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. Casos de Teste

Para avaliar o desempenho dos algoritmos de ordenação de forma sistemática e justa, foram definidos três tipos distintos de cenários para a base de dados, representando diferentes níveis de dificuldade para a execução dos algoritmos. Os casos de teste foram:

a) **Melhor Caso: Compreende os casos em que os dados já estão previamente ordenados.**

Neste cenário, a base de dados foi organizada previamente no formato desejado para cada campo (date, user ou count). Assim, os algoritmos receberam uma lista já ordenada, o que favorece alguns métodos de ordenação, especialmente aqueles que se beneficiam de estruturas já organizadas. O objetivo para esse caso é avaliar o comportamento dos algoritmos na situação mais favorável possível, onde o trabalho de ordenação é mínimo. Por fim, nesse caso houve a compreensão dos dados numa ordenação crescente.

b) **Caso Médio: Dados em ordem aleatória**

Nesse caso, foi utilizado a base de dados no seu estado original, sem qualquer tipo de ordenação aplicada. Essa situação representa o cenário mais comum em aplicações práticas, onde os dados não possuem nenhuma organização prévia. O real objetivo desse caso de teste é medir a eficiência dos algoritmos em condições típicas do mundo real, com entradas aleatórias e sem padrões óbvios.

c) **Pior Caso: Compreende os casos em que os dados já estão previamente ordenados, mas de forma decrescente.**

Neste cenário, a base de dados foi intencionalmente organizada de maneira oposta à ordem desejada. Ou seja, a lista foi classificada em ordem decrescente, sendo assim, no contexto do projeto: para o campo date houve a ordenação do mais recente para o mais antigo; para user foi do Z ao A; para count do maior para o menor. O objetivo desse caso de teste é avaliar o desempenho dos algoritmos diante da situação mais desfavorável, onde será necessário realizar o máximo de operações para ordenar os dados corretamente.

4. Ambientes de Execução

Os testes de desempenho dos algoritmos de ordenação foram realizados em dois ambientes de execução distintos, a fim de validar a consistência dos resultados e analisar eventuais variações provocadas por diferenças de hardware e sistema operacional. As configurações de cada ambiente são descritas a seguir:

a) **Primeiro Ambiente:**
   Processador (CPU): Intel(R) Core(TM) i5-10400F CPU @ 2.90GHz 2.90 GHz
   Memória RAM: 16 GB DDR4
   Sistema Operacional: Windows 10
   Armazenamento: SSD 512 GB
   Ambiente de Desenvolvimento: IntelliJ IDEA
   Versão do Java: Java 23 (OpenJDK)

b) **Segundo Ambiente:**
   Processador (CPU): Intel Core i5-13450HX 2.40 GHz
   Memória RAM: 16GB DDR5
   Sistema Operacional: Windows 11
   Armazenamento: SSD 512GB
   Ambiente de Desenvolvimento: intellij IDEA
   Versão do Java: Java 23

Durante a execução dos algoritmos de ordenação sobre a base de 1.048.575 tweets, foi observado que os algoritmos Insertion Sort e Selection Sort não

completavam a execução dentro de um tempo aceitável, em nenhuma das duas máquinas.

Essa limitação está relacionada à complexidade de tempo desses algoritmos, que é $O(n^2)$. Isso significa que para n registros, o número de operações cresce quadraticamente. No caso desta base com mais de 1 milhão de registros, seriam necessárias mais de 1 trilhão de comparações e movimentações de elementos — tornando a execução inviável na prática.

Além disso, foi identificado que a versão clássica do QuickSort com pivô fixo no último elemento (arr[dir]) apresenta risco de StackOverflowError ao ordenar grandes conjuntos de dados já ordenados ou parcialmente ordenados, devido à divisão desbalanceada das partições recursivas.

É de suma importância relatar que entre os dois ambientes de teste houve diferenças nos tempos de execução, como esperado devido às diferenças de hardware e sistema, mas sem impacto nas conclusões sobre a eficiência relativa dos algoritmos e a possível melhor escolha para cada caso.

5. 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**.

6. 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`**.

7. Resultados:

7.1: Campo Date

algoritmo	campo	caso	tempo
mergeSort	date	medioCaso	1973396300
mergeSort	date	melhorCaso	1709535600
mergeSort	date	piorCaso	1733630700
heapSort	date	medioCaso	5671191600
heapSort	date	melhorCaso	5182658500
heapSort	date	piorCaso	5597461300
quickSortMediar	date	medioCaso	2075288100
quickSortMediar	date	melhorCaso	2171646800
quickSortMediar	date	piorCaso	2199406400
QuickSort	date	medioCaso	1741032600
QuickSort	date	melhorCaso	2050124300
QuickSort	date	piorCaso	2489341000

No campo date, o algoritmo que obteve o melhor desempenho no melhor caso foi o Merge Sort, com um tempo de 1.709.535.600 ms. Isso reforça a eficiência do

algoritmo quando os dados estão pré-ordenados ou próximos da ordenação desejada. Sua abordagem estável e dividida recursivamente o torna altamente eficaz nesse cenário.

No caso médio, o Merge Sort se destacou, com 1.973.396.300 ms, sendo mais rápido que o Quick Sort (1.741.032.600 ms) e o Quick Sort com Mediana de Três (2.075.288.100 ms). Apesar de o Quick Sort tradicional ter mostrado um tempo ligeiramente menor, a diferença não é suficiente para superar a consistência e estabilidade do Merge Sort, principalmente levando em conta sua estabilidade (importante quando múltiplos campos estão envolvidos).

Já no pior caso, o algoritmo com melhor desempenho foi o Merge Sort, com 1.733.630.700 ns. Ele superou tanto o Heap Sort (5.597.461.300 ms) quanto o Quick Sort (2.489.341.000 ns) e o Quick Sort com Mediana de Três (2.199.406.400 ms). Isso mostra a robustez do Merge Sort mesmo em situações desfavoráveis, enquanto algoritmos como Heap Sort sofrem bastante em termos de desempenho nesse cenário.

7.2: Campo User

algoritmo	campo	caso	tempo
mergeSort	user	medioCaso	1116911700
mergeSort	user	melhorCaso	663107400
mergeSort	user	piorCaso	596138000
heapSort	user	medioCaso	2341151000
heapSort	user	melhorCaso	934557100
heapSort	user	piorCaso	1112516400
quickSortMedian	user	medioCaso	1238168300
quickSortMedian	user	melhorCaso	850601700
quickSortMedian	user	piorCaso	850417600
QuickSort	user	medioCaso	715302300
QuickSort	user	melhorCaso	1300712100
QuickSort	user	piorCaso	1690024400

No campo user, o algoritmo que apresentou o melhor desempenho no melhor caso foi o Merge Sort, com tempo de execução de 663.107.400 ms. Sua estabilidade e forma eficiente de divisão tornam-no particularmente eficaz quando os dados já estão ordenados ou quase ordenados.

No caso médio, o Quick Sort se destacou com 715.302.300 ms, superando algoritmos como Merge Sort (1.116.911.700 ms), Quick Sort com Mediana de Três (1.238.168.300 ms) e Heap Sort (2.341.151.000 ms). Isso mostra que, com boas escolhas de pivô e dados medianamente desordenados, o Quick Sort tradicional consegue entregar uma performance sólida e competitiva.

Já no pior caso, o algoritmo mais eficiente foi o Merge Sort novamente, com 596.138.000 ns, mostrando sua consistência mesmo em situações adversas. Enquanto isso, algoritmos como Quick Sort (1.690.024.400 ms) e Heap Sort (1.112.516.400 ms) apresentaram tempos significativamente maiores, o que confirma a vantagem do Merge Sort em manter um desempenho estável independentemente da ordem dos dados.

7.3: Campo Count

algoritmo	campo	caso	tempo
mergeSort	count	medioCaso	232281200
mergeSort	count	melhorCaso	219910700
mergeSort	count	piorCaso	222118800
countingSort	count	medioCaso	74461700
countingSort	count	melhorCaso	80995400
countingSort	count	piorCaso	59122100
heapSort	count	medioCaso	170435700
heapSort	count	melhorCaso	175214700
heapSort	count	piorCaso	169935700
quickSortMediar	count	medioCaso	202033600
quickSortMediar	count	melhorCaso	238335900
quickSortMediar	count	piorCaso	185773700
QuickSort	count	medioCaso	214135800
QuickSort	count	melhorCaso	238102300
QuickSort	count	piorCaso	277341000

No campo count, o algoritmo com melhor desempenho no melhor caso foi o Counting Sort, com tempo de execução de 80.995.400 ms. Esse resultado era esperado, uma vez que o Counting Sort é extremamente eficiente para campos com dados inteiros e faixas de valores limitadas — exatamente o caso de um campo como "count", que representa quantidades.

No caso médio, novamente o Counting Sort foi o mais eficiente, com 74.461.700 ms, superando todos os demais algoritmos por uma margem significativa. O Heap Sort (170.435.700 ms) e o Merge Sort (232.281.200 ms) ficaram muito atrás. Isso destaca a adequação perfeita do Counting Sort para esse tipo de dado, já que ele não depende de comparações e possui complexidade linear em relação ao tamanho do intervalo de entrada.

No pior caso, o Counting Sort continuou sendo o mais eficaz, com 59.122.100 ns, desempenho bastante estável mesmo sob condições desfavoráveis. Enquanto isso, algoritmos como o Quick Sort (277.341.000 ms) e o Quick Sort com Mediana de Três (185.773.700 ms) mostraram tempos consideravelmente maiores, refletindo sua menor previsibilidade de desempenho nesse cenário.

8. 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.

No campo user, o Quick Sort se mostrou o mais eficiente no médio caso, mas não conseguiu manter um desempenho tão bom no caso melhor e no pior caso, onde o Heap Sort foi o mais eficiente. O Heap Sort se mostrou uma escolha segura em contextos onde a estrutura dos dados é incerta. O Merge Sort também se destacou, em relação ao melhor caso, mantendo a consistência e robustez de seu desempenho.

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, o Quick Sort e o Quick Sort com Mediana de Três apresentaram resultados mais competitivos. No pior caso, o Quick Sort mostrou seu desempenho decaindo, enquanto o Heap Sort foi uma opção sólida, apesar da grande variabilidade do Quick Sort.

Essas análises destacam a importância de entender as características de cada algoritmo de ordenação e de realizar testes práticos antes de utilizá-los em ambientes de produção. Embora a teoria forneça uma base confiável, são os testes em condições reais que permitem validar qual algoritmo oferece o melhor

desempenho em diferentes contextos. Em resumo, a escolha do algoritmo ideal deve considerar não apenas a natureza dos dados, mas também as especificidades do problema e o comportamento observado durante os testes práticos.