Universidade Estadual da Paraíba - UEPB

Centro de Ciências e Tecnologia - CCT

Curso de Ciências da Computação - CC

Relatório do projeto da disciplina de Laboratório de Estrutura de Dados
Análise comparativa dos algoritmos de ordenação

Laryssa Finizola Costa da Silva

Ludmila Maria Pereira da Silva

Campina Grande

2025

Projeto desenvolvido junto ao curso de
Ciências da Computação da UEPB
na cadeira de Laboratório de Estrutura
de Dados, como requisito parcial
à obtenção do título de Bacharel.

Orientador: Prof. Fábio Leite

SUMÁRIO

1. Introdução:

Este relatório é realizado como material de estudos para a disciplina de Laboratório de Estrutura de Dados e tem como objetivo apresentar uma análise comparativa de diferentes algoritmos de ordenação aplicados a uma

base de dados contendo tweets. O objetivo do experimento é avaliar o desempenho dos algoritmos nas ordenações por usuário, data e quantidade de menções, considerando diferentes cenários: melhor caso, caso médio e pior caso.

A partir desses testes, foram realizadas análises com base em diferentes critérios, a fim de identificar o algoritmo de ordenação mais eficiente para cada situação específica, utilizando como métrica principal o **tempo de execução em nanosegundos (ns)**.

2. Estrutura da Base de Dados:

O arquivo original **tweets.csv** foi processado e transformado pelo programa em um novo arquivo chamado **tweets_mentioned_persons.csv**, adequado para a aplicação dos algoritmos de ordenação. Durante essa etapa de transformação, os dados foram organizados e formatados conforme os seguintes critérios:

- **Texto do tweet**: conteúdo textual completo da publicação original;
- **Data do tweet**: convertida para o formato **yyyyMMdd** (ano, mês, dia), facilitando a ordenação cronológica;
- **Quantidade de menções**: calculada com base na contagem de palavras precedidas pelo caractere **@**, indicando perfis mencionados no tweet.

Essa estrutura padronizada permitiu a aplicação eficiente dos algoritmos de ordenação, possibilitando uma análise comparativa precisa quanto ao desempenho de cada técnica em diferentes contextos (melhor caso, caso médio e pior caso).

3. Algoritmos Avaliados

Os algoritmos de ordenação que foram avaliados são todos aqueles abordados na disciplina, são eles:

- **Merge Sort**
- **Quick Sort**
- **Quick Sort com Mediana de Três**
- **Selection Sort**
- **Insertion Sort**
- **Heap Sort**
- **Counting Sort**

4. Metodologia:

Para cada combinação de algoritmo e campo de ordenação, foram considerados três cenários:

- **Melhor caso:** dados previamente ordenados
- **Caso médio:** dados em ordem aleatória (original da base)
- **Pior caso:** dados ordenados em ordem inversa

Os tempos de execução foram medidos em milissegundos e registrados no arquivo `resultados_tempos.csv`.

5. Resultados:

5.1: Campo Date

mergeSort	date	piorCaso	53
mergeSort	date	medioCaso	1881
mergeSort	date	melhorCaso	48
insertionSort	date	piorCaso	41
selectionSort	date	medioCaso	44203
selectionSort	date	melhorCaso	102
selectionSort	date	piorCaso	109
quickSort	date	medioCaso	4.128
quickSort	date	melhorCaso	109
quickSort	date	piorCaso	123
quickSortMediar	date	medioCaso	82013
quickSortMediar	date	melhorCaso	87
quickSortMediar	date	piorCaso	68
heapSort	date	medioCaso	20449
heapSort	date	melhorCaso	53
heapSort	date	piorCaso	52

No campo date, o algoritmo que apresentou melhor desempenho no melhor caso foi o Merge Sort, com tempo de execução de 48 ms. Sua estabilidade e a forma eficiente como divide os dados tornam-no especialmente eficaz quando a lista já está ordenada. No caso médio, o Merge Sort novamente se destacou, alcançando o menor tempo (1881 ms) e superando algoritmos como o Heap Sort (20449 ms) e o Quick Sort (4128 ms). Já no pior caso, o algoritmo com melhor desempenho foi o Heap Sort, com 52 ms, demonstrando sua capacidade de manter um desempenho estável mesmo diante de entradas desfavoráveis, sendo mais eficiente que alternativas como Quick Sort e Selection Sort nesse cenário.

5.1: Campo User

mergeSort	user	medioCaso	239
mergeSort	user	melhorCaso	7
mergeSort	user	piorCaso	46
insertionSort	user	medioCaso	1.699
insertionSort	user	melhorCaso	67
insertionSort	user	piorCaso	46
selectionSort	user	medioCaso	241
selectionSort	user	melhorCaso	58
selectionSort	user	piorCaso	53
quickSort	user	medioCaso	1.265
quickSort	user	melhorCaso	98
quickSort	user	piorCaso	118
quickSortMediar	user	medioCaso	459
quickSortMediar	user	melhorCaso	101
quickSortMediar	user	piorCaso	83
heapSort	user	medioCaso	222
heapSort	user	melhorCaso	68
heapSort	user	piorCaso	55

No campo user, o algoritmo com melhor desempenho no melhor caso foi o Selection Sort, com 58 ms, superando os demais. Sua simplicidade favorece a performance em listas já ordenadas. No caso médio, o algoritmo mais eficiente foi o Heap Sort, com 222 ms, apresentando melhor desempenho que o Merge Sort (239 ms) e o Quick Sort (1265 ms), além de se manter consistente mesmo em entradas aleatórias. Já no pior caso, o destaque foi o Merge Sort, que teve o menor tempo de execução (46 ms) e demonstrou grande estabilidade entre os diferentes cenários de teste.

5.1: Campo Count

mergeSort	count	medioCaso	242
mergeSort	count	melhorCaso	51
mergeSort	count	piorCaso	42
insertionSort	count	medioCaso	152
insertionSort	count	melhorCaso	64
insertionSort	count	piorCaso	68
countingSort	count	medioCaso	2.352
countingSort	count	melhorCaso	42
countingSort	count	piorCaso	46
selectionSort	count	medioCaso	275
selectionSort	count	melhorCaso	57
selectionSort	count	piorCaso	45
quickSort	count	medioCaso	366
quickSort	count	melhorCaso	72
quickSort	count	piorCaso	75
quickSortMediar	count	medioCaso	47
quickSortMediar	count	melhorCaso	63
quickSortMediar	count	piorCaso	57
heapSort	count	medioCaso	198
heapSort	count	melhorCaso	84
heapSort	count	piorCaso	93

No campo `count`, o Counting Sort foi o mais eficiente no melhor caso, com 42 ms, sendo ideal para ordenar números inteiros com faixa limitada. No caso médio, o Insertion Sort se destacou com 152 ms, superando até mesmo o Counting Sort (2352 ms), que teve desempenho inferior nesse cenário devido à sobrecarga de preparação da estrutura interna. Já no pior caso, o Merge Sort foi o mais eficiente, também com 42 ms, demonstrando excelente estabilidade e performance mesmo diante de dados ordenados de forma inversa, enquanto o Counting Sort teve dificuldades nesse tipo de entrada**.**

6. Conclusão:

Com base nos experimentos realizados, foi possível observar como diferentes algoritmos de ordenação se comportam diante de cenários variados e tipos distintos de dados. Cada algoritmo possui suas características específicas que os tornam mais ou menos adequados a determinadas situações, evidenciando que não existe uma solução universalmente superior. A escolha do algoritmo ideal depende fortemente do contexto em que será aplicado.

No campo date, o Merge Sort demonstrou excelente desempenho em cenários ordenados e semi-ordenados, mostrando-se o mais eficiente tanto no melhor quanto no caso médio. No entanto, foi o Heap Sort que se destacou no pior caso, graças à sua estabilidade e à capacidade de manter um tempo de execução consistente mesmo em situações menos favoráveis. Isso reforça a importância de considerar não apenas a complexidade teórica, mas também a natureza dos dados ao escolher um algoritmo de ordenação.

Já no campo user, observamos que algoritmos mais simples, como o Selection Sort, podem apresentar resultados surpreendentemente bons em dados já organizados. No entanto, em situações intermediárias, o Heap Sort foi novamente o mais eficiente, revelando-se uma escolha segura em contextos onde a estrutura dos dados é incerta. No cenário mais crítico, o Merge Sort voltou a se destacar, reforçando sua confiabilidade e robustez.

Por fim, no campo count, o Counting Sort foi imbatível no melhor caso, como era esperado, dado que trabalha de forma extremamente eficiente com inteiros em faixas limitadas. No caso médio, porém, o Insertion Sort superou-o, beneficiando-se da simplicidade e da eficiência em listas parcialmente organizadas. No pior caso, o Merge Sort mostrou-se novamente estável e eficaz, lidando melhor com listas em ordem inversa que o próprio Counting Sort, cuja preparação interna comprometeu o desempenho.

Essas análises reforçam a importância de compreender as características dos algoritmos de ordenação e de realizar testes práticos antes de adotá-los em aplicações reais. A teoria oferece uma base sólida, mas somente com a prática é

possível validar quais estratégias realmente entregam os melhores resultados em cada situação.