



Complexidade de Algoritmos

Professor: Elton Sarmanho¹

E-mail: eltonss@ufpa.br



¹Faculdade de Sistemas de Informação - UFPA/CUTINS

5 de junho de 2025

Roteiro

Análise Assintótica de Algoritmos
Objetivos

Tipos Abstratos de Dados
Conceitos Gerais



Roteiro

Crescimento de funções

- Conceitos Gerais

- Notação O

- Notação Ω

- Notação Θ

- Notação Assintótica

- Trabalho 1 - Complexidade de Algoritmos

Referências Bibliográficas



└ Análise Assintótica de Algoritmos

└ Objetivos

- ▶ Esta aula apresenta conceitos gerais sobre complexidade de algoritmos. Ao final, você deverá compreender os seguintes tópicos:
 - ▶ Entender e compreender a importância de medir desempenho do algoritmo.
 - ▶ Saber medir desempenho de um algoritmo através da Anotação assintótica.



└ Análise Assintótica de Algoritmos

└ Objetivos

- ▶ Esta aula apresenta conceitos gerais sobre complexidade de algoritmos:
 - ▶ Dados e TAD's
 - ▶ Crescimento de funções
 - ▶ Notações
 - ▶ Funções



└ Tipos Abstratos de Dados

└ Conceitos Gerais

- ▶ Programas possuem tipos de dados que são característicos de cada implementação
 - ▶ Preciso definir claramente os **tipos de dados**
- ▶ Tipo de dados é um conjunto de valores munido de um conjunto de operações
 - ▶ *int, float, char e etc*



- ▶ Um **Tipo Abstrato de Dados** (TAD) especifica um comportamento definido pelo usuário em termos de suas propriedades abstratas:
 - ▶ Descreve o **comportamento** de um objeto que independe da sua implementação e linguagem de programação, unicamente através dessas propriedades abstratas



Características do TAD

- ▶ Define o comportamento de um tipo de dado sem se preocupar com sua implementação.
- ▶ Necessário uma representação concreta:
 - ▶ Como TAD é implementado
 - ▶ como seus dados são manipulados com suas operações
- ▶ Base da POO



Implementação do TAD

- ▶ Por exemplo, pode-se definir um TAD chamado **pilha**.
- ▶ Nele os operadores seriam **inserção** e **remoção** da pilha, ocorrendo no topo da estrutura.
- ▶ Os dados seriam elementos da pilha
- ▶ A implementação é dependente das estruturas disponíveis na linguagem utilizada e de opções de modelagem (estruturas estáticas ou dinâmicas)



Implementação do TAD

- ▶ Considere uma aplicação que utilize uma lista de inteiros. Poderíamos definir TAD Lista, com as seguintes operações
 - ▶ Faça lista vazia
 - ▶ obtenha o primeiro elemento da lista; se a lista estiver vazia, então retorne nulo;
 - ▶ insira um elemento na lista.
- ▶ Há várias opções de estruturas de dados que permitem uma implementação eficiente para listas (por ex., o tipo estruturado arranjo).



Implementação do TAD

- ▶ Cada operação do tipo abstrato de dados é implementada como um procedimento na linguagem de programação escolhida
- ▶ Qualquer alteração na implementação do TAD fica restrita à parte encapsulada, sem causar impactos em outras partes do código.
- ▶ Cada conjunto diferente de operações define um TAD diferente, mesmo atuando sob um mesmo modelo matemático.
- ▶ A escolha adequada de uma implementação depende fortemente das operações a serem realizadas sobre o modelo.



- ▶ A complexidade de algoritmos é fundamental para projetar algoritmos eficientes.
- ▶ Caracteriza a **complexidade de tempo** em função do tamanho da entrada (n)
- ▶ um algoritmo assintoticamente mais eficiente é a melhor escolha para todas as entradas, **exceto as de tamanho pequeno**.
 - ▶ Conta-se o número de operações consideradas relevantes realizadas pelo algoritmo e expressa-se esse número como uma função de n .
 - ▶ Essas operações podem ser comparações, operações aritméticas, movimento de dados, etc
 - ▶ permite analisar a complexidade de um algoritmo independente do ambiente computacional utilizado



Pior caso, melhor caso, caso médio

- ▶ O número de operações realizadas por um determinado algoritmo pode depender da particular instância da entrada. Em geral interessa-nos o **pior caso**, i.e., o maior número de operações usadas para qualquer entrada de tamanho n .
- ▶ Análises também podem ser feitas para o **melhor caso** e o **caso médio**. Neste último, supõe-se conhecida uma certa distribuição da entrada.



Definição

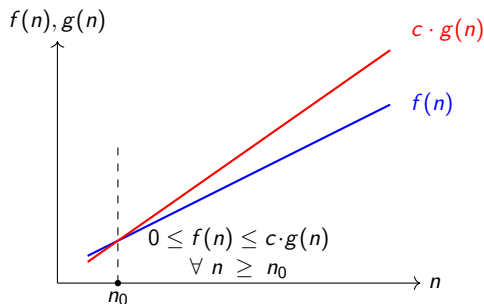
- ▶ Para uma dada função $g(n)$, denotamos $O(g(n))$ o conjunto de funções como:
 $O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq c \cdot g(n) \text{ para todo } n \geq n_0\}$
- ▶ Dizemos que $f(n) \in O(g(n))$, porém podemos adotar $f(n) = O(g(n))$ (Abuso da notação de igualdade)
- ▶ Na maioria dos casos estamos interessados no limite superior, pois queremos saber no pior caso, qual a complexidade de tempo



└ Crescimento de funções

└ Notação O

- ▶ Para todos os valores de n à direita de n_0 , o valor de $f(n)$ reside em $c \cdot g(n)$ ou abaixo desse
- ▶ A função $g(n)$ estabelece um limite Assintótico superior para $f(n)$
- ▶ Exemplo: $an + b = O(n^2)$
 - ▶ Podemos pensar nessa equação como sendo $an + b \leq O(n^2)$ ou $an + b \in O(n^2)$
 - ▶ Pois a taxa de Crescimento linear é **menor ou igual** a taxa quadrática



- ▶ Como abuso de notação, vamos sempre escrever $f(n) = O(g(n))$ ao invés de $f(n) \in O(g(n))$
- ▶ Algoritmo 1: $f_1(n) = 2n^2 + 5n = O(n^2)$
Algoritmo 2: $f_2(n) = 500n + 400 = O(n)$
 - ▶ Um polinômio de grau d é de ordem $O(n^d)$. Como uma constante pode ser considerada como um polinômio de grau 0, logo dizemos que uma constante é $O(n^0) = O(1)$
 - ▶ Podemos descartar os termos de mais baixa ordem e coeficientes do termo de mais alta ordem



Exemplos

1. $n + 5$ é $O(n)$?



Exemplos

1. $n + 5$ é $O(n)$?

Vamos encontrar c e n_0



Exemplos

1. $n + 5$ é $O(n)$?

Vamos encontrar c e n_0

$$f(n) \leq c.n$$



Exemplos

1. $n + 5$ é $O(n)$?

Vamos encontrar c e n_0

$$f(n) \leq c.n$$

$$n + 5 \leq c.n$$



Exemplos

1. $n + 5$ é $O(n)$?

Vamos encontrar c e n_0

$$f(n) \leq c.n$$

$$n + 5 \leq c.n$$

$$5 \leq c.n - n$$



Exemplos

1. $n + 5$ é $O(n)$?

Vamos encontrar c e n_0

$$f(n) \leq c.n$$

$$n + 5 \leq c.n$$

$$5 \leq c.n - n$$

$$c.n - n \geq 5$$



Exemplos

1. $n + 5$ é $O(n)$?

Vamos encontrar c e n_0

$$f(n) \leq c.n$$

$$n + 5 \leq c.n$$

$$5 \leq c.n - n$$

$$c.n - n \geq 5$$

$$n(c - 1) \geq 5$$



Exemplos

1. $n + 5$ é $O(n)$?

Vamos encontrar c e n_0

$$f(n) \leq c.n$$

$$n + 5 \leq c.n$$

$$5 \leq c.n - n$$

$$c.n - n \geq 5$$

$$n(c - 1) \geq 5$$

$$n \geq \frac{5}{c - 1}$$



Exemplos

1. $n + 5$ é $O(n)$?

Vamos encontrar c e n_0

$$f(n) \leq c.n$$

$$n + 5 \leq c.n$$

$$5 \leq c.n - n$$

$$c.n - n \geq 5$$

$$n(c - 1) \geq 5$$

$$n \geq \frac{5}{c - 1}$$

$$c = 2 \text{ e } n_0 = 5$$



Exemplos - Aluno

2. $2n + 10$ é $O(n)$ Qual valor de c e n_0 ?



Definição

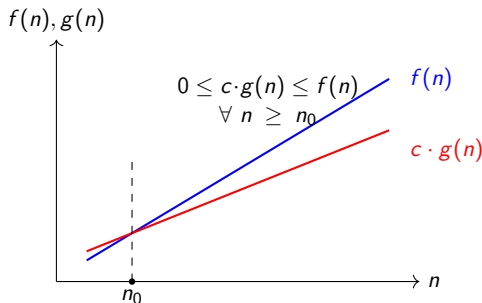
- ▶ Para uma dada função $g(n)$, denotamos $\Omega(g(n))$ o conjunto de funções como:
$$\Omega(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq c \cdot g(n) \leq f(n) \text{ para todo } n \geq n_0\}$$
- ▶ uma função $f(n)$ pertence ao conjunto $\Omega(g(n))$ se existe uma constante positiva c de forma que ela possa estar limitada por $c \cdot g(n)$ para um valor de n suficientemente grande.
- ▶ Dizemos que $f(n) \in \Omega(g(n))$, porém podemos adotar $f(n) = \Omega(g(n))$ (Abuso da notação de igualdade)



└ Crescimento de funções

└ Notação Ω

- ▶ Para todos os valores de n à direita de n_0 , o valor de $f(n)$ reside em $c \cdot g(n)$ ou acima desse
- ▶ A função $g(n)$ estabelece um limite Assintótico inferior para $f(n)$
- ▶ Exemplo: $2n^2 + n = \Omega(n)$
 - ▶ Podemos pensar nessa equação como sendo $2n^2 + n \geq \Omega(n)$, ou seja, a taxa de Crescimento de $2n^2 + n$ é **maior ou igual** a taxa de n



Exemplo 2

- ▶ $6n^2 + n = \Omega(n)$
- ▶ $6n^2 + n \geq c.n$
- ▶ $6n + 1 \geq c$
- ▶ $6n \geq c - 1$
- ▶ $n \geq \frac{c - 1}{6}$
- ▶ Para $n_0 = 1$ temos de tomar $c = 7$



Definição

- ▶ Para uma dada função $g(n)$, denotamos $\Theta(g(n))$ o conjunto de funções como:

$\Theta(g(n)) = \{f(n) : \text{existem constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tais que } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ para todo } n \geq n_0\}$

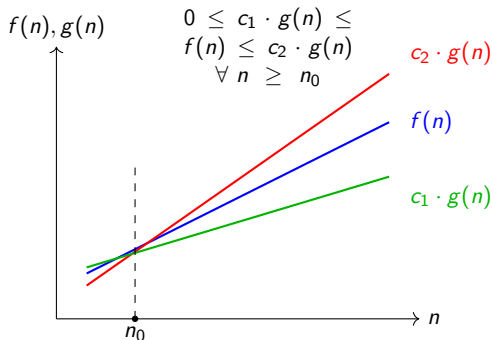
- ▶ uma função $f(n)$ pertence ao conjunto $\Theta(g(n))$ se existem constantes positiva c_1 e c_2 de forma que ela possa estar limitada por $c_1 \cdot g(n)$ e $c_2 \cdot g(n)$ para um valor de n suficientemente grande.



└ Crescimento de funções

└ Notação Θ

- ▶ Para todos os valores de n à direita de n_0 , o valor de $f(n)$ reside em $c_1 \cdot g(n)$ ou acima dele e $c_2 \cdot g(n)$ ou abaixo dele.
- ▶ A função $g(n)$ estabelece um **limite Assintótico restrito** para $f(n)$



limite superior

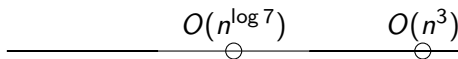
- ▶ Seja dado um problema, multiplicação de duas matrizes quadradas $n \times n$
- ▶ Conhecemos um algoritmo para resolver este problema (pelo método trivial) de complexidade $O(n^3)$
- ▶ Sabemos que a complexidade deste problema não deve superar $O(n^3)$, uma vez que existe um algoritmo que o resolve com essa complexidade.
- ▶ O **limite superior** (*upper bound*) deste problema é $O(n^3)$
 - ▶ **limite superior** de um problema pode mudar se alguém descobrir um outro algoritmo melhor

$$\underline{\quad\quad\quad} \bigcirc O(n^3)$$



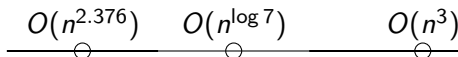
limite superior

- O algoritmo de Strassen reduziu a complexidade para $O(n^{\log 7})$. Então um novo limite superior é criado.



limite superior

- ▶ O algoritmo de Strassen reduziu a complexidade para $O(n^{\log 7})$. Então um novo limite superior é criado.
- ▶ Coppersmith e Winograd melhoraram ainda para $O(n^{2.376})$
- ▶ Note que limite superior de um problema depende do algoritmo. Pode diminuir quando aparecer um algoritmo melhor.

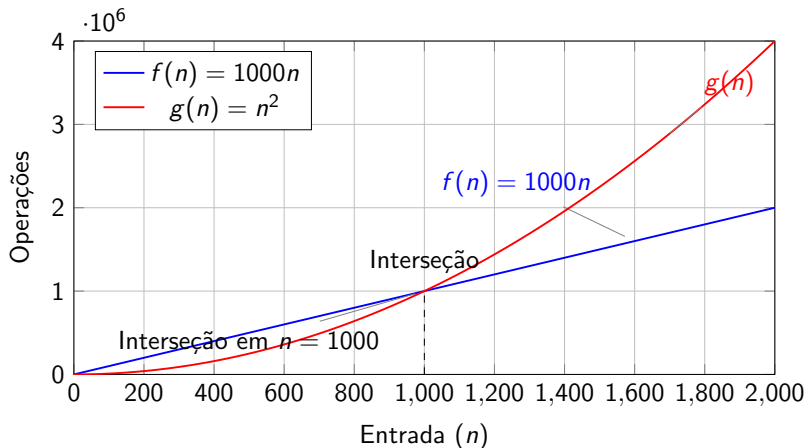


Análise do uso da notação assintótica

- ▶ Para dois algoritmos, considere as funções de eficiência:
 - ▶ $f(n) = 1000n$
 - ▶ $g(n) = n^2$
- ▶ f é maior do que g para valores pequenos de n
- ▶ g cresce mais rapidamente e podemos observar que no ponto onde $n = 1.000$ ocorre a mudança de comportamento $g > f$
- ▶ Conforme as notações vistas, se existe um n_0 a partir do qual $c \cdot f(n)$ é pelo menos tão grande quanto $g(n)$, então, desprezando os fatores constantes e considerando $n_0 = 1000$ e $c = 1$:
 - ▶ $f(n) = O(n^2)$
 - ▶ O que mostra que para $n \geq 1000$, n^2 é um limitante superior para $f(n)$



Análise do uso da notação assintótica



- └ Crescimento de funções

- └ Notação Assintótica

Análise do uso da notação assintótica: Funções

$T(n) = O(1)$: constante

$T(n) = O(\log \log n)$: super-rápido

$T(n) = O(\log n)$: logarítmico – muito bom

$T(n) = O(n)$: linear – toda a entrada é visitada

$T(n) = O(n \log n)$: limite de muitos problemas

$T(n) = O(n^2)$: quadrático

$T(n) = O(n^k)$: polinomial no tamanho da entrada

$T(n) = O(k^n), O(n!), O(n^n)$: exponencial – ruim!



Análise do uso da notação assintótica: Funções

Constante: ≈ 1

- ▶ Independente do tamanho de n , quantidade de operações executadas é uma quantidade fixa de vezes

Logarítmica: $\approx \log_b n$

- ▶ Típica de algoritmos que resolvem um problema transformando-o em problemas menores.

Linear: $\approx n$

- ▶ Uma certa quantidade de operações é processada sobre cada um dos elementos de entrada
- ▶ Situação ideal para quando é preciso processar entrada e obter n elementos de saída



Análise do uso da notação assintótica: Funções

Log Linear(ou n-log-n): $\approx n \cdot \log_b n$

- ▶ São algoritmos que resolvem um problema transformando-o em problemas menores, resolvem cada um de forma independente e depois agrupam as soluções.

Quadrática: $\approx n^2$

- ▶ ocorre quando os dados são processados aos pares, com laços de repetição aninhados
- ▶ São úteis para solucionar problemas de tamanho relativamente pequeno.

Exponencial e Fatorial: $\approx a^n$ e $\approx n!$

- ▶ Normalmente ocorre quando se usa uma solução de **força bruta**
- ▶ não são úteis do ponto de vista prático



Eficiência Assintótica

| 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> |



└ Crescimento de funções

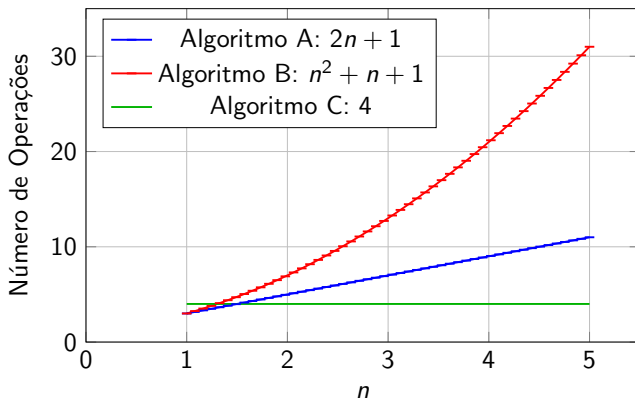
└ Notação Assintótica

Eficiência Assintótica

| | 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 |
| | $O(n)$ | $O(n^2)$ | $O(1)$ |



Eficiência Assintótica



Medida de Complexidade

- ▶ Sejam 5 algoritmos A_1 a A_5 para resolver um mesmo problema, de complexidades diferentes. (Supomos que uma operação leva 1 ms para ser efetuada.)
- ▶ $T_k(n)$ é a complexidade ou seja o número de operações que o algoritmo efetua para n entradas

| n | A_1 $T_1(n)=n$ | A_2 $T_2(n)=n \log n$ | A_3 $T_3(n)=n^2$ | A_4 $T_4(n)=n^3$ | A_5 $T_5(n)=2^n$ |
|-----|---------------------|----------------------------|-----------------------|-----------------------|-----------------------|
| 16 | 0.016s | 0.064s | 0.256s | 4s | 1m4s |
| 32 | 0.032s | 0.16s | 1s | 33s | 46 Dias |
| 512 | 0.512s | 9s | 4m22s | 1 Dia 13h | 10^{137} Séculos |

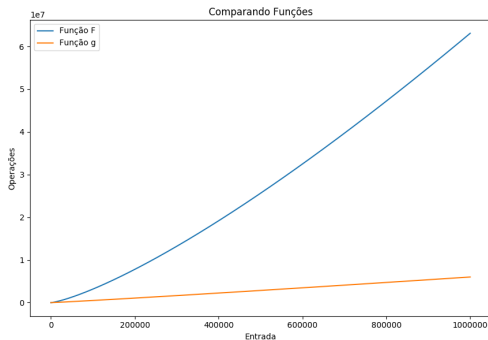


Exercício

- ▶ Um algoritmo tradicional e muito utilizado possui complexidade $f(n) = n^{1.3}$ enquanto um algoritmo novo proposto é da ordem de $g(n) = n \cdot \log n$
- ▶ Qual o melhor algoritmo ?



Exercício



Dicas de análise na prática

- ▶ Se $f(n)$ for um polinômio de grau d então $f(n)$ é $O(n^d)$
 - ▶ despreze os termos de menor ordem
 - ▶ despreze os fatores constantes
- ▶ Use a menor classe de funções possível
 - ▶ $3n$ é $O(n)$, ao invés de $3n$ é $O(3n)$
- ▶ Use a expressão mais simples
 - ▶ $3n + 5$ é $O(n)$, ao invés de $3n + 5$ é $O(3n)$



Analizando Algoritmo

Repetições : o tempo de execução é pelo menos o tempo dos comandos dentro da repetição multiplicada pelo número de vezes que é executada.

► Veja pseudo código abaixo é $O(n)$

$z = 1$

Para x de 1 Ate n Faça:

$z = z * x$



Analizando Algoritmo

Repetições aninhadas : O processo de análise é realizado do processo mais interno ao externo

- ▶ o tempo total é o tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições
- ▶ Veja o exemplo a seguir em que é $O(n^2)$

`matriz = [n,n]`

`Para linha de 1 Ate n Faca:`

`Para coluna de 1 Ate n Faca:`

`printa(matriz[linha,coluna])`



Analizando Algoritmo

Condições : o tempo nunca é maior do que o tempo do teste
entretanto considere **o tempo do maior** entre os
comandos dentro do bloco do “então” e do “senão”.

- Veja o exemplo a seguir em que é $O(n)$. Devido
segundo bloco ser maior Tempo de execução

Se $(x < y)$:

$$x = y + 1$$

Senao:

Para x de 1 Ate n Faça:

$$y = y + x * 2$$



Analizando Algoritmo

Chamadas à subrotinas : a **subrotina deve ser analisada primeiro** e depois ter suas unidades de tempo incorporadas ao programa que a chamou.



Analizando Algoritmo - Exercício

```
busca_linear(vetor, chave) {  
    i = 0;  
    enquanto(i < tamanho(vetor)) {  
        se(vetor[i] == chave)  
            retorna i;  
        ++i;  
    }  
    retorna -1;  
}
```



Analizando Algoritmo - Exercício

```
for(i = 0; i < n-1; ++i)
  for(j = i + 1; j < n; ++j)
    aux = a[i][j];
    a[i][j] = a[j][i];
    a[j][i] = aux;
```



Analizando Algoritmo - Exercício

```
for(i = 0; i < n-1; ++i)
  for(j = i + 1; j < n; ++j)
    aux = a[i][j];
    a[i][j] = a[j][i];
    a[j][i] = aux;
```



Analizando Algoritmo - Exercício - Aluno




```
01  inicio
02      i, j: inteiro
03      A: vetor inteiro de n posicoes
04      i = 1
05
06      enquanto (i < n) faca
07          A[i] = 0
08          i = i + 1
09
10      para i = 1 ate n faca
11          para j = 1 ate n faca
12              A[i] = A[i] + (i*j)
13  fim
```



1. Implemente e calcule a complexidade dos algoritmos
 - a) Escreva um algoritmo para encontrar o maior elemento do Array de tamanho n .
 - b) Escreva um algoritmo para calcular exponenciação de x^y .
 - c) Escreva um algoritmo para verificar se palavra é palíndromo.






Referências I

-  Lee K.D., Hubbard S. (2015) Trees. In: Data Structures and Algorithms with Python. Undergraduate Topics in Computer Science. Springer, Cham. Retrieved from https://doi.org/10.1007/978-3-319-13072-9_6
-  Hubbard, J. (2007). Schaum's Outline of Data Structures with Java. Retrieved from <http://www.amazon.com/Schaums-Outline-Data-Structures-Java/dp/0071476989>
-  Cormen, T. H., Leiserson, C. E., & Stein, R. L. R. E. C. (2012). Algoritmos: teoria e prática. Retrieved from <https://books.google.com.br/books?id=6iA4LgEACAAJ>.



Referências II

-  Ascenio, Ana Fernanda Gomes. Estrutura de dados: Algoritmos, análise da complexidade e implementações em Java e C++. São Paulo: Pearson Prentice Hall, 2010.
-  Szwarcfiter, Jayme Luiz. Estruturas de dados e seus algoritmos / Jayme Luiz Szwarcfiter, Lilian Markenzon. 3.ed. [Reimpr.]. - Rio de Janeiro : LTC, 2015.
-  Capítulo 20: Árvores — documentação Aprenda Computação com Python 3.0 2009.1. Retrieved from https://mange.ifrn.edu.br/python/aprenda-com-py3/capitulo_20.html





Complexidade de Algoritmos

Professor: Elton Sarmanho¹

E-mail: eltonss@ufpa.br



¹Faculdade de Sistemas de Informação - UFPA/CUTINS

5 de junho de 2025