
# Estruturas de Dados Lineares

Introdução à Análise de Algoritmos (cont.)

**Prof. Vinicius Bischoff**

v.08/2022

# Compreensão do "if"

```
def imprimir_if(n: int) -> None:
    if n < 10:          1
        print(n)                1
    else:
        for i in range(n):      n + 1
            print(i)            n
```

Para n < 10, temos o melhor caso com T(n) = 2 $\Rightarrow$ O(1)
Para n >= 10, temos o pior caso com T(n) = 2n + 2 $\Rightarrow$ O(n)

Observa-se que o comando "if" poderá alterar a classe de complexidade e, portanto, deve ser definido quais os casos de análise.

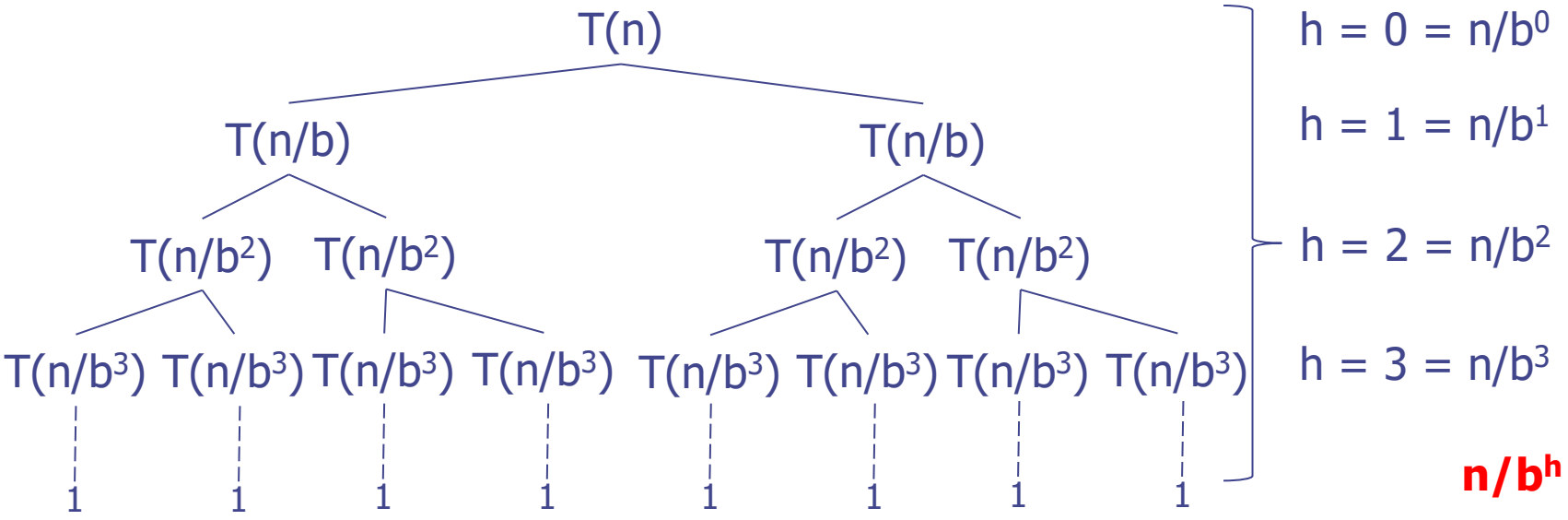
# Algoritmo Constante O(1)

```
def max(a: int, b: int) -> int:
    return a if a > b else b
           1      1         1

                        T(n) = 2 = O(1)
```

```
def somar1ate10() -> int:
    soma: int = 0                   1
    for i in range(1, 11):          11
        soma += i                   10
    return soma                     1

                        T(n) = 23 = O(1)
```

Observa-se que um algoritmo de ordem constante pode-se utilizar de instruções de repetição, como é o caso do segundo algoritmo.

# Algoritmo Logarítmico O(log(n))

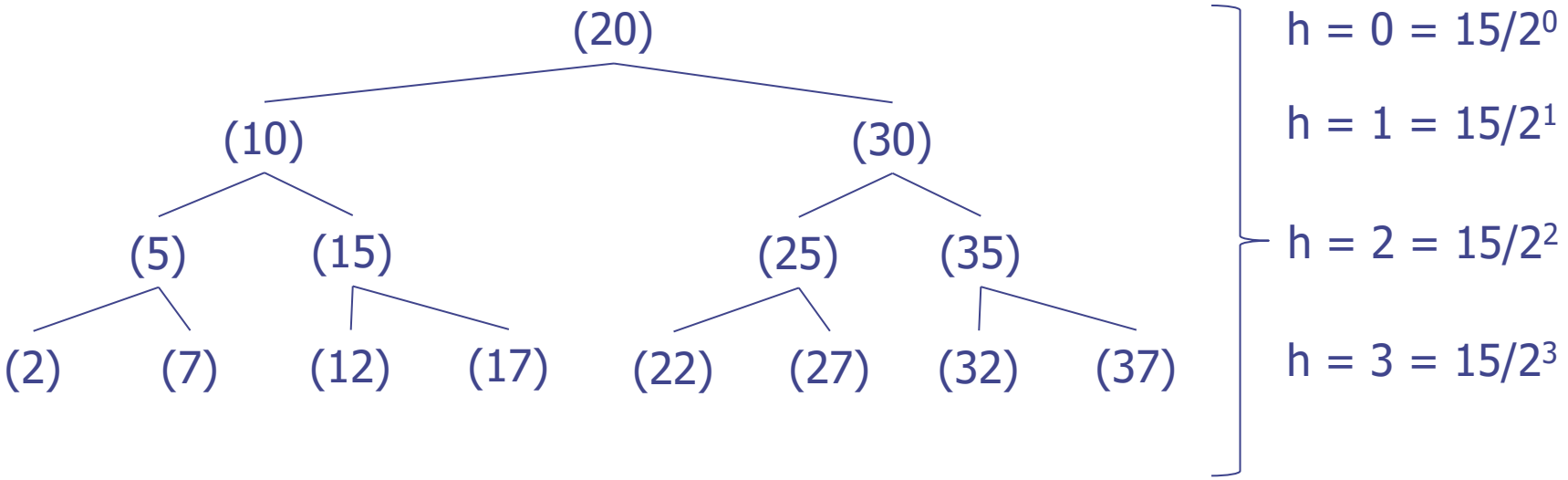



Estamos definindo que o pior caso é quando o elemento está na folha em uma BST cheia, ou seja, quando atingimos 1. Temos:

$$\frac{n}{b^h} = 1 \Longrightarrow n = b^h \Longrightarrow \mathbf{h} = \lfloor \boldsymbol{log_b(n)} \rfloor + \mathbf{1}$$

**(log é função inversa da exponencial)**

# Algoritmo Logarítmico O(log(n))




Pesquisando o número 7 resultará em 4 acessos, logo:

$$\lfloor log_b(n) \rfloor + 1 = \lfloor \log_2(15) \rfloor + 1 = 4\ acessos$$

# Algoritmo Logarítmico O(log(n))

```
def imprimir_pulando(n: int) -> None:
    i: int = 1              1
    while i < n:            log2(n) + 1
        print(i)            log2(n)
        i *= 2              log2(n)
```

$$T(n) \approx 3\log_2(n) + 2 = O(\log(n))$$

| i | n = 8 | n = 10 |
|---|---|---|
| 1 | imprime 1 | imprime 1 |
| 2 | imprime 2 | imprime 2 |
| 4 | imprime 4 | imprime 4 |
| 8 | sai do for | imprime 8 |
| 16 | | sai do for |

3 (n = 8) | 4 (n = 10)

| i | i *= 2 |
|---|---|
| 1 | $2^0$ |
| 2 | $2^1$ |
| 4 | $2^2$ |
| 8 | $2^3$ |
| | $2^k$ |

$i \geq n \Rightarrow 2^k \geq n \Rightarrow k = \log_2(n)$

Observa-se que $\log_2(8) = 3$ e $\log_2(10) = 3.32$. Portanto, o 3.32 não representa as 4 iterações quando o n = 10. Neste caso, quando o cálculo do log resultar em decimal, devemos utilizar a função teto (ceiling). Ficará **$T(n) = 3\lceil \log_2(n) \rceil + 2$**.

# Algoritmo Logarítmico O(log(n))

```
def imprimir_pulando_decrescente(n: int) -> None:
    i: int = n              1
    while i >= 1:           log2(n) + 2
        print(i)            log2(n) + 1
        i //= 2             log2(n) + 1
```

$$T(n) \approx 3\log_2(n) + 5 = O(\log(n))$$

| i = 8 | i /= 2 |
|---|---|
| 8 | $8/2^0$ |
| 4 | $8/2^1$ |
| 2 | $8/2^2$ |
| 1 | $8/2^3$ |
| | $n/2^k$ |
| $i < 1 \Rightarrow n/2^k < 1 \Rightarrow n = 2^k \Rightarrow k = \log_2(n)$ | |

Princípio da pesquisa binária!

Observa-se que o algoritmo imprime o número em ordem decrescente e, por isso, possui limites diferentes do algoritmo anterior. Neste caso, usaremos a função floor (piso) para arredondar o resultado do logaritmo. Ficará **$T(n) = 3\lfloor\log_2(n)\rfloor + 5$**.

# Algoritmo Linear O(n)

```
def imprimir_linear(a: np.array) -> None:
    for elm in a:       n + 1
        print(elm)      n
```

**T(n) = 2n + 1 = O(n)**

```
def imprimir_linear_decrescente(a: np.array) -> None:
    for elm in reversed(a):    n + 1
        print(elm)             n
```

**T(n) = 2n + 1 = O(n)**

```
def imprimir_pulando(a: np.array) -> None:
    for elm in a[::2]:  n / 2 + 1
        print(elm)              n / 2
```

**T(n) = n + 1 = O(n)**

**Metade da função de tempo das anteriores!**

**Importante: omitida a função teto em n / 2.**

# Algoritmo Linearítmico O(nlog(n))

```
def imprimir_linearitmico(n: int) -> None:
    for i in range(n):                    n + 1
        j: int = 1                        n
        while j < n:                      n.(log2(n) + 1) ⇒ nlog2(n) + n
            print(j, end= ' ‘)            n.log2(n) ⇒ nlog2(n)
            j *= 2                        n.log2(n) ⇒ nlog2(n)
        print()                           n
```

$T(n) \approx 3n\log_2(n) + 4n + 1 = O(n\log(n))$

Observar a função teto no logaritmo.

# Algoritmo Quadrático $O(n^2)$

```
def imprimir_quadratico(a: np.matrix) -> None:
  for i, _ in enumerate(a):                n + 1
    for _, elm in enumerate(a[i]):         n.(n + 1) ⇒ n² + n
        print(elm, end = ' ‘)              n.n ⇒ n²
    print()                                n
```

$T(n) = 2n^2 + 3n + 1 = O(n^2)$

**Importante:** a classe/ordem considera o polinômio de maior grau.

Considera-se o exemplo acima como uma matriz quadrática.

# Algoritmo Quadrático O(n²)

```
def imprimir_triangulo(n: int) -> None:
    for i in range(n):
        for j in range(i):
            print(i, end= ' ')
        print()
```

n + 1 → $\frac{n(n+1)}{2}$

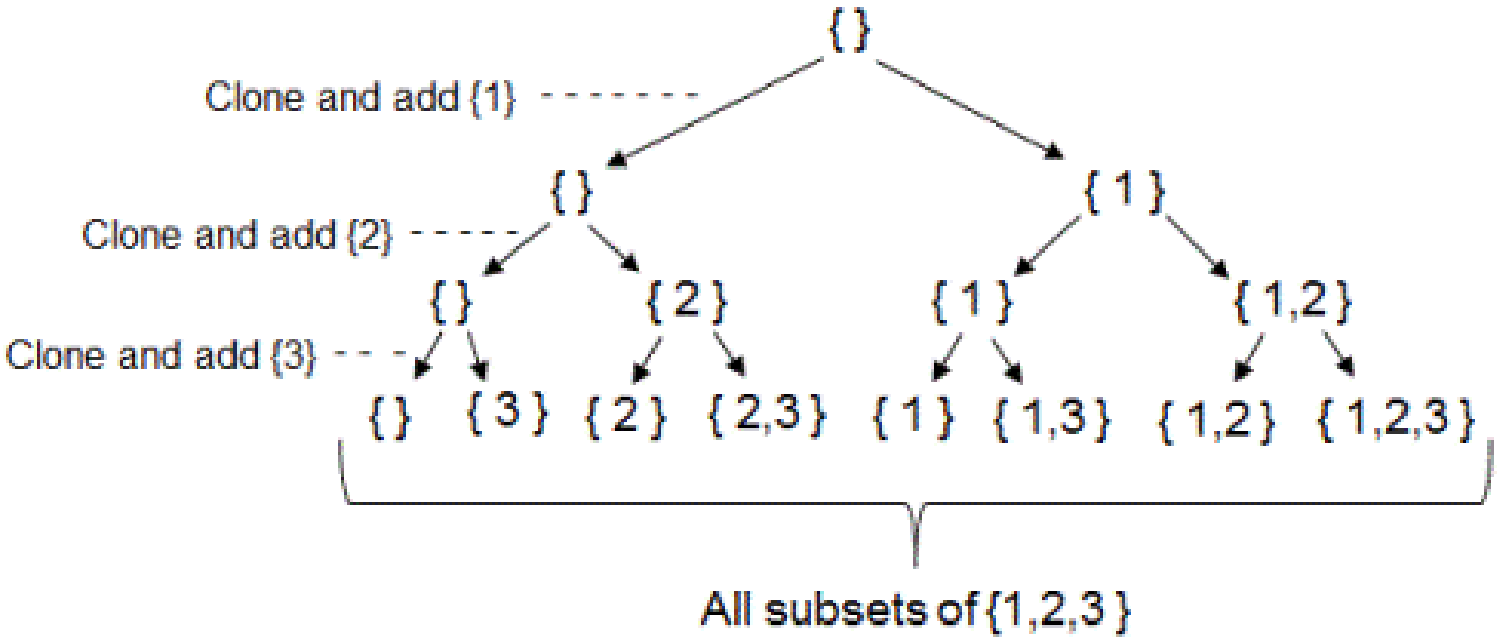
n → $\frac{n(n-1)}{2}$

$$\frac{n^2+n+n^2-n}{2} = \frac{2n^2}{2} = n^2$$

$$T(n) = n^2 + 2n + 1 = O(n^2)$$

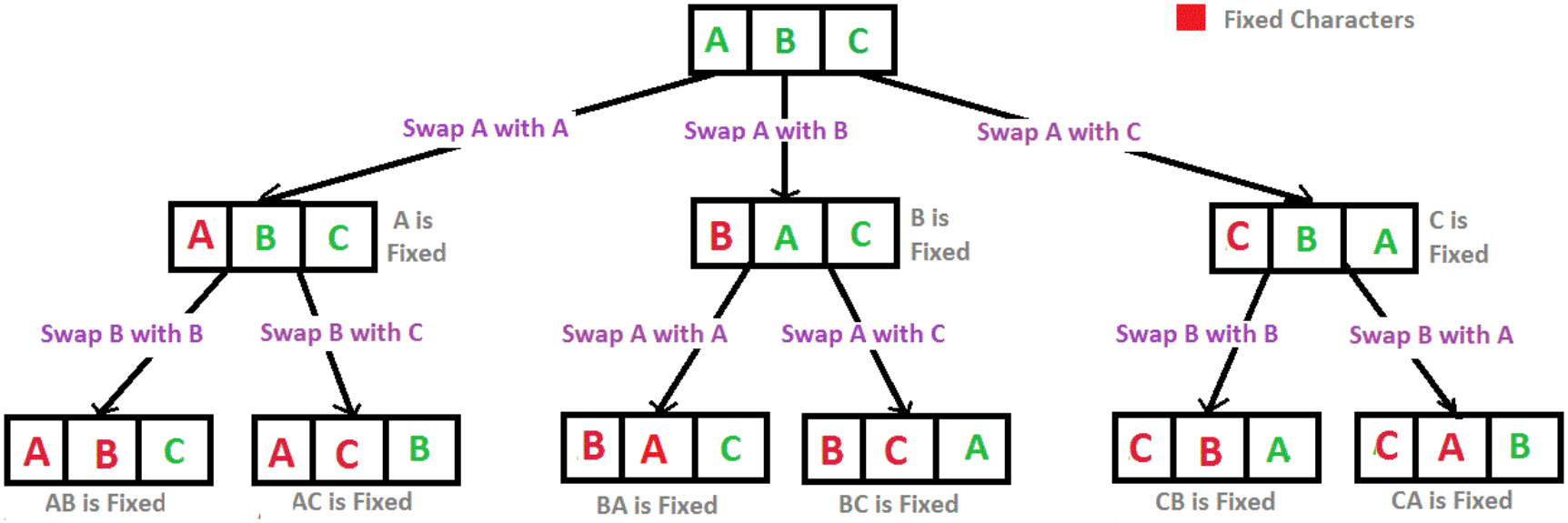
| i | j | print(i) |
|---|---|---|
| 0 | 1 iteração | - |
| 1 | 2 iterações | 1 |
| 2 | 3 iterações | 22 |
| 3 | 4 iterações | 333 |
| 4 | 5 iterações | 4444 |
| | PA=1+2+3+...+n=$\frac{n(n+1)}{2}$ | PA=1+2+3+...+n-1=$\frac{n(n-1)}{2}$ |

# Algoritmo Exponencial $O(2^n)$




**Problema:** todos os subconjuntos de um conjunto.
**Importante:** $2^3 \Rightarrow 8$ subconjuntos. Para 4 caracteres, seriam $2^4 \Rightarrow 16$ subconjuntos.

# Algoritmo Fatorial O(n!)




**Recursion Tree for Permutations of String "ABC"**

**Problema:** permutações (todas combinações) de dígitos.
**Cálculo:** 3! ⇒ 6 combinações. Para 4 letras, seriam 4! ⇒ 24 combinações.

# Outras Classes O(log(n)²) – Polilogaritmo

```
def imprimir_log_n(n: int) -> None:
    i: int = 1                          1
    while i < n:                        log2(n) + 1
        j: int = 1                      log2(n)
        while j < n:                    log2(n).(log2(n) + 1) ⇒ log2(n)² + log2(n)
            print(j, end= ' ‘)          log2(n).log2(n) ⇒ log2(n)²
            j *= 2                      log2(n).log2(n) ⇒ log2(n)²
        i *= 2                          log2(n)
        print()                         log2(n)
```

$$T(n) \approx 3\log_2(n)^2 + 5\log_2(n) + 2 = O(\log(n)^2)$$

Observa-se que a classe não está listada nas classes padrões. Porém, destaca-se que está acima de O(log(n)) e abaixo de O(n). Ainda, observar a função teto no logaritmo.

Importante: cuidar para não confundir a exponenciação de logaritmos, pois o comportamento quadrático é sobre o logaritmo e não sobre o “n”.

# Outras Classes O($\sqrt{n}$) – Sublinear

```
def imprimir_raiz(n: int) -> None:
    i: int = 0                      1
    while i * i < n:                √n + 1
        print(i)                    √n
        i += 1                      √n
```

$$T(n) = 3\lceil\sqrt{n}\rceil + 2 = O(\sqrt{n})$$

Observa-se que a classe não está listada nas classes padrões. Porém, destaca-se que está acima de O(log(n)) e abaixo de O(n).

# Outras Classes O($\log\log(n)$) – Logaritmo duplo

```
def imprimir_logaritmo_duplo(n: int) -> None:
    p: int = 0
    i: int = 1
    while i < n:
        p += 1          log2(n) ⇒ p
        i *= 2
    i = 1               1
    while i < p:        log2(p) + 1 ⇒ log2(log2(n)) + 1
        print(i)        log2(p) ⇒ log2(log2(n))
        i *= 2          log2(p) ⇒ log2(log2(n))
```

$$T(n) = 3\lceil\log_2\lceil\log_2(n)\rceil\rceil + 2 = O(\text{loglog}(n))$$

Observa-se que a classe não está listada nas classes padrões. Porém, destaca-se que está acima de O(1) e abaixo de O(log(n)). Estamos considerando apenas a análise de complexidade do segundo “for”, destacado em amarelo.

# Resumo

```
for i in range(n):                  ⇒ O(n)

for i in range(0, n, 2):            ⇒ n / 2 ⇒ O(n)

for i in range(0, n, 10):           ⇒ n / 10 ⇒ O(n)

for i in reversed(range(n)):        ⇒ O(n)

while i < n:                        ⇒ log2(n) ⇒ O(log(n))
    i *= 2

while i < n:                        ⇒ log3(n) ⇒ O(log(n))
    i *= 3

while i >= 1:                       ⇒ log2(n) ⇒ O(log(n))
    i //= 2
```

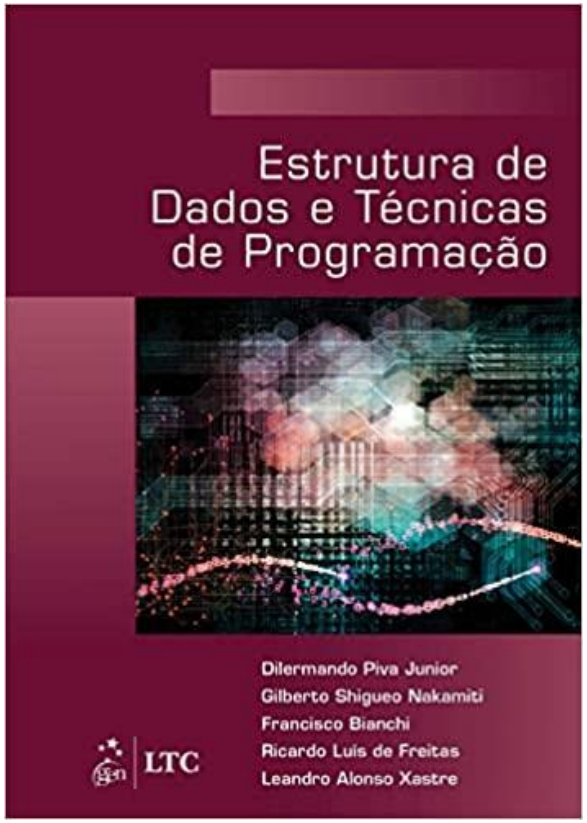
$$b^k = n \rightarrow \log_b n = k$$

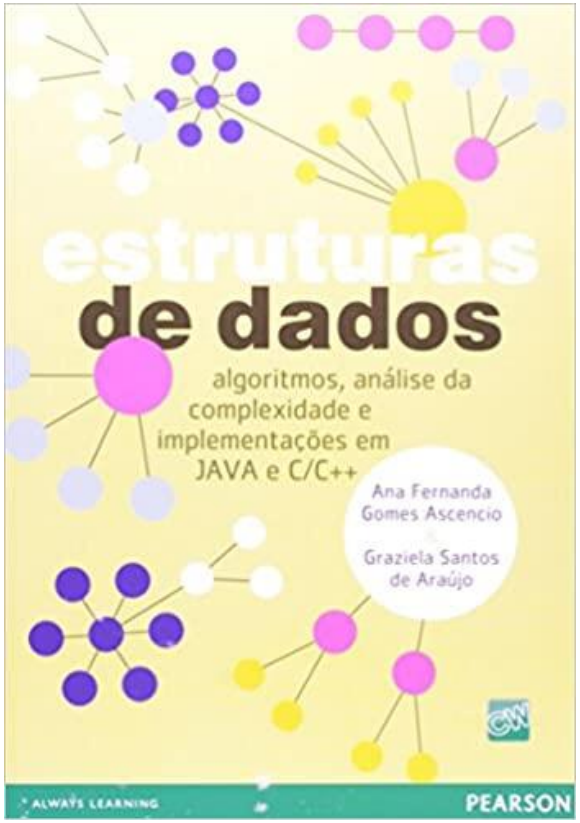
$$P.A = 1 + 2 + 3 + \cdots + n \rightarrow S_n = \frac{n(n+1)}{2}$$

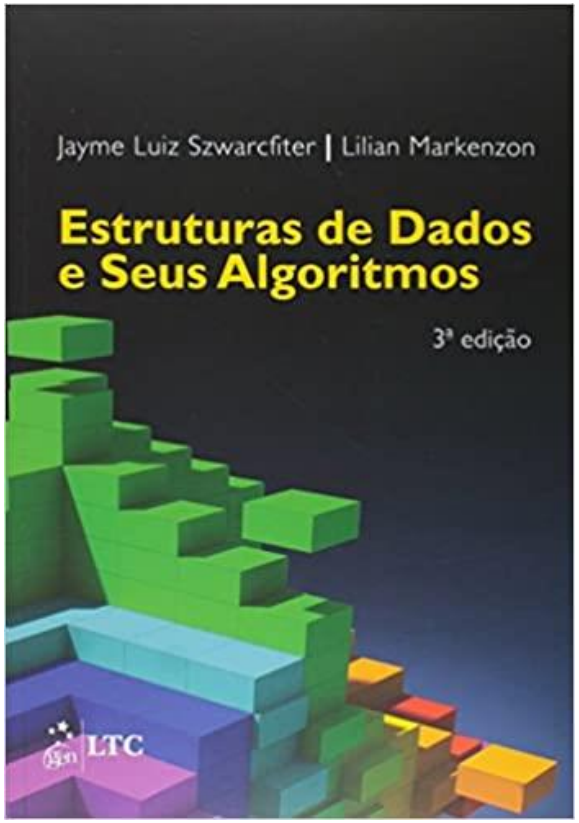
$$\lceil 3.5 \rceil = 4 \ \ e \ \lfloor 3.5 \rfloor = 3$$

$$\log {}^k n = \log(n)^k = (\log(n))^k$$

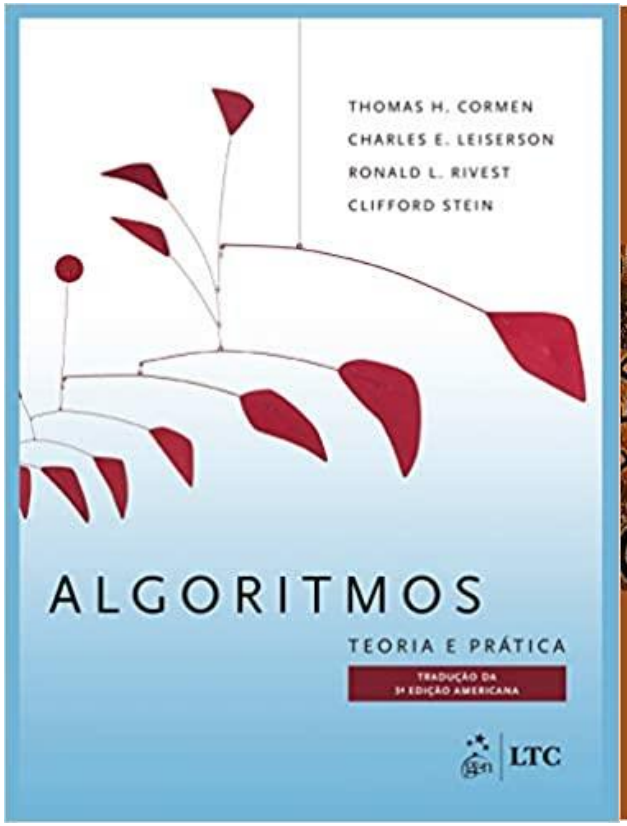
# Referências Bibliográficas

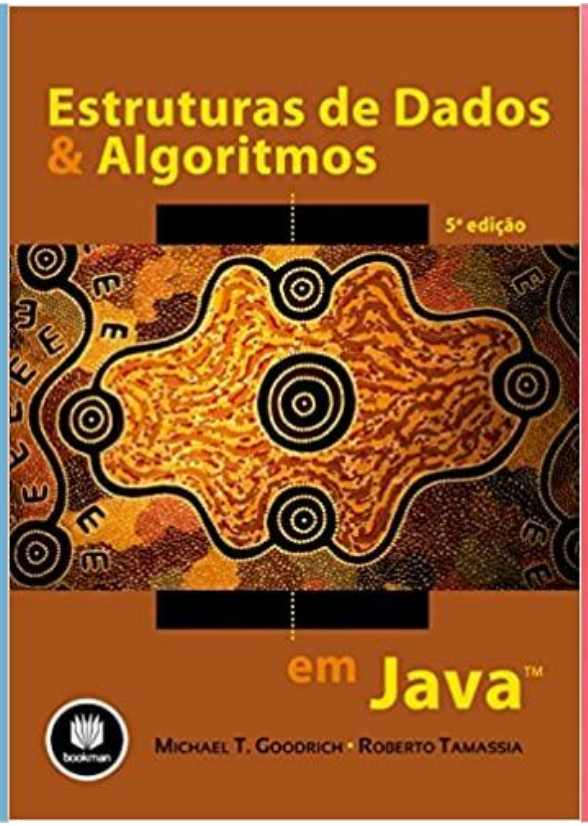


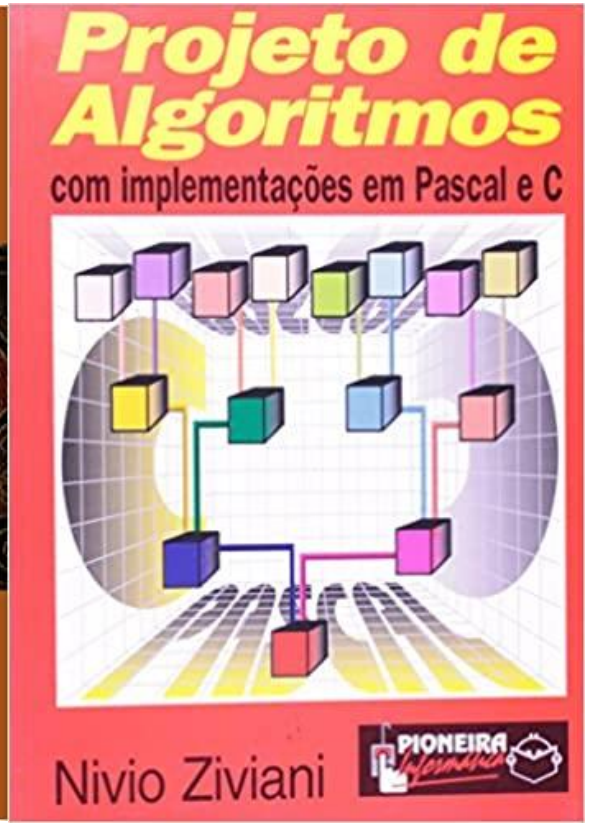



# Referências Bibliográficas




Abdul Bari.
https://www.youtube.com/channel/UCZCFT11CWBi3MHNlGf019nw