

Aula 1 - Notação Assintótica



Disciplina: Algoritmos e Estruturas de Dados II

Álvaro Luiz Fazenda

alvaro.fazenda@unifesp.br

Bibliografia básica

- Projeto de algoritmos – com implementações em PASCAL e C
 - Nívio Ziviani – Thomson – 2004
 - Capítulo 1 (seção 1.3 e 1.4)

- Algoritmos – Teoria e Prática
 - Thomas H. Cormen et al – Elsevier – 2002
 - Capítulo 3

Porque estudar algoritmos?

- Computadores reais apresentam restrições em velocidade de processamento e de espaço de armazenamento, que variam conforme o custo financeiro do sistema utilizado
 - **Tempo de computação e espaço de armazenagem são recursos limitados que devem ser usados de forma otimizada!**
- O desempenho total do sistema depende tanto da escolha de algoritmos eficientes quanto do hardware

Porque estudar algoritmos? (II)

- O projeto de algoritmos é fortemente influenciado pelo estudo de seus comportamentos ou eficiência
 - Existem várias opções de algoritmos a serem utilizados, variando os aspectos de tempo de execução e espaço de armazenamento ocupado
 - Estudo comum nas áreas de: pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras

Análise de algoritmos

(Cormen et al., 2002)

- **Analisar** um Algoritmo significa prever os recursos de que o algoritmo necessitará
 - Preocupação principal: tempo de execução
 - Analisando-se vários algoritmos candidatos para resolver um problema pode-se identificar os mais eficientes
 - E descartar os de qualidade inferior
- Pode-se considerar o uso de um computador genérico:
 - *RAM (Random-access machine)*
 - Execuções de instruções sequenciais (sem concorrência)
 - Instruções comuns em computadores reais
 - Sem hierarquia de acesso a memória

Análise de um algoritmo particular

- Qual é o custo de se usar um dado algoritmo para resolver um problema específico, independentemente do hardware?
 - Para uma dada instância do programa, o custo depende, entre outros, dos seguintes fatores:
 - Quantidade de dados de entrada;
 - Forma dos dados de entrada;
 - Espécie de dispositivos de armazenamento utilizados.
 - Características que devem ser investigadas:
 - Número de vezes que cada parte do algoritmo deve ser executada para uma dada instância;
 - Quantidade de memória necessária.

Análise de uma classe de algoritmos

- Toda uma família de algoritmos pode ser investigada
- Procura-se identificar um que seja o melhor possível para algoritmos da mesma classe
- Coloca-se limites para a complexidade computacional dos algoritmos pertencentes à classe

Custo computacional de um Algoritmo

- Determinando o menor custo possível para resolver problemas de uma dada classe, temos a medida da dificuldade inerente para resolver o problema
- Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é ótimo para a medida de custo considerada
- Podem existir vários algoritmos para resolver o mesmo problema
- Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado

Medindo tempo do programa executável

- Problemas:
 - Resultados são dependentes do compilador e de opções de compilação;
 - Resultados são dependentes do hardware;
 - Acesso a memória pode causar lentidão quando se testam grandes quantidades de dados.
- Vantagens:
 - Serão considerados tanto os custos reais das operações como os custos não aparentes, tais como alocação de memória, indexação, carga, dentre outros.

Tempo executável: Exemplo

```
#include<time.h>

int main() {

    clock_t start = clock();
    long int t, n, a;

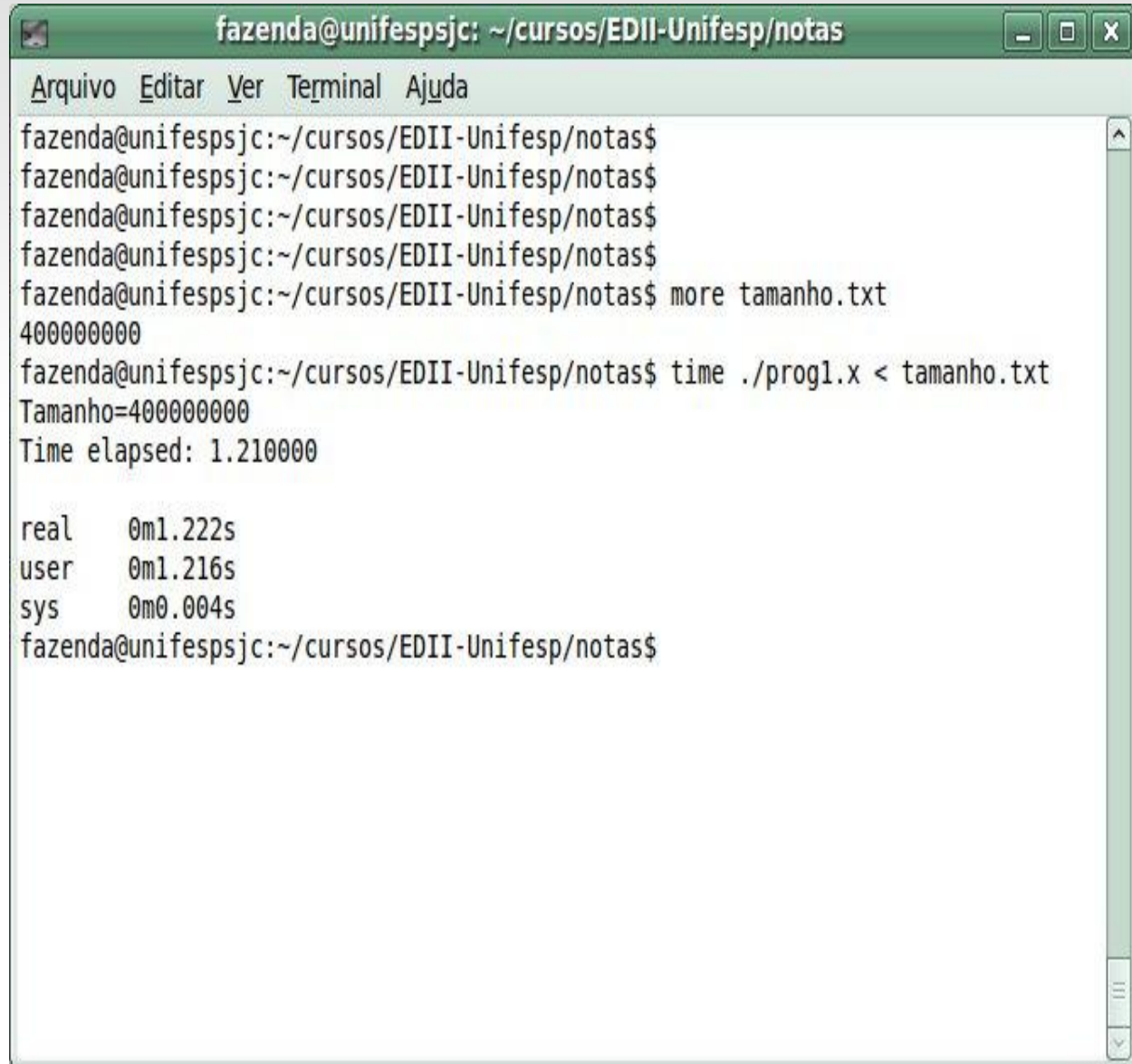
    scanf("%ld", &n);

    for(t=0; t<n; t++)

        a++;

    printf("Time elapsed: %f\n",
        ((double)clock() - start)/
        CLOCKS_PER_SEC);

    return;
}
```



A terminal window titled 'fazenda@unifespjic: ~/cursos/EDII-Unifesp/notas' showing the execution of the program. The user enters 'more tamanho.txt' to view the input file, which contains '400000000'. Then, the user runs 'time ./progl.x < tamanho.txt'. The output shows 'Tamanho=400000000' and 'Time elapsed: 1.210000'. Below this, the system reports the execution time breakdown: 'real 0m1.222s', 'user 0m1.216s', and 'sys 0m0.004s'.

```
fazenda@unifespjic: ~/cursos/EDII-Unifesp/notas
Arquivo Editar Ver Terminal Ajuda
fazenda@unifespjic:~/cursos/EDII-Unifesp/notas$
fazenda@unifespjic:~/cursos/EDII-Unifesp/notas$
fazenda@unifespjic:~/cursos/EDII-Unifesp/notas$
fazenda@unifespjic:~/cursos/EDII-Unifesp/notas$
fazenda@unifespjic:~/cursos/EDII-Unifesp/notas$ more tamanho.txt
400000000
fazenda@unifespjic:~/cursos/EDII-Unifesp/notas$ time ./progl.x < tamanho.txt
Tamanho=400000000
Time elapsed: 1.210000

real    0m1.222s
user    0m1.216s
sys     0m0.004s
fazenda@unifespjic:~/cursos/EDII-Unifesp/notas$
```

Modelo matemático de custo computacional

- Usa-se um modelo matemático baseado em um computador idealizado
- Deve ser especificado o conjunto de operações e seus custos de execuções
- Ignora-se o custo de algumas das operações e considera-se apenas as operações mais significativas
 - Ex.: Algoritmos de ordenação: Consideramos o número de comparações entre os elementos do conjunto a ser ordenado e ignoramos as operações aritméticas, de atribuição e manipulações de índices, etc.

Análise inicial do custo de um programa

```
#include<time.h>
```

```
int main() {
```

```
    clock_t start = clock();
```

```
    long int t, n, a;
```

```
    scanf("%ld", &n);
```

```
    for(t=0; t<n; t++)
```

```
        a++;
```

```
    printf("Time elapsed: %f\n",  
          ((double)clock() - start)/  
          CLOCKS_PER_SEC);
```

```
    return;
```

```
}
```

C1

C2

C4

$n \cdot C5$

$n \cdot C6$

$n \cdot C7$

C8

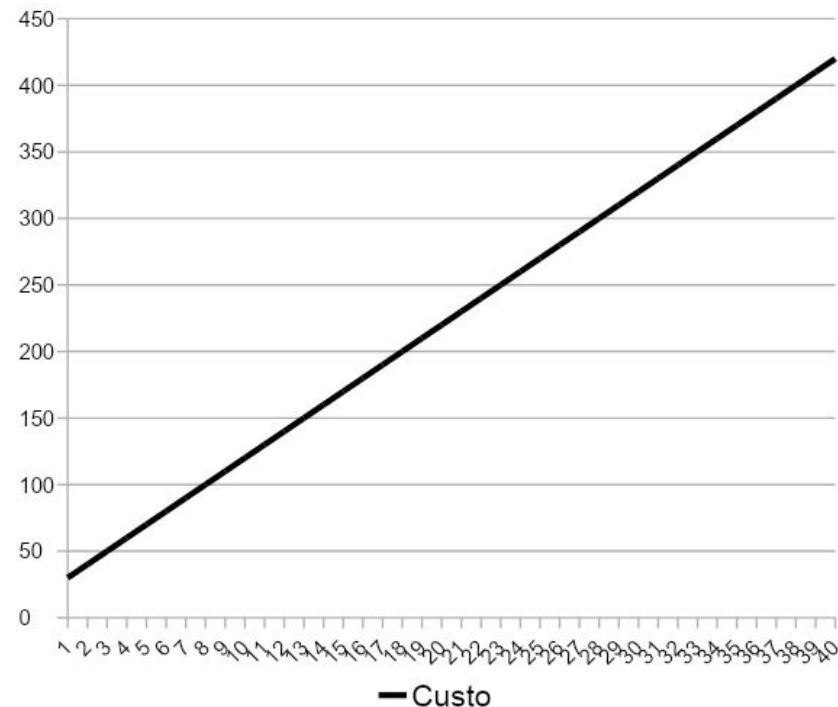
Custo total do programa

- Tempo total depende do tempo de cada instrução C_x , assim:
 - $C_{tot} = C1 + C2 + C3 + n * C4 + n * C5 + n * C6 + C7$
 - $C_{tot} = C1 + C2 + C3 + C7 + n(C4 + C5 + C6)$
 - $C8 = C1 + C2 + C3 + C7$
 - $C9 = C4 + C5 + C6$
 - **$C_{tot} = C8 + nC9$**

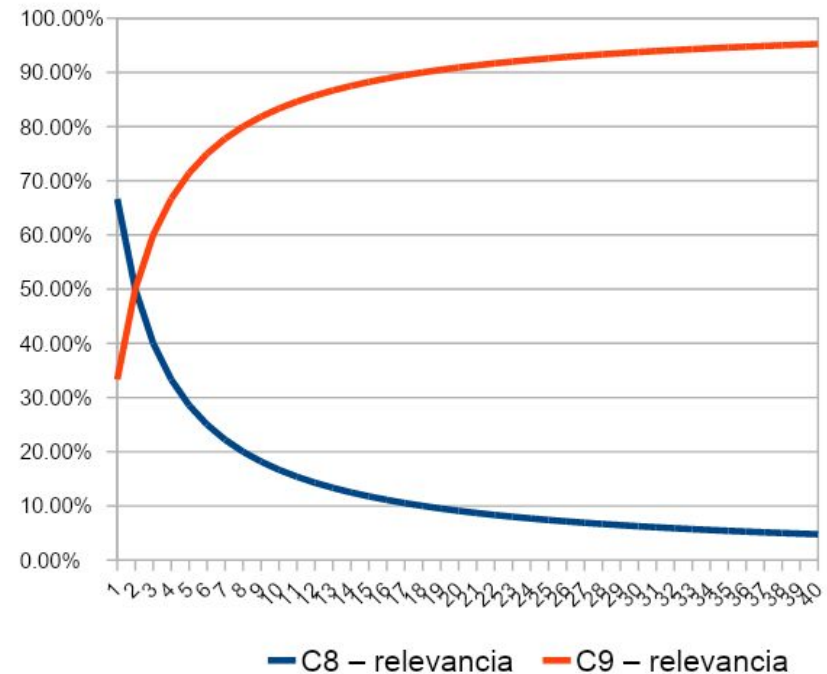
Custo total do programa (II)

■ Considerando:

- $C8 = 20 \text{ s}$
- $C9 = 10 \text{ s}$
- $N = 1 \dots 40$



- Vê-se que o custo de $f(n) = n * C$ domina o problema para grandes valores de n



Custo total do programa (III)

- Para grandes valores de n pode-se desconsiderar C8 na análise de custo do programa exemplo
- Pode-se considerar, também, uma função $f(n)$ que deverá indicar a quantidade de instruções a ser executada no laço, dentro do programa exemplo
 - A Função $f(n)$ deverá indicar a complexidade do algoritmo estudado

Função de Complexidade $f(n)$

- $f(n)$ é denominada **Função de Complexidade de Tempo**, e representa a medida de custo necessária para executar um algoritmo para um problema de tamanho n
 - A complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada
- Pode-se usar também uma função para se medir o custo de memória (Função de complexidade de espaço)

Exemplo: Maior Elemento

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros $A[n]$, $n \geq 1$

```
int Max(int* A, int n) {  
    int i, Temp;  
  
    Temp = A[0];  
    for (i = 1; i < n; i++)  
        if (Temp < A[i])  
            Temp = A[i];  
    return Temp;  
}
```

- Seja f uma função de complexidade tal que $f(n)$ é o número de comparações envolvendo os elementos de A , se A contiver n elementos. **Qual é a função $f(n)$?**

Exemplo: Maior Elemento

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros $A[n]$, $n \geq 1$

```
int Max(int* A, int n) {  
    int i, Temp;  
  
    Temp = A[0];  
    for (i = 1; i < n; i++)  
        if (Temp < A[i]) Temp = A[i];  
    return Temp;  
}
```

($n-1$)

- Seja f uma função de complexidade tal que $f(n)$ é o número de comparações envolvendo os elementos de A , se A contiver n elementos. Qual é a função $f(n)$?

Exemplo: Maior Elemento

Teorema

Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, $n \geq 1$, faz pelo menos $n - 1$ comparações.

Prova

Cada um dos $n - 1$ elementos tem de ser investigado por meio de comparações, que é menor do que algum outro elemento. Logo, $n - 1$ comparações são necessárias.

Tamanho da entrada de dados

- Conforme já visto, a medida do custo de execução de um algoritmo **depende principalmente do tamanho da entrada dos dados**
- É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada

Tamanho da entrada de dados

(II)

- No caso da função *Max* do programa do exemplo, o **custo é uniforme** sobre todos os problemas de tamanho n
- Já para um algoritmo de ordenação isso não ocorre: se os dados de entrada já estiverem quase ordenados, então pode ser que o algoritmo trabalhe menos

Casos a Serem Analisados

- **Melhor caso:** menor tempo de execução sobre todas as entradas de tamanho n
- **Pior caso:** maior tempo de execução sobre todas as entradas de tamanho n
- **Caso médio (ou caso esperado):** média dos tempos de execução de todas as entradas de tamanho n
- $Melhor\ Caso \leq Caso\ Médio \leq Pior\ Caso$

Maior e Menor Elemento

Seja $f(n)$ o número de comparações entre os elementos de A . Logo, $f(n) = 2(n - 1)$ para o melhor caso, pior caso e caso médio.

```
void MaxMin1(int* A, int n, int* pMax, int* pMin)
{
    int i;

    *pMax = A[0];
    *pMin = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i],
        if (A[i] < *pMin) *pMin = A[i];
    }
}
```

$2*(n-1)$

Maior e Menor Elemento (II)

MinMax1 pode ser facilmente melhorado: a comparação $A[i] < *pMin$ só é necessária quando a comparação $A[i] > *pMax$ dá falso.

```
void MaxMin2(int* A, int n, int* pMax, int* pMin)
{
    int i;

    *pMax = A[0];
    *pMin = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        else if (A[i] < *pMin) *pMin = A[i];
    }
}
```


Maior e Menor Elemento (III)

- Melhor caso:
 - quando os elementos estão em ordem crescente
 - $f(n) = n - 1$
- Pior caso:
 - quando o maior elemento é o primeiro no vetor
 - $f(n) = 2(n - 1)$
- Caso médio:
 - $A[i]$ é maior do que Max a metade das vezes
 - $f(n) = 3n/2 - 3/2$

Maior e Menor Elemento (IV)

- Considerando o número de comparações realizadas, é possível obter um algoritmo mais eficiente:
 - Compare os elementos de A aos pares, separando-os em dois subconjuntos (maiores em um e menores em outro), a um custo de $n/2$ comparações.
 - O máximo é obtido do subconjunto que contém os maiores elementos, a um custo de $n/2 - 1$ comparações.
 - O mínimo é obtido do subconjunto que contém os menores elementos, a um custo de $n/2 - 1$ comparações.

Maior e Menor Elemento (V)

```
void MaxMin3(int* A, int n, int* pMax, int* pMin)
{
    int i;

    if ((n % 2) > 0)        { *pMax = A[n-1]; *pMin = A[n-1]; }
    else
    {
        if (A[n-2] > A[n-1]) { *pMax = A[n-2]; *pMin = A[n-1]; } → Comparação 1
        else                { *pMax = A[n-1]; *pMin = A[n-2]; }
    }

    for (i = 1; i < n-1; i += 2)
    {
        if (A[i - 1] > A[i]) → Comparação 2
        {
            if (A[i - 1] > *pMax) *pMax = A[i - 1]; → Comparação 3
            if (A[i] < *pMin) *pMin = A[i]; → Comparação 4
        }
        else
        {
            if (A[i - 1] < *pMin) *pMin = A[i - 1]; → Comparação 3
            if (A[i] > *pMax) *pMax = A[i]; → Comparação 4
        }
    }
}
```

Maior e Menor Elemento (VI)

- Quantas comparações são feitas em MaxMin3?
 - 1ª comparação feita 1 vez
 - 2ª comparação feita $n/2 - 1$ vezes
 - 3ª e 4ª comparações feitas $n/2 - 1$ vezes
 - $f(n) = 1 + n/2 - 1 + 2 * (n/2 - 1)$
 - $f(n) = (3n - 6)/2 + 1$
 - $f(n) = 3n/2 - 3 + 1 = 3n/2 - 2$
 - No pior caso, melhor caso e caso médio

Comparação dos algoritmos

- Os algoritmos *MaxMin2* e *MaxMin3* são superiores ao algoritmo *MaxMin1* de forma geral.
- O algoritmo *MaxMin3* é superior ao algoritmo *MaxMin2* com relação ao pior caso e bastante próximo quanto ao caso médio

Os três algoritmos	$f(n)$		
	Melhor caso	Pior caso	Caso médio
MaxMin1	$2(n - 1)$	$2(n - 1)$	$2(n - 1)$
MaxMin2	$n - 1$	$2(n - 1)$	$3n/2 - 3/2$
MaxMin3	$3n/2 - 2$	$3n/2 - 2$	$3n/2 - 2$

Comportamento assintótico

- O parâmetro n fornece uma medida da dificuldade para se resolver o problema, que tem complexidade $f(n)$
- Para valores suficientemente pequenos de n , qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes
 - A escolha do algoritmo não é um problema crítico para problemas de tamanho pequeno
- Logo, a análise de algoritmos é realizada para valores grandes de n
- Estuda-se o comportamento assintótico das funções de custo (ou funções de complexidade de tempo), ou seja, o comportamento destas funções, ou ordem de crescimento, para valores grandes de n

Exemplo

- Diferentes formas de se calcular: $1 + 2 + \dots + n$

Algorithm A	Algorithm B	Algorithm C
<pre>sum = 0 for i = 1 to n sum = sum + i</pre>	<pre>sum = 0 for i = 1 to n { for j = 1 to i sum = sum + 1 }</pre>	<pre>sum = n * (n + 1) / 2</pre>

Exemplo – Algoritmo 1

```
for i = 1 to n  
  sum = sum + i
```



1



2



3

...

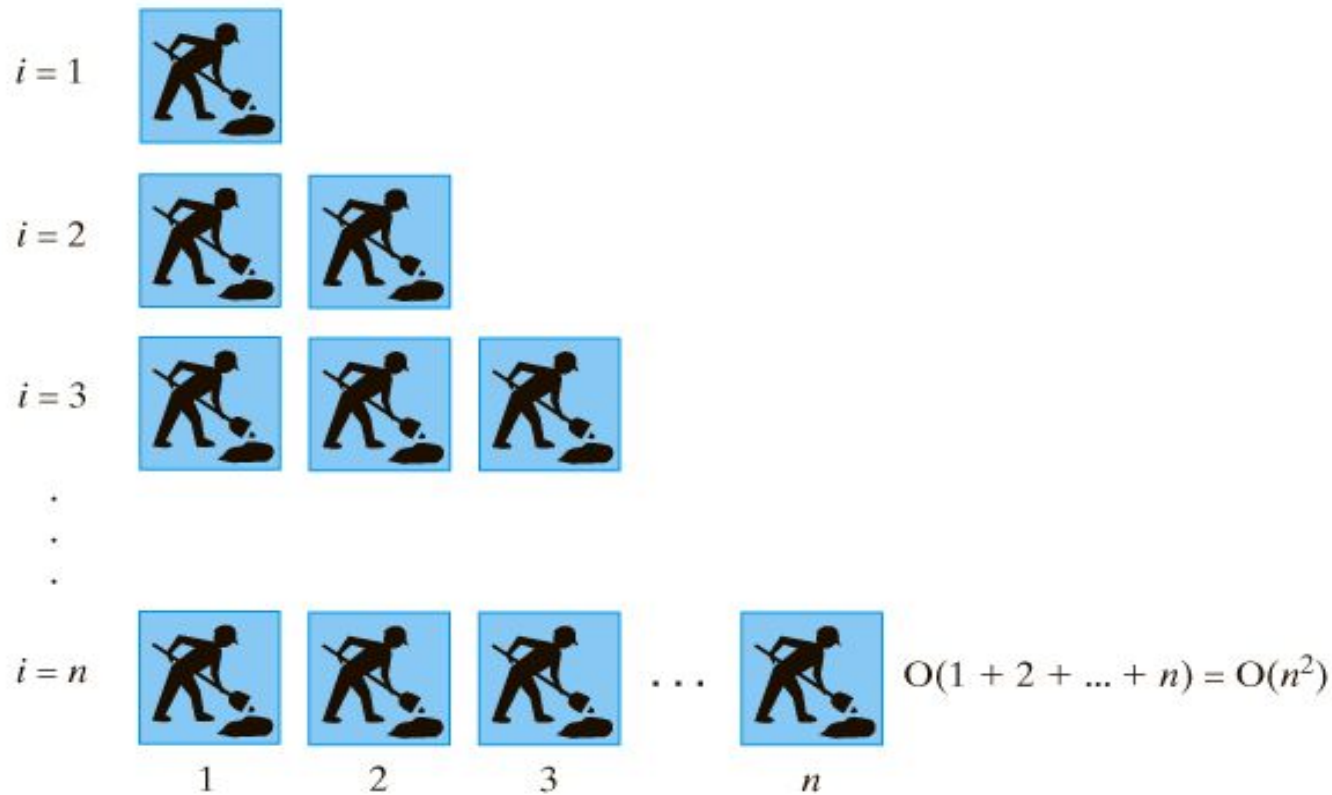


n

$O(n)$

Exemplo – Algoritmo 2

```
for i = 1 to n  
{ for j = 1 to i  
  sum = sum + 1  
}
```



Número de operações

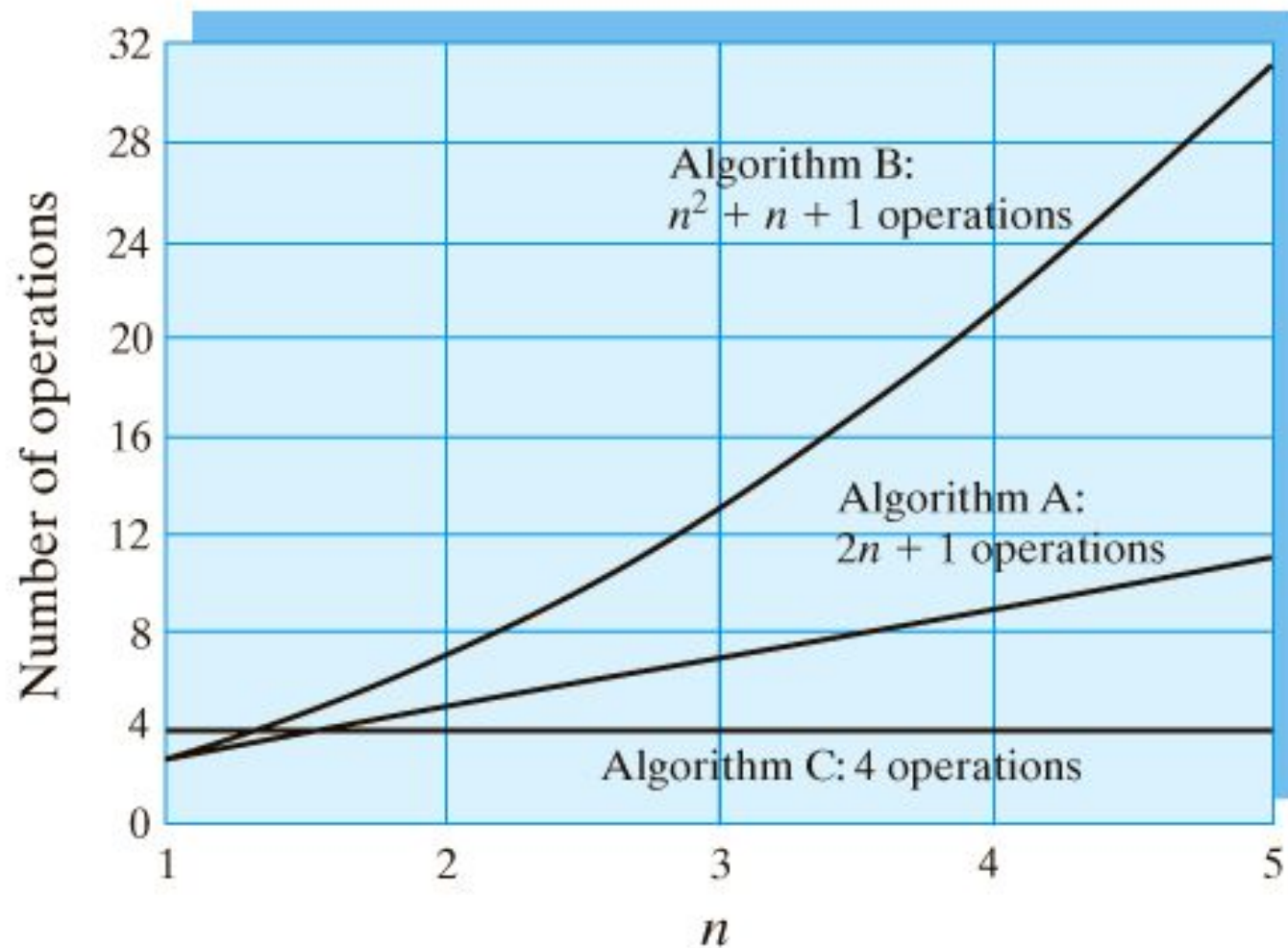
	Algorithm A	Algorithm B	Algorithm C
Assignments	$n + 1$	$1 + n(n + 1) / 2$	1
Additions	n	$n(n + 1) / 2$	1
Multiplications			1
Divisions			1
Total operations	$2n + 1$	$n^2 + n + 1$	4

$O(n)$

$O(n^2)$

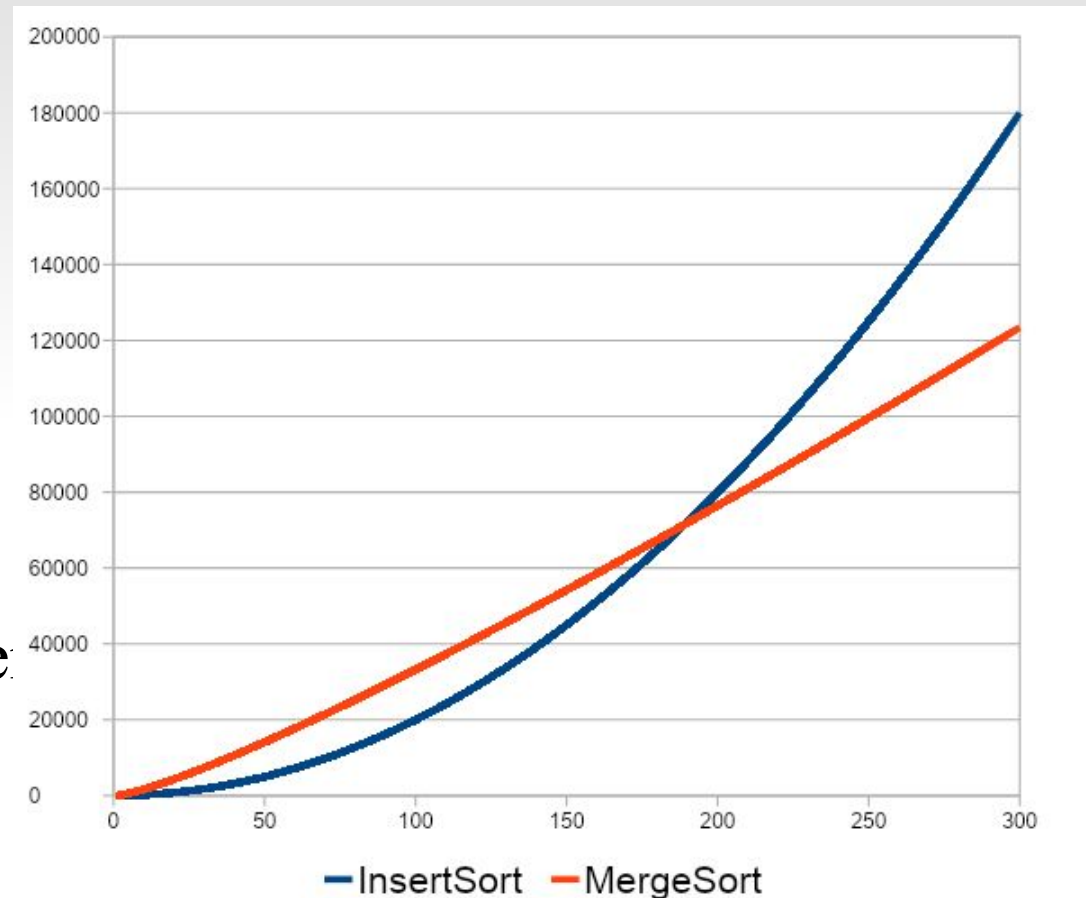
$O(1)$

Gráfico do Número de operações



Exemplo prático - eficiência:

- Ordenação por Inserção
(*InsertSort*):
 $f(n) = C1 n^2$
- Ordenação por Intercalação
(*MergeSort*):
 $f(n) = C2 n \lg(n)$
 - Onde $C1$ e $C2$ não depende de n (constantes)
 - $C1 < C2$
($C1=2$ e $C2=50$)



Ex. prático – eficiência (II)

- Computador A:
 - 1 *Tops* (*Tera Operations per seconds* – 1 trilhão de oper. por seg.)
 - *InsertSort*
- Computador B:
 - 1 *Gops* (*Giga operations per seconds* – 1 Bilhão ...)
 - *MergeSort*



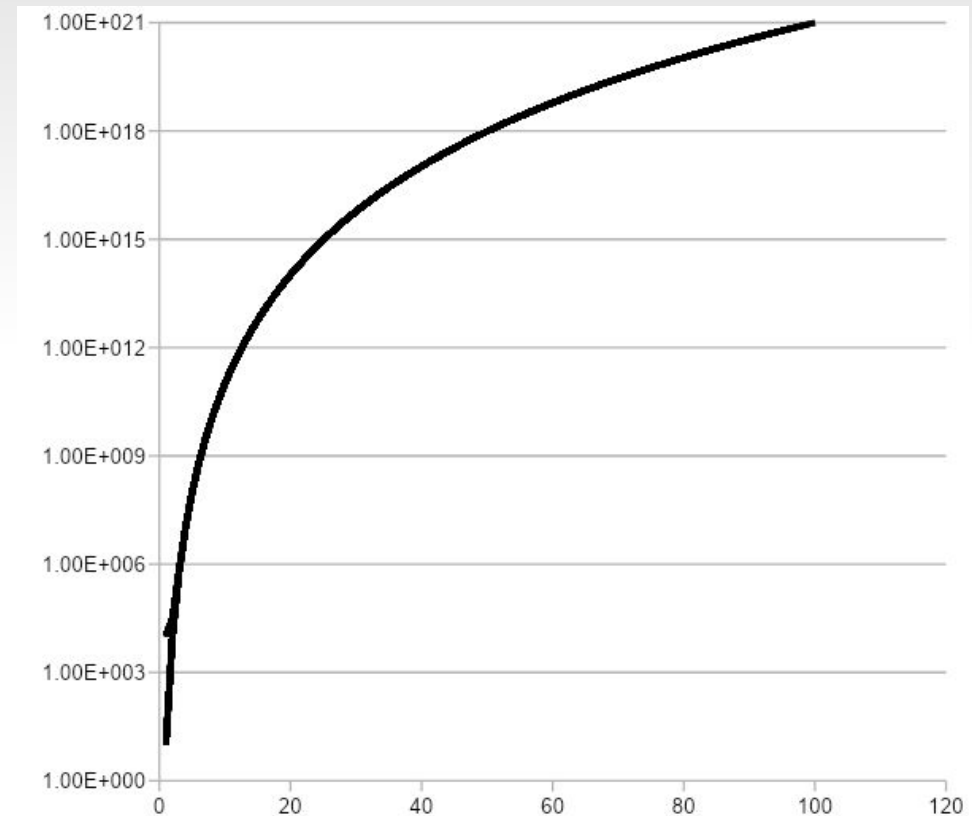
Comportamento assintótico de funções

O que acontece quando n aumenta?

$$P_1(n) = 10n^{10} + 100n^2 + 10000n + 1/n$$

$$P_2(n) = 10n^{10}$$

Somente o termo de mais alta ordem (n^{10}) é relevante para grandes valores de n



Comportamento assint. de funções (II)

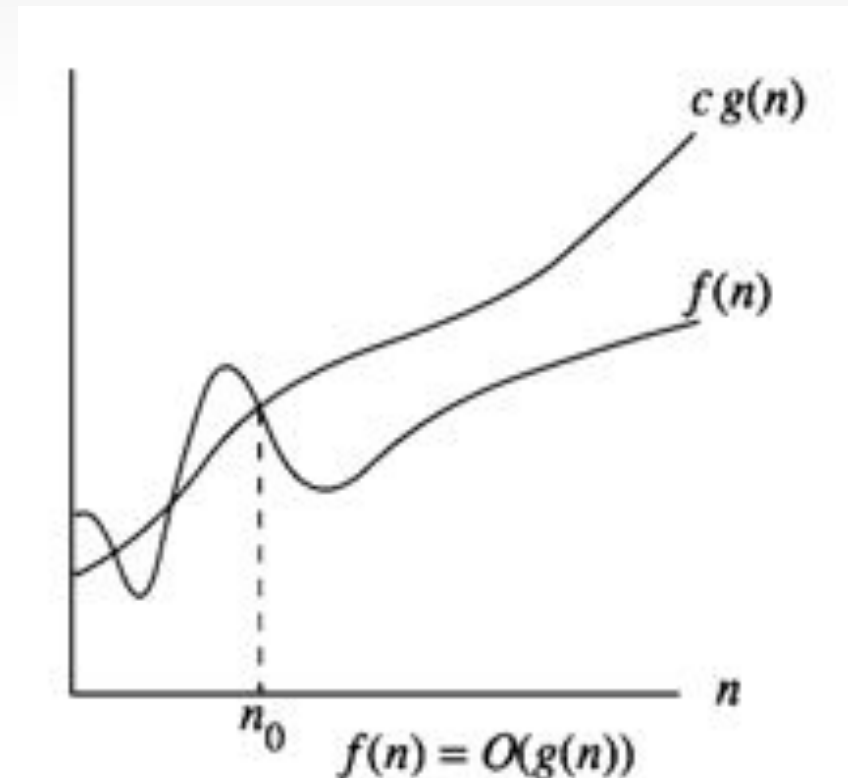
- Eficiência Assintótica: relevante apenas a ordem de crescimento do tempo de execução do algoritmo
 - Preocupa-se, apenas, com a maneira pela qual o tempo de execução aumenta com a variação no tamanho dos dados de entrada
 - Em geral, um algoritmo assintoticamente mais eficiente é a melhor opção de escolha para a maioria das variações de entrada, exceto as muito pequenas

Notação assintótica para Algoritmos

- Permite classificar classes de algoritmos conforme sua complexidade computacional:
 - $\Theta(f(n))$ - "Teta"
 - $O(f(n))$ - "Ozão"
 - $o(f(n))$ - "Ozinho"
 - $\Omega(f(n))$ - "Omega"
 - $\omega(f(n))$ - "Omegazinho"

Notação O (Limite assintótico superior)

- Dada uma função $g(n)$, denota-se por $O(g(n))$ o conjunto de funções:
 - $O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq cg(n) \text{ para todo } n \geq n_0\}$
- Ou seja, uma função $f(n)$ pertence ao conjunto $O(g(n))$ se existir uma constante positiva c tal que ela possa estar com custo abaixo ou igual a $cg(n)$, para n suficientemente grande



Notação O (II)

- Dominação assintótica:
 - Definição: Uma função $g(n)$ domina assintoticamente outra função $f(n)$ se $f(n) = O(g(n))$
 - Exemplo: quando dizemos que o tempo de execução $T(n)$ de um programa é $O(n^2)$, significa que existem constantes c e n_0 tais que, para valores de $n \geq n_0$, $T(n) \leq cn^2$

Notação O - exemplo

- $f(n) = (n + 1)^2$
- Logo $f(n)$ é $O(n^2)$, quando $n_0 = 1$ e $c = 4$
- Isto porque $(n + 1)^2 \leq 4n^2$ para $n \geq 1$

Notação O - exemplo

- $g(n) = n$ e $f(n) = n^2$
- Sabemos que $g(n)$ é $O(n^2)$, pois para $n \geq 1$, $n \leq n^2$
- Entretanto, $f(n)$ não é $O(n)$
- Suponha que existam constantes c e n_0 tais que para todo $n \geq n_0$, $n^2 \leq cn$
- Se $c \geq n$ para qualquer $n \geq n_0$, então deveria existir um valor para c que possa ser maior ou igual n para todo n

Notação O – exemplo 2

- $f(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$
- Basta mostrar que $3n^3 + 2n^2 + n \leq 6n^3$ para $n \geq 0$
- A função $f(n) = 3n^3 + 2n^2 + n$ é também $O(n^4)$,
entretanto, essa afirmação é mais fraca do que dizer
que $f(n)$ é $O(n^3)$

Operações com a Notação O

$$f(n) = O(f(n))$$

$$c \times O(f(n)) = O(f(n)) \quad c = \text{constante}$$

$$O(f(n)) + O(f(n)) = O(f(n))$$

$$O(O(f(n))) = O(f(n))$$

$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

$$O(f(n))O(g(n)) = O(f(n)g(n))$$

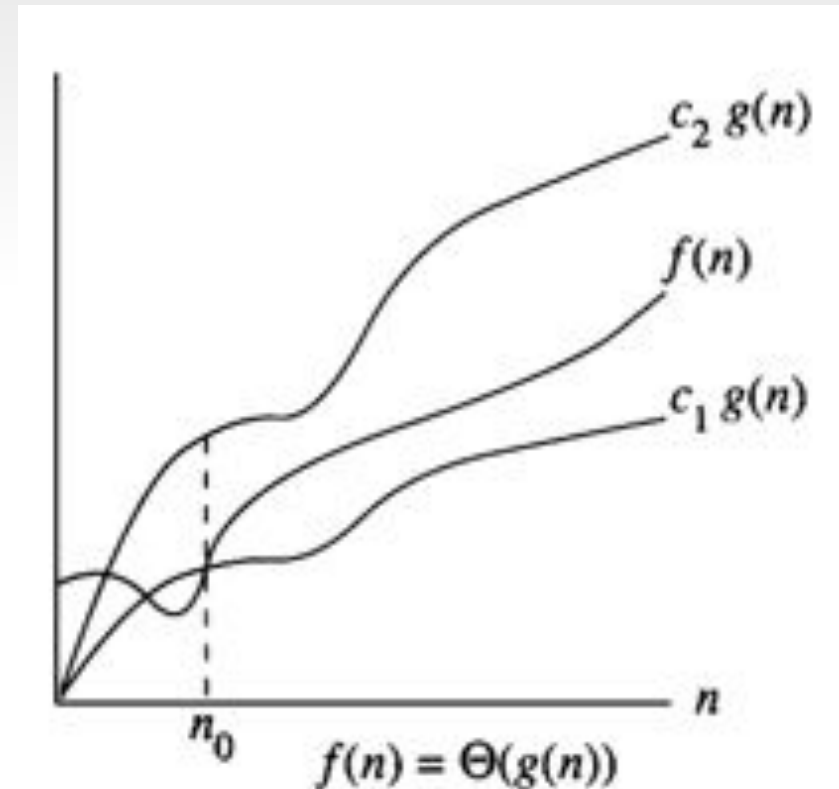
$$f(n)O(g(n)) = O(f(n)g(n))$$

Notação Θ

- Dada uma função $g(n)$, denota-se por $\Theta(g(n))$ o conjunto de funções:
 - $\Theta(g(n)) = \{f(n) : \text{existem constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tais que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0\}$
 - Ou seja, uma função $f(n)$ pertence ao conjunto $\Theta(g(n))$ se existirem constantes positivas c_1 e c_2 tais que ela possa ser imprensada entre $c_1 g(n)$ e $c_2 g(n)$, para n suficientemente grande.

Notação Θ (II)

- Como $\Theta(g(n))$ é um conjunto:
 $f(n) \in \Theta(g(n))$
 - Comumente escreve-se:
” $f(n) = \Theta(g(n))$ ”
- $g(n)$ é um limite assintoticamente restrito para $f(n)$
 - Θ define **limites superior e inferior**



Notação Θ (III)

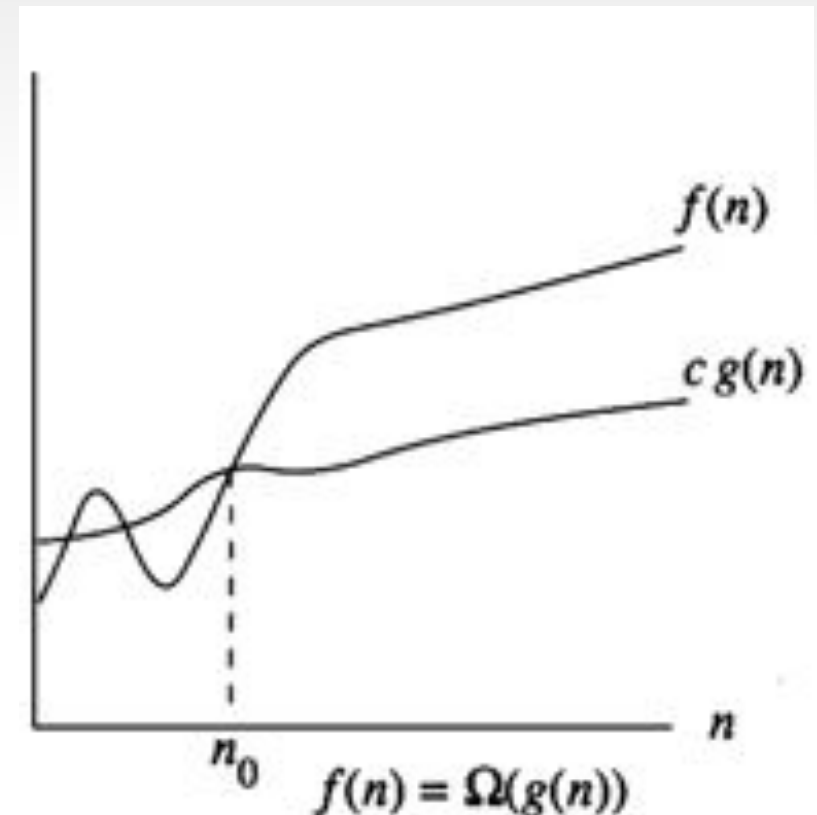
- Exemplo: $f(n) \in \Theta(n^2)$:
 - Considerando: $f(n) = \frac{1}{2}n^2 - 3n$
 - $f(n) = \Theta(n^2)$??
 - $c_1 n^2 \leq (\frac{1}{2}n^2 - 3n) \leq c_2 n^2$ (Para todo $n \geq n_0$)
 - Dividindo a equação por n^2 :
 - $c_1 \leq (1/2 - 3/n) \leq c_2$
 - Desigualdade da direita: $c_2 = 1/2$ para qualquer $n > 0$
 - Desigualdade da esquerda: $c_1 = 1/14$ para $n \geq 7$
 - Assim: $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
 - Existem outras combinações possíveis
 - Importante encontrar **uma**!

Notação Θ (IV)

- Exemplo: $f(n) \notin \Theta(n^2)$:
 - Considerando: $f(n) = 6n^3$, $f(n) \neq \Theta(n^2)$??
 - Supondo que existam c_2 e n_0 tais que:
 $6n^3 \leq c_2 n^2$ para todo $n \geq n_0$
 - Implica: $n \leq c_2/6$
Que não é válido!! pois c_2 é constante e sempre poderá existir um valor de n suficientemente grande
 - Pois n pode crescer até o infinito a partir de n_0 ,

Notação Ω (Limite assintótico inferior)

- Dada uma função $g(n)$, denota-se por $\Omega(g(n))$ o conjunto de funções:
- $\Omega(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq c g(n) \leq f(n) \text{ para todo } n \geq n_0\}$
- Ou seja, uma função $f(n)$ pertence ao conjunto $\Omega(g(n))$ se existir uma constante positiva c tal que ela possa estar com custo superior ou igual a $c g(n)$, para n suficientemente grande



Notação o (limite estritamente superior)

- Dada uma função $g(n)$, denota-se por $o(g(n))$ o conjunto de funções:
 - $o(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) < cg(n) \text{ para todo } n \geq n_0\}$
- Ou seja, uma função $f(n)$ pertence ao conjunto $o(g(n))$ se existir uma constante positiva c tal que ela possa estar com custo abaixo a $cg(n)$, para n suficientemente grande
- A função $f(n)$ se torna insignificante em relação a $g(n)$ à medida que n se aproxima do infinito:

Notação ω (limite estritamente inferior)

- Dada uma função $g(n)$, denota-se por $\omega(g(n))$ o conjunto de funções:
 - $\omega(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq cg(n) < f(n) \text{ para todo } n \geq n_0\}$
 - Ou seja, uma função $f(n)$ pertence ao conjunto $\omega(g(n))$ se existir uma constante positiva c tal que ela possa estar com custo acima de $cg(n)$, para n suficientemente grande
- A função $f(n)$ se torna arbitrariamente grande em relação a $g(n)$ à medida que n se aproxima do infinito:

Transitividade:

Transitivity:

-

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$,

$f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$,

$f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$,

$f(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$,

$f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$.

Reflexividade:

Reflexivity:

-

$$f(n) = \Theta(f(n)),$$

$$f(n) = O(f(n)),$$

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

Simetria:

Symmetry:

$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

Simetria de transposição

Transpose symmetry:

-

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$,
 $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Classes de Comport.

Assintótico

- Se f é uma função de complexidade para um algoritmo F , então $O(f)$ é considerada a complexidade assintótica ou o comportamento assintótico do algoritmo F
- Se as funções f e g dominam assintoticamente uma a outra, os algoritmos associados são equivalentes
 - Nestes casos, o comportamento assintótico não serve para comparar os algoritmos
 - Exemplo: dois algoritmos F e G aplicados à mesma classe de problemas, sendo que F leva três vezes o tempo de G ao serem executados, isto é, $f(n) = 3g(n)$, sendo que $O(f(n)) = O(g(n))$
 - O comportamento assintótico não serve para comparar os algoritmos, pois eles diferem apenas por uma constante

Complexidade Constante

- $f(n) = O(1)$
 - Algoritmos de complexidade $O(1)$ são ditos de **complexidade constante**
 - As instruções do algoritmo são executadas um número fixo de vezes

• Complexidade Logarítmica

- $f(n) = O(\log n)$
 - Típico em algoritmos que transformam um problema em outros menores
 - Quando n é mil, $\log_2 n \approx 10$, quando n é 1 milhão, $\log_2 n \approx 20$
 - Para dobrar o valor de $\log n$ temos de considerar o quadrado de n
 - A base do logaritmo muda pouco estes valores: quando n é 1 milhão, o $\log_2 n$ é 20 e o $\log_{10} n$ é 6

Complexidade Linear

- $f(n) = O(n)$
 - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada
 - É a melhor situação possível para um algoritmo que tem de processar/produzir n elementos de entrada/saída
 - Cada vez que n dobra de tamanho, o custo de execução dobra

Complexidade $n \log n$

- $f(n) = O(n \log n)$
 - Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e ajuntando as soluções depois
 - Quando n é 1 milhão, $n \log_2 n$ é cerca de 20 milhões
 - Quando n é 2 milhões, $n \log_2 n$ é cerca de 42 milhões, pouco mais do que o dobro

Complexidade Quadrática e cúbica

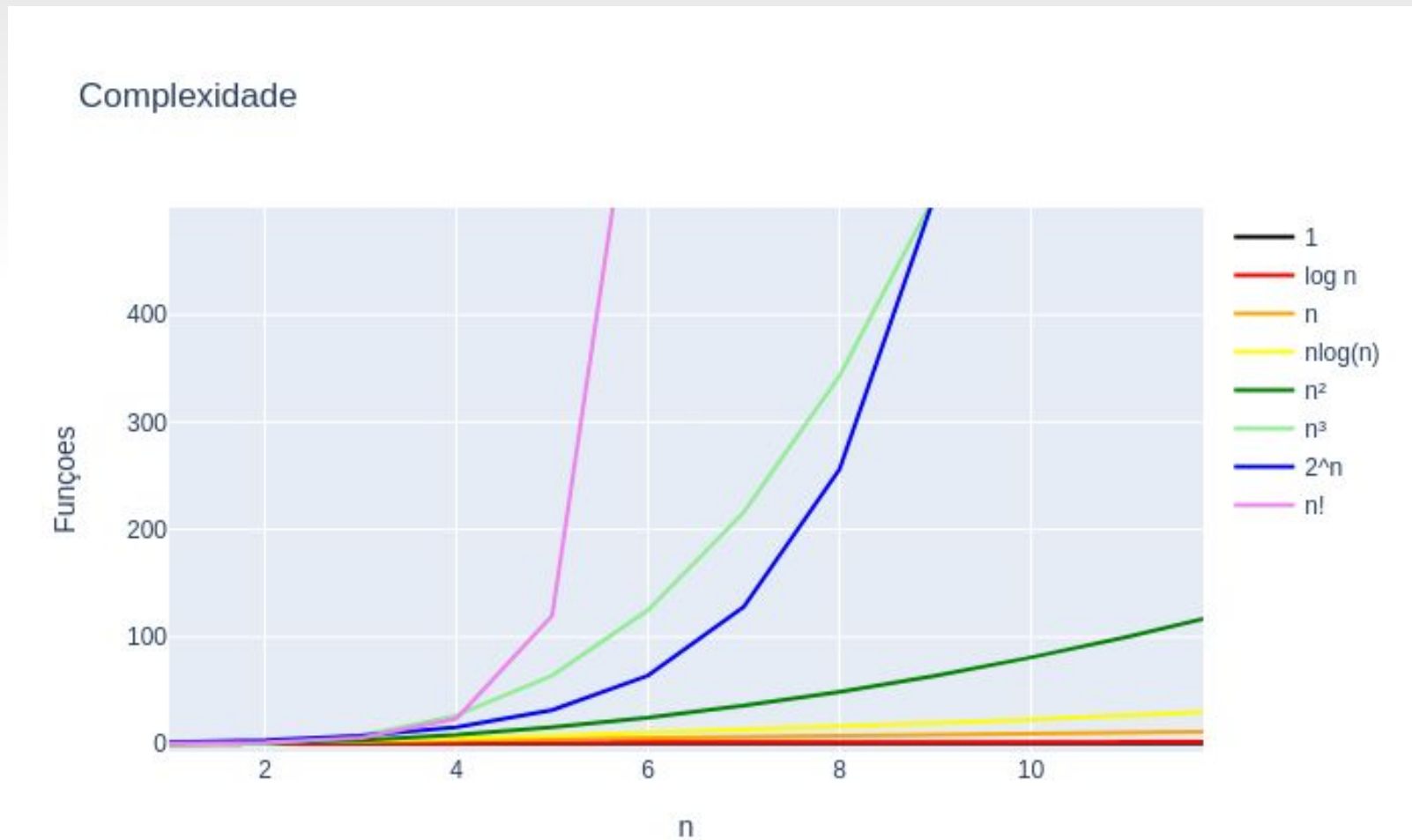
- Comuns em programas para computação científica
- Quadrática: $f(n) = O(n^2)$
 - Ocorrem quando os itens de dados são processados aos pares, muitas vezes em um laço aninhado dentro de outro
 - Quando n é mil, o número de operações é da ordem de 1 milhão
 - Sempre que n dobra, o custo de execução é multiplicado por 4
- Cúbica: $f(n) = O(n^3)$
 - Sempre que n dobra, o tempo de execução fica multiplicado por 8

Complexidade Exponencial

- $f(n) = O(2^n)$
 - Comum na solução de problemas quando se usa força bruta para resolvê-los
 - Quando n é 20, o tempo de execução é cerca de 1 milhão. Quando n dobra, o custo é elevado ao quadrado
- $f(n) = O(n!)$
 - Um algoritmo de complexidade $O(n!)$ é dito ter complexidade exponencial, apesar de $O(n!)$ ter comportamento muito pior do que $O(2^n)$
 - $n = 20 \rightarrow 20! = 2432902008176640000$
 - $n = 40 \rightarrow$ número com 48 dígitos

Hierarquia:

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



Comparação de Funções de Complexidade

- Algoritmo Linear $O(1)$ executa 1 MOPS (milhões de operações por segundo) – Tabela de Garey & Johnson (1979)

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	0,0.35 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.

Comparação de Funções de Complexidade (II)

Função de custo de tempo	Computador atual	Computador 100 vezes mais rápido	Computador 1.000 vezes mais rápido
n	t_1	$100 t_1$	$1000 t_1$
n^2	t_2	$10 t_2$	$31,6 t_2$
n^3	t_3	$4,6 t_3$	$10 t_3$
2^n	t_4	$t_4 + 6,6$	$t_4 + 10$

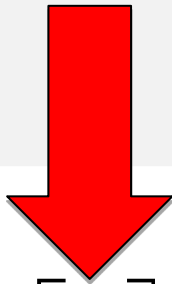
Exercício

```
void exercicio1(int n)
{
    int i, a;
    a=0; i=0;
    while (i<n)
    {
        a+=i;
        i+=2;
    }
}
```

```
void exercicio2(int n)
{
    int i, j, a;
    a=0;
    for (i=0; i<n; i++)
        for (j=0; j<i; j++)
            a+=i+j;
}
```

Exercício

```
void exercicio1(int n)
{
    int i, a;
    a=0; i=0;
    while (i<n)
    {
        a+=i;
        i+=2;
    }
}
```



$$\left\lceil \frac{n}{2} \right\rceil$$

$O(n)$

```
void exercicio2(int n)
{
    int i, j, a;
    a=0;
    for (i=0; i<n; i++)
        for (j=0; j<i; j++)
            a+=i+j;
}
```



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

$O(n^2)$

Obs: Adaptado de Jurandy Almeida Jr