

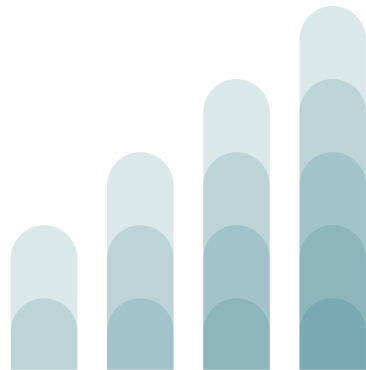
INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Campos do Jordão

ESTRUTURA DE DADOS

Análise de Algoritmos e Complexidade

Introdução e Conceitos.

Professor Mestre Igor de Moraes Sampaio
igor.sampaio@ifsp.edu.br





Análise de Algoritmos





Análise de Algoritmos

- A Análise de Algoritmos é um campo fundamental da Ciência da Computação.
- É o processo de estudar e prever o desempenho de um algoritmo antes ou depois de sua implementação, focando principalmente em tempo de execução e uso de memória.
- Busca determinar a eficiência de um algoritmo ao processar entradas de diferentes tamanhos.
- Seu principal objetivo é prever a escalabilidade e o impacto dos recursos computacionais.



Por que Analisar Algoritmos?

- Escolher a solução mais eficiente para um problema.
- Prever o comportamento em entradas grandes.
- Comparar diferentes abordagens.
- Identificar gargalos e otimizar o código.



Tipos de Complexidade

A complexidade é dividida em dois aspectos principais:

- **Complexidade de Tempo**
 - Mede a quantidade de operações executadas pelo algoritmo conforme o tamanho da entrada cresce.
- **Complexidade de Espaço**
 - Mede a quantidade de memória extra necessária para armazenar variáveis, pilhas de recursão e estruturas auxiliares.



Abordagens de Análise

- Análise teórica (matemática)
 - Modela o algoritmo como uma sequência de passos.
 - Conta o número de operações em função do tamanho da entrada n .
 - Usa notação assintótica para simplificar.
- Análise empírica (experimental)
 - Executa o algoritmo em diferentes tamanhos de entrada.
 - Mede o tempo e o uso de memória na prática.
 - Pode variar dependendo do hardware, compilador e linguagem.



Tipos de Análise

Tipo	Descrição	Exemplo
Pior caso	Tempo máximo que o algoritmo pode levar	Busca por um elemento inexistente na pesquisa sequencial
Melhor caso	Tempo mínimo possível	Encontrar o elemento logo na primeira comparação
Caso médio	Tempo esperado na maioria das vezes	Elemento no meio da lista



Complexidade





Complexidade

- A Complexidade de um algoritmo mede a quantidade de recursos necessários (tempo, memória, largura de banda...) em função do tamanho da entrada.
- Os dois tipos mais comuns:
 - Complexidade de tempo: quantas operações o algoritmo executa.
 - Complexidade de espaço: quanta memória o algoritmo usa.



Notações Assintóticas

- Quando analisamos um algoritmo, não nos preocupamos com o tempo exato (em segundos) ou o número exato de passos, mas sim como esse tempo cresce conforme o tamanho da entrada aumenta.
- Para isso usamos notações assintóticas, que representam o comportamento do algoritmo para valores grandes de n .



Notação Big-O - O

- Significa: limite superior do crescimento da função.
- Usada para: indicar o pior caso (ou tempo máximo que o algoritmo pode levar).
- Leitura: "A complexidade é no máximo dessa ordem de crescimento".
- Exemplo:
 - Pesquisa Sequencial $\rightarrow O(n)$: no pior caso, percorre todos os elementos.
 - Bubble Sort $\rightarrow O(n^2)$: no pior caso, faz $n \times n$ comparações.
- Analogia: é como dizer que "vou demorar no máximo 30 minutos para chegar", mesmo que às vezes demore menos.



Notação Ômega – Ω

- Significa: limite inferior do crescimento da função.
- Usada para: indicar o **melhor caso** (tempo mínimo que o algoritmo pode levar).
- Leitura: “O algoritmo vai levar pelo menos esse tempo”.
- Exemplo:
 - Pesquisa Sequencial $\rightarrow \Omega(1)$ no melhor caso (achou no primeiro elemento).
 - Bubble Sort $\rightarrow \Omega(n)$ no melhor caso (ainda percorre pelo menos uma vez a lista).
- Analogia: é como dizer “demoro no mínimo 10 minutos para chegar”, nunca menos que isso.



Notação Teta – Θ

- Significa: o crescimento exato (limite superior e inferior são iguais).
- Usada para: indicar o caso médio ou quando melhor e pior caso têm a mesma ordem.
- Leitura: “O tempo de execução cresce exatamente dessa ordem”.
- Exemplo:
 - Algoritmo que sempre percorre a lista exatamente uma vez $\rightarrow \Theta(n)$.
 - Merge Sort $\rightarrow \Theta(n \log n)$ no melhor, pior e caso médio.
- Analogia: é como dizer “sempre levo exatamente 20 minutos para chegar”.

- O pequeno – o
 - Significa: crescimento estritamente menor que a função indicada.
- Ω pequeno – ω
 - Significa: crescimento estritamente maior que a função indicada.



Resumo

Notação	Indica	Representa
$O(g(n))$	Limite superior	Pior caso
$\Omega(g(n))$	Limite inferior	Melhor caso
$\Theta(g(n))$	Limite exato	Crescimento igual nos casos
$o(g(n))$	Crescimento menor	Estritamente mais rápido
$\omega(g(n))$	Crescimento maior	Estritamente mais lento



A Importância da Notação Big-O na Complexidade

- O Big-O é uma garantia de desempenho máximo — diz:
 - “No pior cenário, este algoritmo não será mais lento que esta função.”
- Isso é útil porque:
 - Evita surpresas desagradáveis em entradas grandes.
 - Permite escolher algoritmos seguros para cenários críticos (como sistemas bancários ou tempo real).
- Exemplo: Pesquisa Sequencial $\rightarrow O(n)$
 - Mesmo que às vezes termine na primeira comparação, sabemos que nunca será pior que percorrer toda a lista.



O Crescimento com Entradas Grandes

- Quando um algoritmo é executado, o tempo exato depende de:
 - Velocidade do processador
 - Linguagem e compilador usados
 - Otimizações do sistema
 - Constantes e termos menores
- Esses fatores variam de máquina para máquina, mas a taxa de crescimento com o tamanho da entrada n é o que realmente define se um algoritmo vai escalar bem.
- Exemplo:
 - Algoritmo A: $5n + 3$ operações
 - Algoritmo B: $n^2 + 2$ operações
 - Para $n = 10$, A e B podem ter tempos parecidos.
 - Para $n = 1.000$, A executa ~5.000 operações e B executa ~1.000.002 → diferença absurda.
 - O Big-O ignora o “+3” e o “5x” e mostra apenas que A é $O(n)$ e B é $O(n^2)$.



Complexidade de Tempo

- Quantifica o número de operações básicas realizadas pelo algoritmo.

Notação	Nome	Exemplo típico
$O(1)$	Constante	Acesso direto em array
$O(\log n)$	Logarítmica	Busca binária
$O(n)$	Linear	Pesquisa sequencial
$O(n \log n)$	Linearítmica	Merge Sort, Quick Sort (médio caso)
$O(n^2)$	Quadrática	Bubble Sort, Insertion Sort
$O(2^n)$	Exponencial	Algoritmos de força bruta em conjuntos
$O(n!)$	Fatorial	Permutações completas



Constante — $O(1)$

```
// Executa sempre uma vez, independente de n
void funcConstante(int n) {
    int x = 0; // operação única
}
```



Logarítmica — $O(\log n)$

```
// Loop que cresce dividindo n pela metade a cada passo
void funcLogaritmica(int n) {
    for (int i = 1; i < n; i *= 2) {
        // operação constante
    }
}
```



Linear — $O(n)$

```
// Loop que percorre todos os elementos uma vez  
void funcLinear(int n) {  
    for (int i = 0; i < n; i++) {  
        // operação constante  
    }  
}
```



Linearítmica — $O(n \log n)$

```
// Loop externo linear e interno logarítmico
void funcLinearLog(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 1; j < n; j *= 2) {
            // operação constante
        }
    }
}
```



Quadrática — $O(n^2)$

```
// Dois loops aninhados lineares
void funcQuadratica(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            // operação constante
        }
    }
}
```



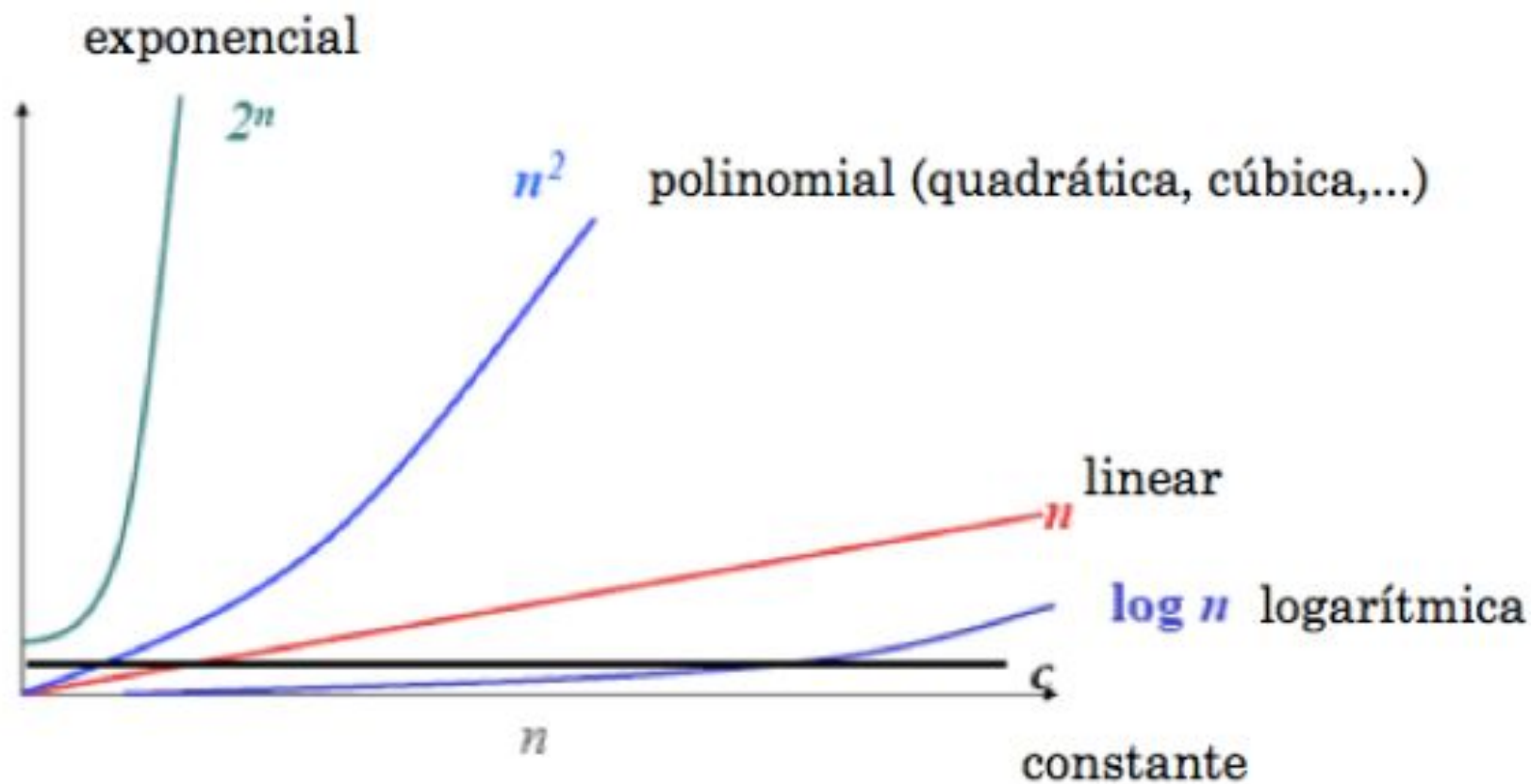
Exponencial — $O(2^n)$

```
// Função recursiva que chama duas vezes a si mesma
void funcExponencial(int n) {
    if (n <= 1) return;
    funcExponencial(n - 1);
    funcExponencial(n - 1);
}
```




Fatorial — $O(n!)$

```
// Exemplo: gera permutações recursivamente (estrutura simplificada)
void funcFatorial(int n, int depth = 0) {
    if (depth == n) return;
    for (int i = depth; i < n; i++) {
        // troca elementos e chama recursivamente
        funcFatorial(n, depth + 1);
        // desfaz troca
    }
}
```





Exemplo

- Algoritmo: Pesquisa Sequencial
- Entrada: Lista com n elementos
- Passos:
 - Melhor caso $\rightarrow O(1)$ (primeiro elemento já é o buscado)
 - Pior caso $\rightarrow O(n)$ (precisa verificar todos)
 - Caso médio $\rightarrow O(n)$ (em média percorre metade, mas em notação assintótica é $O(n)$)

Algoritmo	Melhor Caso	Médio Caso	Pior Caso
Busca Linear	$O(1)$	$O(n)$	$O(n)$
Busca Binária	$O(1)$	$O(\log n)$	$O(\log n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$





Complexidade de Espaço

- Mede a quantidade de memória usada. Pode incluir:
 - Memória fixa (variáveis globais, arrays de tamanho fixo).
 - Memória dinâmica (criada durante a execução).
 - Memória de recursão (chamadas empilhadas).
- Exemplo:
 - Um vetor de tamanho $n \rightarrow$ ocupa $O(n)$ espaço.
 - Um algoritmo que apenas troca valores usando uma variável auxiliar $\rightarrow O(1)$ de espaço.



Desafio





Desafio

Questionário: Questões sobre
Análise de Algoritmos e
Complexidade

Teste seu entendimento da aula.

Disponível no Classroom.