Relatório Técnico: Análise Comparativa de Algoritmos de Ordenação

Jerônimo Rafael Bezerra Filho
20240039188

Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital
IMD0029 - Estrutura de Dados Básicas 1
Professor: João Guilherme
4 de julho de 2025

Sumário

1 Introdução

O presente relatório detalha uma análise experimental comparativa de cinco algoritmos de ordenação fundamentais na ciência da computação: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort e Quick Sort. A ordenação de dados é uma das tarefas mais ubíquas e essenciais em processamento de informações, servindo como base para inúmeras outras operações, como busca, fusão de dados e otimização de consultas em bancos de dados. O objetivo deste trabalho é avaliar o desempenho prático desses algoritmos sob diferentes condições de entrada de dados, medindo o tempo de execução, o número de comparações e o número de trocas. A análise busca não apenas verificar as complexidades teóricas, mas também fornecer uma base empírica para a escolha do algoritmo mais adequado a um determinado cenário de aplicação.

2 Fundamentação Teórica

Nesta seção, descrevemos brevemente cada um dos algoritmos analisados e suas respectivas complexidades teóricas.

- **Bubble Sort:** Itera repetidamente sobre a lista, comparando pares de elementos adjacentes e trocando-os se estiverem na ordem errada. O processo continua até que nenhuma troca seja necessária, indicando que a lista está ordenada.
- **Selection Sort:** Divide a lista em duas partes: uma sublista ordenada e uma sublista com os itens restantes. A cada iteração, o algoritmo encontra o menor elemento da sublista não ordenada e o move para o final da sublista ordenada.
- **Insertion Sort:** Constrói a lista ordenada final um item de cada vez. Ele itera sobre a entrada, e para cada elemento, o insere na posição correta dentro da porção já ordenada da lista.
- **Merge Sort:** É um algoritmo de "dividir para conquistar". Ele divide a lista não ordenada em n sublistas, cada uma contendo um elemento, e então as mescla repetidamente para produzir novas sublistas ordenadas até que reste apenas uma.
- **Quick Sort:** Também utiliza a abordagem de "dividir para conquistar". Ele seleciona um elemento como pivô e particiona os outros elementos em dois sub-arrays, de acordo com se eles são menores ou maiores que o pivô. Os sub-arrays são então ordenados recursivamente.

Tabela 1: Complexidade Teórica dos Algoritmos de Ordenação.

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

3 Metodologia

A análise experimental foi conduzida seguindo uma metodologia estruturada para garantir a consistência e a validade dos resultados.

- **Dados Gerados:** Foram criados três tipos de vetores para cada teste:
  - **Aleatórios:** Vetores com elementos em ordem completamente aleatória.
  - **Quase Ordenados:** Vetores inicialmente ordenados, mas com 10% de seus elementos trocados de posição aleatoriamente para simular dados com pouca desordem.
  - **Inversamente Ordenados:** Vetores com elementos em ordem decrescente, representando o pior cenário para muitos algoritmos.
- **Tamanhos dos Vetores:** Os testes foram executados com vetores de três tamanhos distintos: 1.000, 5.000 e 10.000 elementos.
- **Métricas e Medição:** Para cada execução, foram coletadas três métricas:
  - **Tempo de Execução:** Medido em milissegundos (ms) utilizando a biblioteca `<chrono>` do C++, que oferece medições de alta precisão.
  - **Comparações:** Contagem do número de vezes que dois elementos foram comparados.
  - **Trocas:** Contagem do número de vezes que elementos foram trocados de posição.

4 Resultados e Análise

Nesta seção, apresentamos os dados coletados da execução dos cinco algoritmos de ordenação em diferentes cenários. A análise subsequente compara o desempenho prático observado com as complexidades teóricas esperadas de cada algoritmo.

4.1 Dados Brutos Compilados

Os algoritmos foram testados com vetores de tamanho 1.000, 5.000 e 10.000, em três configurações: aleatória, quase ordenada e inversamente ordenada. A Tabela 2 resume os resultados para o maior conjunto de dados testado (N = 10.000), onde as diferenças de desempenho são mais evidentes.

Tabela 2: Resultados compilados para N = 10.000.

Algoritmo	Tipo de Vetor	Tempo (ms)	Comparações	Trocas
BubbleSort	Aleatorio	279.539	49.995.000	24.905.679
	Quase Ordenado	132.175	49.995.000	5.789.136
	Inversamente	291.275	49.995.000	49.995.000
SelectionSort	Aleatorio	87.335	49.995.000	9.986
	Quase Ordenado	78.038	49.995.000	1.000
	Inversamente	81.429	49.995.000	5.000
InsertionSort	Aleatorio	67.190	24.915.673	24.915.678
	Quase Ordenado	**12.376**	**5.799.133**	**5.799.135**
	Inversamente	118.258	49.995.000	50.004.999
MergeSort	Aleatorio	**1.999**	**120.387**	133.616
	Quase Ordenado	1.504	115.317	133.616
	Inversamente	0.999	64.608	133.616
QuickSort	Aleatorio	**1.000**	166.035	90.630
	Quase Ordenado	2.035	364.475	231.519
	Inversamente	171.322	49.995.000	25.004.999

4.2 Análise Crítica

Os resultados experimentais corroboram fortemente a teoria da complexidade de algoritmos.

Algoritmos de Complexidade Quadrática ($O(n^2)$)

Conforme a teoria, **Bubble Sort** e **Selection Sort** realizaram um número de comparações idêntico e massivo, independentemente da ordem inicial do vetor. O Bubble Sort se mostrou o menos eficiente, com um número de trocas extremamente elevado. Em contrapartida, a grande vantagem do Selection Sort é visível nos dados: o número de trocas é linear ($O(n)$) e drasticamente inferior.

O **Insertion Sort** demonstrou sua natureza *adaptativa*. No cenário "Quase Ordenado", seu desempenho foi notavelmente superior ao dos outros algoritmos $O(n^2)$. Contudo, no pior caso (vetor inverso), seu desempenho degradou para um nível similar ao do Bubble Sort.

Algoritmos de Complexidade Log-Linear ($O(n \log n)$)

O **Merge Sort** apresentou o comportamento mais estável e previsível. Seu tempo de execução foi consistentemente baixo em todos os três cenários, confirmando sua eficiência e imunidade a piores casos.

O **Quick Sort** foi o campeão nos cenários "Aleatorio"e "Quase Ordenado", registrando os menores tempos. No entanto, os dados do cenário "Inversamente Ordenado"expõem sua grande vulnerabilidade: o desempenho degradou catastroficamente para $O(n^2)$, com o número de comparações saltando para quase 50 milhões. Isso ocorre porque a estratégia de pivô adotada é ineficiente para dados já ordenados ou inversamente ordenados.

4.3 Visualização Gráfica

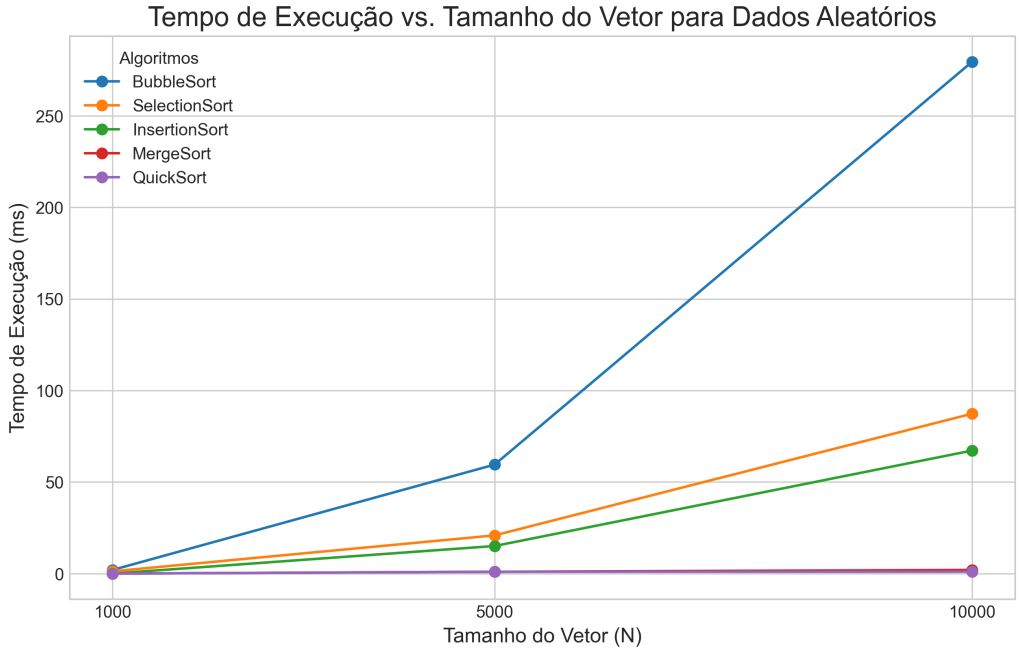

Figura 1: Gráfico de Tempo de Execução (ms) vs. Tamanho do Vetor para dados aleatórios.

Um gráfico de tempo por tamanho, como o exemplificado na Figura 1, ilustra visualmente a diferença de crescimento. Observa-se uma curva de crescimento acentuado para os algoritmos quadráticos e uma curva muito mais suave para os algoritmos log-lineares.

5 Conclusões

Com base na análise experimental, é possível concluir que a escolha de um algoritmo de ordenação depende fortemente das características dos dados de entrada e dos requisitos da aplicação. Os resultados práticos alinham-se com a teoria da complexidade, mas fornecem insights quantitativos sobre as constantes e os cenários que favorecem cada algoritmo.

5.1 Ranking de Desempenho

1. **Merge Sort:** Melhor desempenho geral devido à sua consistência e imunidade a piores casos.
2. **Quick Sort:** Mais rápido no caso médio, mas com risco de degradação em cenários específicos.
3. **Insertion Sort:** Melhor para dados pequenos ou quase ordenados.
4. **Selection Sort:** Lento, mas com um número muito baixo de trocas.
5. **Bubble Sort:** Ineficiente e não recomendado para uso prático.

5.2 Aplicações Recomendadas

- **Cenário Crítico (Estabilidade e Previsibilidade):** Utilizar **Merge Sort**.

- **Desempenho Médio Máximo:** Utilizar **Quick Sort**, preferencialmente com uma estratégia de pivoteamento mais robusta.
- **Dados Quase Ordenados:** Utilizar **Insertion Sort**.
- **Restrição de Escrita na Memória:** Utilizar **Selection Sort**, onde o custo de trocas é mais crítico que o de comparações.
- **Fins Didáticos:** Utilizar **Bubble Sort** para ensinar conceitos básicos de ordenação.