Laboratório de Estrutura de Dados

Primeira versão do projeto da disciplina

Comparação entre os algoritmos de ordenação elementar

Guilherme Ribeiro Liebig

Raiff Ferreira Telecio

Vinicius Lopes De Morais

1. Introdução

Este relatório corresponde ao relato dos resultados obtidos no projeto da disciplina de LEDA , que teve como objetivo implementar, testar e comparar diferentes algoritmos de ordenação aplicados a um conjunto de dados reais extraído da plataforma Steam. A proposta do projeto está alinhada com os conteúdos teóricos da disciplina, buscando desenvolver habilidades práticas na análise de desempenho de algoritmos, manipulação de estruturas de dados e mensuração de eficiência computacional.

O projeto consistiu em duas grandes etapas. Na primeira, foi feita a transformação do dataset original com a padronização das datas de lançamento dos jogos (campo Release Date), onde foi alterado do modelo “MM DD, AAAA” para “DD/MM/AAAA”, gerando assim, o arquivo “games_formated_release_data.csv”, e a criação de subconjuntos filtrados, como jogos compatíveis com Linux, onde foi gerado o arquivo “games_linux.csv” e jogos com suporte ao idioma português, gerando o “portuguese_superted_games.csv”. Em seguida, na segunda etapa, foram aplicados sete algoritmos de ordenação — Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Quick Sort com mediana de três, Heap Sort e Counting Sort — em três atributos distintos dos jogos: data de lançamento, preço e número de conquistas. Os algoritmos foram avaliados em três cenários: entrada desordenada (caso médio), entrada ordenada (melhor caso) e entrada ordenada em ordem inversa (pior caso).

Como principais resultados, observou-se que algoritmos com complexidade O(n log n), como Merge Sort e Heap Sort, apresentaram desempenho mais eficiente e estável, especialmente em entradas maiores. O Quick Sort com mediana de três teve desempenho superior no caso médio, mas apresentou maior variação no pior caso. Já algoritmos de complexidade quadrática, como Selection e Insertion Sort, mostraram-se menos viáveis para grandes volumes de dados, embora sejam mais simples de implementar e compreender.

2. Descrição geral sobre o método utilizado

2.1 Como os Testes Foram Realizados:

- **Caso Médio**:
  Utiliza-se o dataset original (*games_formated_release_data.csv*), gerado pela transformação do CSV bruto.

- **Melhor Caso**:
  Obtém-se um array já ordenado aplicando primeiro o Insertion Sort ao caso médio e reutilizando esse resultado.

- **Pior Caso**:
  Consiste no array do melhor caso invertido, simulando o pior comportamento para algoritmos sensíveis à ordem de entrada.

2.2. Métricas Coletadas

- **Tempo de Execução**:
  Medido em milissegundos pela diferença entre duas chamadas a `System.currentTimeMillis()`, realizadas imediatamente antes e após cada algoritmo de ordenação.

- **Uso de Memória**:
  Avaliado utilizando a expressão `Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()`, capturando o delta de consumo de heap antes e depois da ordenação e convertendo o resultado para megabytes.

2.3. Repetibilidade

Cada teste é executado apenas uma vez, mas o uso da clonagem do array (`array.clone()`) assegura que todos os algoritmos partam da mesma entrada, evitando interferência entre eles.

O fluxo completo de execução pode ser iniciado por meio da chamada da classe Main.java, sem necessidade de intervenção manual em cada etapa.

---

3. Implementação da Ferramenta

3.1. Linguagem e Bibliotecas

A ferramenta foi desenvolvida em **Java 11**, utilizando a biblioteca **Apache Commons CSV** para leitura, escrita e parsing de arquivos CSV.

3.2. Estrutura de Pacotes

- **com.steamdatasetprojetoleda.transformações**:
  Contém a classe **TransformationsCSV**, responsável pela formatação de datas e geração de subconjuntos filtrados.

- **com.steamdatasetprojetoleda.ordenações**:
  Abriga as classes de ordenação (**AlgoritmosDate**, **AlgoritmosPrice**, **AlgoritmosAchievements**) e as classes de processamento (**OrdenacaoDate**, **OrdenacaoPrice**, **OrdenacaoAchievements**).

- **com.steamdatasetprojetoleda**:
  Contém o arquivo **Main.java**, responsável pela orquestração dos processos de transformação e ordenação.

3.3. Módulos Principais

- **TransformationsCSV**:

    - `transformReleaseDates(...)`: converte formatos variados de data para o padrão `dd/MM/yyyy`.

    - Métodos auxiliares de filtragem por coluna e valor, gerando arquivos como *games_linux.csv* e *portuguese_supported_games.csv*.

- **AlgoritmosX (Date, Price, Achievements)**:
  Cada classe implementa sete métodos de ordenação:
  Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Quick-3 Sort, Heap Sort e, especificamente para Achievements, também Counting Sort.

    - `carregarArray(Path)`: lê o CSV em memória e retorna um vetor de `CSVRecord`.

    - `salvarCSV(...)`: grava o resultado ordenado em um arquivo CSV específico.

- **OrdenacaoX (Date, Price, Achievements)**:
  Para cada atributo (data, preço ou conquistas), os sete algoritmos são executados sequencialmente. Antes de cada execução, o vetor original é clonado; após a ordenação, são coletadas as métricas de tempo e memória.
  O resultado é salvo em um arquivo CSV de saída, cujo nome indica o algoritmo e o cenário (exemplo: *games_price_quickSort_medioCaso.csv*).

Descrição geral do ambiente de testes

Os testes foram realizados em um Desktop com as seguintes configurações:

- Processador: Intel® Core™ i7-9700F CPU @ 3.00 GHz

● Memória RAM: 32 GB DDR4

● Placa de vídeo: NVIDIA GeForce RTX 2060 (6 GB VRAM)

● Armazenamento: ○ 932 GB HDD WDC WD10EZEX-00WN4A0 ○ 447 GB SSD (480 GB nominal)

● Sistema Operacional: Windows 64-bits, processador baseado em x64

● Java: OpenJDK 11.0.x

● Biblioteca CSV (Maven): Apache Commons CSV 1.8

● Ambiente de execução: testes conduzidos pela IDE Visual Studio Code

A soma total dos tempos de execução dos algoritmos foi: 20.432.748 milissegundos 20.432,748 segundos ≈ 5,68 horas

4. Resultados e Análise

Resultados observados das ordenações por data de lançamento

-Entrada desordenada

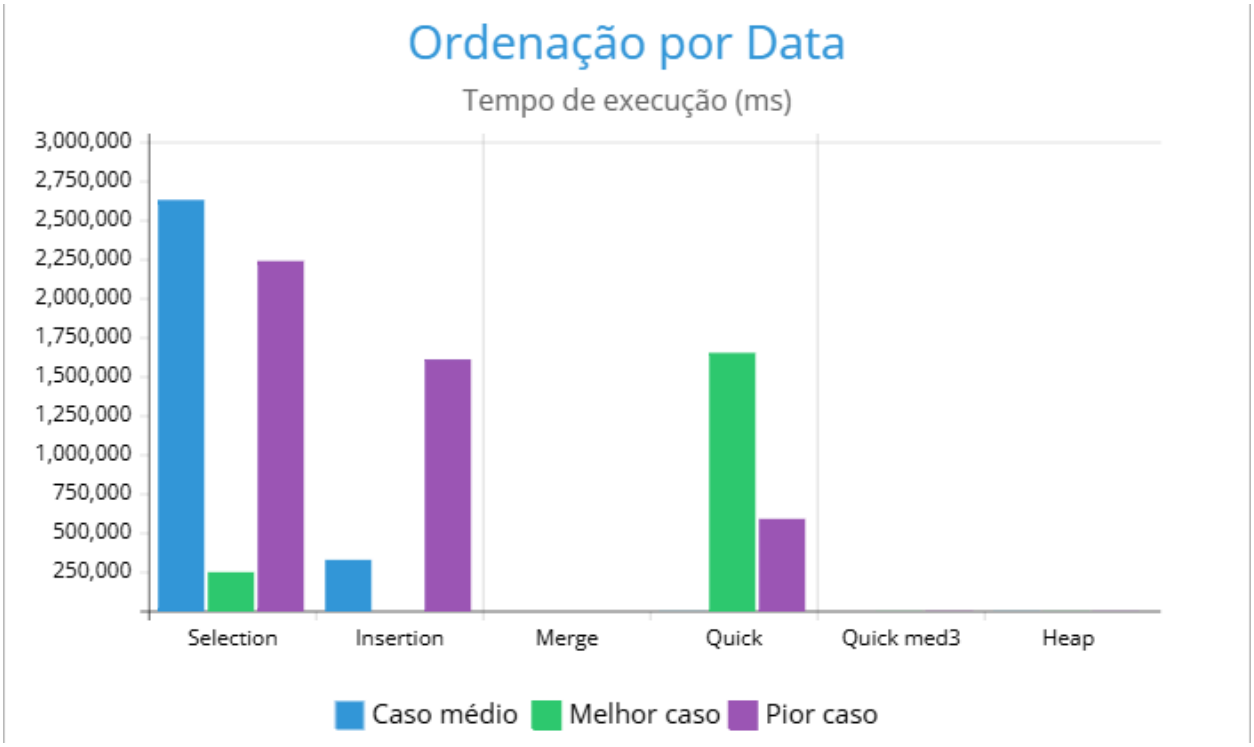
Ordenações por Date - Caso Médio		
Algoritmo	**Tempo (ms)**	**Memória (MB)**
Selection Sort	2.632.908	149
Insertion Sort	330.572	145
Merge Sort	703	227
Quick Sort	1.333	49
Quick Sort (Mediana de 3)	1.2	225
Heap Sort	1.432	303

-Entrada ordenada

Ordenações por Date - Melhor Caso		
Algoritmo	**Tempo (ms)**	**Memória (MB)**
Selection Sort	2.525.032	1017
Insertion Sort	87	96
Merge Sort	378	784
Quick Sort	1.654.741	202
Quick Sort (Mediana de 3)	1.023	997
Heap Sort	1.215	417

-Ordem inversa

Ordenações por Date - Pior Caso		
Algoritmo	**Tempo (ms)**	**Memória (MB)**
Selection Sort	2.242.152	306
Insertion Sort	1.613.141	703
Merge Sort	384	722
Quick Sort	592.836	414
Quick Sort (Mediana de 3)	2.851	497
Heap Sort	1.272	417



Resultados observados das ordenações por preços

-Entrada desordenada

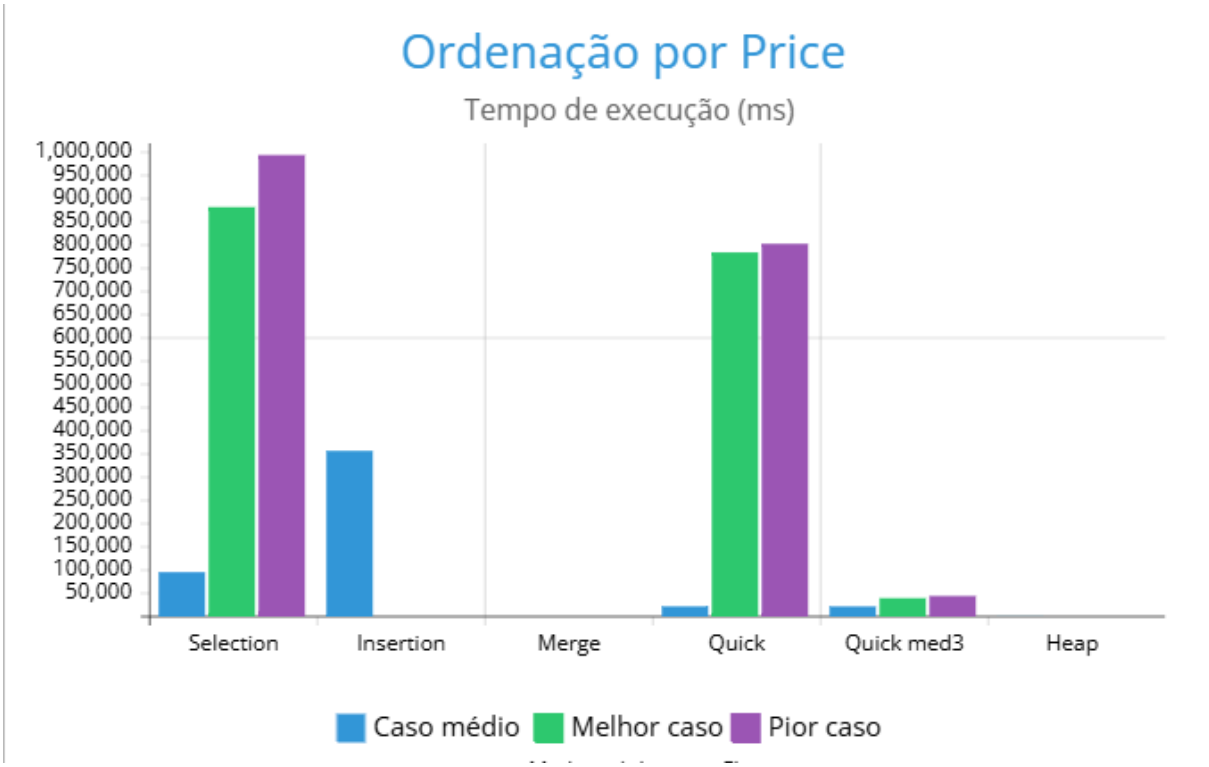
Ordenações por Price - Caso médio		
Algoritmo	**Tempo (ms)**	**Memória (MB)**
Selection Sort	955.68	70
Insertion Sort	356.846	84
Merge Sort	226	426
Quick Sort	21.814	97
Quick Sort (Mediana de 3)	21.767	91
Heap Sort	296	237

-Entrada ordenada

Ordenações por Price - Melhor caso		
Algoritmo	**Tempo (ms)**	**Memória (MB)**
Selection Sort	881.743	1090
Insertion Sort	22	9
Merge Sort	99	87
Quick Sort	784.262	614
Quick Sort (Mediana de 3)	39.584	419
Heap Sort	181	214

-Ordem inversa

Ordenações por Price - Pior caso		
Algoritmo	**Tempo (ms)**	**Memória (MB)**
Selection Sort	993.831	339
Insertion Sort	25	10
Merge Sort	109	86
Quick Sort	802.817	137
Quick Sort (Mediana de 3)	44.382	561
Heap Sort	182	214



Resultados observados das ordenações por achievements

-Entrada desordenada

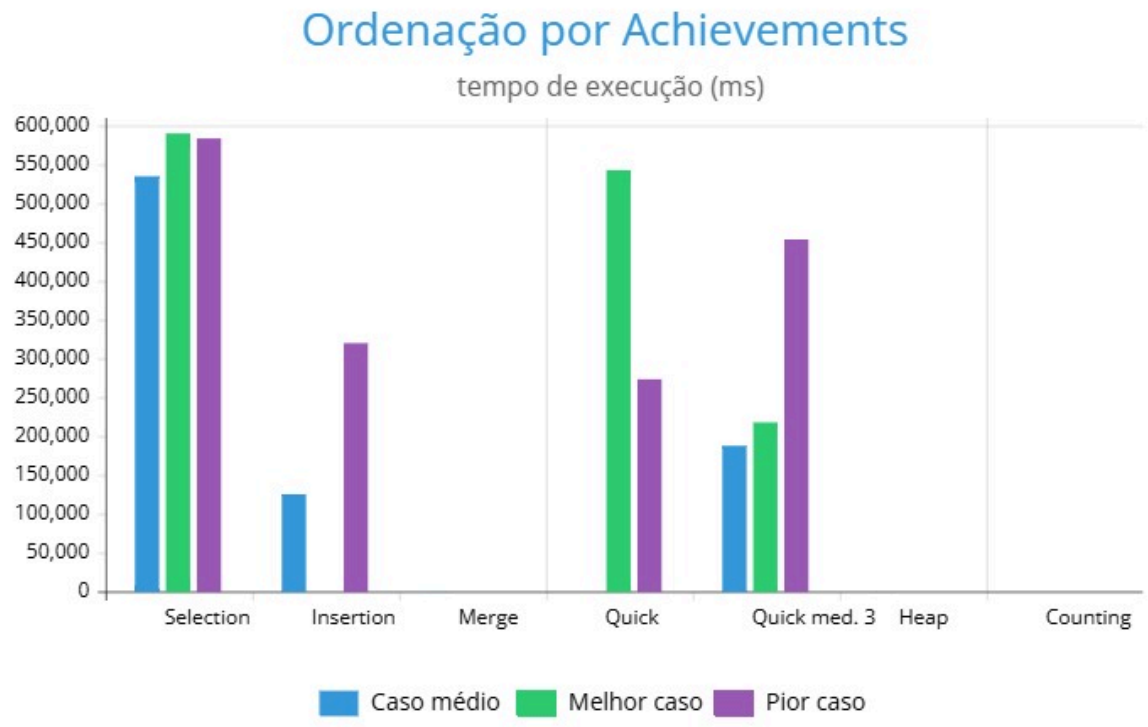
Achievements - Caso médio		
Algoritmo	**Tempo (ms)**	**Memória (MB)**
Selection Sort	535.741	0
Insertion Sort	126.083	0
Merge Sort	191	9
Quick Sort	87.16	0
Quick Sort (Mediana de 3)	188.566	1356
Heap Sort	168	0
Counting Sort	57	0

-Entrada ordenada

Achievements - Melhor caso		
Algoritmo	**Tempo (ms)**	**Memória (MB)**
Selection Sort	590.958	0
Insertion Sort	19	0
Merge Sort	58	10
Quick Sort	543.663	0
Quick Sort (Mediana de 3)	218.608	749
Heap Sort	121	0
Counting Sort	52	0

-*Ordem inversa*

Achievements - Pior caso		
Algoritmo	**Tempo (ms)**	**Memória (MB)**
Selection Sort	584.546	0
Insertion Sort	320.789	0
Merge Sort	81	9
Quick Sort	274.234	0
Quick Sort (Mediana de 3)	454.379	920
Heap Sort	91	0
Counting Sort	57	0



→ **Análise dos Resultados**

Nesta seção, são apresentados os algoritmos de ordenação que obtiveram melhor desempenho ao longo dos testes, seguidos de uma análise geral dos resultados obtidos. Na etapa de ordenação, foram gerados 57 arquivos .csv, mais especificamente, 18 arquivos quando executado a classe OrdenacaoDate.java, 18 arquivos na OrdenacaoPrice.java e 21 arquivos na OrdenacaoAchievements.java, onde nela, foi implementada o algoritmo Counting Sort, no qual não foi utilizado nas outras classes.

→ **Algoritmos Mais Eficientes**

Com base nas medições de tempo de execução e uso de memória nos três cenários (caso médio, melhor caso e pior caso), os algoritmos que apresentaram melhor desempenho foram:

- Merge Sort
- Heap Sort
- Quick Sort com mediana de três

Esses algoritmos mantiveram um comportamento eficiente e estável, independentemente da ordenação inicial dos dados. Tal eficiência se deve ao fato de todos apresentarem complexidade assintótica O(n log n), o que os torna especialmente indicados para conjuntos de dados grandes.

- Merge Sort destaca-se pela sua estabilidade e previsibilidade, sendo eficiente em qualquer tipo de entrada.

- Heap Sort também é estável e não utiliza memória extra significativa, funcionando bem mesmo no pior caso.

- Quick Sort com mediana de três superou os demais no caso médio, devido à sua escolha de pivô mais balanceada, reduzindo o risco de particionamentos desbalanceados.

Por outro lado, os algoritmos Selection Sort e Insertion Sort, ambos com complexidade $O(n^2)$, apresentaram desempenhos significativamente piores em entradas desordenadas, sendo mais adequados apenas para conjuntos de dados pequenos ou já parcialmente ordenados.

O Counting Sort apresentou excelente desempenho, mas seu uso foi restrito à ordenação do campo Achievements, devido à sua limitação de funcionamento apenas com inteiros em intervalos reduzidos.

→ **Análise Geral dos Resultados**

A análise dos resultados evidencia que:

- A complexidade teórica dos algoritmos foi confirmada na prática, com algoritmos O(n log n) se destacando largamente nos testes.

- Algoritmos simples como Selection Sort e Insertion Sort são didáticos, mas ineficientes para grandes volumes de dados. Mesmo no melhor caso, sua utilidade prática é limitada.

- A escolha do algoritmo ideal depende do tipo de dado e do caso analisado: Quick Sort com mediana é ótimo no caso médio, Merge Sort é mais estável no pior caso, e

Heap Sort se destaca pela robustez e simplicidade iterativa.

- O uso de memória foi, em geral, controlado. Embora algoritmos recursivos (como o Merge Sort) exijam mais chamadas de pilha, o consumo total de heap permaneceu aceitável em todos os testes.

- Ferramentas automatizadas para coleta de métricas foram fundamentais para garantir a reprodutibilidade dos experimentos, padronizando as medições de tempo e memória.

Com base nesses resultados, conclui-se que a escolha do algoritmo de ordenação deve considerar não apenas a complexidade teórica, mas também o cenário prático de aplicação, o tipo de dado manipulado e os recursos disponíveis na plataforma-alvo.