

# Estrutura de Dados

*Estrutura de Dados*

*Complexidade de Algoritmos*

# Sumário

- 1 Introdução
- 2 Algoritmos
- 3 Análise de Algoritmos
- 4 Definição

- 5 Como Analisar
- 6 Comportamento Assintótico
- 7 Notação Big-O
- 8 Exemplos

# Introdução

**Nota:**

**Pense em um número entre 1 e 100!**



*Uma tentativa ruim de acertar o número.*

(Bhargava, 2016)



(Bhargava, 2016)



(Bhargava, 2016)



(Bhargava, 2016)



# Algoritmos

## Como resolver um problema?

- › Descrever o problema de forma clara e precisa;
- › Escrever o algoritmo correspondente;
- › Sequência de instruções simples e objetiva.

## Vários algoritmos para o mesmo problema:

- › Algoritmos diferem na forma como utilizam os recursos do computador.
- › Principais recursos que influenciam:
  - » Tempo de execução
  - » Uso de memória

Um algoritmo eficiente resolve o problema corretamente usando menos recursos.

# Análise de Algoritmos

## Análise de Algoritmos

- › Área de pesquisa cujo foco são os algoritmos;
- › Busca responder a seguinte pergunta: **podemos fazer um algoritmo mais eficiente?**
- › Algoritmos diferentes, mas capazes de resolver o mesmo problema, não necessariamente o fazem com a mesma eficiência.

# Definição

## Definição

A **complexidade de um algoritmo** mede o custo computacional (em tempo e/ou espaço) necessário para executar um algoritmo em função do tamanho da entrada ( $n$ ).

### Por que é importante?

- › Permite **comparar algoritmos** com base em sua eficiência.
- › Ajuda a **prever desempenho** para grandes volumes de dados.
- › Suporta **decisões de projeto** em sistemas de alto desempenho.
- › Auxilia na **otimização** de programas.

Formas de determinar a eficiência de um algoritmo:

- 1 Análise empírica;
  - » Executamos o algoritmo e medimos seu tempo de execução.
- 2 Análise matemática.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 int main(void){
5     clock_t inicio, fim;
6     inicio = clock();
7     algoritmo_analisado();
8     fim = clock();
9     tempo = (fim - inicio) * 1000 / CLOCKS_PER_SEC;
10    printf("Tempo: %lu milissegundos!\n", tempo);
11 }
```

Código 1: Análise empírica.

## Análise Empírica

### Vantagens:

- › Avalia desempenho em uma configuração específica de computador/linguagem;
- › Considera custos não aparentes (ex.: alocação de memória);
- › Permite comparar computadores e linguagens.

### Desvantagens:

- › Necessidade de implementar o algoritmo;
- › Depende da habilidade do programador;
- › Resultados podem ser influenciados por hardware e software;
- › Depende da natureza dos dados (reais, aleatórios, perversos).

## Análise Matemática

### Vantagens:

- › Estudo formal da ideia do algoritmo;
- › Usa simplificações e computador idealizado;
- › Ignora detalhes de hardware, linguagem e CPU;
- › Permite entender o comportamento à medida que os dados crescem.

### Desvantagens:

- › Pode não refletir o desempenho real em hardware específico;
- › Não considera eventos de software ou implementação.

- › **Complexidade de Tempo:** mede quantas operações são realizadas.
- › **Complexidade de Espaço:** mede o uso de memória durante a execução.

## Exemplo

Um algoritmo pode ser rápido ( $O(n)$ ), mas usar muita memória ( $O(n^2)$ ).

## Melhor Caso

Situação mais favorável, onde o algoritmo realiza o menor número de operações.

## Pior Caso

Situação mais desfavorável, onde o algoritmo realiza o maior número de operações.

## Caso Médio

Situação típica ou esperada, considerando a distribuição das entradas.

# Como Analisar

**Algoritmo exemplo:** encontra o maior valor em um array  $A$  com  $n$  elementos e armazena em  $M$

```
1 int M = A[0];  
2 for(int i = 0; i < n; i++){  
3     if(A[i] >= M){  
4         M = A[i];  
5     }  
6 }
```

Código 2: Exemplo: encontra o maior valor de um vetor (*array*).

**Instruções simples:**

- › Atribuição de valor a uma variável
- › Acesso a um elemento do array
- › Comparação de valores
- › Incremento
- › Operações aritméticas básicas

### Notas importantes:

- Todas as instruções simples têm o mesmo custo
- Comandos de seleção (if) têm custo zero

**Custo da inicialização de  $M$ :** 1 instrução (atribuição)

**Custo do laço for inicial:** 2 instruções (uma atribuição + uma comparação)

**Custo do laço for em execução:**  $2n$  instruções (incremento + comparação, executadas  $n$  vezes)

**Custo total antes do conteúdo do laço:**  $3 + 2n$  instruções



**Considerando comandos dentro do for:**

- › if: 1 instrução sempre executada
- › atribuição: 1 instrução, depende do if

**Pior caso:** array em ordem crescente

- › Valor de  $M$  sempre substituído
- › Total de instruções adicionais:  $2n$

Função de custo no pior caso:

$$f(n) = 3 + 2n + 2n = 4n + 3$$

```
1 int M = A[0];  
2 for(int i = 0; i < n; i++){  
3     if(A[i] >= M){  
4         M = A[i];  
5     }  
6 }
```

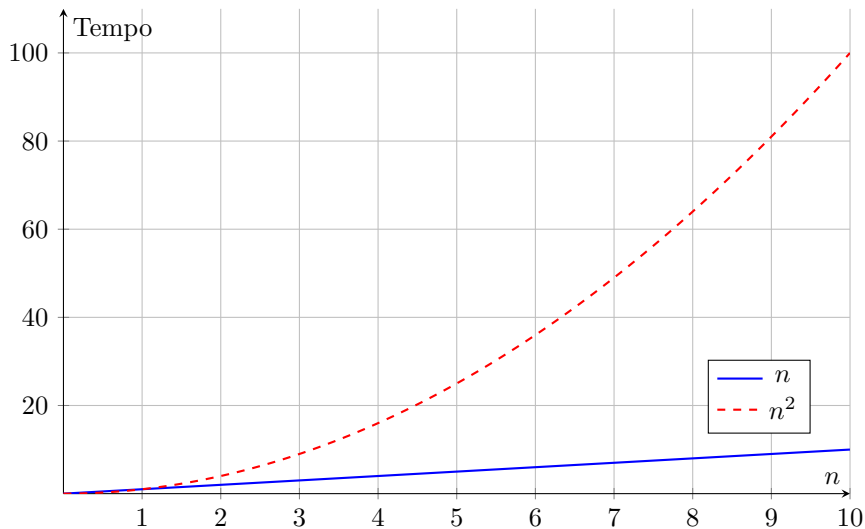
Código 3: Exemplo: encontra o maior valor de um vetor (*array*).

**Exemplo para matriz  $A$  de tamanho  $n \times n$ :**

$$f(n) = 1 + 2 + 2n + n(2 + 2n + 2n) = 3 + 4n + 4n^2$$

```
1 int M = A[0][0];  
2 for(int i = 0; i < n; i++){  
3     for(int j = 0; j < n; j++){  
4         if(A[j] >= M){  
5             M = A[j];  
6         }  
7     }  
8 }
```

Código 4: Exemplo: encontra o maior valor de uma matriz.



# Comportamento Assintótico

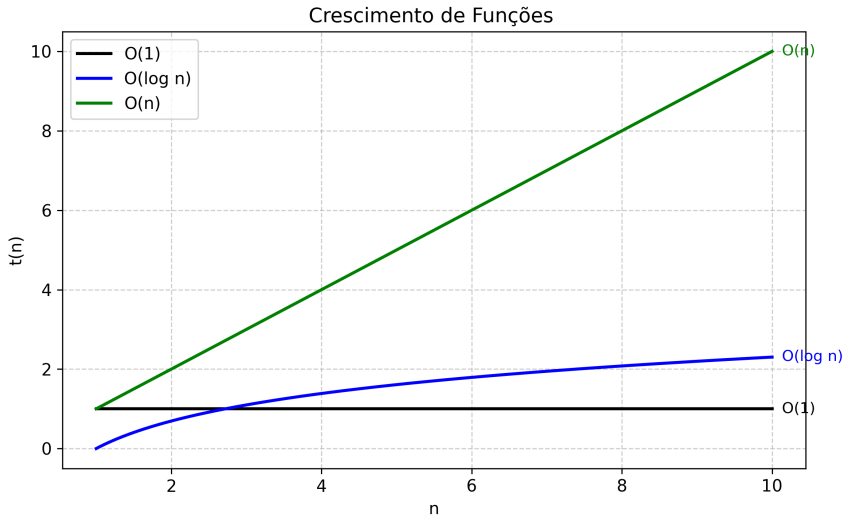
- › O custo do algoritmo pode ser representado por uma função de complexidade
- › Exemplo:  $f(n) = 4n + 3$
- › Nos dá uma ideia do custo de execução para um problema de tamanho  $n$
- › Podemos também analisar o espaço gasto pelo algoritmo



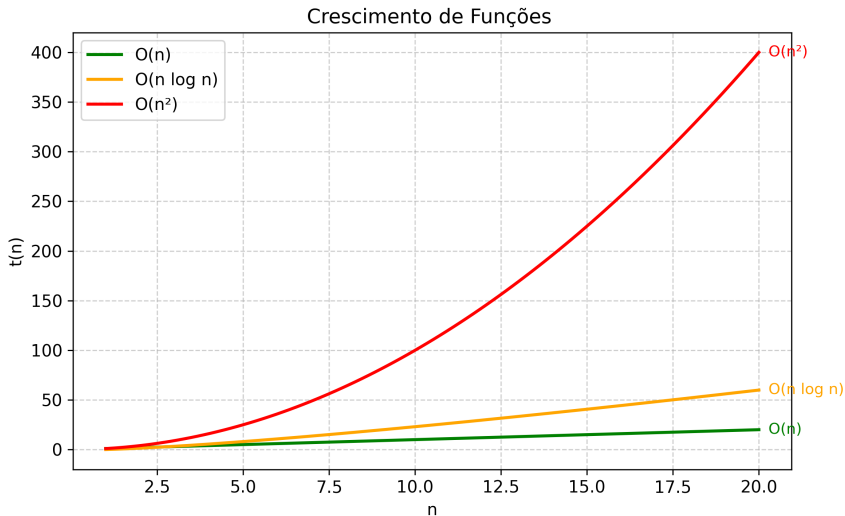
- › Nem todos os termos da função são importantes para avaliar o custo
- › Devemos manter apenas os termos que crescem mais rápido quando  $n$  aumenta
- › Termos constantes ou de crescimento lento podem ser descartados

- › Função:  $f(n) = 4n + 3$
- › 3 é constante e não muda com o crescimento de  $n \Rightarrow$  pode ser descartada
- › Multiplicadores constantes também podem ser descartados
- › Assim,  $f(n) = 4n \Rightarrow f(n) = n$  (análise assintótica)

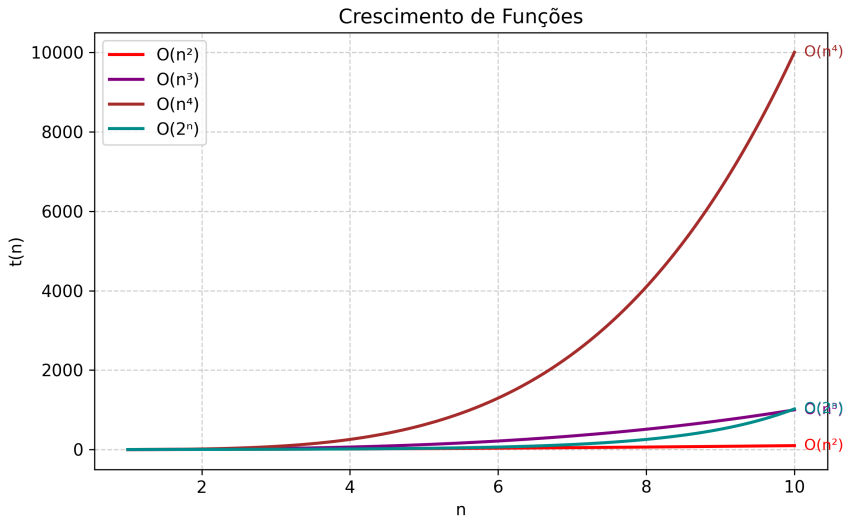
- › O termo de maior crescimento domina a função
- › Exemplo:
  - ›  $g(n) = 1000n + 500 \Rightarrow O(n)$
  - ›  $h(n) = n^2 + n + 1 \Rightarrow O(n^2)$
- › Podemos ignorar termos e constantes menos relevantes



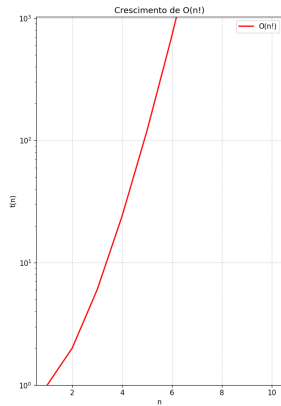
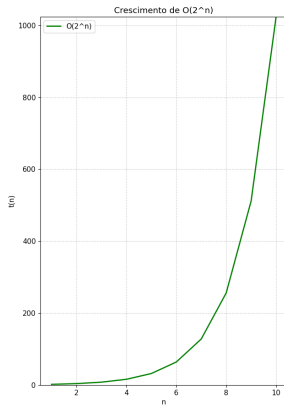
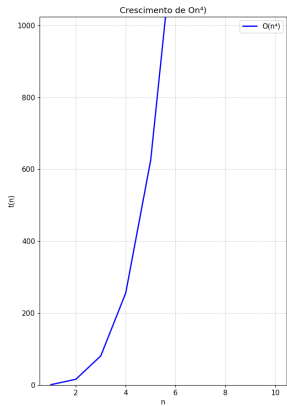
# Comportamento Assintótico II



# Comportamento Assintótico III



# Comportamento Assintótico IV



- ›  $f(n) = 105 \Rightarrow O(1)$
- ›  $f(n) = 15n + 2 \Rightarrow O(n)$
- ›  $f(n) = n^2 + 5n + 2 \Rightarrow O(n^2)$
- ›  $f(n) = 5n^3 + 200n^2 + 112 \Rightarrow O(n^3)$



## Hierarquia assintótica de funções

Do crescimento mais lento para o mais rápido:

$$\underbrace{\varepsilon}_{0 < \varepsilon < 1} < \underbrace{1}_{\text{constante}} < \underbrace{\log n}_{\text{logaritmo}} < \underbrace{(\log n)^k}_{\text{potência de log}} < \underbrace{n^\varepsilon}_{\text{potência fracionária}} < \underbrace{n}_{\text{linear}} < \underbrace{n^c}_{\text{polinomial com } c > 1} < \underbrace{c^n}_{\text{exponencial}} < \underbrace{n!}_{\text{fatorial}} < \underbrace{n^n}_{\text{quase exponencial}}$$

*ou de forma simplificada:*

$$\varepsilon < 1 < \log n < (\log n)^k < n^\varepsilon < n < n^c < c^n < n! < n^n$$

- › Sem laço (exceto recursão):  $f(n) = 1$
- › Um laço de 1 a  $n$ :  $f(n) = n$
- › Dois laços aninhados:  $f(n) = n^2$
- › Três laços aninhados:  $f(n) = n^3$
- › Etc.

# Notação Big-O

- › Representa o limite superior do custo do algoritmo
- › Considera o pior caso possível para todas as entradas de tamanho  $n$
- › Permite saber que o algoritmo não vai ultrapassar determinado custo

- › Podemos criar um laço interno "pior caso" sempre executando  $n$  vezes
- › Função de custo aproximada:  $f(n) = n^2$
- › Notação grande-O:  $O(n^2)$
- › Indica limite superior do custo do algoritmo

## Definições

- ›  $O(f(n))$ : Limite superior — crescimento máximo (pior caso).
- ›  $\Omega(f(n))$ : Limite inferior — crescimento mínimo (melhor caso).
- ›  $\Theta(f(n))$ : Crescimento assintótico exato (caso médio).

## Exemplo

---

Para o algoritmo Bubble Sort:

- › Melhor caso:  $\Omega(n)$
- › Pior caso:  $O(n^2)$
- › Caso médio:  $\Theta(n^2)$

| Complexidade  | Descrição    | Exemplo                     |
|---------------|--------------|-----------------------------|
| $O(1)$        | Constante    | Acesso direto a vetor       |
| $O(\log n)$   | Logarítmica  | Busca binária               |
| $O(n)$        | Linear       | Busca sequencial            |
| $O(n \log n)$ | Quase linear | Merge Sort                  |
| $O(n^2)$      | Quadrática   | Bubble Sort                 |
| $O(2^n)$      | Exponencial  | Força bruta em subconjuntos |
| $O(n!)$       | Fatorial     | Permutações completas       |

Tabela 1: Crescimento de complexidades comuns.



# Exemplos

## Passos para Calcular a Complexidade

- 1 Identifique as operações principais (atribuições, somas, comparações, etc.).
- 2 Conte quantas vezes cada operação é executada.
- 3 Expresse o total de operações como uma função de  $n$ .
- 4 Elimine constantes e mantenha o termo de maior crescimento.

## Exemplo 1 — Código simples

```
int soma = 0;           // 0(1)
for (int i = 0; i < n; i++) { // repete n vezes
    soma = soma + i;     // 0(1) por iteração
}
```

### Análise:

- › Linha 1:  $O(1)$
- › Loop:  $n \times O(1) = O(n)$
- › Total:  $O(1) + O(n) \Rightarrow \boxed{O(n)}$

## Exemplo 2 — Loop duplo

```
for (int i = 0; i < n; i++) {           // O(n)
    for (int j = 0; j < n; j++) {       // O(n) por iteração de i
        soma++;                         // O(1)
    }
}
```

**Análise:**  $n \times n \times O(1) = O(n^2)$

### Exemplo 3 — Recursão simples

```
void soma_recursiva(int n, int *soma) {  
    if (n <= 0) return;  
    (*soma)++;                // O(1)  
    soma_recursiva(n - 1, soma); // O(n)  
}
```

**Análise:**  $n \times O(1) = O(n)$

## Exemplo 4 — Fibonacci Recursivo

```
int fibonacci(int n) {  
    if (n <= 0)  
        return 0;  
    if (n == 1)  
        return 1;  
    return fibonacci(n - 1) + fibonacci(n - 2); // chamadas recursivas  
}
```

**Análise:**  $T(n) = T(n - 1) + T(n - 2) + O(1) \implies O(2^n)$

## Exemplo 5 — Torre de Hanoi Recursiva

```
void hanoi(int n, char origem, char destino, char auxiliar) {  
    if (n == 0) return;  
    hanoi(n - 1, origem, auxiliar, destino);  
    printf("Mover disco %d de %c para %c\n", n, origem, destino);  
    hanoi(n - 1, auxiliar, destino, origem);  
}
```

**Análise:**  $T(n) = 2 \cdot T(n - 1) + O(1) \implies O(2^n)$

## Regra geral

- › Ignore constantes:  $3n + 10 \Rightarrow O(n)$
- › Ignore termos menores:  $n^2 + n \Rightarrow O(n^2)$

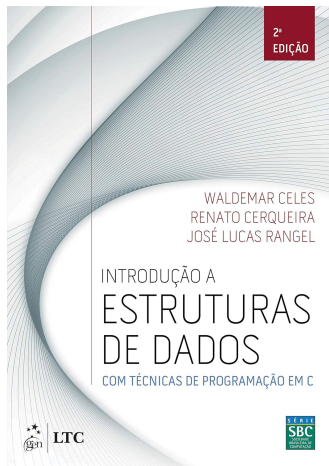


# Complexidade dos Algoritmos de Ordenação I

| Algoritmo      | Melhor Caso   | Médio         | Pior Caso     | Estável |
|----------------|---------------|---------------|---------------|---------|
| Bubble Sort    | $O(n)$        | $O(n^2)$      | $O(n^2)$      | Sim     |
| Insertion Sort | $O(n)$        | $O(n^2)$      | $O(n^2)$      | Sim     |
| Selection Sort | $O(n^2)$      | $O(n^2)$      | $O(n^2)$      | Não     |
| Merge Sort     | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | Sim     |
| Quick Sort     | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$      | Não     |
| Heap Sort      | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | Não     |

Tabela 2: Comparativo de complexidades dos algoritmos de ordenação.

(CELES; CERQUEIRA; RANGEL, 2004) - Capítulo 11



(Bhargava, 2016)





CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. **Introdução a estruturas de dados: com técnicas de programação em C**. Rio de Janeiro: Elsevier, 2004.



DEITEL, Paul; DEITEL, Harvey. **C: Como Programar**. 6. ed. São Paulo: Pearson Universidades, 2011.



SCHILDT, Herbert. **C Completo e Total: O Guia Definitivo para Programação em C**. 3. ed. São Paulo: Makron Books, 1996. ISBN 978-8534606928.

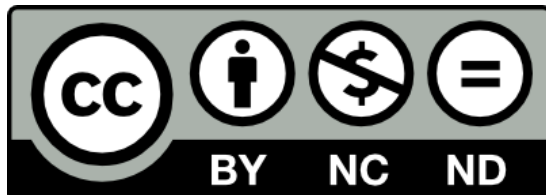


BHARGAVA, Aditya. **Entendendo Algoritmos: Um Guia Ilustrado para Programadores e Outros Curiosos**. 1. ed. São Paulo, Brasil: Novatec, 2016. ISBN 978-85-7522-367-8.



CORMEN, Thomas H. *et al.* **Algoritmos: Teoria e Prática**. 4ª. Rio de Janeiro: LTC, 2024. p. 912. ISBN 9788595159907.

Estes slides estão protegidos por uma licença Creative Commons



Este modelo foi adaptado de Maxime Chupin.

# Estrutura de Dados

*Estrutura de Dados*

*Complexidade de Algoritmos*