Laboratório de Estrutura de Dados

# Primeira versão do projeto da disciplina

## Comparação entre os algoritmos de ordenação elementar

Lucas Silva Nascimento

Manoel Barros da Cruz Neto

Vitor Francisco Farias Souza

# 1. Introdução

Este relatório corresponde ao relato dos resultados obtidos no projeto da disciplina de LEDA que trazem os resultados da comparação entre alguns algoritmos de ordenação, usando uma base de dados fornecida pelo professor, disponibilizada pela plataforma “kaggle”, contendo diversas informações sobre jogos do ambiente de jogo “steam”, sendo dados coletados por meio da Steam Spy, API desse ambiente. O objetivo do projeto, é apresentar e comparar o desempenho dos seguintes ordenação: Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Quick Sort com Mediana de 3, Counting Sort, e Heap Sort. Aplicados, nas situações demandadas pelo professor.

Por conseguinte, no projeto é implementado a transformação das datas de lançamento (campo: Release date) do formato “MM DD, AAAA” para o formato “DD/MM/AAAA”, gerando um arquivo chamado “games_formated_release_date.csv”. Analogamente, foi gerado um arquivo com todos os jogos compatíveis com o sistema operacional linux (campo: Linux) com base no arquivo previamente gerado (games_formated_release_date.csv). Finalmente, é feito um novo arquivo com filtro de jogos que tem suporte a língua portuguesa (campo: Supported languages) utilizando novamente o arquivo das datas de lançamento. Feitas as devidas transformações, foram adaptados os métodos supracitados com finalidade de executar as seguintes ordenações:

- Ordenar pela data de lançamento de forma crescente;
- Ordenar por preço dos jogos de forma crescente;
- Ordenar pelo número de conquistas de forma decrescente.

Além disso, foram propostos três casos para os quais os métodos de ordenação seriam submetidos. São eles: o dataset original (sem nenhuma ordenação prévia), o dataset de maneira já ordenada e o dataset ordenado na sua forma contraria, de maneira que esteja totalmente desordenado.

Ademais, abre-se uma observação, counting sort foi usado apenas para ordenar para a ordenação com base nos números de conquistas. Pois, a implementação do próprio algoritmo exige uma etapa de “contagem” da ocorrência dos valores durante o array, ou seja, entre valores do tipo double, é inviável sua aplicação, uma vez que existem infinitos números entre dois valores inteiros. Desse principio, a ordenação com base nas datas de lançamento, também não foi submetida a esse algoritmo de ordenação.

Em síntese, os testes comprovaram os resultados esperados para cada método de ordenação. Nos quais, pudemos observar que dentre os algoritmos in place simples, nos casos onde o array já estava ordenado, o Insertion Sort obteve baixo tempo de execução. Pois, assim como esperado em seu melhor caso, percorre a lista uma única vez tendo complexidade $O(n)$. Além disso, também pudemos

comprovar que o Selection Sort, conforme já esperado, manteve sua complexidade $O(n^2)$ em todos os casos ao qual foi submetido.

Em relação aos algoritmos mais complexos, os testes confirmaram os comportamentos esperados. O Merge Sort, sendo um algoritmo não in-place, manteve sua complexidade de *O(n log n)* em todos os cenários. O Quick Sort, ao ser aplicado em arrays ordenados, evidenciou seu pior desempenho quando o pivô foi escolhido de forma inadequada, resultando em $O(n^2)$, enquanto a estratégia com mediana de 3 corrigiu esse problema, garantindo um tempo de execução eficiente de *O(n log n)*. Para o Counting Sort, por ser um algoritmo baseado em contagem, mesmo com arrays ordenados, manteve sua complexidade *O(n + k)*. Por fim, o Heap Sort apresentou comportamento consistente como esperado, confirmando sua complexidade de *O(n log n)*.

Resumidamente, o relatório estruturou-se da seguinte forma, foi inicialmente apresentado detalhes de como se deu a implementação do projeto. Posteriormente, os objetivos do projeto, junto a apresentação resumida dos resultados conquistados. Nesse sentido, seguiu-se expondo uma descrição geral sobre a implementação do projeto, descrevendo como se deu o desenvolvimento do código para conquistar os objetivos propostos, bem como, a descrição e especificação do ambiente ao qual o submeteu-se a execução do código. Por fim, foram expostos os resultados, bem como dissertado e analisado cada resultado com apresentação de tabelas e gráficos, para chegar as conclusões coerentes.

## 2. Descrição geral sobre o método utilizado

Para o desenvolvimento desse projeto, foi criado um projeto Maven na IDE Intellij. Incrementando a API Apache Commons CSV, aplicada para leitura e escrita dos arquivos separados por vírgulas, por meio, de objetos “Writer”, “Reader”, “CSVPrinter” e “CSVParser”. Na etapa de transformações, foram convertidas as datas utilizando classes da biblioteca “java.time”, dividindo a etapa de transformações em 4 funções principais: convertDate(), toReleaseDate(), getLinuxGames() e getPortugueseGames().

De modo que, a função convertDate() é utilizada pelo método toReleaseDate() que tem funcionamento semelhante aos demais métodos. Com isso, cada um dos métodos, exceto o que executa a conversão das datas tem seu funcionamento da seguinte forma, todo o csv é lido e armazenado em um objeto CSVParser e sobre ele é iterado até o fim de todas as linhas fazendo as devidas verificações para efetuar as transformações, tudo isso na classe criada, chamada de “Transformations.java”.

Então, feitas as transformações, efetuam-se as ordenações. De maneira, que para cada tipo de ordenação solicitado foram criadas classes para cumprir com todas as tarefas(OrdenationsByDate.java, OrdenationsByAchievements.java e OrdenationsByPrice.java). Nas

quais, todas possuem funcionamento bem próximo, exceto suas ordenações, onde cada uma possui seus métodos de ordenação adaptados para cumprirem suas necessidades.

Assim, cada uma dessas classes tem uma função independente de leitura e escrita de arquivos separados por vírgulas, para serem lidos e armazenados em um array do tipo CSVRecord retornado pela função getArray(). Logo, é chamado cada método de ordenação recebendo como parâmetro um clone desse array, marcando também o tempo de execução de cada método. Após finalizada cada ordenação, é exibido o tempo de execução e a partir do array ordenado é criado um novo arquivo separado por vírgulas contendo o dataset ordenado conforme solicitado em cada etapa, pela função writeToCsv() que cria o arquivo com o nome devido.

Assim, para o pleno funcionamento do código foi criada uma classe Main.java que invoca o método principal de cada uma das classes anteriores, garantindo assim que o código funcione perfeitamente. Ademais, vale ressaltar que todos os caminhos para os arquivos foram criados usando um objeto Path, garantindo sua portabilidade em diversos sistemas operacionais.

### Descrição geral do ambiente de testes

Os testes foram realizados em um notebook com o sistema operacional Ubuntu 20.04 (dual boot), possuindo 16gb (gigabytes) de memória ram, um processador AMD A10-9600P Radeon R5, frequência de 2.40GHz. A IDE utilizada para os testes foi o Intellij Community 2024.2.1, com o aplicativo instalado no SSD de 512gb. O tempo total para execução de todos os testes foi, aproximadamente, 51.331.443 milissegundos, ou 51331,443 segundos (aproximadamente 14 horas). Vale ressaltar, que o notebook esteve a todo momento ligado na tomada e somente com a IDE em execução, assim garantindo a melhor análise possível.

## 3. Resultados e Análise

Resultados observados das ordenações por data de lançamento, em ordem crescente:

Tabela 1: Tempo de execução para ordenações por datas.

| ORDENAÇÃO POR DATAS | | | |
|---|---|---|---|
| DATASET | DECRESCENTE | DESORDENADO | CRESCENTE |
| TIPO DE ORDENAÇÃO | TEMPO DE EXECUÇÃO (ms) | TEMPO DE EXECUÇÃO (ms) | TEMPO DE EXECUÇÃO (ms) |
| SELECTION SORT | 6500139 | 6759165 | 6793006 |
| INSERTION SORT | 4020298 | 1026800 | 150 |
| MERGE SORT | 1528 | 1878 | 1175 |
| QUICK SORT | 1665988 | 3578 | 4149878 |
| QUICK SORT(MEDIANA DE 3) | 8527 | 3110 | 3208 |

| | | | |
|---|---|---|---|
| HEAP SORT | 4090 | 4578 | 4257 |

Fonte: Tabela do autor.

Gráfico 1: Tempo de execução utilizado para cada método relacionado a preços.


Fonte: Gráfico do autor.

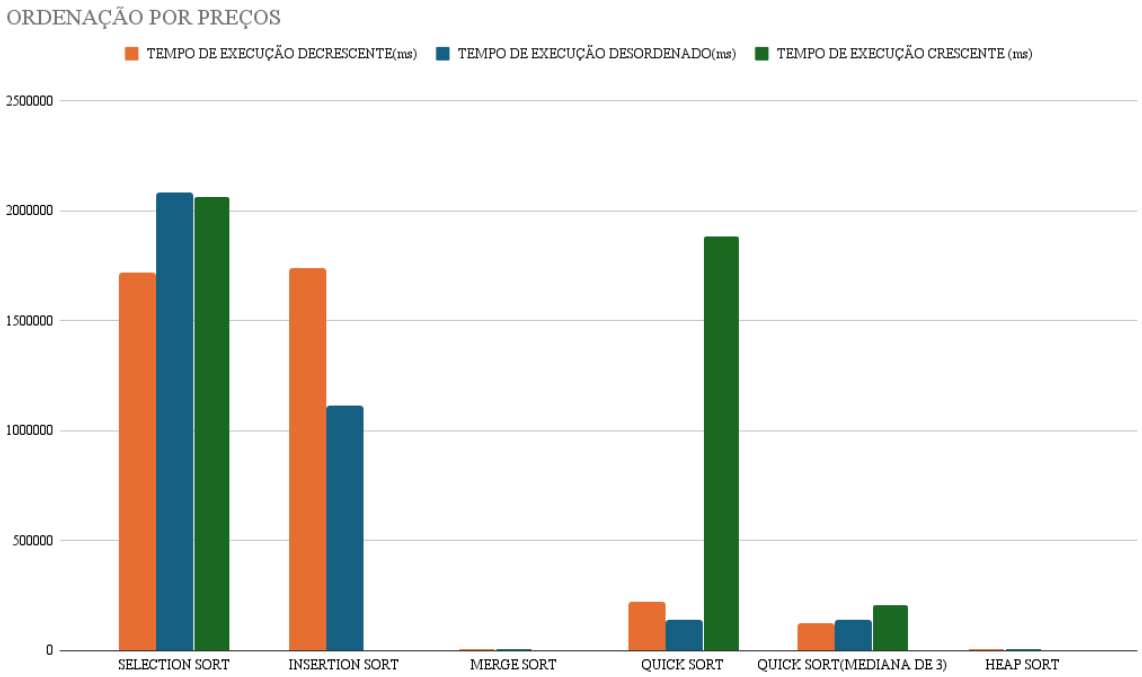
Resultado observados das ordenações por preços, em ordem crescente:

Tabela 2: Tempo de execução para ordenações por preços.

| ORDENAÇÃO POR PREÇOS | | | |
|---|---|---|---|
| DATASET | DECRESCENTE | CRESCENTE | ORDENADO |
| TIPO DE ORDENAÇÃO | TEMPO DE EXECUÇÃO (ms) | TEMPO DE EXECUÇÃO (ms) | TEMPO DE EXECUÇÃO (ms) |
| SELECTION SORT | 1714682 | 2077792 | 2065434 |
| INSERTION SORT | 1733504 | 1110704 | 50 |
| MERGE SORT | 231 | 434 | 369 |
| QUICK SORT | 215568 | 131756 | 1883280 |
| QUICK SORT(MEDIANA DE 3) | 119615 | 134999 | 204290 |
| HEAP SORT | 423 | 687 | 481 |

Fonte: Tabela do autor.

Gráfico 2: Tempo de execução utilizado para cada método relacionado a preços.




Fonte: Gráfico do autor.

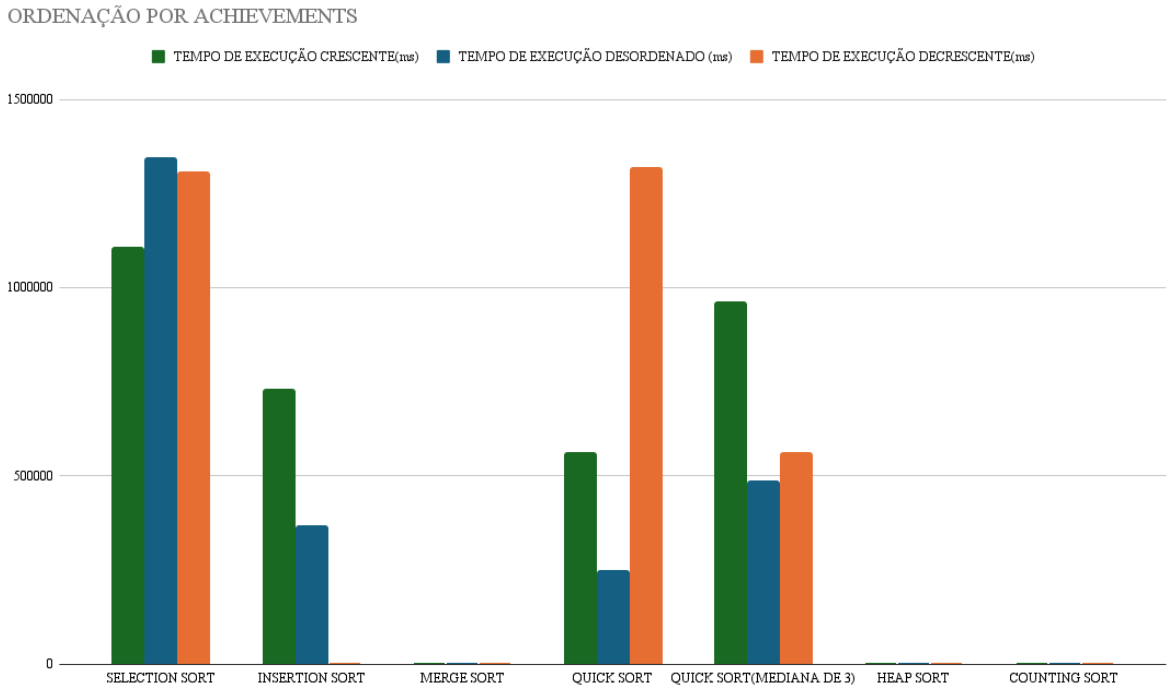
Resultados observados das ordenações por achievements, feita em ordem decrescente:

Tabela 3: Tempos de execução das Ordenações por achievements.

| ORDENAÇÃO POR ACHIEVEMENTS | | | |
|---|---|---|---|
| | CRESCENTE | DESORDENADO | DECRESCENTE |
| TIPO DE ORDENAÇÃO | TEMPO DE EXECUÇÃO (ms) | TEMPO DE EXECUÇÃO (ms) | TEMPO DE EXECUÇÃO (ms) |
| SELECTION SORT | 1106689 | 1342809 | 1306722 |
| INSERTION SORT | 729095 | 365979 | 40 |
| MERGE SORT | 241 | 232 | 198 |
| QUICK SORT | 561091 | 247246 | 1318445 |
| QUICK SORT(MEDIANA DE 3) | 960363 | 484811 | 560572 |
| HEAP SORT | 267 | 391 | 311 |
| COUNTING SORT | 100 | 99 | 90 |

Fonte: Tabela do autor.

Gráfico 3: Tempo de execução utilizado para cada método relacionado à Achievements.




Fonte: Gráfico do autor.

Com base nos resultados obtidos, conseguimos conquistar todos os objetivos propostos dentro do esperado. Com isso, pudemos observar que o algoritmo de ordenação Selection Sort, teve sua complexidade $O(n^2)$ mantida para todos os casos ao qual foi submetido, assim, possuindo o pior desempenho em todos os casos, comparado aos outros métodos. Desse modo, confirmando sua ineficiência para bases de dados grandes em qualquer caso.

Além disso, pudemos observar a variação de desempenho do Insertion Sort, uma vez que para os casos onde o dataset já estava ordenado conforme solicitado, ele obteve seu melhor desempenho tendo como complexidade $O(n)$ sobressaindo-se sobre os demais. Entretanto, quando foi submetido aos outros casos de ordenações, teve resultado discrepante com o já citado, comprovando sua complexidade $O(n^2)$ e denotando sua ineficiência para bases de dados extensas e desordenadas.

Ademais, sobre o Merge Sort foi comprovada sua estabilidade de desempenho em todos os casos onde foi submetido, nos quais, pudemos provar que sua complexidade mantém-se constante em todas as ocasiões, sendo $O(nlogn)$ para todos os casos. De modo, que se torna eficiente em todos os casos submetidos.

Analogamente, analisando o desempenho do Quick Sort pudemos observar seu péssimo desempenho para bases já ordenadas, onde ocorre um desbalanceamento das partições, evidenciando sua complexidade $O(n^2)$. Contemplando assim, um desempenho discrepante para o caso em questão,

com relação aos demais casos aos quais foi submetido, que quando possuem um balanceamento das partições obtém complexidade O*(nlogn)*, melhorando consideravelmente o desempenho.

Além disso, corrigindo o problema de escolha do pivô, que podia gerar um desbalanceamento das partições do Quick Sort, foi implementada uma abordagem de mediana de 3 valores. Desse modo, balanceando o particionamento e mantendo uma complexidade menor para o algoritmo, levando sua complexidade em ambos casos para *O(nlogn)*, assim melhorando seu desempenho com relação à abordagem anterior do mesmo.

Bem como o Merge Sort, o Heap Sort manteve seu desempenho constante em ambos os casos aos quais foi submetido. Demonstrando, por meio de sua boa performance e desempenho, sua complexidade *O(nlogn)* para todas as bases de dados as quais teve que ordenar. Embora seja estável, ainda foi inferior ao Merge Sort.

Sobre o Counting Sort, vale ressaltar que sua implementação foi aplicada apenas para as ordenações de dados do tipo inteiro, de maneira a qual foi aplicada apenas para as ordenações por quantidade de conquistas. Entretanto, pode-se evidenciar seu excelente desempenho em todos os cenários aos quais foi submetido, demonstrando sua complexidade *O(n+k)* em todos os casos.

Diante do exposto, pudemos comprovar que o desempenho de cada método de ordenação pode variar conforme a base de dados ao qual é submetida. Então, com base nos resultados observamos que os métodos de ordenação Selection Sort e Insertion Sort são ineficientes para grandes bases de dados, com exceção ao caso onde o dataset já está ordenado para o Insertion Sort que evidencia seu melhor caso, apenas efetuando as comparações simples entre os elementos.

Nesse viés, pudemos constatar que o Merge Sort pode ser uma escolha extremamente sólida em questão de desempenho, independentemente da disposição dos dados. Além disso, sobre o Quick Sort pudemos ver sua discrepância de desempenho com relação à abordagem de pivoteamento, onde, quando aplicada a estratégia de mediana de 3, tornou-se eficiente em desempenho. Entretanto, quando não aplicada essa estratégia, torna-se inviável sua utilização para bases ordenadas.

Por fim, embora com desempenho inferior ao Merge Sort, o Heap Sort manteve seu desempenho constante em todos os casos, destacando sua estabilidade em termos de eficiência. Além disso, o Counting Sort embora limitado em sua implementação, mostrou-se constante e eficaz em todos os casos aos quais foi submetido, sendo eficiente independente do dataset.