

# **Complexidade de Algoritmos**

**Kaio Christaldo**  
**Fabricio Matsunaga**

# Introdução à Complexidade de Algoritmos

- **Complexidade de um algoritmo:** o quanto o tempo de execução e o uso de memória de um algoritmo aumentam à medida que o tamanho da entrada cresce.
- **Importância de analisar a complexidade:** isso ajuda a entender a eficiência do algoritmo em termos de desempenho, especialmente para entradas grandes.

# Notações de Complexidade

## 1. Notação Big-O ( $O$ ) – Pior caso

- a. Indica o tempo máximo que um algoritmo pode levar.
- b. Mostra como o algoritmo escala com o aumento da entrada.
- c. É a notação mais usada em competições, entrevistas e prática.

## 2. Notação Ômega ( $\Omega$ ) – Melhor caso

- a. Representa o tempo mínimo que o algoritmo pode levar.
- b. Ex: Quando um algoritmo termina logo na primeira tentativa.

## 3. Notação Teta ( $\Theta$ ) – Caso médio

- a. Usada quando o pior e o melhor caso têm a mesma ordem.
- b. Indica que o tempo de execução é sempre proporcional a  $f(n)$ .

# Notações de Complexidade

## Exemplo de Algoritmos de Busca

```
1  int buscaBinaria(vector<int>& v, int alvo) {  
2      int inicio = 0;  
3      int fim = v.size() - 1;  
4  
5      while (inicio <= fim) {  
6          int meio = inicio + (fim - inicio) / 2;  
7  
8          if (v[meio] == alvo)  
9              return meio;  
10         else if (v[meio] < alvo)  
11             inicio = meio + 1;  
12         else  
13             fim = meio - 1;  
14     }  
15  
16     return -1;  
17 }
```

```
1  int buscaLinear(vector<int>& v, int alvo) {  
2      for (int i = 0; i < v.size(); i++) {  
3          if (v[i] == alvo)  
4              return i;  
5      }  
6      return -1;  
7  }  
8
```

# Notações de Complexidade

## Busca Binária

```
1  int buscaBinaria(vector<int>& v, int alvo) {
2      int inicio = 0;
3      int fim = v.size() - 1;
4
5      while (inicio <= fim) {
6          int meio = inicio + (fim - inicio) / 2;
7
8          if (v[meio] == alvo)
9              return meio;
10         else if (v[meio] < alvo)
11             inicio = meio + 1;
12         else
13             fim = meio - 1;
14     }
15
16     return -1;
17 }
```

## Complexidade:

- **Melhor caso:**  $\Omega(1)$  (se o elemento está no meio)
- **Pior caso:**  $O(\log n)$
- **Vantagem:** Vetor Ordenado

# Notações de Complexidade

## Busca Linear ou Sequencial

```
1  int buscaLinear(vector<int>& v, int alvo) {  
2      for (int i = 0; i < v.size(); i++) {  
3          if (v[i] == alvo)  
4              return i;  
5      }  
6      return -1;  
7  }  
8
```

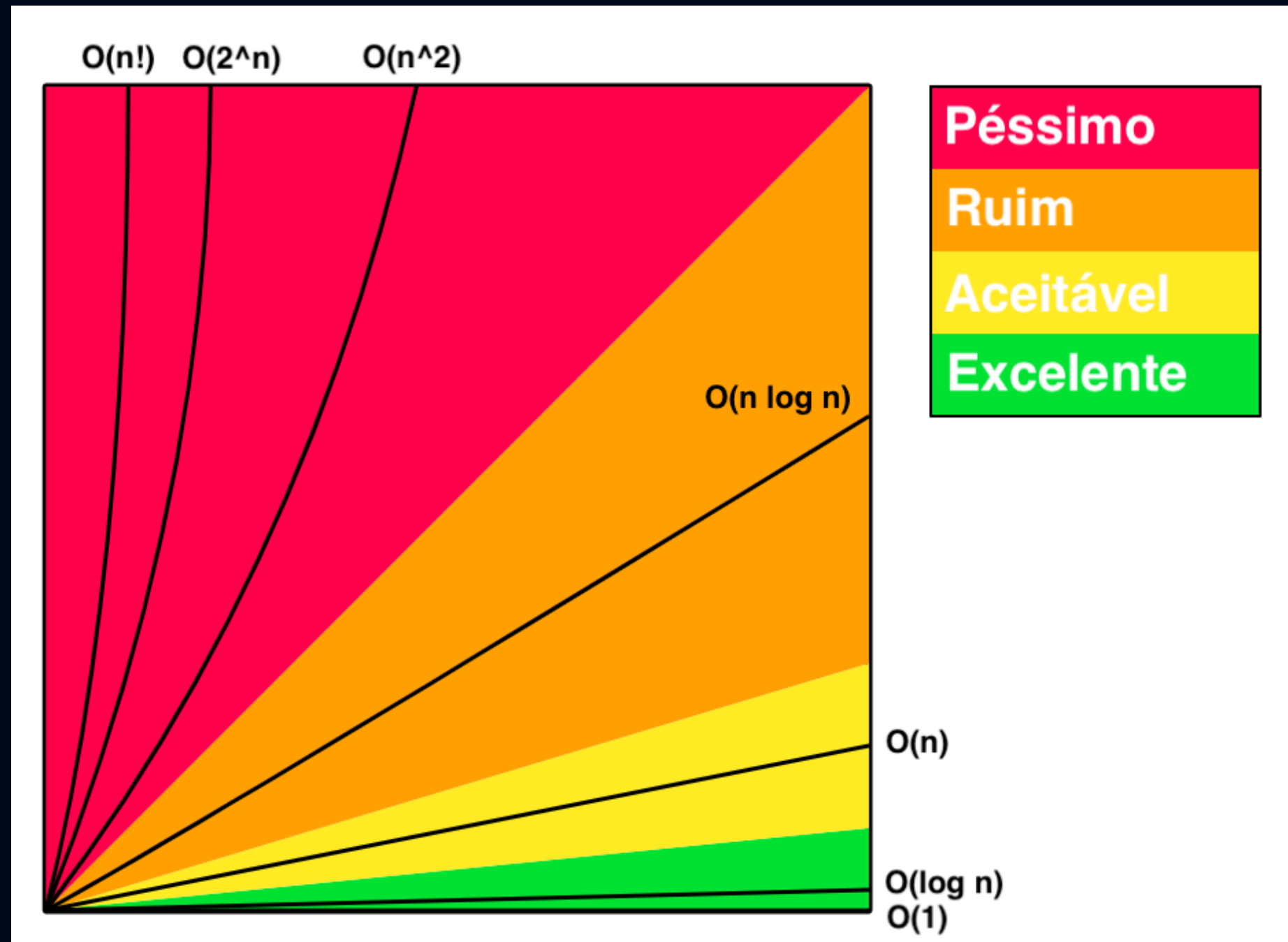
### Complexidade:

- **Melhor caso:**  $\Omega(1)$  (se o elemento está no começo)
- **Pior caso:**  $O(n)$
- **Útil:** Vetor não ordenado

# Tabela de Complexidade – Piores Casos

Notação	Nome	Exemplo de algoritmo
$O(1)$	Constante	Acesso direto em array
$O(\log n)$	Logarítmica	Busca binária
$O(n)$	Linear	Percorrer vetor
$O(n \log n)$	Linearítmica	Merge Sort, Quick Sort
$O(n^2)$	Quadrática	Bubble Sort, seleção, inserção
$O(2^n)$	Exponencial	Algoritmo de força bruta
$O(n!)$	Fatorial	Problema do Caixeiro Viajante

# Tabela de Complexidade – Piores Casos





# Dicas para estimar a complexidade esperada

## 1. Olhe para os limites de entrada (n)

- Geralmente, o valor de **n** já te dá uma boa ideia da complexidade máxima permitida.

Complexidade	n máximo seguro em ~1 segundo
$O(1)$	Qualquer valor
$O(\log n)$	Até $10^9$ ou mais
$O(n)$	Até $10^7$ (com leitura rápida)
$O(n \log n)$	Até $10^5 \sim 2 \cdot 10^5$
$O(n^2)$	Até $10^3 \sim 3 \cdot 10^3$
$O(n^3)$	Até 500 ou menos
$O(2^n)$	Até $n = 20$
$O(n!)$	Até $n = 10$

# Dicas para estimar a complexidade esperada

## 2. Analise os loops

- Um **único** for até  $n$  –  $O(n)$
- **Dois** for aninhados –  $O(n^2)$
- Um **for** e dentro dele uma **busca binária** –  $O(n \log n)$
- Recursões com divisão pela metade (como Merge Sort) –  $O(n \log n)$

# Dicas para estimar a complexidade esperada

## 3. Fique atento à estrutura de dados usada

- **sort, set, map, priority\_queue** – geralmente  $O(\log n)$  por operação
- **unordered\_map, unordered\_set** –  $O(1)$  no caso médio
- **vector** com **push\_back** –  $O(1)$  amortizado

# Dicas para estimar a complexidade esperada


## 4. Estime quantas operações o algoritmo faz

- Se um problema envolve ordenação – pense em  $O(n \log n)$
- Se envolve combinações, subconjuntos, permutações – pode ser  $O(2^n)$  ou  $O(n!)$
- Se envolve "contar pares", "inversões", "distância mínima" – geralmente tem um  $O(n \log n)$  com merge sort ou segment tree

# Dicas para estimar a complexidade esperada

## 5. Atenção com entradas e saídas

- Se o problema tem muita entrada,



```
1 ios::sync_with_stdio(false); cin.tie(NULL); // inicio da main()
```

- Evite **endl**, prefira **\n** para evitar **TLE** só por causa do flush do output



```
1 cout << "\n"; // Melhor que cout << endl;
```



**Fim.**