UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
IMD0029 - ESTRUTURAS DE DADOS BÁSICAS I

Análise Comparativa de Algoritmos de Ordenação

Componente Curricular: IMD0029 - Estruturas de Dados Básicas I
Docente: João Guilherme Domingos de Oliveira
Discente: Marília Yanis Cândido Costa

Natal, RN
2025

RESUMO

Este relatório apresenta uma análise de desempenho experimental de cinco algoritmos de ordenação: Bubble Sort, Insertion Sort, Selection Sort, Merge Sort e Quick Sort. O objetivo foi comparar o comportamento prático destes algoritmos com suas complexidades teóricas, submetendo-os a três cenários de dados distintos: aleatórios, quase ordenados e inversamente ordenados. A metodologia envolveu a implementação dos algoritmos em C++, a coleta de métricas de tempo, comparações e trocas, e a subsequente análise e visualização dos dados com scripts em Python. Os resultados confirmaram a superioridade dos algoritmos de complexidade O(nlogn) (Quick Sort e Merge Sort) em cenários gerais, mas também destacaram casos de exceção, como a degradação do Quick Sort para $O(n^2)$ em dados inversos e a eficiência do Insertion Sort em dados quase ordenados. Conclui-se com uma recomendação de uso para cada algoritmo baseada nas evidências empíricas coletadas.

Palavras-chave: Algoritmos de Ordenação. Análise de Algoritmos. Estrutura de Dados. Medição de Desempenho.

SUMÁRIO

1. INTRODUÇÃO

A ordenação de conjuntos de dados é uma das tarefas mais fundamentais na ciência da computação, sendo um pilar para a otimização de operações complexas como buscas, intercalações e análises estatísticas. A eficiência com que os dados são organizados pode determinar a performance de um sistema computacional. Embora diversos algoritmos de ordenação existam, não há uma solução universalmente excelente; a escolha do método mais adequado está intrinsecamente ligada às características da entrada, como volume de dados, grau de ordenação inicial e restrições de memória. O objetivo deste trabalho, portanto, é realizar uma análise experimental e comparativa do desempenho de cinco algoritmos de ordenação clássicos: Bubble Sort, Insertion Sort, Selection Sort, Merge Sort e Quick Sort. A análise visa confrontar a complexidade teórica destes algoritmos com seu comportamento prático, medindo métricas de tempo de execução, comparações e trocas sob diferentes cenários de dados: aleatórios, quase ordenados e inversamente ordenados.

2. DESENVOLVIMENTO

2.1. FUNDAMENTAÇÃO TEÓRICA

2.1.1. DESCRIÇÃO DOS ALGORITMOS

A. Bubble Sort: Algoritmo que percorre a lista diversas vezes, a cada passagem comparando elementos adjacentes e trocando-os de lugar se estiverem na ordem errada;
B. Insertion Sort: Constrói a lista ordenada final um elemento de cada vez, percorrendo a lista e inserindo cada elemento em sua posição correta na parte já ordenada;
C. Selection Sort: A cada passo, busca o menor elemento na porção não ordenada da lista e o posiciona no início da porção ordenada;
D. Merge Sort: Baseado na estratégia de "dividir para conquistar", divide a lista recursivamente ao meio até que cada sub-lista contenha um único elemento, e então as mescla de forma ordenada;
E. Quick Sort: Também de "dividir para conquistar", escolhe um elemento como pivô e particiona a lista, colocando os

elementos menores que o pivô à sua esquerda e os maiores à sua direita, ordenando as sub-listas recursivamente.

2.1.2. TABELA DE COMPLEXIDADES TEÓRICAS

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Complexidade de Espaço
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)

Tabela 1 – Complexidades Teóricas

2.2. METODOLOGIA

A análise experimental foi conduzida com base em um código-fonte desenvolvido em C++ , que implementa os algoritmos, gera os conjuntos de dados e realiza a coleta precisa das métricas de desempenho.

2.2.1. DESCRIÇÃO DOS TIPOS DE DADOS GERADOS

Foram gerados três tipos de conjuntos de dados para os testes:

A. Aleatórios: Vetores preenchidos com inteiros a partir de um gerador de números aleatórios (*std::mt19937*) da biblioteca padrão do C++.
B. Quase Ordenados: Vetores inicialmente ordenados onde 5% dos seus elementos tiveram suas posições trocadas aleatoriamente.
C. Inversamente Ordenados: Vetores preenchidos com elementos em ordem decrescente.

2.2.2. TAMANHO DOS VETORES TESTADOS

Os testes foram executados com vetores de quatro tamanhos distintos: 100, 1.000, 5.000 e 10.000 elementos.

2.2.3. MÉTRICAS COLETADAS E MÉTODO DE MEDIÇÃO

A coleta das métricas de desempenho foi realizada de forma precisa para cada execução. O tempo de execução foi aferido em milissegundos, utilizando a biblioteca <*chrono*> para capturar os instantes exatos antes e depois da chamada de cada função de ordenação. De modo complementar, as métricas de trabalho computacional, ou seja, o número de comparações e de trocas de elementos, foram mensuradas por meio de contadores em uma struct, passada por referência às funções. Assim, cada operação de comparação ou troca dentro dos algoritmos incrementava seu respectivo contador, garantindo uma medição detalhada e acurada.

3. RESULTADOS E ANÁLISES

3.1. TABELAS COMPARATIVAS ENTRE ALGORITMOS DE ORDENAÇÃO

A tabela a seguir apresenta um resumo dos dados brutos coletados para o maior vetor testado (N=10.000), demonstrando as diferenças de performance.

Algoritmo	Tempo (ms)	Comparações	Trocas
Quick Sort	02.09	167.892	85.252
Merge Sort	4.91	120.490	133.616
Insertion Sort	179.99	~24.8 milhões	~24.8 milhões
Selection Sort	233.32	~50 milhões	9.992
Bubble Sort	487.12	~50 milhões	~24.8 milhões

Tabela 2 – Comparação entre Algoritmos de Ordenação

3.2. GRÁFICOS E VISUALIZAÇÕES DOS DADOS

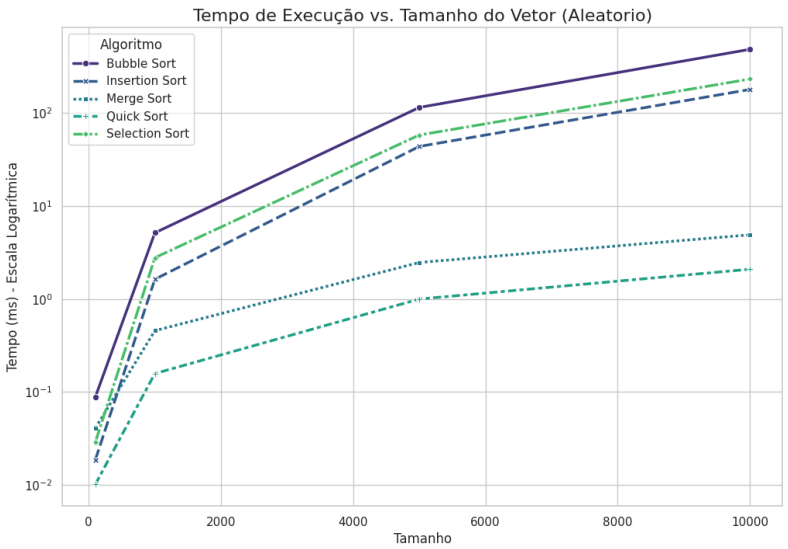

Figura 1 – Tempo de Execução vs. Tamanho do Vetor (Dados Aleatórios)

Observa-se na Figura 1 uma clara segregação dos algoritmos em duas classes de desempenho distintas. Os algoritmos Quick Sort e Merge Sort apresentam um crescimento de tempo suave e quase linear, característico da complexidade O(nlogn). Em contrapartida, os algoritmos Bubble Sort, Insertion Sort e Selection Sort exibem um crescimento exponencial, que corrobora suas complexidades de $O(n^2)$. Para um vetor de 10.000 elementos, essa diferença é massiva: o Quick Sort levou 2,09 ms enquanto o Bubble Sort precisou de 487,12 ms.

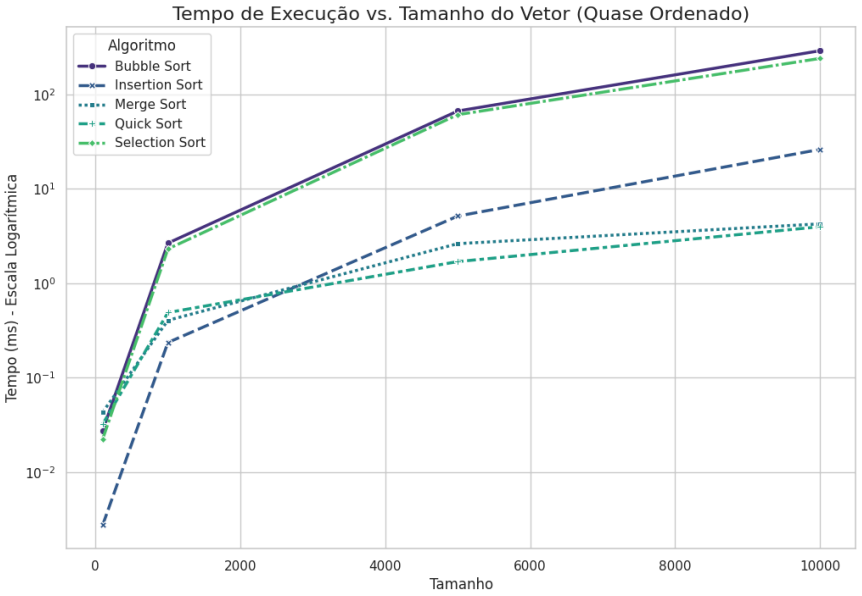

Figura 2 – Tempo de Execução vs. Tamanho do Vetor (Quase Ordenados)

Neste cenário, a principal observação é a notável melhora de desempenho do Insertion Sort. Sua curva de crescimento é significativamente mais suave do que no cenário aleatório, o que evidencia seu melhor caso teórico de complexidade O(n). Para N=10.000, seu tempo foi de 26,14 ms, uma redução drástica em comparação com os 179,99 ms do cenário aleatório.

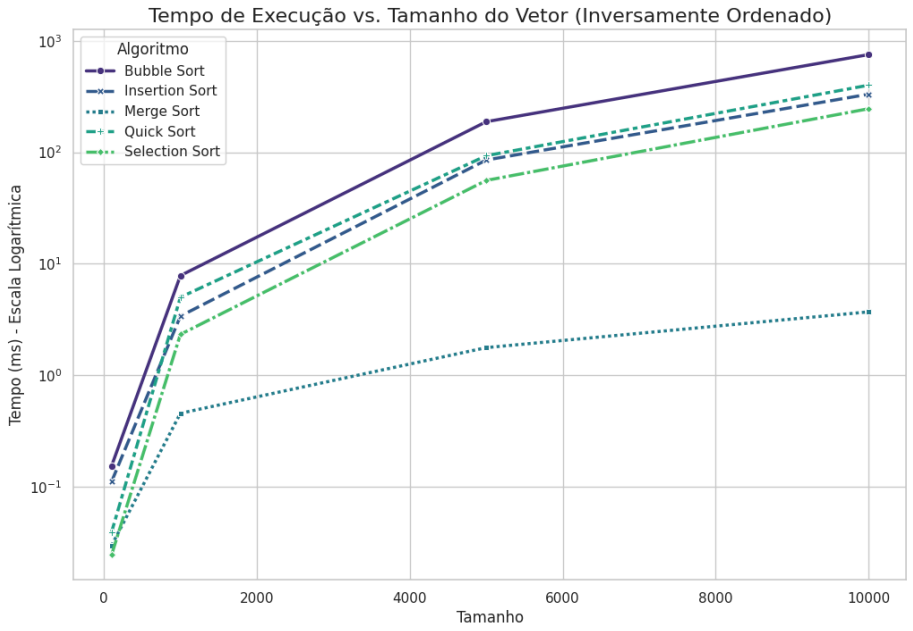

Figura 3 – Tempo de Execução vs. Tamanho do Vetor (Inversamente Ordenado)

Aqui, o gráfico mostra a principal vulnerabilidade do Quick Sort. Sua linha de desempenho, antes a mais eficiente, agora salta para o grupo dos algoritmos de complexidade quadrática. Seu tempo para N=10.000 foi de 399,03 ms, confirmando empiricamente seu pior caso $O(n^2)$. O Merge Sort, em contrapartida, manteve sua performance estável com apenas 3,71 ms, destacando-se como o mais robusto.

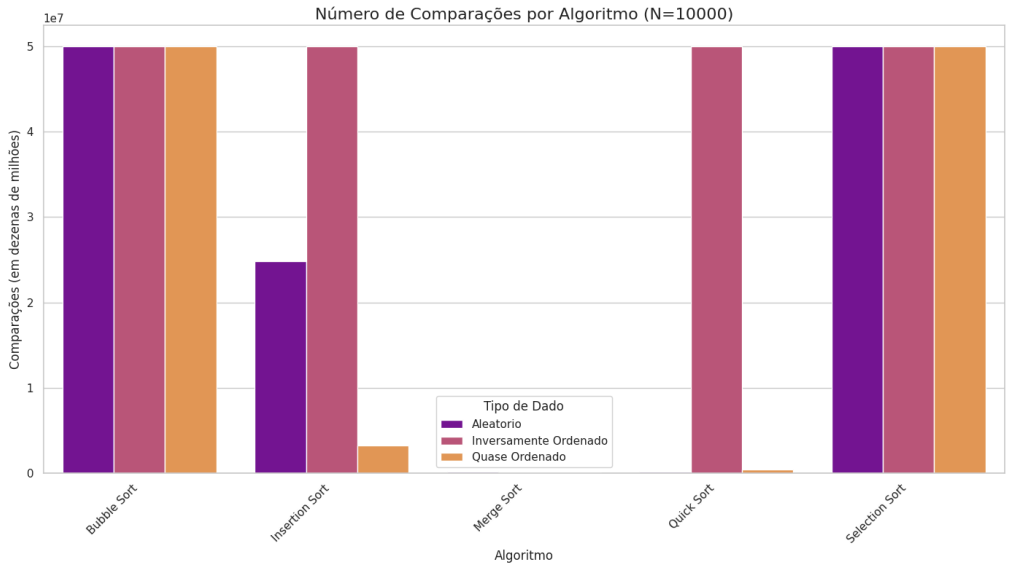

Figura 4 – Número de Comparações por Algoritmo (N = 10000)

O gráfico de barras demonstra que, no cenário inverso, o Quick Sort realizou o mesmo volume de comparações que os algoritmos quadráticos, aproximadamente 50 milhões. Isso explica a degradação de seu tempo de execução, pois seu trabalho computacional tornou-se máximo. O Merge Sort, por outro lado, manteve um baixo número de comparações em todos os cenários.

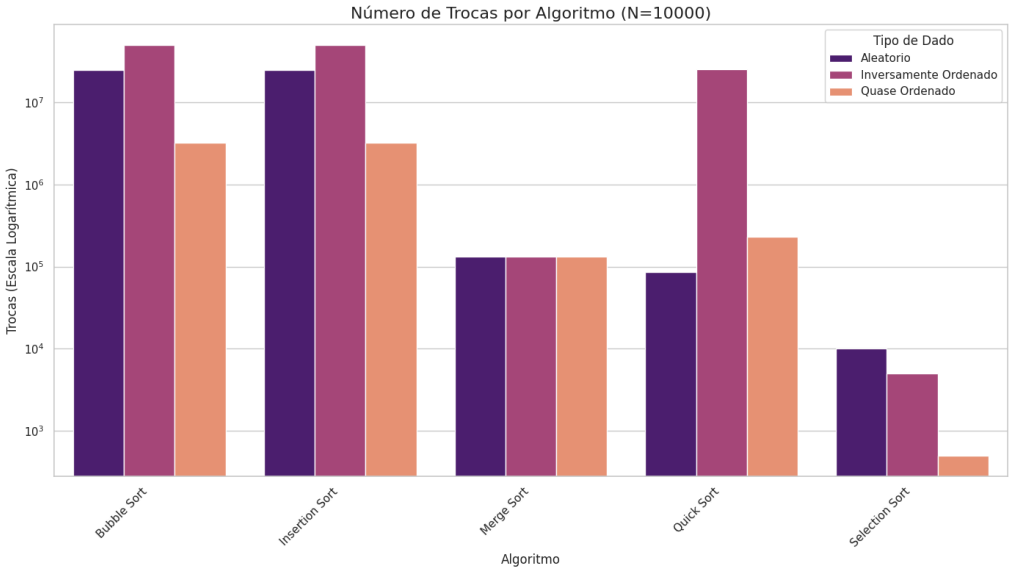

Figura 5 – Número de Trocas por Algoritmo (N = 10000)

Nesta figura, o destaque absoluto é o Selection Sort. As barras correspondentes a este algoritmo são quase imperceptíveis, pois ele foi projetado para minimizar as operações de

escrita na memória, realizando um número de trocas da ordem de O(n). Para N=10.000, ele realizou apenas 5.000 trocas no cenário inverso, enquanto outros algoritmos realizaram milhões.

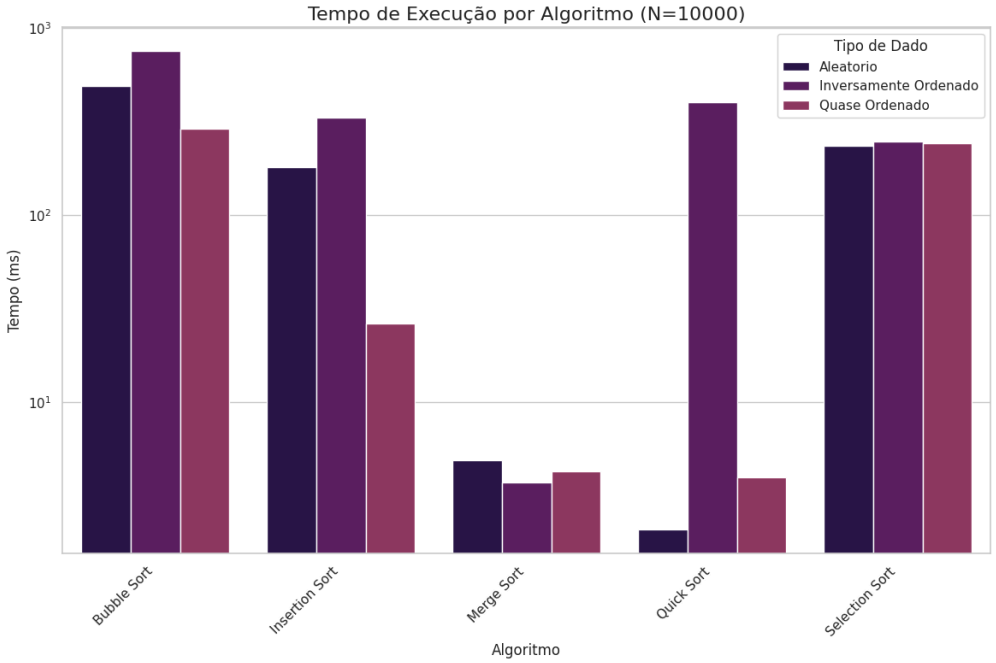

Figura 6 – Tempo de Execução por Algoritmo (N = 10000)

Este gráfico serve como um sumário conclusivo, identificando os algoritmos mais rápidos em cada cenário: o Quick Sort para dados aleatórios e quase ordenados, e o Merge Sort para dados inversamente ordenados, conforme os dados numéricos do relatório.

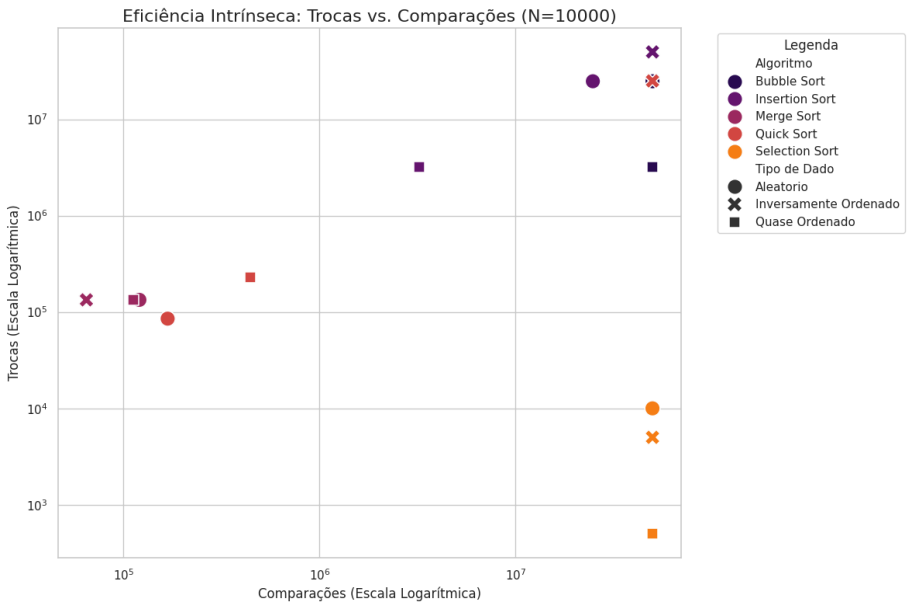

Figura 7 – Eficiência Intrínseca: Trocas vs. Comparações (N = 10000)

Este gráfico de dispersão revela a "assinatura" de cada algoritmo. O Selection Sort é um ponto isolado no canto inferior direito, indicando muitas comparações e poucas trocas. O Merge Sort forma um agrupamento coeso, mostrando seu equilíbrio e consistência. Os demais algoritmos se posicionam no canto superior direito, evidenciando seu alto custo tanto em comparações quanto em trocas.

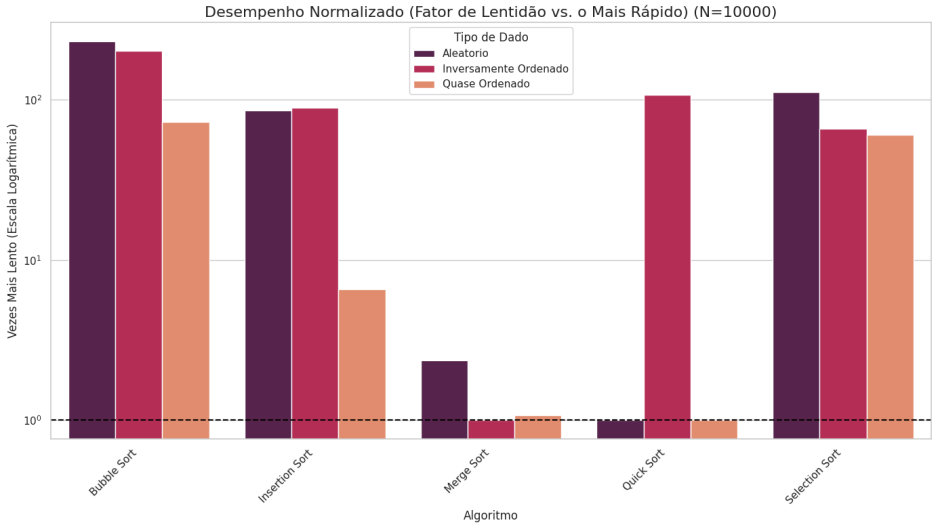

Figura 8 – Desempenho Normalizado (Fator de Lentidão vs. o Mais Rápido) (N = 10000)

Este gráfico traduz a complexidade em um impacto prático. Ele mostra que a escolha de um algoritmo inadequado pode ter um custo de desempenho massivo. Por exemplo, o Bubble Sort foi mais de 200 vezes mais lento que o Quick Sort no cenário aleatório, e o Quick Sort, em seu pior caso, foi mais de 100 vezes mais lento que o Merge Sort.

3.3. DISCUSSÃO CRÍTICA E INSIGHTS

A análise aprofundada dos dados experimentais, corroborada pelas visualizações gráficas, permite extrair insights que transcendem a mera classificação de desempenho, revelando as características intrínsecas e os trade-offs de engenharia de cada algoritmo.

A discussão mais fundamental que os dados proporcionam é a comprovação empírica da disparidade de performance entre os algoritmos de complexidade quadrática ($O(n^2)$) e os de complexidade log-linear ($O(n\log n)$). Conforme ilustrado nos gráficos de tempo versus

tamanho, observa-se uma segregação visual inequívoca, onde as curvas de desempenho para Quick Sort e Merge Sort exibem um crescimento suave e controlado, enquanto as curvas para Bubble Sort, Insertion Sort e Selection Sort ascendem de forma exponencial. Este comportamento não representa uma diferença sutil, mas sim um abismo de performance. O impacto quantitativo torna-se evidente nos dados para N=10.000 no cenário aleatório, onde o Quick Sort finalizou a ordenação em 2,09 ms, ao passo que o Bubble Sort necessitou de 487,12 ms. O gráfico de desempenho normalizado, ou "Fator de Lentidão", dramatiza essa diferença, demonstrando que a escolha de um algoritmo $O(n^2)$ em vez de um O(nlogn) pode resultar em uma penalidade de performance superior a 200 vezes para este conjunto de dados. Fica evidente, portanto, que a complexidade teórica é o principal preditor do desempenho prático para grandes volumes de dados e que a seleção de um algoritmo de uma classe de complexidade inadequada para um problema de larga escala constitui, na prática, um erro de projeto com implicações diretas na eficiência do sistema.

Dentro da classe de algoritmos eficientes, o Quick Sort é frequentemente considerado o mais rápido na prática, um fato que os dados confirmam, porém com uma ressalva crítica que define sua dualidade. Em cenários com dados de entrada aleatórios e quase ordenados, o Quick Sort apresentou consistentemente o menor tempo de execução, performance atribuída a fatores como a excelente localidade de referência (localidade de cache), que otimiza o acesso à memória, e um baixo fator constante em sua complexidade média. Contudo, o ponto mais relevante da análise é a sua falha de desempenho no cenário de dados inversamente ordenados. Nesta situação, seu tempo de execução saltou de 2,09 ms para 399,03 ms. A visualização desta anomalia no gráfico de tempo correspondente mostra a curva do Quick Sort convergindo com a dos algoritmos quadráticos, e a causa é revelada no gráfico de comparações, onde seu número de comparações atingiu o máximo teórico de quase 50 milhões, idêntico ao do Bubble Sort. Este fenômeno ocorre devido à escolha sistemática de pivôs inadequados em dados estruturados, o que gera partições recursivas desbalanceadas (com sub-vetores de tamanho n-1 e 0), degradando a complexidade do algoritmo para seu pior caso teórico de $O(n^2)$.

Em oposição à volatilidade do Quick Sort, o Merge Sort emerge como o modelo de confiabilidade e desempenho previsível. Sua característica mais notável é a estabilidade de desempenho em todos os cenários testados, com tempos de execução para N=10.000 que foram notavelmente consistentes: 4,91 ms (aleatório), 3,71 ms (inverso) e 4,26 ms (quase

ordenado). Além disso, o algoritmo foi o mais performático no cenário de dados inversos, justamente onde o Quick Sort demonstrou sua maior fraqueza. Essa imunidade a padrões nos dados de entrada o torna a escolha ideal para aplicações de missão crítica, onde picos de latência inesperados são inaceitáveis. O Merge Sort representa, portanto, um clássico trade-off de engenharia de software: sacrifica-se uma fração da performance no caso médio em favor de uma garantia de desempenho O(nlogn) no pior caso. Seu principal custo não é temporal, mas espacial, devido à sua complexidade de espaço de O(n) para os arrays auxiliares de mesclagem, o que pode ser uma restrição em sistemas com memória limitada.

Ainda assim, a análise revela que mesmo os algoritmos quadráticos, embora superados em cenários gerais, possuem relevância em contextos específicos. A análise do cenário de dados quase ordenados, por exemplo, destaca a força do Insertion Sort. Sua performance melhorou drasticamente, caindo de 179,99 ms em dados aleatórios para 26,14 ms. Este comportamento alinha-se à sua complexidade de melhor caso de O(n), pois quando os dados estão próximos de suas posições finais, cada inserção requer um número mínimo de deslocamentos, tornando-o imbatível para volumes de dados menores ou como componente de algoritmos híbridos. De forma similar, o Selection Sort justifica sua relevância pela análise da métrica de trocas. Posicionando-se como um ponto isolado nos gráficos, ele é caracterizado por um número massivo de comparações, mas um número mínimo e previsível de trocas (O(n)). Em um cenário onde a operação de escrita em memória é computacionalmente cara ou fisicamente limitada, como em memórias Flash, o Selection Sort, apesar de seu tempo de execução elevado, representa a escolha de engenharia correta para maximizar a durabilidade do hardware.

4. CONCLUSÃO

Com base na análise experimental, conclui-se que a escolha do algoritmo de ordenação ideal é dependente do cenário de aplicação. Para dados aleatórios de grande volume, o Quick Sort oferece o melhor desempenho médio. Para aplicações que exigem garantia de performance e estabilidade, o Merge Sort é a escolha superior, como demonstrado em seu desempenho consistente em todos os cenários. O Insertion Sort prova seu valor em conjuntos de dados pequenos ou quase ordenados, enquanto o Selection Sort é útil em contextos onde a operação de troca de elementos é computacionalmente cara. Os resultados práticos validaram as predições teóricas de complexidade, reforçando a importância da análise de algoritmos na engenharia de software.