Prof.º
Alexandre Gomes

“Complexidade de algoritmos”

Complexidade de algoritmos

O que é Complexidade de Algoritmos?

A complexidade de um algoritmo mede quanto recurso computacional (tempo e espaço) ele consome para resolver um problema em função do tamanho da entrada.

Dois tipos principais:

- Complexidade de Tempo (Time Complexity): mede o número de operações.
- Complexidade de Espaço (Space Complexity): mede o uso de memória.

Notações Assintóticas (Big-O)

Big-O é uma “ferramenta” usada para medir a eficiência de algoritmos. Ele nos ajuda a entender como o tempo de execução ou o uso de memória de um algoritmo cresce à medida que a entrada aumenta.

Pense assim: se um programa roda rápido para 10 dados, ele continuará rápido para 1 milhão? O Big-O nos dá uma forma de responder essa pergunta.

Por que o Big O é importante?

Imagine que você está construindo um aplicativo que precisa buscar informaçõesem uma lista enorme. Se seu código não for eficiente, pode funcionar bem em testes pequenos, mas travar quando for para produção.

O Big-O ajuda a prever problemas antes que eles aconteçam, tornando seu código mais escalável e performático.

Notações (Big-O)

Notação	Nome	Exemplo de desempenho
O(1)	Constante	Acesso direto a vetor
O(log n)	Logarítmica	Busca binária
O(n)	Linear	Percorrer um array
O(n log n)	Linearítmica	Quicksort, Mergesort
$O(n^2)$	Quadrática	Bubble Sort, Selection Sort
$O(2^n)$	Exponencial	Problemas de combinação
O(n!)	Fatorial	Algoritmos de permutação

Tabela de Crescimento

n (entrada)	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(2^n)$
10	1	3	10	30	100	1.024
100	1	6	100	600	10.000	~1.27e30
1.000	1	10	1.000	10.000	1M	~1e301

Exemplo com JavaScript

O(1) – Tempo Constante

```
function getFirstElement(arr) {
  return arr[0];
}
```

*Sempre executa o mesmo número de operações, não importa o valor de [n].

O(log n) – Logarítmica

```
function buscaBinaria( vetor, valorBusca ){
    let ini = 0
    let fim = vetor.length - 1
    while(fim >= ini){
        let meio =  Math.floor((ini + fim) / 2)
        if( valorBusca === vetor[meio]){
            return meio
        }
        else if(valorBusca > vetor[meio]){
            ini = meio + 1
        }else{
            fim = meio - 1
        }
    }
    return -1
}
```

*Divide o problema pela metade a cada passo.

O(n) – Linear

```
function buscaSequencial(vetor, valorBusca){
    for(let i=0; i < vetor.length; i++ ){
        if(vetor[i] === valorBusca) return i
    }
    return -1
}
```

*Percorre todos os elementos da entrada uma vez.

O(n log n) – Linearítmica

*Divide e conquista com chamadas recursivas.

```
function mergeSort(vetor){
    if(vetor.length < 2) return vetor
    let meio = Math.floor(vetor.length / 2)
    let vetEsq = vetor.slice(0, meio)
    let vetDir = vetor.slice(meio)
    div++
    vetEsq = mergeSort(vetEsq)
    vetDir = mergeSort(vetDir)
    let posEsq = 0, posDir = 0, vetRes = []
    while(posEsq < vetEsq.length && posDir < vetDir.length){
        if(vetEsq[posEsq] < vetDir[posDir]){
            vetRes.push(vetEsq[posEsq])
            posEsq++
        }else{
            vetRes.push(vetDir[posDir])
            posDir++
        }
    }
    let sobra
    if(posEsq < posDir){
        sobra = vetEsq.slice(posEsq)
    }
    else{
    sobra = vetDir.slice(posDir)
    }
    jun++
    return[...vetRes,...sobra]
}
```

O(n^2) –Quadrático

```
function selectionSort(vetor){
    for(let posSel = 0; posSel < vetor.length - 1; posSel++){
        let posMenor = posSel + 1
        for(let i = posMenor + 1; i < vetor.length; i++){
            if(vetor[posMenor] > vetor[i] ){
                posMenor = i
            }
        }
        if(vetor[posSel] > vetor[posMenor]){
            [ vetor[posSel], vetor[posMenor] ] = [ vetor[posMenor], vetor[posSel] ]
        }
    }
}
```

*Loops aninhados sobre a mesma entrada.

O(2ⁿ) – Exponencial

```
function fibonacci(n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

*Cada chamada gera duas novas chamadas.

O(n!) – Fatorial

```
function permutacoes(arr) {
    if (arr.length <= 1) return [arr];
    let resultado = [];
    for (let i = 0; i < arr.length; i++) {
      let fixo = arr[i];
      let restantes = arr.slice(0, i).concat(arr.slice(i + 1));
      for (let p of permutacoes(restantes)) {
        resultado.push([fixo, ...p]);
      }
    }
    return resultado;
  }
```

*Explora todas as permutações possíveis.

(Analogia: Todas as ordens possíveis para livros numa prateleira.)

Análise de Casos: Melhor, Médio e Pior

- **Melhor caso:** quando o algoritmo encontra uma solução rapidamente.

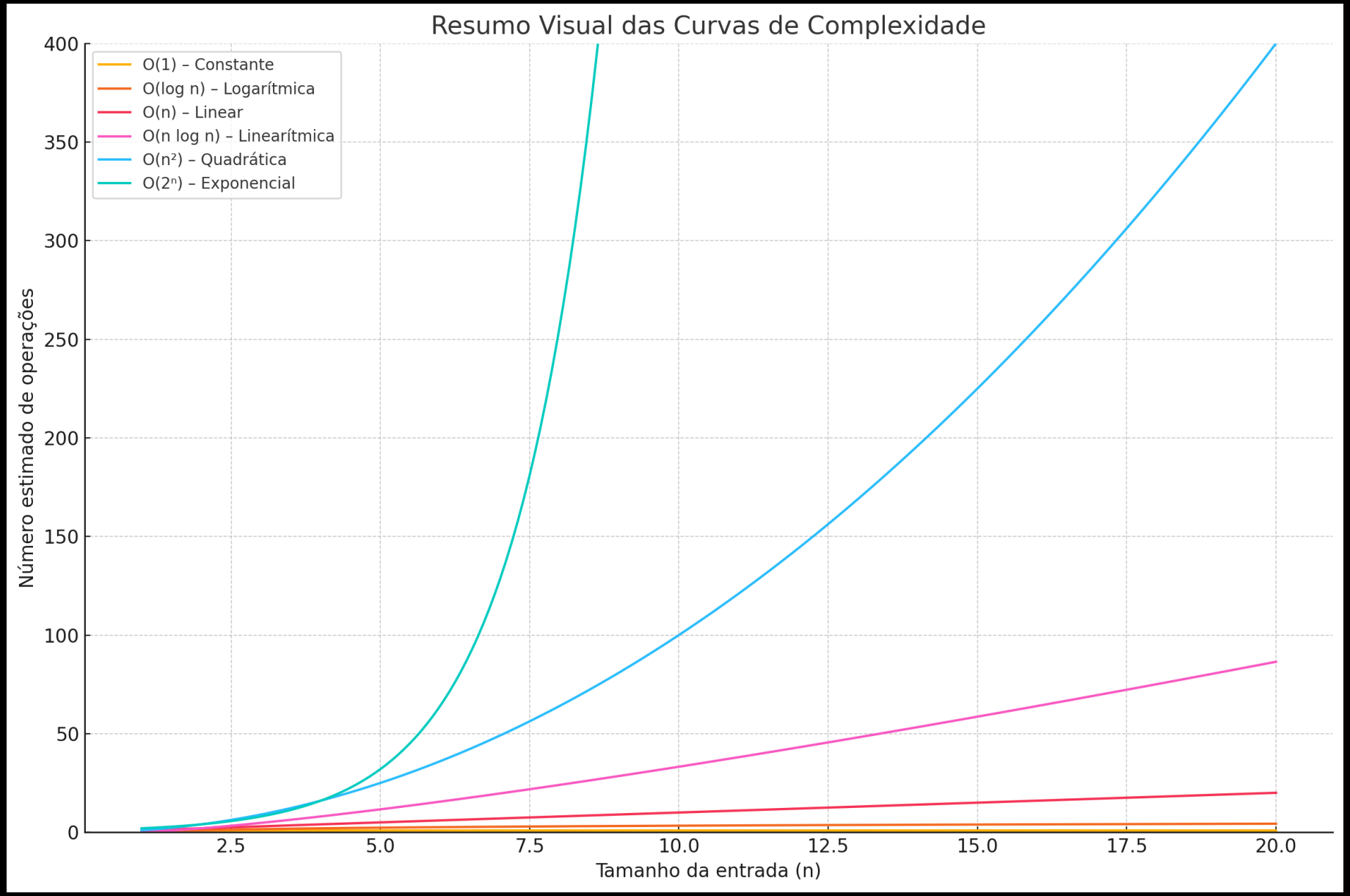
- **Caso médio:** comportamento esperado em geral.

- **Pior caso:** quando percorre todos os caminhos possíveis.

Objetivos da Análise Assintótica

- Saber qual algoritmo é mais eficiente quando o volume de dados cresce.
- Estimar o desempenho relativo.
- Escolher a estrutura de dados adequada.

Resumo visual das curvas de complexidade



Quando usar?

Tipo de crescimento	Quando usar
$O(1)$, $O(\log n)$	Sempre que possível (super eficientes)
$O(n)$, $O(n \log n)$	Aceitável mesmo com milhões de elementos
$O(n^2)$	Somente para poucos elementos (< 1000)
$O(2^n)$, $O(n!)$	**Evite!** Só em casos pequenos e inevitáveis