

# Complexidade de Algoritmos

## Análise e notação assintótica

Prof. Marcelo de Souza

45EST – Algoritmos e Estruturas de Dados  
Universidade do Estado de Santa Catarina



## Leitura principal:

- ▶ Capítulo 4 de [Goodrich e Tamassia \(2013\)](#)<sup>1</sup> – Ferramentas de análise.

## Leitura complementar:

- ▶ Capítulos 1 a 3 de [Toscani e Veloso \(2012\)](#)<sup>2</sup>.
- ▶ Capítulos 2 e 3 de [Preiss \(2001\)](#)<sup>3</sup>.

---

<sup>1</sup>Michael T Goodrich e Roberto Tamassia (2013). *Estruturas de Dados & Algoritmos em Java*. 5ª ed. Bookman Editora.

<sup>2</sup>Laira Vieira Toscani e Paulo A. S. Veloso (2012). *Complexidade de Algoritmos*. 3ª ed. Vol. 13. Bookman: Porto Alegre.

<sup>3</sup>Bruno R Preiss (2001). *Estruturas de dados e algoritmos: padrões de projetos orientados a objetos com Java*. Campus.



Um **problema** é caracterizado pela descrição da sua **entrada** e **saída**.



Um **problema** é caracterizado pela descrição da sua **entrada** e **saída**.

Exemplo:

## Problema da ordenação (não decrescente)

**Entrada:** uma sequência  $\langle a_1, a_2, \dots, a_n \rangle$  de  $n$  números.

**Saída:** uma permutação dos números  $\langle a'_1, a'_2, \dots, a'_n \rangle$  tal que  $a'_1 \leq a'_2 \leq \dots, a'_n$ .



Um **problema** é caracterizado pela descrição da sua **entrada** e **saída**.

Exemplo:

## Problema da ordenação (não decrescente)

**Entrada:** uma sequência  $\langle a_1, a_2, \dots, a_n \rangle$  de  $n$  números.

**Saída:** uma permutação dos números  $\langle a'_1, a'_2, \dots, a'_n \rangle$  tal que  $a'_1 \leq a'_2 \leq \dots, a'_n$ .

## Exemplo concreto do problema da ordenação

**Entrada:** 7 8 1 4 6 3 9 → **instância**

**Saída:** 1 3 4 6 7 8 9 → **solução/resultado**



Dado um problema:

- ▶ Como saber se um algoritmo é bom para resolvê-lo?
- ▶ Como saber qual é o melhor algoritmo entre duas opções?



Dado um problema:

- ▶ Como saber se um algoritmo é bom para resolvê-lo?
- ▶ Como saber qual é o melhor algoritmo entre duas opções?

Qual medida usar para definir **bom** ou **melhor**?

- ▶ Correção;
- ▶ Simplicidade;
- ▶ Facilidade em codificar;
- ▶ Facilidade em manter;
- ▶ Tempo de processamento;
- ▶ Consumo de memória.



Dado um problema:

- ▶ Como saber se um algoritmo é bom para resolvê-lo?
- ▶ Como saber qual é o melhor algoritmo entre duas opções?

Qual medida usar para definir **bom** ou **melhor**?

- ▶ Correção;
- ▶ Simplicidade;
- ▶ Facilidade em codificar;
- ▶ Facilidade em manter;
- ▶ **Tempo de processamento**;
- ▶ **Consumo de memória.**

Análise de algoritmos (complexidade)

Como medir a complexidade de um algoritmo?





# Análise de algoritmos

Como medir a complexidade?

Podemos executar o(s) algoritmo(s) e medir/plotar os resultados. Problemas:

- ▶ Sensível às entradas escolhidas, ao *software* e ao *hardware* usados;
- ▶ Comparação prejudicada;
- ▶ Necessário implementar e executar todos os algoritmos.



# Análise de algoritmos

## Como medir a complexidade?

Podemos executar o(s) algoritmo(s) e medir/plotar os resultados. Problemas:

- ▶ Sensível às entradas escolhidas, ao *software* e ao *hardware* usados;
- ▶ Comparação prejudicada;
- ▶ Necessário implementar e executar todos os algoritmos.

**Solução:** métodos analíticos.

- ▶ Definem a complexidade como uma função  $f(n)$  do tamanho  $n$  da entrada.



# Análise de algoritmos

## Como medir a complexidade?

Podemos executar o(s) algoritmo(s) e medir/plotar os resultados. Problemas:

- ▶ Sensível às entradas escolhidas, ao *software* e ao *hardware* usados;
- ▶ Comparação prejudicada;
- ▶ Necessário implementar e executar todos os algoritmos.

**Solução:** métodos analíticos.

- ▶ Definem a complexidade como uma função  $f(n)$  do tamanho  $n$  da entrada.

**Ideia geral** (complexidade de tempo):

- ▶ Contar o número de operações primitivas executadas pelo algoritmo;
- ▶ Cada operação primitiva executa em um tempo constante;
- ▶ Quanto menor o número de operações, mais eficiente é o algoritmo.



### Operações primitivas são passos básicos do algoritmo:

- ▶ Atribuição de valores;
- ▶ Operações aritméticas ou lógicas;
- ▶ Comparação de valores;
- ▶ Acesso a um elemento de um vetor;
- ▶ Recuperar a referência de um objeto;
- ▶ Chamada de um método;
- ▶ Retorno de um método.

### Exemplos:

```
int a = 10;  
int b = a - 7  
if (b < 5)  
    int c = v[3];  
Object x = this;  
this.compute();  
return result;
```



# Análise de algoritmos

## Exemplo I

Algoritmo arrayMax(A, n):

```
1  // Entrada: um vetor A com  $n \geq 1$  elementos inteiros.  
2  // Saída: o maior elemento de A.  
3  
4  currentMax  $\leftarrow$  A[0]  
5  for i  $\leftarrow$  1 to n - 1 do  
6      if currentMax < A[i] then  
7          currentMax  $\leftarrow$  A[i]  
8  
9  return currentMax
```

# Análise de algoritmos

## Exemplo I

Algoritmo arrayMax(A, n):

```
1 // Entrada: um vetor A com  $n \geq 1$  elementos inteiros.
2 // Saída: o maior elemento de A.
3
4 currentMax  $\leftarrow$  A[0]
5 for i  $\leftarrow$  1 to n - 1 do
6   if currentMax < A[i] then
7     currentMax  $\leftarrow$  A[i]
8
9 return currentMax
```

Linha	Operações	
4	1 acesso ao vetor + 1 atribuição	2
5	1 inicialização + $n$ comparações + $2(n - 1)$ incrementos	$3n - 1$
6	1 acesso ao vetor + 1 comparação, repetidos $n - 1$ vezes $\rightarrow 2(n - 1)$	$2n - 2$
7	0 [cond. nunca satisfeito] a $2(n - 1)$ [cond. sempre satisfeito]	$[0, 2n - 2]$
9	1 retorno	1

## Exemplo I

Complexidade de tempo no **melhor caso** ( $A[0]$  é o maior elemento):

►  $T(n) = 2 + 3n - 1 + 2n - 2 + 1 = 5n.$

Complexidade de tempo no **pior caso** ( $A[n - 1]$  é o maior elemento):

►  $T(n) = 2 + 3n - 1 + 2n - 2 + 2n - 2 + 1 = 7n - 2.$

Complexidade de tempo no **caso médio**:

► Depende da distribuição das entradas e do uso de teoria de probabilidades.

# Análise de algoritmos



## Exemplo I

Complexidade de tempo no **melhor caso** ( $A[0]$  é o maior elemento):

►  $T(n) = 2 + 3n - 1 + 2n - 2 + 1 = 5n.$

Complexidade de tempo no **pior caso** ( $A[n - 1]$  é o maior elemento):

►  $T(n) = 2 + 3n - 1 + 2n - 2 + 2n - 2 + 1 = 7n - 2.$

Complexidade de tempo no **caso médio**:

- Depende da distribuição das entradas e do uso de teoria de probabilidades.

Normalmente se considera a complexidade no **pior caso**, pois fornece um limite superior do tempo de execução. Logo:

- O algoritmo arrayMax executará no máximo  $7n - 2$  operações para cumprir sua tarefa;
- Seja  $\alpha$  o tempo gasto na operação primitiva mais complexa sob determinados *hardware* e *software*, o tempo de execução do algoritmo arrayMax será de, no máximo,  $\alpha(7n - 2).$



Qual a complexidade de tempo (operações) no pior caso do algoritmo abaixo?

```
1  for(int i = 0; i < n; i++) {  
2      for(int j = 0; j < n; j++) {  
3          if(matriz[i][j] != 0)  
4              // Operação com complexidade 1  
5      }  
6  }
```



Qual a complexidade de tempo (operações) no pior caso do algoritmo abaixo?

```
1  for(int i = 0; i < n; i++) {  
2      for(int j = 0; j < n; j++) {  
3          if(matriz[i][j] != 0)  
4              // Operação com complexidade 1  
5      }  
6  }
```

- ▶ O laço externo executa  $3n + 2$  operações.
  - ▶ Inicialização (1), comparações ( $n + 1$ ) e incremento ( $2n$ ).
- ▶ Mesma complexidade do laço interno, que repete  $n$  vezes. Logo,  $n(3n + 2) = 3n^2 + 2n$ .
- ▶ O condicional é executado  $n^2$  vezes. Logo,  $2n^2$ .
- ▶ A operação interna é executada  $n^2$  vezes, no pior caso.
- ▶  $T(n) = 3n + 2 + 3n^2 + 2n + 2n^2 + n^2 \quad \therefore \quad T(n) = 6n^2 + 5n + 2$ .

# Análise de algoritmos

## Taxa de crescimento



Note que  $T(n) = 7n - 2$  é uma função linear.

- ▶ O tempo de processamento cresce na mesma proporção do tamanho da entrada ( $n$ );
- ▶ A complexidade tempo desse algoritmo é **linear**.

Note que  $T(n) = 7n - 2$  é uma função linear.

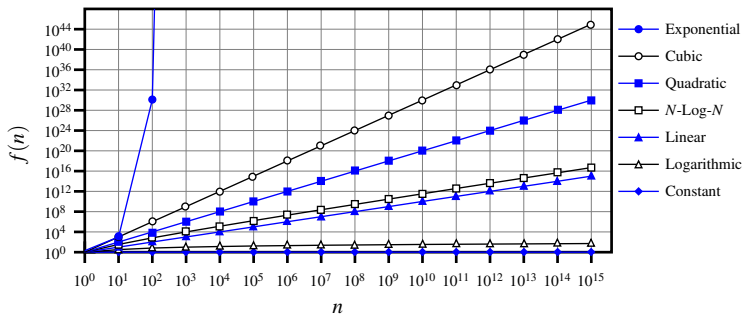
- ▶ O tempo de processamento cresce na mesma proporção do tamanho da entrada ( $n$ );
- ▶ A complexidade tempo desse algoritmo é **linear**.

A complexidade  $T(n)$  pode ser definida por funções com diferentes taxas de crescimento:

- ▶ constante  $\approx 1$
- ▶ logarítmica  $\approx \log n$
- ▶ linear  $\approx n$
- ▶  $n$ -log- $n$   $\approx n \log n$
- ▶ quadrática  $\approx n^2$
- ▶ cúbica  $\approx n^3$
- ▶ polinomial  $\approx n^k$
- ▶ exponencial  $\approx a^n \quad (a > 1)$

# Análise de algoritmos

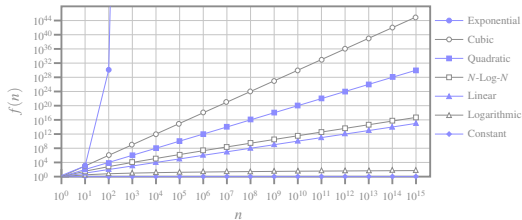
## Taxa de crescimento das funções de complexidade





# Análise de algoritmos

## Taxa de crescimento das funções de complexidade



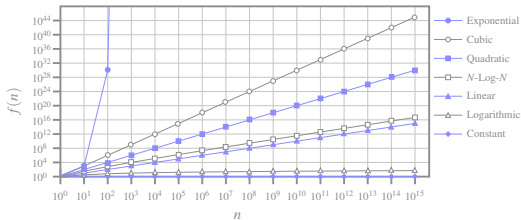
## Número de operações

$n$	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$
8	3	8	24	64	512	256
16	4	16	64	256	4 096	65 536
32	5	32	160	1 024	32 768	$4,3 \times 10^9$
64	6	64	384	4 096	262 144	$1,8 \times 10^{19}$
128	7	128	896	16 384	$2,1 \times 10^6$	$3,4 \times 10^{38}$
256	8	256	2 048	65 536	$1,7 \times 10^7$	$1,2 \times 10^{77}$
512	9	512	4 608	262 144	$1,3 \times 10^8$	$1,3 \times 10^{154}$



# Análise de algoritmos

## Taxa de crescimento das funções de complexidade



### Número de operações

$n$	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$
8	3	8	24	64	512	256
16	4	16	64	256	4 096	65 536
32	5	32	160	1 024	32 768	$4,3 \times 10^9$
64	6	64	384	4 096	262 144	$1,8 \times 10^{19}$
128	7	128	896	16 384	$2,1 \times 10^6$	$3,4 \times 10^{38}$
256	8	256	2 048	65 536	$1,7 \times 10^7$	$1,2 \times 10^{77}$
512	9	512	4 608	262 144	$1,3 \times 10^8$	$1,3 \times 10^{154}$

### Tempo de processamento

$n$	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$
100	< 1s	< 1s	< 1s	< 1s	< 1s	$10^{13}$ anos
1000	< 1s	< 1s	< 1s	< 1s	1s	$10^{284}$ anos
10000	< 1s	< 1s	< 1s	< 1s	16 min	$10^{2993}$ anos
100000	< 1s	< 1s	< 1s	10 s	12 dias	—
1000000	< 1s	< 1s	< 1s	16 min	32 anos	—



# Análise de algoritmos

## Taxa de crescimento das funções de complexidade

**Exemplo:** em um jogo existem 70 itens para compra (e.g., materiais, poderes e armas). Cada item tem um custo e fornece algum benefício. Itens combinados fornecem benefícios diferenciados. A fim de tomar a melhor decisão possível, queremos avaliar toda combinação possível de compra de itens, verificando o custo total e os benefícios esperados.

- ▶ Podemos representar uma compra usando um vetor binário  $V \in \{0, 1\}^{70}$ , onde o valor de uma posição  $i \in [0, 70]$  indica se o item  $i$  será comprado ou não.
- ▶ Devemos avaliar toda combinação possível de valores a  $V$ .



# Análise de algoritmos

## Taxa de crescimento das funções de complexidade



**Exemplo:** em um jogo existem 70 itens para compra (e.g., materiais, poderes e armas). Cada item tem um custo e fornece algum benefício. Itens combinados fornecem benefícios diferenciados. A fim de tomar a melhor decisão possível, queremos avaliar toda combinação possível de compra de itens, verificando o custo total e os benefícios esperados.

- ▶ Podemos representar uma compra usando um vetor binário  $V \in \{0, 1\}^{70}$ , onde o valor de uma posição  $i \in [0, 70]$  indica se o item  $i$  será comprado ou não.
- ▶ Devemos avaliar toda combinação possível de valores a  $V$ .

**Resultado:** o algoritmo de avaliação terá complexidade de tempo **exponencial**  $\rightarrow 2^n$ .

# Análise de algoritmos



## Taxa de crescimento das funções de complexidade

Um algoritmo com complexidade  $2^n$ , para  $n = 70$ , executando em um computador capaz de processar  $10^9$  operações por segundo, demoraria **37 436** anos para terminar sua execução!



# Análise de algoritmos

## Taxa de crescimento das funções de complexidade

Um algoritmo com complexidade  $2^n$ , para  $n = 70$ , executando em um computador capaz de processar  $10^9$  operações por segundo, demoraria **37 436** anos para terminar sua execução!

### **E se usarmos um computador mais rápido?**

- ▶ 100× mais rápido → 374 anos;
- ▶ 1 000× mais rápido → 37 anos;
- ▶ 1 000 000× mais rápido → 136 dias;
- ▶ 100 000 000× mais rápido → 1 dia.
  - ▶ Para  $n = 80$ , demoraria 140 dias;
  - ▶ Para  $n = 100$ , demoraria 401 969 anos.



# Análise de algoritmos

## Taxa de crescimento das funções de complexidade

Um algoritmo com complexidade  $2^n$ , para  $n = 70$ , executando em um computador capaz de processar  $10^9$  operações por segundo, demoraria **37 436** anos para terminar sua execução!

### E se usarmos um computador mais rápido?

- ▶ 100× mais rápido → 374 anos;
- ▶ 1 000× mais rápido → 37 anos;
- ▶ 1 000 000× mais rápido → 136 dias;
- ▶ 100 000 000× mais rápido → 1 dia.
  - ▶ Para  $n = 80$ , demoraria 140 dias;
  - ▶ Para  $n = 100$ , demoraria 401 969 anos.

### Moral da história

Um algoritmo melhor executando em um computador mais lento **ganhará sempre** de um algoritmo pior em um computador mais rápido, para instâncias suficientemente grandes.



A análise completa (contagem de operações) é muito detalhada e onerosa.



A análise completa (contagem de operações) é muito detalhada e onerosa.

Além disso, o que importa na prática é a **taxa de crescimento** da função de complexidade!



A análise completa (contagem de operações) é muito detalhada e onerosa.

Além disso, o que importa na prática é a **taxa de crescimento** da função de complexidade!

A **análise assintótica** foca em descrever a taxa de crescimento da complexidade de um algoritmo em função do tamanho  $n$  da entrada.

# Análise assintótica



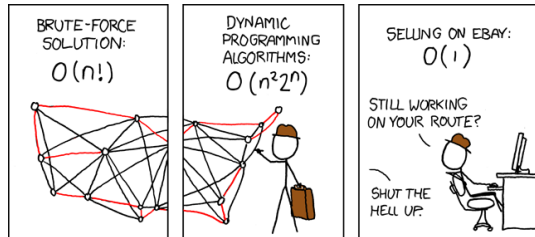
A análise completa (contagem de operações) é muito detalhada e onerosa.

Além disso, o que importa na prática é a **taxa de crescimento** da função de complexidade!

A **análise assintótica** foca em descrever a taxa de crescimento da complexidade de um algoritmo em função do tamanho  $n$  da entrada.

Para isso, usaremos a notação  $\mathcal{O}$  (*big-oh*).

- Bem como as notações  $\Theta$  e  $\Omega$ .

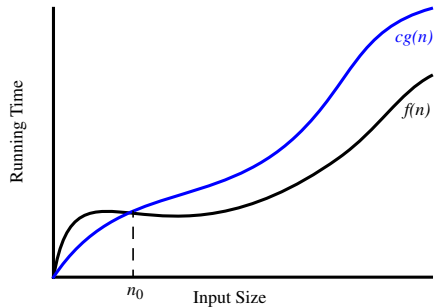




# Análise assintótica

Notação  $\mathcal{O}$  – definição formal

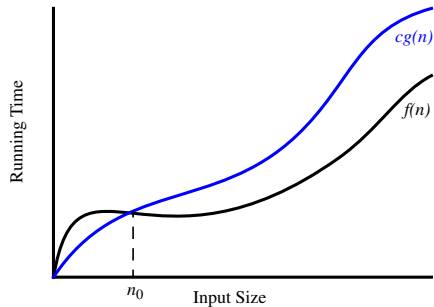
Sejam  $f(n)$  e  $g(n)$  funções que mapeiam o tamanho da entrada no tempo de processamento, dizemos que  $f(n)$  é  $\mathcal{O}(g(n))$  se existe uma constante real  $c > 0$  e uma constante inteira  $n_0 \geq 1$  tais que  $f(n) \leq cg(n)$  para todo inteiro  $n \geq n_0$ .



# Análise assintótica

Notação  $\mathcal{O}$  – definição formal

Sejam  $f(n)$  e  $g(n)$  funções que mapeiam o tamanho da entrada no tempo de processamento, dizemos que  $f(n)$  é  $\mathcal{O}(g(n))$  se existe uma constante real  $c > 0$  e uma constante inteira  $n_0 \geq 1$  tais que  $f(n) \leq cg(n)$  para todo inteiro  $n \geq n_0$ .



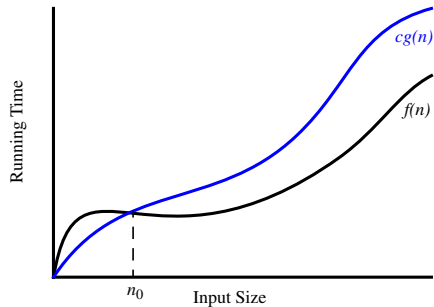
## Na prática:

- ▶ Se  $f(n)$  é  $\mathcal{O}(g(n))$ , então  $f(n)$  é “menor ou igual” a  $g(n)$  a medida que  $n$  cresce.
- ▶ Com isso,  $g(n)$  é um limite superior para  $f(n)$ .
- ▶ Ou seja,  $f(n)$  é tão boa quando  $g(n)$ .

# Análise assintótica

## Notação $\mathcal{O}$ – definição formal

Sejam  $f(n)$  e  $g(n)$  funções que mapeiam o tamanho da entrada no tempo de processamento, dizemos que  $f(n)$  é  $\mathcal{O}(g(n))$  se existe uma constante real  $c > 0$  e uma constante inteira  $n_0 \geq 1$  tais que  $f(n) \leq cg(n)$  para todo inteiro  $n \geq n_0$ .



### Na prática:

- ▶ Se  $f(n)$  é  $\mathcal{O}(g(n))$ , então  $f(n)$  é “menor ou igual” a  $g(n)$  a medida que  $n$  cresce.
- ▶ Com isso,  $g(n)$  é um limite superior para  $f(n)$ .
- ▶ Ou seja,  $f(n)$  é tão boa quanto  $g(n)$ .

**Exemplo:** a função  $T(n) = 7n - 2$  é  $\mathcal{O}(n)$ .

- ▶ Para  $c = 7$  e  $n_0 = 1$ , temos que  $7n - 2 \leq cn$ , para todo  $n \geq n_0$ . Logo  $T(n)$  é  $\mathcal{O}(n)$ .



# Análise assintótica

## Exemplo concreto

Qual a complexidade assintótica do algoritmo arrayMax?

```
1 // Entrada: um vetor A com  $n \geq 1$  elementos inteiros.  
2 // Saída: o maior elemento de A.  
3  
4 currentMax  $\leftarrow$  A[0]  
5 for i  $\leftarrow$  1 to n - 1 do  
6   if currentMax < A[i] then  
7     currentMax  $\leftarrow$  A[i]  
8  
9 return currentMax
```

# Análise assintótica

## Exemplo concreto

Qual a complexidade assintótica do algoritmo arrayMax?

```
1 // Entrada: um vetor A com  $n \geq 1$  elementos inteiros.  
2 // Saída: o maior elemento de A.  
3  
4 currentMax  $\leftarrow$  A[0]  
5 for i  $\leftarrow$  1 to n - 1 do  
6   if currentMax < A[i] then  
7     currentMax  $\leftarrow$  A[i]  
8  
9 return currentMax
```

Sabemos que, no pior caso, são executadas  $7n - 2$  operações. Logo, esse algoritmo tem complexidade  $\mathcal{O}(n)$ . Isto é, complexidade linear.

- ▶ Não precisamos contar todas as operações. Basta identificarmos o termo de maior complexidade (neste caso,  $n$ ), pois é quem define a taxa de crescimento da função!



# Análise assintótica

## Regras básicas I

**Função polinomial: sempre considerar o maior grau.**

- ▶  $5n^4 + 3n^3 + 2n^2 + 4n + 1$  é  $\mathcal{O}(n^4)$ .
- ▶  $n^3 + 600n$  é  $\mathcal{O}(n^3)$ .



**Função polinomial: sempre considerar o maior grau.**

- ▶  $5n^4 + 3n^3 + 2n^2 + 4n + 1$  é  $\mathcal{O}(n^4)$ .
- ▶  $n^3 + 600n$  é  $\mathcal{O}(n^3)$ .

**Constantes e multiplicadores são eliminados.**

- ▶  $2^{n+2} + 4$  é  $\mathcal{O}(2^n)$ .
- ▶  $4n^3$  é  $\mathcal{O}(n^3)$ .



**Função polinomial: sempre considerar o maior grau.**

- ▶  $5n^4 + 3n^3 + 2n^2 + 4n + 1$  é  $\mathcal{O}(n^4)$ .
- ▶  $n^3 + 600n$  é  $\mathcal{O}(n^3)$ .

**Constantes e multiplicadores são eliminados.**

- ▶  $2^{n+2} + 4$  é  $\mathcal{O}(2^n)$ .
- ▶  $4n^3$  é  $\mathcal{O}(n^3)$ .

**Função mista: sempre considerar o termo de maior complexidade.**

- ▶  $5n^2 + 3n \log n + 2n + 5$  é  $\mathcal{O}(n^2)$ .
- ▶  $2n + 100 \log n$  é  $\mathcal{O}(n)$ .





**Sempre considerar a menor complexidade possível.**

- ▶ É verdade que  $4n^2 + 10$  é  $\mathcal{O}(n^4)$ , mas é melhor dizer que é  $\mathcal{O}(n^2)$  [limite mais ajustado].



**Sempre considerar a menor complexidade possível.**

- ▶ É verdade que  $4n^2 + 10$  é  $\mathcal{O}(n^4)$ , mas é melhor dizer que é  $\mathcal{O}(n^2)$  [limite mais ajustado].

**Sempre considerar a representação mais simples.**

- ▶  $4n^2 + 2 \log n$  é  $\mathcal{O}(n^2)$ , o que é melhor que  $\mathcal{O}(n^2 + \log n)$ .

# Análise assintótica

Notações  $\Theta$  e  $\Omega$



## Notação $\mathcal{O}$ (majorante)

Se  $f(n)$  é  $\mathcal{O}(g(n))$ , então  $cg(n)$  é um **limite superior** para  $f(n)$ . Ou seja,  $f(n)$  **não é pior** que  $cg(n)$ .

# Análise assintótica

Notações  $\Theta$  e  $\Omega$



## Notação $\mathcal{O}$ (majorante)

Se  $f(n)$  é  $\mathcal{O}(g(n))$ , então  $cg(n)$  é um **limite superior** para  $f(n)$ . Ou seja,  $f(n)$  **não é pior** que  $cg(n)$ .

## Notação $\Theta$ (minorante)

Se  $f(n)$  é  $\Theta(g(n))$ , então  $cg(n)$  é um **limite inferior** para  $f(n)$ . Ou seja,  $f(n)$  **não é melhor** que  $cg(n)$ .

# Análise assintótica

Notações  $\Theta$  e  $\Omega$



## Notação $\mathcal{O}$ (majorante)

Se  $f(n)$  é  $\mathcal{O}(g(n))$ , então  $cg(n)$  é um **limite superior** para  $f(n)$ . Ou seja,  $f(n)$  **não é pior** que  $cg(n)$ .

## Notação $\Theta$ (minorante)

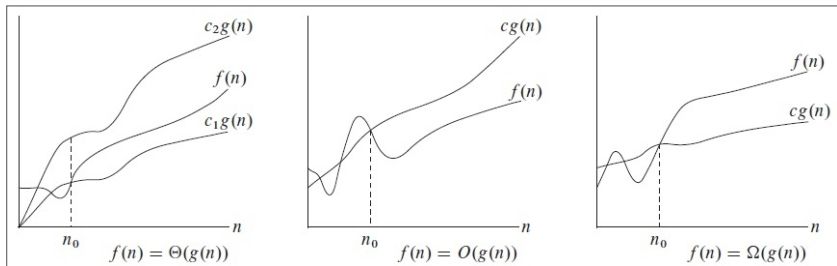
Se  $f(n)$  é  $\Theta(g(n))$ , então  $cg(n)$  é um **limite inferior** para  $f(n)$ . Ou seja,  $f(n)$  **não é melhor** que  $cg(n)$ .

## Notação $\Omega$ (limite “apertado” – majorante e minorante)

Se  $f(n)$  é  $\Omega(g(n))$ , então  $c_1g(n)$  é um **limite inferior** para  $f(n)$  e  $c_2g(n)$  é um **limite superior** para  $f(n)$ . Ou seja,  $f(n)$  **é igual** a  $cg(n)$ .

# Análise assintótica

## Notações $\Theta$ e $\Omega$



### Detalhes:

- ▶  $f(n)$  é  $\Theta(g(n)) \iff f(n)$  é  $O(g(n))$  e  $f(n)$  é  $\Omega(g(n))$ .
- ▶  $f(n)$  é  $\Theta(g(n)) \iff g(n)$  é  $\Theta(f(n))$ .
- ▶  $f(n)$  é  $O(g(n)) \iff g(n)$  é  $\Omega(f(n))$ .

# Análise assintótica

## Exemplo I

Algoritmo sumNumbers(n1, n2):

```
1  // Soma dois números inteiros.  
2  public static int sumNumbers(int n1, int n2) {  
3      int result = n1 + n2;  
4      return result;  
5  }
```

# Análise assintótica

## Exemplo I



Algoritmo sumNumbers(n1, n2):

```
1 // Soma dois números inteiros.  
2 public static int sumNumbers(int n1, int n2) {  
3     int result = n1 + n2;  
4     return result;  
5 }
```

### Análise:

- ▶ Linhas 3 e 4 executam operações de tempo constante;
- ▶ Complexidade **constante**:  $\mathcal{O}(1)$ ;
- ▶ Algoritmo de tempo constante.



# Análise assintótica

## Exemplo II



Algoritmo disjoint1(int[] vA, int[] vB, int[] vC):

```
1  // Retorna true se não existe nenhum elemento comum nos três grupos.
2  // Cada vetor possui elementos distintos dentro de si.
3  public static boolean disjoint1(int[] vA, int[] vB, int[] vC) {
4      for (int a : vA)
5          for (int b : vB)
6              for (int c : vC)
7                  if ((a == b) && (b == c))
8                      return false;
9      return true;
10 }
```



Algoritmo disjoint1(int[] vA, int[] vB, int[] vC):

```
1  // Retorna true se não existe nenhum elemento comum nos três grupos.
2  // Cada vetor possui elementos distintos dentro de si.
3  public static boolean disjoint1(int[] vA, int[] vB, int[] vC) {
4      for (int a : vA)
5          for (int b : vB)
6              for (int c : vC)
7                  if ((a == b) && (b == c))
8                      return false;
9      return true;
10 }
```

### Análise:

- ▶ A operação constante da linha 7 é repetida  $n \times n \times n = n^3$  vezes;
- ▶ Complexidade **cúbica**:  $\mathcal{O}(n^3)$ ;
- ▶ Algoritmo de tempo cúbico.

# Análise assintótica

## Exemplo III



Algoritmo disjoint2(int[] vA, int[] vB, int[] vC):

```
1  // Retorna true se não existe nenhum elemento comum nos três grupos.
2  // Cada vetor possui elementos distintos dentro de si.
3  public static boolean disjoint2(int[] vA, int[] vB, int[] vC) {
4      for (int a : vA)
5          for (int b : vB)
6              if (a == b)
7                  for (int c : vC)
8                      if (a == c) return false;
9      return true;
10 }
```



Algoritmo disjoint2(int[] vA, int[] vB, int[] vC):

```
1 // Retorna true se não existe nenhum elemento comum nos três grupos.
2 // Cada vetor possui elementos distintos dentro de si.
3 public static boolean disjoint2(int[] vA, int[] vB, int[] vC) {
4     for (int a : vA)
5         for (int b : vB)
6             if (a == b)
7                 for (int c : vC)
8                     if (a == c) return false;
9     return true;
10 }
```

### Análise:

- ▶ Os laços das linhas 4 e 5 sempre são executados –  $\mathcal{O}(n^2)$ ;
- ▶ No máximo  $n$  pares são iguais (lin. 6), então o laço da linha 7 executa no máximo  $n$  vezes;
- ▶ Complexidade **quadrática**:  $n^2 + n^2 \iff \mathcal{O}(n^2)$ .



Algoritmo repeat1(char c, int n):

```
1  // Compõe uma String com o caractere c repetido n vezes.
2  public static String repeat1(char c, int n) {
3      String answer = "";
4      for (int j = 0; j < n; j++)
5          answer += c;
6      return answer;
7  }
```



Algoritmo repeat1(char c, int n):

```
1 // Compõe uma String com o caractere c repetido n vezes.
2 public static String repeat1(char c, int n) {
3     String answer = "";
4     for (int j = 0; j < n; j++)
5         answer += c;
6     return answer;
7 }
```

### Análise:

- ▶ Strings são imutáveis em Java: o comando `answer += c` implica em criar uma nova String, copiar cada caractere da String antiga para ela, e acrescentar o caractere `c`;
- ▶ A linha 5 executa operações conforme o tamanho de `answer`:  $1 + 2 + \dots + n - 1$ ;
- ▶ Logo, sua complexidade é  $\sum_{j=0}^{n-1} j = n(n+1)/2$ ;
- ▶ Complexidade **quadrática**:  $\mathcal{O}(n^2)$ .



Algoritmo repeat2(char c, int n):

```
1 // Compõe uma String com o caractere c repetido n vezes.
2 public static String repeat2(char c, int n) {
3     StringBuilder sb = new StringBuilder();
4     for (int j = 0; j < n; j++)
5         sb.append(c);
6     return sb.toString();
7 }
```



Algoritmo repeat2(char c, int n):

```
1 // Compõe uma String com o caractere c repetido n vezes.
2 public static String repeat2(char c, int n) {
3     StringBuilder sb = new StringBuilder();
4     for (int j = 0; j < n; j++)
5         sb.append(c);
6     return sb.toString();
7 }
```

### Análise:

- ▶ A classe `StringBuilder` implementa uma lista dinâmica, que permite executar a operação `sb.append(c)` em tempo constante;
- ▶ Complexidade **linear**:  $\mathcal{O}(n)$ ;
- ▶ O algoritmo é o mesmo, o que muda é a estrutura de dados.



Algoritmo unique1(int[] data):

```
1 // Retorna true se não existe elemento duplicado no vetor.
2 public static boolean unique1(int[] data) {
3     int n = data.length;
4     for (int j = 0; j < n - 1; j++)
5         for (int k = j + 1; k < n; k++)
6             if (data[j] == data[k])
7                 return false;
8     return true;
9 }
```



Algoritmo unique1(int[] data):

```
1 // Retorna true se não existe elemento duplicado no vetor.
2 public static boolean unique1(int[] data) {
3     int n = data.length;
4     for (int j = 0; j < n - 1; j++)
5         for (int k = j + 1; k < n; k++)
6             if (data[j] == data[k])
7                 return false;
8     return true;
9 }
```

### Análise:

- ▶ O laço interno é executado  $(n - 1) + (n - 2) + \dots + 2 + 1$  vezes;
- ▶ Complexidade **quadrática**:  $\mathcal{O}(n^2)$ .



Algoritmo unique2(int[] data):

```
1 // Retorna true se não existe elemento duplicado no vetor.
2 // O vetor é ordenado para verificar apenas elementos subsequentes.
3 public static boolean unique2(int[] data) {
4     int n = data.length;
5     Arrays.sort(data); // Operação  $O(n \log n)$ 
6     for (int j = 0; j < n - 1; j++)
7         if (data[j] == data[j+1])
8             return false;
9     return true;
10 }
```



Algoritmo unique2(int[] data):

```
1 // Retorna true se não existe elemento duplicado no vetor.
2 // O vetor é ordenado para verificar apenas elementos subsequentes.
3 public static boolean unique2(int[] data) {
4     int n = data.length;
5     Arrays.sort(data); // Operação  $\mathcal{O}(n \log n)$ 
6     for (int j = 0; j < n - 1; j++)
7         if (data[j] == data[j+1])
8             return false;
9     return true;
10 }
```

### Análise:

- ▶ A ordenação custa  $n \log n$  e percorrer o vetor custa  $n$ ;
- ▶ Complexidade  **$n \log n$** :  $n \log n + n \iff \mathcal{O}(n \log n)$ .

