n+4

# Análise do Tempo de Execução

## Contando o número de execuções de cada comando

- Exemplo 2:

**Algoritmo 2 (n):**

**inicio**

**para**  $i \leftarrow 1$  **até**  $n$  **faça**       $\{ n+1 \}$   
     $v[i] \leftarrow i$                            $\{ n \}$   
**para**  $j \leftarrow 2$  **até**  $n$  **faça**       $\{ n*n = n^2 \}$   
     $p \leftarrow v[i] * j$                        $\{ n*(n-1) = n^2 - n \}$

**fim-para**

**fim**

**Tempo de Execução por Comando**

**se**  $n > 0$  :       $T(n) = 2n^2 + n + 1$

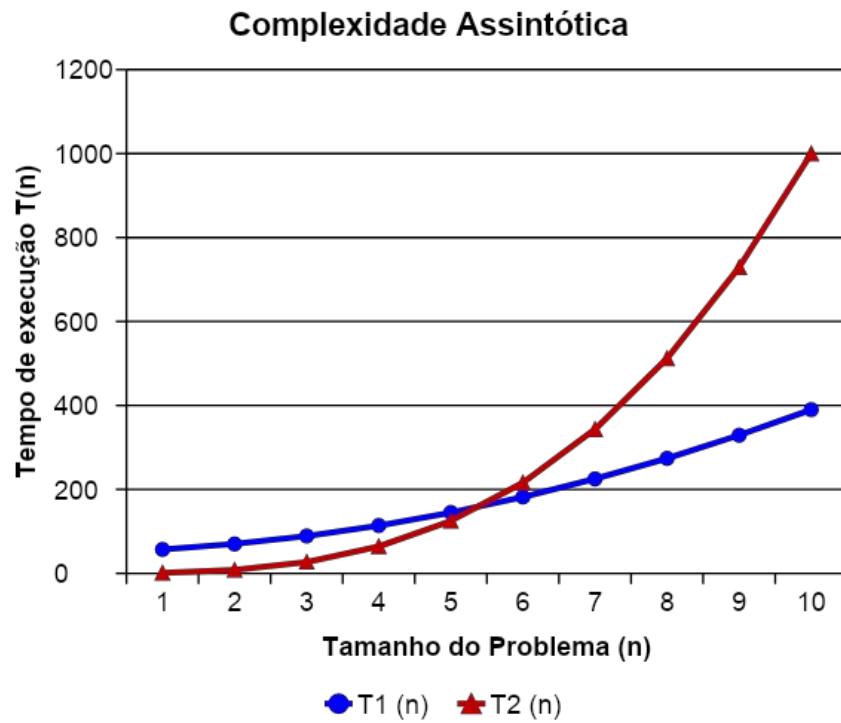
# Análise de Algoritmos

## Notação Assintótica

- **Limitantes assintóticas:**
- Na prática, ao medir o tempo de execução de um algoritmo podemos desprezar constantes aditivas e multiplicativas.
- Somente são levados em conta os termos de maior grau.

- **Exemplo:**

- $T(n) = an + b = O(n)$
- $T(n) = an^2+bn+c = O(n^2)$



# Notação Assintótica O

## Limite Assintótico Superior

---

- **Limite assintótico Superior:**
- Uma função  $T(n)$  é de ordem  $O(f(n))$ ,  $T(n) \in O(f(n))$ ,  
se existem constantes positivas  $c$  e  $N$ , tal que,  $T(n) \leq c f(n)$ ,  $\forall n \geq N$ .
- Ou seja:
- $f(n)$  supera  $T(n)$ , para  $n$  suficientemente grande;
- $f(n)$  é um **limite assintótico superior** para  $T(n)$ ;

# Notação Assintótica O

## Limite Assintótico Superior

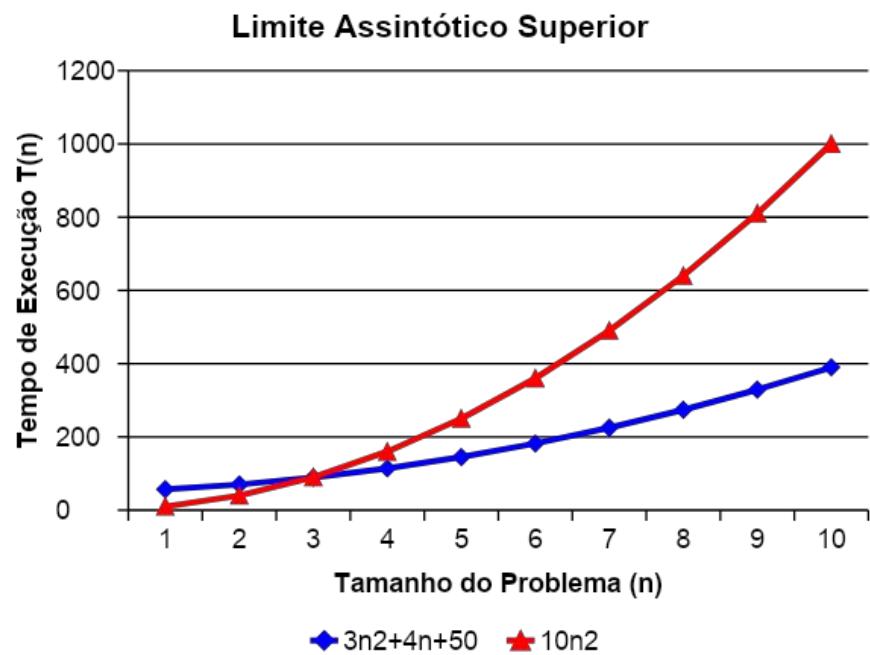
- **Exemplo:**
- Mostrar que  $T(n) = 3n^2 + 4n + 50$  é  $O(n^2)$
- Aplicando a definição, temos que  $f(n) = n^2$ .
- Devemos achar valores apropriados para  $c$  e  $N$ , de maneira que:  
 $T(n) \leq c f(n), \forall n \geq N$
- Temos que:  $T(n) = 3n^2 + 4n + 50$
- Logo, para  $N=1$  e  $c = 57$  temos que:  
$$3n^2 + 4n + 50 \leq 57n^2, \forall n \geq 1$$
- Alternativamente,  $N=3$  e  $c = 10$  temos que:  
$$3n^2 + 4n + 50 \leq 10n^2, \forall n \geq 3$$

# Notação Assintótica O

## Limite Assintótico Superior

- Exemplo:

	$3n^2+4n+50$	$57n^2$	$10n^2$
1	57	57	10
2	70	228	40
3	89	513	90
4	114	912	160
5	145	1425	250
6	182	2052	360
7	225	2793	490
8	274	3648	640
9	329	4617	810
10	390	5700	1000



# Técnicas para Análise de Algoritmos

## Operações com o Operador $O$

---

- **Exemplo:**
  - Se  $T(n) = 3n^3 + 4n^2 + 20$  então  $T(n)$  é de ordem  $O(n^3)$
  - Se  $T(n) = 25n^2 + 20n$  então  $T(n)$  é de ordem  $O(n^2)$
  - Se  $T(n) = 37n - 45$  então  $T(n)$  é de ordem  $O(n)$
  - Se  $T(n) = 59$  então  $T(n)$  é de ordem  $O(1)$
- Em geral:
  - Se  $T(n) = C$  então  $T(n)$  é de ordem  $O(1)$

# Técnicas para Análise de Algoritmos

## Blocos Simples, Condicionais: Se .. Então

- **Comandos Simples:**

```
v [i] ← 100;    // T(n) = c   O(1)  
v [j] ← v[i] - 200; // T(n) = c   O(1)
```

- **Bloco de comandos simples:**

**inicio**

```
v [i] ← 100; // T(n) = c   O(1)  
v [j] ← v[i] - 200; // T(n) = c   O(1)  
v [j] ← v[i] - 200; // T(n) = c   O(1)
```

O(1)

**fim**

- **Bloco Se .. Então:**

**Se** n=0 **então** // T(n) = c O(1)  
v [i] ← 100; // T(n) = c O(1)

O(1)

**Senão**

```
v [i] ← 100; // T(n) = c O(1)  
v [j] ← v[i] - 200; // T(n) = c O(1)  
v [j] ← v[i] - 200; // T(n) = c O(1)
```

O(1)

O(1)

**fim**

# Técnicas para Análise de Algoritmos

## Blocos de Repetição: Para .. Fim-Para

- **Bloco Para .. Fim-Para:**

```
Para i ← 1 ate n faça // T(n) = n O(n)
    v [i] ← 100;        // T(n) = c O(1)
```

**Fim-Para**

O(n)

```
Para i ← 1 ate n faça // T(n) = n O(n)
```

```
    Para j ← 1 ate n faça // T(n) = n O(n)
```

```
        w [i,j] ← 500;    // T(n) = c O(1)
```

**Fim-Para**

**Fim-Para**

O( $n^2$ )

# Técnicas para Análise de Algoritmos

## Blocos Combinados

- **Bloco Combinado:**

**Se  $w[i,i]=0$  entao**

```
Para i ← 1 ate n faca    // T(n) = n   O(n)
    Para j ← 1 ate n faca // T(n) = n   O(n)
        w [i,j] ← 500;      // T(n) = c   O(1)
    Fim-Para
Fim-Para
```

$O(n^2)$

**senão**

```
Para i ← 1 ate n faca    // T(n) = n   O(n)
    v [i] ← 100;           // T(n) = c   O(1)
Fim-Para
```

$O(n^2)$

$O(n)$

**fimse**

# Eficiência de Algoritmos

## Notação Big-O

- Dada a velocidade dos computadores atuais, não existe interesse em calcular uma medida exata da eficiência de um algoritmo, porém existe interesse em ter uma medida geral da sua ordem de magnitude.
- Se analise de dois algoritmos mostra que um deles executa 15 iterações e outro 25 iterações, eles serão tão rápidos que não perceberemos a diferença.
- Mas se por outro lado, um deles iterar 15 vezes enquanto o outro 1500 vezes, esta diferença é significativa.
- Mostramos que uma medida da eficiência de um algoritmo pode ser expressado mediante uma função  $f(n)$  onde  $n$  representa o número de dados a serem processados.
- Embora a função  $f(n)$  possa ter uma forma complexa, podemos aproximar ela, identificando o fator dominante dessa função, o que serve para determinar a ordem de magnitude da função.
- Com isso, não precisamos calcular a medida completa ou precisa de eficiência, apenas o fator que determina a sua eficiência.
- Este fator dominante é chamado big-O, e pode ser interpretado como “na ordem de magnitude de”, e é denotado como  $O(f(n))$ .

# Eficiência de Algoritmos

## Análise Big-O

- A notação Big-O pode ser calculada a partir de  $f(n)$  utilizando os seguintes passos:
  - Em cada termo de  $f(n)$  fazer o coeficiente igual a um. Ou seja, descartar os coeficientes dos termos da função  $f(n)$ .
  - Manter apenas o termo de maior magnitude da função  $f(n)$ .
- Os termos podem ser classificados em ordem de magnitude de menor a maior da seguinte maneira:

log n   n   n log n    $n^2$     $n^3 \dots n^k$     $2^n$     $n!$

# Notação Big-O

## Exemplo 1

- Considere a seguinte expressão:

$$f(n) = n \frac{(n+1)}{2} = \frac{1}{2} n^2 + \frac{1}{2} n$$

- Para calcular a expressão  $O(f(n))$  que mede o desempenho de pior caso do algoritmo com base na função  $f(n)$ .
- Primeiro, eliminam-se todos os coeficientes da expressão, temos assim:

$$n^2 + n$$

- Logo, escolhe-se o termo de maior magnitude e descartam-se os outros, temos assim:

$$n^2$$

- Com isso podemos afirmar que a ordem de  $f(n)$  de pior caso é:

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

# Notação Big-O

## Exemplo 2

- Considere a seguinte expressão polinomial:

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_2 n^2 + a_1 n + a_0$$

- Para calcular a expressão  $O(f(n))$  que mede o desempenho do algoritmo de pior caso com base na função  $f(n)$ .
- Primeiro, elimina-se todos os coeficientes da expressão:

$$f(n) = n^k + n^{k-1} + \dots + n^2 + n + 1$$

- Logo, escolhe-se o termo de maior valor e descartam-se os outros termos.

$$f(n) = n^k$$

- Com isso podemos afirmar que a ordem de uma expressão polinomial é:

$$O(f(n)) = O(n^k)$$

# Eficiência de Algoritmos

## Medidas Padrão de Eficiência

- Os cientistas da computação definiram sete categorias de eficiência de algoritmos principais ou mais representativos. Essas categorias são mostradas na tabela, em ordem de maior eficiência para menor eficiência.

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	microseconds
Linear	$O(n)$	10,000	seconds
Linear logarithmic	$O(n \log n)$	140,000	seconds
Quadratic	$O(n^2)$	$10,000^2$	minutes
Polynomial	$O(n^k)$	$10,000^k$	hours
Exponential	$O(c^n)$	$2^{10,000}$	intractable
Factorial	$O(n!)$	$10,000!$	intractable

O tempo real depende da velocidade do processador

Não muda o fato de que existem categorias de problemas intratáveis.

### Measures of Efficiency for $n = 10,000$

- Observe que o tamanho da amostra do problema ilustrado é 10.000. Além disso, observe que neste caso existe um incremento muito grande entre o número de iterações de uma categoria para a próxima de menor eficiência.

# Eficiência de Algoritmos

## Medidas Padrão de Eficiência

- Na tabela anterior, se dá uma estimativa do número de iterações, de acordo com a ordem de magnitude, para  $n = 10.000$ .

Eficiência	Big-O	Iterações	Tempo Estimado
Logaritmica	$\log n$	13,2877	13,28 Microsegundos
Linear	$n$	10.000	10 Milisegundos
Linear Logaritmica	$n \log n$	132.877	0,132 Segundos
Quadrática	$n^2$	$10.000^2$	1,6 Minutos
Polinomial	$n^3$	$10.000^3$	277,77 Horas
Exponencial	$2^n$	$2^{10.000}$	Inratável
Factorial	$n!$	$10.000!$	Inratável

- Para calcular o tempo estimado, estima-se o tempo para 1 operação. Neste caso, considera-se que somente 1 iteração é realizada por iteração.
- Por exemplo:

$$1 \text{ operação} = 10^{-6} \text{ segundos} = 1 \text{ microsegundo}$$

# Complexidade

## Tempo Computacional segundo Complexidade

**Tabela 1.3** Comparação de várias funções de complexidade

Função de custo	Tamanho $n$					
	10	20	30	40	50	60
$n$	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
$n^2$	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0035 s	0,0036 s
$n^3$	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0,316 s
$n^5$	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
$2^n$	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
$3^n$	0,059 s	58 min	6,5 anos	3855 séc.	$10^8$ séc.	$10^{13}$ séc.

Considera-se que operação custa 1 microsegundo.



# Eficiência de Algoritmos

## Medidas Padrão de Eficiência

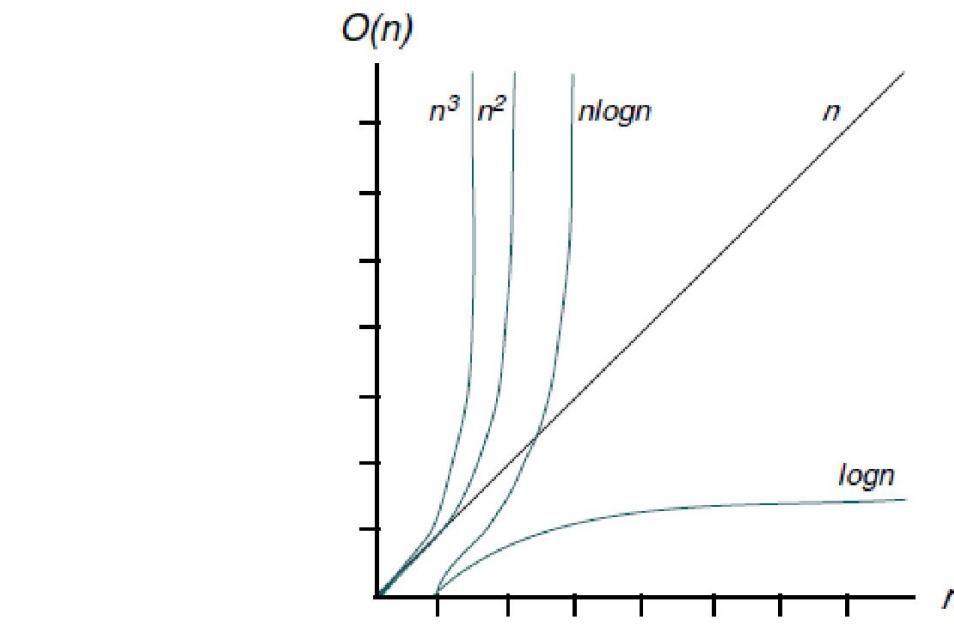
---

- Estas categorias ou medidas de eficiência ressaltam a diferença teórica entre algoritmos de cada categoria a medida que o tamanho dos dados processados aumenta.
- Enquanto lidamos com uma quantidade de elementos **pequena**, embora exista uma diferença entre o número de iterações realizada por algoritmos de categorias de eficiência diferente, essa diferença pode não ser perceptível, principalmente se tratando de frações de segundo.
- Para quantidade de dados  $n$ , suficientemente **grande**, a diferença entre o número de iterações realizadas por algoritmos de categorias diferente não somente se torna perceptível como pode ser **imensa**.

# Eficiência de Algoritmos

## Medidas Padrão de Eficiência

- Podemos pensar que cada categoria de eficiência dos algoritmos representa a ordem de crescimento do número de iterações realizadas por um algoritmo a medida que o número de dados  $n$  cresce.



Plot of Efficiency Measures

# Eficiência de Algoritmos

## Exemplos da Análise Big-O

---

- Para demonstrar os conceitos discutidos, examinaremos dois algoritmos conhecidos:
  - i) Soma de matrizes e
  - ii) Multiplicação de matrizes.

# Analise Big-O

## Exemplo 1 – Soma de matrizes

- Para somar duas matrizes quadradas, precisamos somar os seus elementos correspondentes.
- Soma-se o primeiro elemento da primeira matriz ao primeiro elemento da segunda matriz, o segundo elemento da primeira matriz ao segundo elemento da segunda matriz e assim por diante até chegar ao último elemento.

---

$$\begin{array}{|c|c|c|} \hline 4 & 2 & 1 \\ \hline 0 & -3 & 4 \\ \hline 5 & 6 & 2 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline 6 & 1 & 7 \\ \hline 3 & 2 & -1 \\ \hline 4 & 6 & 2 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 10 & 3 & 8 \\ \hline 3 & -1 & 3 \\ \hline 9 & 12 & 4 \\ \hline \end{array}$$

---

Add Matrices

---

- Como as matrizes tem duas dimensões: linhas e colunas. Precisamos utilizar as duas dimensões para percorrer todos os elementos de cada matriz.
- Em uma matriz quadrada, as duas dimensões são de igual tamanho, denotada como  $n$ . Assim, percorremos um total de  $n^2$  elementos para cada matriz.
- Resolvemos o problema tipicamente mediante o uso de dois loops aninhados.

# Analise Big-O

## Exemplo 1 – Soma de matrizes

- Resolvemos o problema tipicamente mediante o uso de dois loops aninhados.

### Add Two Matrices

```
Algorithm addMatrix (matrix1, matrix2, size, matrix3)
Add matrix1 to matrix2 and place results in matrix3
Pre matrix1 and matrix2 have data
size is number of columns or rows in matrix
Post matrices added--result in matrix3
1 loop (not end of row)
  1 loop (not end of column)
    1 add matrix1 and matrix2 cells
    2 store sum in matrix3
  2 end loop
2 end loop
end addMatrix
```

Para percorrer as linhas

Para percorrer as colunas

Somar células correspondentes das matrizes 1 e 2

Armazenar na matriz 3

- Observe que o número de iterações de ambos os loops é  $n^2$ .
- Já o número de operações realizadas é um múltiplo de  $n^2$ . Por exemplo, se considerarmos apenas, uma soma e uma atribuição por iteração, teríamos  $2n^2$ .
- O importante é que para a análise de eficiência do algoritmo trata-se de um algoritmo quadrático  $O(n^2)$ .

# Analise Big-O

## Exemplo2 – Multiplicação de matrizes

- Para multiplicar duas matrizes quadradas, devemos calcular o valor de cada elemento  $c[i, j]$  na matriz produto.
- Para isso, devemos multiplicar cada elemento da linha  $i$  da primeira matriz, pelo elemento correspondente na coluna  $j$  da segunda matriz. O valor resultante é a soma dos produtos.
- Generalizando o conceito, temos:

```
matrix3 [row, col] =  
    matrix1[row, 0] x matrix2[0, col]  
    +      matrix1[row, 1] x matrix2[1, col]  
    +      matrix1[row, 2] x matrix2[2, col]  
    ...  
    +      matrix1[row, s-1] x matrix2[s-1, col]  
where s = size of matrix
```

# Analise Big-O

## Exemplo 2 – Multiplicação de matrizes

- A figura ilustra um exemplo de como duas matrizes são multiplicadas.

4	2	1
0	-3	4
5	6	2

6	1	7
3	2	-1
4	6	2

$$(a) 4 \times 6 + 2 \times 3 + 1 \times 4 = 34$$

4	2	1
0	-3	4
5	6	2

6	1	7
3	2	-1
4	6	2

$$(b) 4 \times 1 + 2 \times 2 + 1 \times 6 = 14$$

4	2	1
0	-3	4
5	6	2

6	1	7
3	2	-1
4	6	2

$$(c) 0 \times 7 + (-3) \times (-1) + 4 \times 2 = 11$$

# Analise Big-O

## Exemplo2 – Multiplicação de matrizes

---

- Para resolver este problema, devemos realizar pelo menos  $n^2$  iterações para preencher todos os valores da matriz produto. No entanto, existe o trabalho extra de calcular o produto de uma linha por uma coluna específicas. Como a matriz é quadrada este trabalho extra requer  $n$  iterações para multiplicar cada elemento de uma linha pelo seu correspondente na coluna.
- Dessa forma o algoritmo de multiplicação realizaria  $n^3$  iterações.
- Resolvemos o problema tipicamente mediante o uso de três loops aninhados.

# Analise Big-O

## Exemplo2 – Multiplicação de matrizes

- Resolvemos o problema tipicamente mediante o uso de três loops aninhados.

### Multiply Two Matrices

```
Algorithm multiMatrix (matrix1, matrix2, size, matrix3)
Multiply matrix1 by matrix2 and place product in matrix3
Pre matrix1 and matrix2 have data
size is number of columns and rows in matrix
Post matrices multiplied--result in matrix3
1 loop (not end of row)
  1 loop (not end of column)
    1 loop (size of row times)
      1 calculate sum of
        (all row cells) * (all column cells)
      2 store sum in matrix3
    2 end loop
  2 end loop
3 return
end multiMatrix
```

Para percorrer as linhas

Para percorrer as colunas

Para escolher uma posição na linha e coluna

Multiplica um elemento da linha por seu correspondente na coluna

Armazenar na matriz 3

- Observe que o número de iterações de ambos os loops é  $n^3$ . Já o número de operações realizadas é um múltiplo de  $n^3$ . Poderíamos calcular uma aproximação. Mas o importante é que para a análise de eficiência do algoritmo trata-se de um algoritmo cúbico  $O(n^3)$ .

# Referências

---

- Gilberg, R.F. e Forouzan, B.A. Data Structures\_A Pseudocode Approach with C. Capítulo I. Basic Concepts. Segunda Edição. Editora Cengage, Thomson Learning, 2005.