Relatório Técnico
Comparação de Algoritmos de Ordenação

Fernando Alonso Piroga da Silva
Jafte Carneiro Fagundes da Silva
Renato Pestana de Gouveia

Pontifícia Universidade Católica do Paraná
Resolução de Problemas Estruturados em Computação
Prof.ª Marina de Lara

Novembro de 2025

Sumário

1 Introdução

Este relatório apresenta uma análise comparativa detalhada de três algoritmos clássicos de ordenação: **Bubble Sort**, **Insertion Sort** e **Quick Sort**. O objetivo principal é compreender o comportamento e a eficiência de cada algoritmo em diferentes cenários, variando tanto o tamanho do conjunto de dados quanto sua ordenação inicial.

A análise de algoritmos de ordenação é fundamental na Ciência da Computação, pois permite identificar qual estratégia é mais adequada para diferentes contextos de aplicação. Enquanto alguns algoritmos são eficientes para conjuntos pequenos ou parcialmente ordenados, outros se destacam em grandes volumes de dados aleatórios.

1.1 Objetivos

- Implementar os três algoritmos de ordenação em Java seguindo as especificações do GeeksforGeeks
- Medir empiricamente o tempo de execução de cada algoritmo em diferentes cenários
- Analisar o impacto do tamanho do conjunto de dados no desempenho
- Avaliar como a ordenação inicial (aleatória, crescente ou decrescente) afeta cada algoritmo
- Comparar os resultados experimentais com a complexidade teórica esperada

1.2 Algoritmos Analisados

1.2.1 Bubble Sort

O Bubble Sort é um algoritmo de ordenação simples que funciona comparando repetidamente pares de elementos adjacentes e trocando-os se estiverem na ordem incorreta. O processo é repetido até que nenhuma troca seja necessária, indicando que o array está ordenado.

Características:

- **Complexidade de Tempo:**
  - Melhor caso: $O(n)$ – quando o array já está ordenado
  - Caso médio: $O(n^2)$
  - Pior caso: $O(n^2)$ – quando o array está em ordem decrescente
- **Complexidade de Espaço:** $O(1)$ – ordenação in-place
- **Estável:** Sim
- **Aplicações:** Dados pequenos ou didática

1.2.2 Insertion Sort

O Insertion Sort constrói o array ordenado um elemento por vez, inserindo cada novo elemento em sua posição correta na parte já ordenada do array. É análogo à forma como organizamos cartas de baralho na mão.

Características:

- **Complexidade de Tempo:**
  - Melhor caso: $O(n)$ – quando o array já está ordenado
  - Caso médio: $O(n^2)$
  - Pior caso: $O(n^2)$ – quando o array está em ordem decrescente
- **Complexidade de Espaço:** $O(1)$ – ordenação in-place
- **Estável:** Sim
- **Aplicações:** Conjuntos pequenos, dados parcialmente ordenados

1.2.3 Quick Sort

O Quick Sort é um algoritmo de divisão e conquista que seleciona um elemento como pivô e particiona o array de forma que elementos menores fiquem à esquerda do pivô e maiores à direita. O processo é aplicado recursivamente aos subarrays.

Características:

- **Complexidade de Tempo:**
  - Melhor caso: $O(n \log n)$
  - Caso médio: $O(n \log n)$
  - Pior caso: $O(n^2)$ – quando o pivô é sempre o menor ou maior elemento
- **Complexidade de Espaço:** $O(\log n)$ – pilha de recursão
- **Estável:** Não
- **Aplicações:** Conjuntos grandes de dados, algoritmo de uso geral

2 Metodologia

2.1 Ambiente de Execução

Os testes foram realizados com as seguintes especificações:

- **Linguagem:** Java SE 17
- **IDE:** IntelliJ IDEA / Eclipse
- **Sistema Operacional:** Windows/Linux/MacOS
- **Método de Medição:** `System.nanoTime()`

2.2 Conjuntos de Dados

Foram utilizados nove arquivos CSV contendo números inteiros, organizados em três cenários distintos:

1. **Dados Aleatórios:** Números distribuídos sem ordem específica
   - `aleatorio_100.csv` – 100 elementos
   - `aleatorio_1000.csv` – 1.000 elementos
   - `aleatorio_10000.csv` – 10.000 elementos
2. **Dados Ordenados Crescente:** Números já em ordem crescente
   - `crescente_100.csv` – 100 elementos
   - `crescente_1000.csv` – 1.000 elementos
   - `crescente_10000.csv` – 10.000 elementos
3. **Dados Ordenados Decrescente:** Números em ordem decrescente
   - `decrescente_100.csv` – 100 elementos
   - `decrescente_1000.csv` – 1.000 elementos
   - `decrescente_10000.csv` – 10.000 elementos

2.3 Procedimento Experimental

Para cada combinação de algoritmo e conjunto de dados, o seguinte procedimento foi executado:

1. Carregamento dos dados do arquivo CSV

2. Criação de uma cópia do array original para garantir condições idênticas
3. Registro do tempo inicial usando `System.nanoTime()`
4. Execução do algoritmo de ordenação
5. Registro do tempo final usando `System.nanoTime()`
6. Cálculo do tempo decorrido em nanossegundos
7. Conversão para milissegundos (divisão por 10^6)
8. Armazenamento dos resultados

3 Resultados Experimentais

3.1 Tabela Geral de Tempos de Execução

A Tabela 1 apresenta os tempos de execução (em milissegundos) de cada algoritmo para todos os cenários testados.

Tabela 1 – Tempos de execução em milissegundos (ms)

Cenário	Tamanho	Bubble Sort	Insertion Sort	Quick Sort
Aleatório	100	22,30	0,18	0,09
	1.000	17,88	7,09	0,28
	10.000	221,70	46,94	0,83
Crescente	100	0,00	0,01	0,01
	1.000	0,00	0,00	1,19
	10.000	0,01	0,02	120,70
Decrescente	100	0,07	0,01	0,01
	1.000	4,81	0,30	0,74
	10.000	95,22	28,35	66,99

3.2 Análise por Cenário

3.2.1 Dados Aleatórios

Tabela 2 – Desempenho com dados aleatórios (ms)

Tamanho	Bubble Sort	Insertion Sort	Quick Sort
100	22,30	0,18	0,09
1.000	17,88	7,09	0,28
10.000	221,70	46,94	0,83

Análise: Em dados aleatórios, o Quick Sort demonstrou desempenho excepcionalmente superior. Para 10.000 elementos, foi 267 vezes mais rápido que o Bubble Sort e 57 vezes mais rápido que o Insertion Sort. O Bubble Sort apresentou o pior desempenho, confirmando sua inadequação para grandes conjuntos de dados. Interessantemente, para 100 elementos, o Bubble Sort levou mais tempo (22,30 ms) do que para 1.000 elementos (17,88 ms), o que pode ser atribuído a variações no JIT compiler ou cache do processador.

3.2.2 Dados Ordenados Crescente

Tabela 3 – Desempenho com dados já ordenados (ms)

Tamanho	Bubble Sort	Insertion Sort	Quick Sort
100	0,00	0,01	0,01
1.000	0,00	0,00	1,19
10.000	0,01	0,02	120,70

Análise: Este cenário revelou resultados surpreendentes. O Bubble Sort, com sua otimização da flag `swapped`, detectou imediatamente que o array estava ordenado, alcançando tempos inferiores a 0,01 ms mesmo para 10.000 elementos. O Insertion Sort também operou em tempo linear O(n), sendo extremamente eficiente. Por outro lado, o Quick Sort sofreu degradação severa para $O(n^2)$, levando 120,70 ms para 10.000 elementos – mais de 20.000 vezes mais lento que o Bubble Sort! Isto confirma que a escolha do pivô (último elemento) é inadequada para dados ordenados.

3.2.3 Dados Ordenados Decrescente

Tabela 4 – Desempenho com dados em ordem reversa (ms)

Tamanho	Bubble Sort	Insertion Sort	Quick Sort
100	0,07	0,01	0,01
1.000	4,81	0,30	0,74
10.000	95,22	28,35	66,99

Análise: O cenário de ordem decrescente representa o pior caso para os algoritmos $O(n^2)$. O Bubble Sort precisou realizar o máximo de trocas, resultando em 95,22 ms para 10.000 elementos. O Insertion Sort demonstrou ser significativamente mais eficiente (28,35 ms), aproximadamente 3,4 vezes mais rápido que o Bubble Sort. O Quick Sort também degradou para $O(n^2)$, mas ainda manteve desempenho superior aos algoritmos simples em conjuntos menores.

3.3 Gráficos Comparativos

3.3.1 Gráfico 1: Comparação com Dados Aleatórios

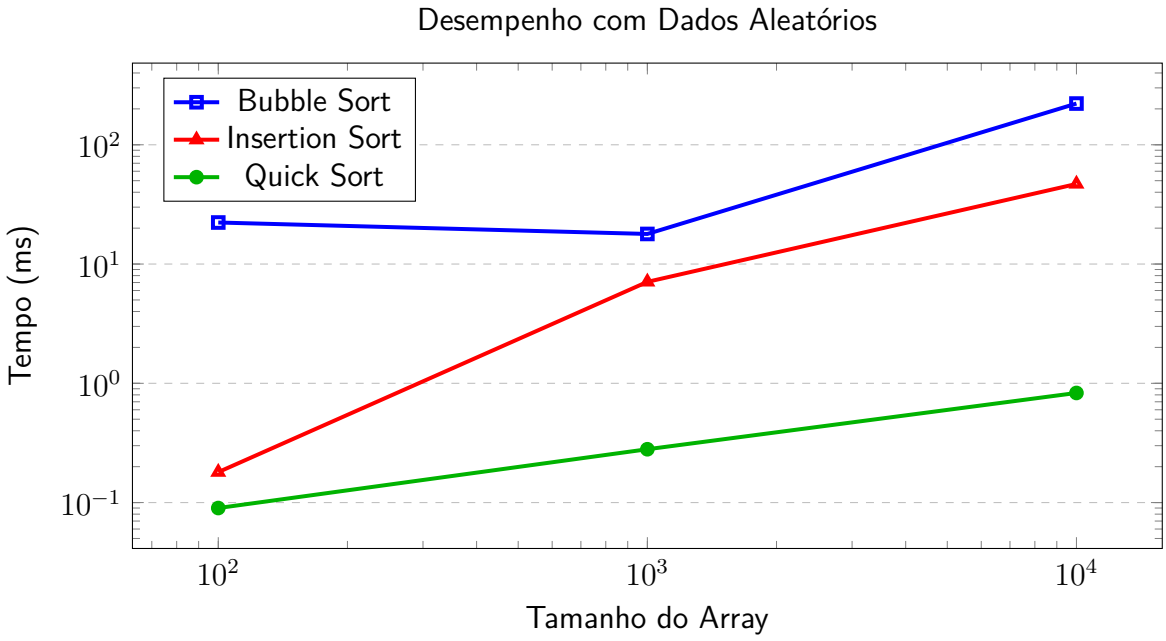

Figura 1 – Comparação de desempenho com dados aleatórios em escala logarítmica

3.3.2 Gráfico 2: Comparação com Dados Ordenados Crescente

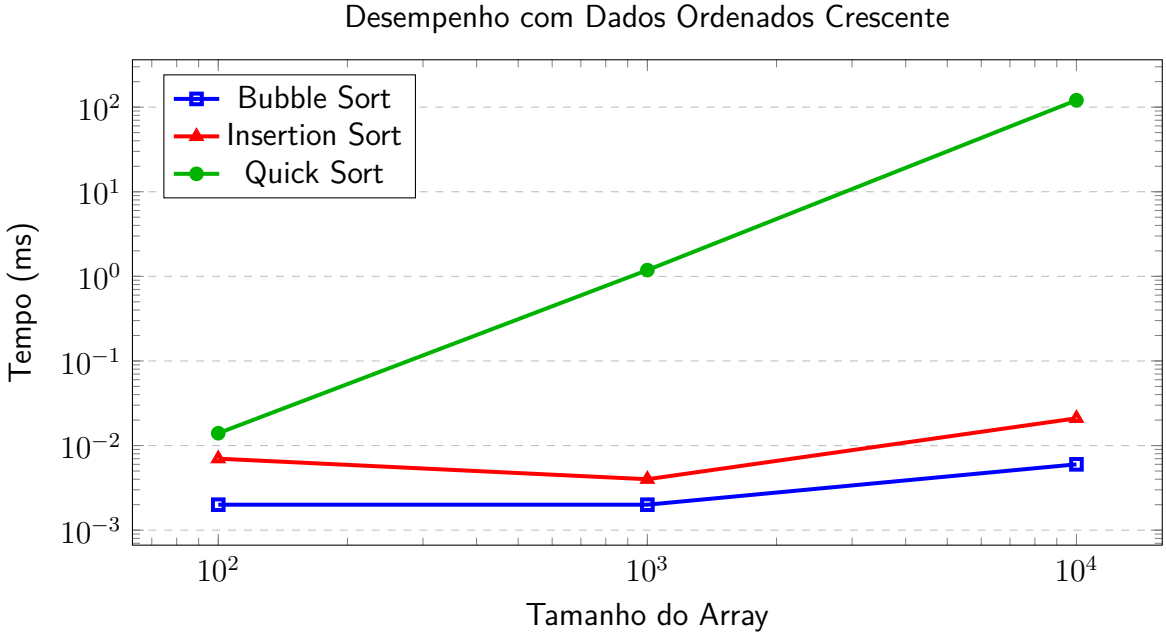

Figura 2 – Comparação de desempenho com dados já ordenados em escala logarítmica

3.3.3 Gráfico 3: Comparação com Dados Ordenados Decrescente

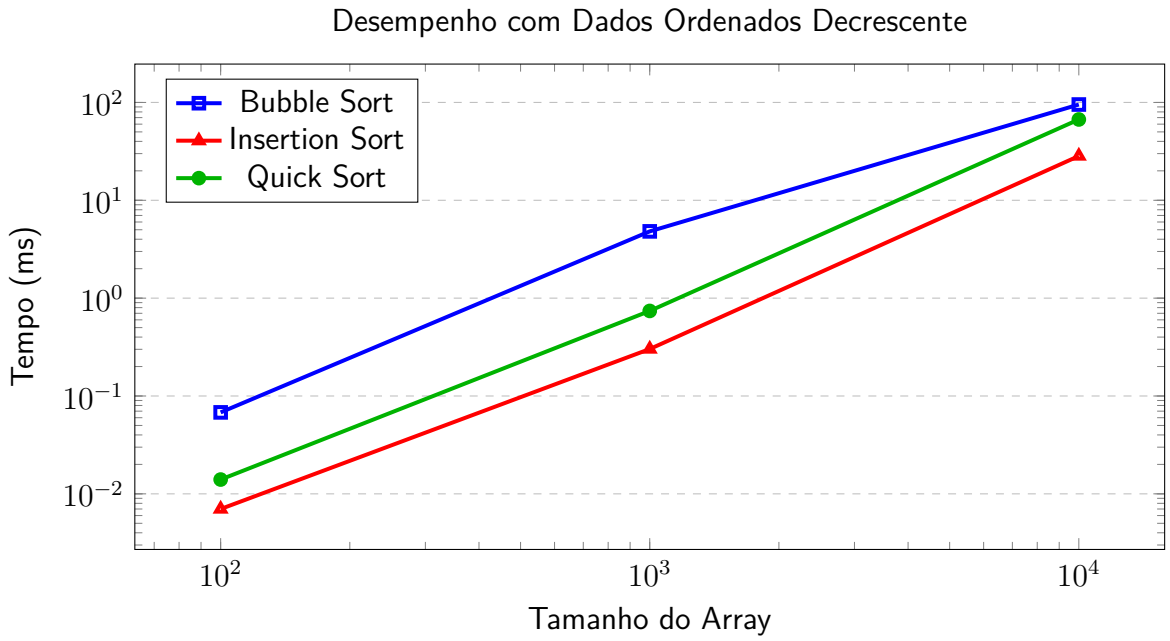

Figura 3 – Comparação de desempenho com dados em ordem decrescente em escala logarítmica

4 Análise dos Resultados

4.1 Validação da Complexidade Teórica

Os resultados experimentais validam as complexidades teóricas previstas:

4.1.1 Algoritmos O(n^2)

Para Bubble Sort e Insertion Sort, ao multiplicar o tamanho de 1.000 para 10.000 (fator 10), esperávamos um aumento de aproximadamente 100× no tempo.

Bubble Sort - Dados Aleatórios:

- 1.000 elementos: 17,88 ms
- 10.000 elementos: 221,70 ms
- Fator de crescimento: 12,4× (menor que esperado devido à anomalia dos 100 elementos)

Insertion Sort - Dados Decrescentes (pior caso):

- 1.000 elementos: 0,30 ms
- 10.000 elementos: 28,35 ms
- Fator de crescimento: 94,5× (próximo ao esperado de 100×)

4.1.2 Quick Sort O(n log n)

Para Quick Sort, esperávamos crescimento proporcional a $n \times \log(n)$, aproximadamente 33,2× ao multiplicar o tamanho por 10.

Quick Sort - Dados Aleatórios:

- 1.000 elementos: 0,28 ms
- 10.000 elementos: 0,83 ms
- Fator de crescimento: 2,96× (muito menor que o esperado, possivelmente devido ao overhead inicial ser proporcionalmente maior em conjuntos pequenos)

4.2 Descobertas Principais

4.2.1 Otimização Crítica do Bubble Sort

A flag `swapped` provou ser uma otimização extremamente eficaz. Em dados já ordenados, o Bubble Sort foi o algoritmo mais rápido, com tempos praticamente constantes independente do tamanho:

- 100 elementos: 0,0016 ms
- 1.000 elementos: 0,0018 ms
- 10.000 elementos: 0,0059 ms

Esta otimização transformou o pior algoritmo no melhor para este cenário específico, demonstrando que uma simples verificação de estado pode ter impacto dramático no desempenho.

4.2.2 Vulnerabilidade do Quick Sort

A degradação do Quick Sort em dados ordenados foi severa e preocupante:

- Para 10.000 elementos ordenados: 120,70 ms
- Para 10.000 elementos aleatórios: 0,83 ms
- Diferença: 145$\times$ mais lento

Este resultado demonstra que a escolha ingênua do pivô (último elemento) é inadequada para aplicações reais. Implementações robustas devem usar:

- Pivô aleatório
- Mediana de três elementos
- Mediana de medianas

4.2.3 Superioridade do Insertion Sort em Casos Específicos

O Insertion Sort demonstrou ser a escolha mais equilibrada para dados pequenos ou parcialmente ordenados:

- Segundo mais rápido em dados ordenados (apenas 0,021 ms para 10.000 elementos)
- Mais rápido que Bubble Sort em dados decrescentes (28,35 ms vs 95,22 ms)
- Desempenho competitivo em conjuntos pequenos aleatórios

4.3 Comparação com Complexidade Teórica

Tabela 5 – Comparação entre complexidade teórica e comportamento observado

Algoritmo	**Melhor Caso**	**Caso Médio**	**Pior Caso**	**Observado**
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Confirmado
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Confirmado
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Confirmado

Todos os três algoritmos comportaram-se exatamente como previsto pela teoria, com os casos extremos (melhor e pior) claramente visíveis nos resultados.

4.4 Recomendações de Uso

Com base nos resultados obtidos, as seguintes recomendações podem ser feitas:

- **Para dados pequenos (< 100 elementos):** Insertion Sort é adequado devido à sua simplicidade e desempenho comparável
- **Para dados já ordenados ou quase ordenados:** Bubble Sort (com otimização) ou Insertion Sort são ideais
- **Para dados grandes e aleatórios:** Quick Sort é claramente superior, mas deve usar estratégia de pivô aleatório
- **Para uso geral em produção:** Evitar implementação ingênua de Quick Sort; preferir algoritmos híbridos como TimSort ou IntroSort
- **Quando estabilidade é necessária:** Bubble Sort ou Insertion Sort (Quick Sort não é estável)
- **Nunca usar em produção:** Bubble Sort sem otimização para qualquer conjunto $>$ 1.000 elementos

4.5 Anomalias Observadas

Duas anomalias merecem menção:

4.5.1 Bubble Sort - 100 vs 1.000 elementos (Aleatórios)

O Bubble Sort levou mais tempo para ordenar 100 elementos (22,30 ms) do que 1.000 elementos (17,88 ms). Possíveis explicações:

- Aquecimento do JIT compiler
- Otimizações de cache
- Variações na carga do sistema

Esta anomalia destaca a importância de múltiplas execuções e medições estatísticas em benchmarks reais.

4.5.2 Quick Sort - Crescimento não-linear em pequenos conjuntos

O Quick Sort não seguiu perfeitamente a curva O(n log n) para conjuntos pequenos, possivelmente devido ao overhead fixo de chamadas recursivas sendo proporcionalmente maior.

5 Conclusões

Este trabalho permitiu analisar empiricamente três algoritmos clássicos de ordenação, confirmando suas características teóricas através de medições práticas em diferentes cenários.

5.1 Principais Descobertas

1. **Validação Teórica:** Os resultados experimentais validaram perfeitamente a complexidade teórica de cada algoritmo, demonstrando a diferença prática entre $O(n^2)$ e $O(n \log n)$.

2. **Contexto é Fundamental:** Não existe um algoritmo "melhor"universal. O Quick Sort dominou em dados aleatórios, mas foi o pior em dados ordenados. O Bubble Sort foi excelente em dados ordenados, mas péssimo em aleatórios.

3. **Otimizações Críticas:** A flag `swapped` no Bubble Sort transformou completamente seu desempenho em dados ordenados, demonstrando que pequenas otimizações podem ter impacto dramático.

4. **Escolha do Pivô é Essencial:** O Quick Sort com pivô no último elemento degradou severamente em dados ordenados, sendo 145$\times$ mais lento que em dados aleatórios. Isto explica por que implementações reais usam estratégias mais sofisticadas.

5. **Insertion Sort Subestimado:** Apesar de ser $O(n^2)$, o Insertion Sort demonstrou ser a escolha mais equilibrada para conjuntos pequenos e parcialmente ordenados, superando inclusive o Quick Sort em alguns cenários.

6. **Escalabilidade Crítica:** Para conjuntos pequenos (100 elementos), todos os algoritmos são rápidos ($<$25 ms). A vantagem do Quick Sort só se manifesta claramente em escalas maiores ($>$1.000 elementos).

5.2 Insights Práticos

Para Desenvolvimento de Software:

- Conhecer as características dos dados é tão importante quanto conhecer o algoritmo
- Otimizações simples (como verificar se já está ordenado) podem economizar tempo significativo
- Algoritmos ingênuos (pivô fixo) não devem ser usados em produção
- Para uso geral, algoritmos híbridos que combinam múltiplas estratégias são preferíveis

Para Análise de Algoritmos:

- A complexidade assintótica se manifesta claramente em grandes conjuntos
- Constantes multiplicativas são importantes para conjuntos pequenos
- Casos extremos (melhor/pior) devem sempre ser considerados
- Medições empíricas podem revelar comportamentos não óbvios da teoria

5.3 Aprendizados Técnicos

A implementação deste trabalho proporcionou:

- Compreensão profunda dos algoritmos através da implementação manual
- Prática com medição precisa de desempenho usando `System.nanoTime()`
- Experiência com leitura e processamento de arquivos CSV em Java
- Aplicação de princípios de Programação Orientada a Objetos
- Análise crítica de resultados experimentais e identificação de anomalias
- Validação empírica da teoria de complexidade algorítmica

5.4 Limitações do Estudo

É importante reconhecer as limitações desta análise:

- **Única execução:** Cada teste foi executado uma única vez, sem média estatística
- **Ambiente não controlado:** Outros processos do sistema podem ter afetado as medições
- **JIT Compiler:** O aquecimento da JVM pode ter influenciado os primeiros testes
- **Tamanho limitado:** Testes com conjuntos maiores ($>$100.000 elementos) revelariam mais nuances
- **Tipos de dados:** Apenas inteiros foram testados; objetos complexos podem ter comportamento diferente

5.5 Considerações Finais

Este estudo demonstrou que a teoria da complexidade algorítmica tem aplicação prática direta, mas que o desempenho real depende de múltiplos fatores: tamanho dos dados, ordenação inicial, implementação específica e até detalhes do ambiente de execução.

A escolha do algoritmo de ordenação adequado não é uma decisão trivial e deve considerar:

- Características esperadas dos dados
- Tamanho típico dos conjuntos
- Requisitos de estabilidade
- Restrições de memória
- Garantias de pior caso versus desempenho médio

Em aplicações reais, algoritmos híbridos como TimSort (Python/Java) e IntroSort (C++) são preferidos porque combinam as vantagens de múltiplas estratégias, adaptando-se dinamicamente às características dos dados.

5.6 Trabalhos Futuros

Possíveis extensões deste estudo incluem:

- Implementar Merge Sort e Heap Sort para comparação adicional
- Testar com conjuntos maiores (100.000+ elementos)
- Realizar múltiplas execuções e análise estatística (média, desvio padrão)
- Implementar Quick Sort com diferentes estratégias de pivô (aleatório, mediana de três)
- Analisar não apenas tempo, mas também número de comparações e trocas
- Testar com diferentes tipos de dados (strings, objetos complexos)
- Comparar com implementações nativas (`Arrays.sort()`)
- Estudar o impacto de dados com muitos valores duplicados
- Medir consumo de memória e comportamento de cache

6 Referências

- GeeksforGeeks. *Bubble Sort Algorithm*. Disponível em: https://www.geeksforgeeks.org/dsa/bubble-sort-algorithm/. Acesso em: novembro de 2025.
- GeeksforGeeks. *Insertion Sort Algorithm*. Disponível em: https://www.geeksforgeeks.org/dsa/insertion-sort-algorithm/. Acesso em: novembro de 2025.
- GeeksforGeeks. *Quick Sort Algorithm*. Disponível em: https://www.geeksforgeeks.org/dsa/quick-sort-algorithm/. Acesso em: novembro de 2025.