# Relatório de Análise de Algoritmos de Ordenação

**Identificação**

* **Disciplina:** Resolução de Problemas Estruturados em Computação
* **Trabalho:** TDE 04 – Comparação de Algoritmos de Ordenação
* **Professor(a):** Marina de Lara
* **Acadêmico(s):** **Bernardo Plottegher e Arthur Bertoni Gurkewicz**
* **Data:** **14/11/2025**

Sumário

[Relatório de Análise de Algoritmos de Ordenação 1](#_Toc214005245)

[1. Introdução 3](#_Toc214005246)

[2. Metodologia 4](#_Toc214005247)

[2.1 Conjuntos de dados 4](#_Toc214005248)

[2.2 Implementação dos algoritmos 4](#_Toc214005249)

[2.3 Medição do tempo 4](#_Toc214005250)

[3. Resultados 5](#_Toc214005251)

[3.1 Tabela de tempos de execução 5](#_Toc214005252)

[3.2 Gráficos comparativos 6](#_Toc214005253)

[4.Análise dos resultados 7](#_Toc214005254)

[4.1 Dados aleatórios 7](#_Toc214005255)

[4.2 Dados em ordem crescente 7](#_Toc214005256)

[4.3 Dados em ordem decrescente 7](#_Toc214005257)

[5. Conclusão 9](#_Toc214005258)

## 1. Introdução

A ordenação de dados é uma operação fundamental em computação, estando presente em diversas etapas de processamento de informação, como busca, análise de dados e organização de estruturas em memória. Neste trabalho, foram implementados e comparados três algoritmos de ordenação clássicos: Bubble Sort, Insertion Sort e Quick Sort.

O objetivo é analisar o desempenho desses algoritmos em diferentes cenários de entrada, variando tanto o tamanho dos vetores quanto o tipo de ordenação inicial (aleatória, crescente e decrescente). A partir da medição de tempo de execução, busca‑se compreender em quais situações cada algoritmo apresenta melhor (ou pior) desempenho e relacionar esses resultados com a complexidade teórica de cada método.

## 2. Metodologia

### 2.1 Conjuntos de dados

Foram utilizados os arquivos .csv disponibilizados no enunciado do TDE, contendo vetores de inteiros com diferentes tamanhos e ordens iniciais:

* **Aleatório:** aleatorio\_100.csv, aleatorio\_1000.csv, aleatorio\_10000.csv
* **Crescente:** crescente\_100.csv, crescente\_1000.csv, crescente\_10000.csv
* **Decrescente:** decrescente\_100.csv, decrescente\_1000.csv, decrescente\_10000.csv

Cada arquivo representa um vetor de tamanho 100, 1.000 ou 10.000 elementos, organizado conforme o tipo de ordenação inicial.

### 2.2 Implementação dos algoritmos

Os três algoritmos foram implementados em linguagem **[preencher: Java/Python/etc.]**, seguindo as versões clássicas:

* **Bubble Sort:** percorre o vetor diversas vezes, comparando pares adjacentes e trocando‑os se estiverem fora de ordem.
* **Insertion Sort:** constrói a ordenação final inserindo, a cada passo, o elemento atual na posição correta em relação aos anteriores.
* **Quick Sort:** algoritmo recursivo que escolhe um pivô, particiona o vetor em elementos menores e maiores que o pivô e, em seguida, ordena recursivamente as partições.

Cada algoritmo recebe uma cópia independente do vetor original, garantindo que todos processem exatamente os mesmos dados de entrada.

### 2.3 Medição do tempo

Para medir o tempo de execução de cada algoritmo em cada arquivo, foi utilizada a função:

* System.nanoTime() (ou função equivalente, caso outra linguagem tenha sido usada),

registrando o tempo imediatamente antes do início da ordenação e após a conclusão. A diferença entre esses valores, convertida para milissegundos (ms), foi registrada para compor as tabelas de resultados.

## 3. Resultados

### 3.1 Tabela de tempos de execução

A Tabela 1 apresenta os tempos medidos para cada combinação de conjunto de dados e algoritmo de ordenação.

**Tabela 1 – Tempos de execução (em milissegundos)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Conjunto de dados** | **Bubble Sort (ms)** | **Insertion Sort (ms)** | **Quick Sort (ms)** |
| aleatorio\_100 | 0,89 | 0,05 | 0,02 |
| aleatorio\_1000 | 5,91 | 2,38 | 0,33 |
| aleatorio\_10000 | 102,48 | 33,21 | 0,91 |
| crescente\_100 | 0,01 | 0,03 | 0,02 |
| crescente\_1000 | 0,20 | 0,03 | 0,55 |
| crescente\_10000 | 22,80 | 0,07 | 65,76 |
| decrescente\_100 | 0,01 | 0,00 | 0,01 |
| decrescente\_1000 | 0,58 | 0,15 | 0,39 |
| decrescente\_10000 | 59,63 | 14,21 | 38,03 |

### 3.2 Gráficos comparativos

Abaixo estão os gráficos comparativos sugeridos, baseados nos dados da Tabela 1.

Gráfico 1: Tempo de execução × Tamanho do vetor (Dados Aleatórios)

Comparação do desempenho dos três algoritmos para vetores de 100, 1.000 e 10.000 elementos aleatórios.

Gráfico 2: Tempo de execução × Tamanho do vetor (Dados Crescentes)

Comparação do desempenho dos três algoritmos para vetores de 100, 1.000 e 10.000 elementos em ordem crescente.

Gráfico 3: Tempo de execução × Tamanho do vetor (Dados Decrescentes)

Comparação do desempenho dos três algoritmos para vetores de 100, 1.000 e 10.000 elementos em ordem decrescente.

## 4.Análise dos resultados

Nesta seção, são discutidos os principais comportamentos observados a partir da Tabela 1 e dos gráficos.

### 4.1 Dados aleatórios

Nos conjuntos aleatórios, observa‑se que:

* O Quick Sort apresenta o menor tempo de execução em todos os tamanhos, mantendo valores bem inferiores aos demais algoritmos.
* O Insertion Sort é consistentemente mais rápido que o Bubble Sort, mas ainda apresenta tempos significativamente maiores que o Quick Sort à medida que o tamanho do vetor aumenta.
* O Bubble Sort demonstra crescimento acentuado do tempo de execução, especialmente para 10.000 elementos, o que é coerente com sua complexidade média de O(n²).

Esses resultados são compatíveis com a complexidade típica dos algoritmos: Quick Sort com complexidade média O(n log n), enquanto Bubble Sort e Insertion Sort apresentam O(n²) em dados sem ordenação prévia.

### 4.2 Dados em ordem crescente

Para vetores já ordenados de forma crescente:

* Bubble Sort e Insertion Sort apresentam tempos muito baixos, principalmente para 100 e 1.000 elementos.
* O Insertion Sort se beneficia bastante do fato de o vetor já estar ordenado, alcançando tempos praticamente constantes, o que é esperado, já que seu melhor caso é O(n).
* O Quick Sort, por outro lado, teve um tempo bem maior para 10.000 elementos (65,76 ms), o que indica que a estratégia de escolha de pivô utilizada provavelmente levou o algoritmo a um caso próximo do pior cenário (O(n²)), com muitas partições desbalanceadas.

### 4.3 Dados em ordem decrescente

Para vetores inicialmente decrescentes:

* O Quick Sort mantém vantagem em alguns tamanhos, mas perde desempenho para 10.000 elementos em comparação com o Insertion Sort.
* O Insertion Sort apresenta tempos menores do que o Bubble Sort em todos os tamanhos, embora ainda cresça quadraticamente.
* O Bubble Sort continua sendo o algoritmo mais lento, especialmente para o maior conjunto.

Esse comportamento reforça que, em cenários muito desfavoráveis (como dados já ordenados ou inversamente ordenados), a implementação específica do Quick Sort pode acabar sofrendo com partições ruins, enquanto o Insertion Sort, apesar de ser O(n²), mantém uma execução relativamente previsível.

## 5. Conclusão

Com base nos resultados obtidos, é possível destacar os seguintes pontos principais:

* O **Quick Sort** é, em geral, o algoritmo mais eficiente, principalmente para dados aleatórios e vetores de maior tamanho, confirmando sua complexidade média.
* O **Insertion Sort** apresenta bom desempenho para vetores pequenos e/ou quase ordenados, sendo particularmente eficiente no caso crescente.
* O **Bubble Sort** mostrou‑se o menos eficiente em praticamente todos os cenários, sendo pouco indicado para grandes volumes de dados.