Laboratório de Estrutura de Dados

Segunda versão do projeto da disciplina

Comparação entre os algoritmos de ordenação elementar

T5-Steam Games Dataset

Arthur Barbosa, Lucas Defensor e Manoel Firmino

• Introdução

Este relatório corresponde ao relato dos resultados obtidos no segundo projeto da disciplina de LEDA (LABORATÓRIO DE ESTRUTURA DE DADOS) que teve como principal objetivo a implementação de estruturas de dados que serão aplicadas no projeto, além de aplicar técnicas de transformação, filtragem e ordenação de dados sobre um conjunto real extraído da plataforma Steam, referente a jogos. A proposta visa consolidar o conhecimento sobre as várias estruturas de dados e suas características, algoritmos de ordenação, além de manipulação de arquivos CSV. Para realização dessa segunda parte do projeto foram utilizadas as transformações que foram apresentadas na primeira parte do mesmo projeto . Por fim, realizou-se ordenações pela data de lançamento em ordem crescente, pelo preço em ordem crescente e pelo número de conquistas, em ordem decrescente. Como resultado, foram gerados diversos arquivos CSV contendo os dados transformados, filtrados e ordenados conforme as especificações. Os testes demonstraram a variação de desempenho da utilização das diferentes estruturas de dados que foram introduzidas no projeto para buscar uma melhora no tempo, dos algoritmos de ordenação de acordo com o tipo de dado e a situação (melhor, médio ou pior caso). O Counting Sort não foi utilizado para ordenar os campos de “Release date” e “Price” nos dataset “games.csv”, pois é um algoritmo de ordenação eficiente e apropriado para ordenar valores inteiros não-negativos que estão no intervalo conhecido e relativamente pequeno.

• Descrição geral sobre o método utilizado

Descrição dos Testes Realizados

Para avaliar o desempenho dos algoritmos, foram realizados testes com diferentes conjuntos de dados (obtidos das filtragens e transformações descritas anteriormente). Os testes seguiram os seguintes critérios:

• Medição de tempo de execução: Utilizamos a ferramenta VisualVM, conforme especificado na descrição do projeto, para monitorar e registrar o tempo de execução dos algoritmos. Essa aplicação permitiu a análise precisa do desempenho em termos de tempo e uso de recursos.

• Cenários testados**:**

- Melhor caso
- Caso médio
- Pior caso

• Critérios de avaliação:

- Tempo de execução (em segundos).
- Estimativa de custo teórico (complexidade assintótica dos algoritmos).
- Comparação entre abordagens de divisão e conquista (QuickSort, MergeSort) e métodos mais simples (SelectionSort, InsertionSort, etc.)

Foram geradas tabelas contendo os tempos de execução médios de cada algoritmo, variações conforme o tamanho do dataset, e considerações sobre eficiência prática, comparando o comportamento esperado (teórico) com os resultados obtidos.

Descrição da Implementação da Ferramenta

A ferramenta foi desenvolvida na linguagem Java, utilizando bibliotecas padrão para manipulação de arquivos e estruturas de dados. O foco principal da implementação foi:

• **Leitura e transformação de dados** a partir de arquivos CSV originais extraídos da plataforma Steam.

• **Conversão de datas** no campo “Release date” para o formato brasileiro padrão (DD/MM/AAAA).

• **Filtragem de dados** com base em dois critérios: compatibilidade com o sistema operacional Linux e suporte ao idioma português.

• **Ordenação de dados** utilizando diferentes algoritmos, conforme os critérios:

  • Data de lançamento (ordem crescente).

  • Preço do jogo (ordem crescente).

  • Quantidade de conquistas (ordem decrescente).

A estrutura geral da ferramenta foi dividida em módulos principais:

  • **Leitor de CSV**: responsável por carregar e processar o arquivo original.

  • **Transformador de Dados**: realiza conversões de tipos e formatos.

  • **Filtro**: aplica as condições de seleção.

  • **Ordenador**: implementa diferentes algoritmos de ordenação (como QuickSort, MergeSort, SelectionSort, InsertionSort e BubbleSort).

• **Exportador de CSV:** gera arquivos com os dados resultantes das etapas anteriores.

• Estruturas de Dados

- **Lista Duplamente Encadeada**

A lista duplamente encadeada foi utilizada para realizar a leitura do arquivo CSV na ordenação referente a preço, e depois transformar em array para a ordenação ser realizada. O problema que foi resolvido com a implementação dessa estrutura foi que não precisamos saber o tamanho exato do arquivo que estamos tratando, assim, nos poupando de realizar um loop extra sobre o arquivo inteiro apenas para saber seu tamanho e só assim criar o array definitivo que vai armazenar os dados do arquivo. Realizamos a transformação para array para realizar a ordenação devido aos custos da maioria dos métodos de presentes na lista duplamente encadeada, por exemplo o método “get” têm um custo O(n) e por isso a utilização de arrays se mostra muito mais eficiente uma vez que o custo do mesmo método referente ao array é O(1) . A implementação se mostra na imagem a seguir:

```
ListaDuplamenteEncadeada<CSVRecord> listaDuplamenteEncadeada = new ListaDuplamenteEncadeada<>();
for(CSVRecord record : parser) {
    if(record.getRecordNumber() == 1){
        escritorDeArquivo.printRecord(record);
    }
    else if(record.size() > 2){
        listaDuplamenteEncadeada.insert(record);

    }

}
```

- **Árvore AVL**

A árvore AVL também foi utilizada para realizar a leitura do arquivo CSV só que desta vez na ordenação referente a conquistas, e depois transformar em array para a ordenação ser realizada. A árvore foi implementada com base na comparação do ID de cada jogo, como está apresentado no comparador logo na imagem abaixo, para não ter nenhuma interferência na ordenação que vai ser realizada logo depois, referente às

conquistas dos jogos. O problema que foi resolvido com a implementação dessa estrutura foi a mesma da anterior, ou seja, que não precisamos saber o tamanho exato do arquivo que estamos tratando, assim, nos poupando de realizar um loop extra sobre o arquivo inteiro apenas para saber seu tamanho e só assim criar o array definitivo que vai armazenar os dados do arquivo. Realizamos a transformação para array para realizar a ordenação devido aos custos da maioria dos métodos de presentes na lista duplamente encadeada, por exemplo o método “get” têm um custo O(log n), que apesar de ser mais eficiente que a primeira estrutura utilizada, não chega a ser mais eficiente que o custo do mesmo método em uma array, por isso se justifica a utilização de arrays para realizar a ordenação do arquivo. A implementação se mostra na imagem a seguir:

```
Comparator<CSVRecord> comparadorPorId = (record1, record2) -> {
    String idStr1 = record1.get(0);
    String idStr2 = record2.get(0);

    Integer id1 = Integer.parseInt(idStr1.trim());
    Integer id2 = Integer.parseInt(idStr2.trim());

    return id1.compareTo(id2);
};

FileReader leitorFinal = new FileReader(caminhoArquivoParaSerLido);
CSVPrinter escritorDeArquivo = new CSVPrinter(new FileWriter(CAMINHO_ARQUIVO_GERADO, true), CSVFormat.DEFAULT);
CSVParser parser = CSVFormat.RFC4180.parse(leitorFinal);

ArvoreAVL<CSVRecord> arvore = new ArvoreAVL<>(comparadorPorId);


for(CSVRecord record : parser) {
    if(record.getRecordNumber() == 1){
        escritorDeArquivo.printRecord(record);
    }
    else if(record.size() > 2){
        arvore.inserir(record);
    }

}
```

- **Fila**

A fila, por sua vez, foi utilizada para realizar a escrita do arquivo CSV na ordenação referente a preço e também de conquistas. O problema que foi resolvido com a implementação dessa estrutura foi que os elementos que se encontram na fila já estão na ordem exata que devem ser inseridos no arquivo CSV, dificultando algum erro que insira algum elemento diferente, já que em uma fila só temos acesso ao elemento que se encontra na cabeça da fila naquele momento, ao contrário do array que podemos acessar qualquer posição e qualquer erro no acesso dos elementos poderia causar um erro na inserção dos elementos dentro do arquivo. A implementação se mostra na imagem a seguir:

```
for(int k = 0; k < filaEscrita.getQuantidadeDeElementos(); k++){
    escritorDeArquivo.printRecord(filaEscrita.desenfileirar());
}
```

Descrição geral do ambiente de testes

Os testes foram realizados em um computador com as seguintes configurações de hardware e software:

• Processador: Intel Core i5 (13ª geração)

• Memória RAM: 8 GB

• Armazenamento: SSD de 256 GB

• Sistema Operacional: Windows 10 Pro, 64 bits

• Resultados e Análise

Apresentaremos uma análise comparativa do desempenho dos algoritmos Merge Sort, Quick Sort, Quick Sort com Mediana de 3, Selection Sort, Insertion Sort, Counting Sort e

Heap Sort utilizando também agora as três estruturas de dados que foram implementadas nessa segunda parte do projeto. Todos os algoritmos foram testados utilizando a ferramenta VisualVM, que permitiu a medição precisa do tempo de execução e do consumo de recursos durante o processamento dos dados.

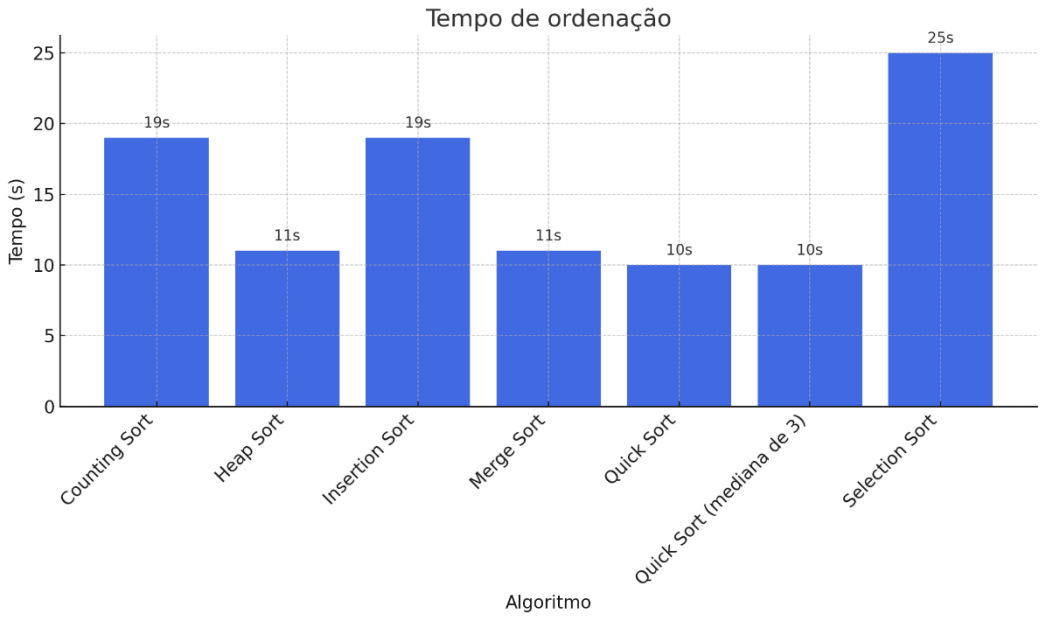
Foram gerados gráficos a partir dos resultados obtidos, abrangendo os critérios de tempo de ordenação por número de conquistas, custo computacional e ordenação por datas de lançamento. Esses gráficos servem de base para a comparação da utilização das diferentes estruturas de dados e entre os algoritmos, permitindo avaliar seu comportamento prático em diferentes situações e sobre distintos tipos de dados extraídos do dataset de jogos da Steam.

- **Conquistas**

O gráfico a seguir apresenta o tempo de execução (em segundos) dos algoritmos de ordenação aplicados ao campo "número de conquistas" dos jogos, com base em um conjunto de dados real extraído da plataforma Steam.

Tempos de ordenação (em segundos)

Algoritmo	Tempo (segundos)
Counting Sort	19
Heap Sort	11
Insertion Sort	19
Merge Sort	11
Quick Sort	10
Quick Sort Mediana 3	10
Selection Sort	25



Análise de desempenho (do melhor para o pior):

• O **Quick Sort com mediana de 3** foi o algoritmo com melhor desempenho nos testes, apresentando o menor tempo de execução geral. Sua estratégia de escolha do pivô pela mediana de três valores reduziu significativamente os riscos de particionamentos desbalanceados, aumentando a eficiência na prática.

• O **Quick Sort** também se destacou, empatando em tempo com sua versão otimizada. Sua estrutura baseada em divisão e conquista permitiu uma ordenação rápida e eficiente, especialmente em dados aleatórios. O algoritmo se mostrou eficaz, mantendo desempenho equivalente ao da sua versão com mediana, mas com implementação mais simples..

• O **Merge Sort** apresentou desempenho muito estável, com tempo de execução semelhante ao Quick sort (normal e mediana de 3). Sua complexidade garantida de O(nlogn) e comportamento consistente, independentemente da distribuição dos dados, o tornam uma das alternativas mais confiáveis.

• O **Heap Sort**, apesar de sua complexidade teórica igual à do Merge Sort, apresentou desempenho prático inferior. Sua estrutura baseada em heap adiciona um custo extra

de reorganização durante o processo de ordenação. Ainda assim, manteve um tempo estável.

• O **Insertion Sort** surpreendeu positivamente, considerando que sua complexidade é $O(n^2)$. As otimizações aplicadas no código, como a separação entre os dados completos e os valores a serem ordenados, conseguiu tempos de execução muito bons. Embora normalmente seja ineficiente para grandes volumes de dados.

• O **Counting Sort**, apesar de possuir complexidade linear O(n+k) teve desempenho abaixo do esperado nos testes realizados. Isso ocorreu devido à grande faixa de valores presente no campo de dados analisado, o que aumentou o custo de alocação de memória e processamento interno.

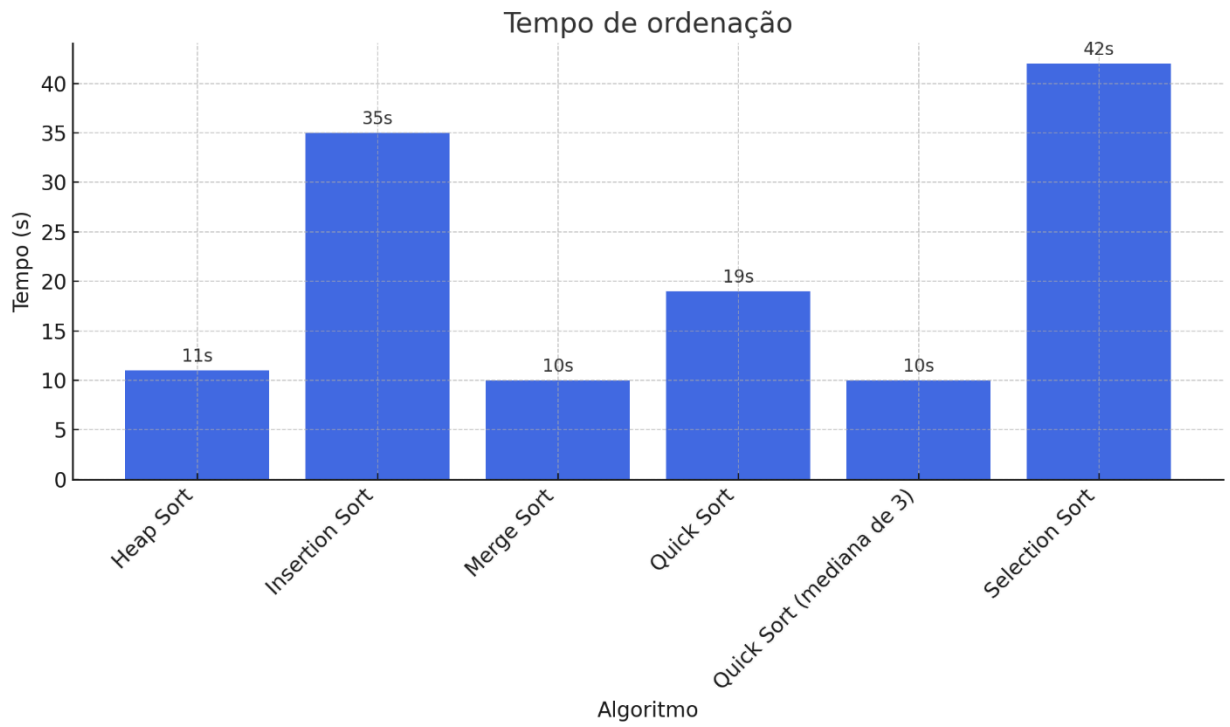
• O **Selection Sort** foi o algoritmo que apresentou o pior desempenho geral, registrando o maior tempo de execução entre todos. Mesmo com uma estrutura simples e previsível, sua complexidade quadrática e o grande número de comparações e trocas necessárias afetou negativamente a performance.

• Preços

A seguir, são apresentados os gráficos que comparam o tempo de execução dos algoritmos na tarefa de ordenação pelo campo "preço" dos jogos. Essa análise visa observar como cada algoritmo se comporta diante de dados numéricos contínuos, como é o caso dos preços, que variam amplamente em valor e distribuição.

Tempos de ordenação (em segundos)

Algoritmo	Tempo (segundos)
Heap Sort	11
Insertion Sort	35
Merge Sort	10
Quick Sort	19
Quick Sort Mediana 3	10
Selection Sort	42



A ordem de desempenho, do melhor para o pior, foi a seguinte:

• O **Merge Sort** Apresentou um dos melhores desempenho geral na ordenação por preço, foi um dos melhores tempos, ele se beneficiou de sua complexidade garantida de O(nlogn) e comportamento estável, independente da ordenação inicial dos dados.

• O **Quick Sort com mediana de 3** também foi o outro que apresentou desempenho excelente, com tempo igual ao merge sort, ele mostrou novamente sua eficácia ao evitar os piores casos por meio da escolha refinada do pivô, entregando alta performance e equilíbrio em diversos cenários.

• O **Heap Sort** apareceu logo em seguida, com tempo de 11 segundos. Apesar de sua estrutura ser mais complexa, baseada em árvores binárias de heap, seu desempenho foi razoavelmente bom. A reorganização constante dos elementos durante a ordenação gerou um pequeno custo adicional, mas ainda assim ele manteve-se com um ótimo tempo.

• O **Quick Sort** apresentou tempo de 19 segundos, teve desempenho consideravelmente inferior ao de sua versão com mediana de 3. Isso indica que, neste

caso, a escolha do pivô tradicional não conseguiu dividir bem os dados, resultando em particionamentos desequilibrados e maior profundidade recursiva. Mesmo sendo um algoritmo geralmente rápido, sua eficiência foi prejudicada neste cenário específico.

• O **Insertion Sort**, com 35 segundos, apresentou desempenho bastante inferior aos demais, como esperado pela sua complexidade $O(n^2)$. Ainda que possa ser eficiente para conjuntos pequenos ou quase ordenados, seu tempo de execução se elevou consideravelmente com o aumento dos dados, mostrando as limitações desse algoritmo em aplicações maiores.

• O **Selection Sort** foi o algoritmo com pior desempenho, registrando o maior tempo de execução: 42 segundos. Seu funcionamento, que exige múltiplas passagens completas pelos dados para encontrar o menor valor restante, torna-o extremamente ineficiente em grandes volumes.

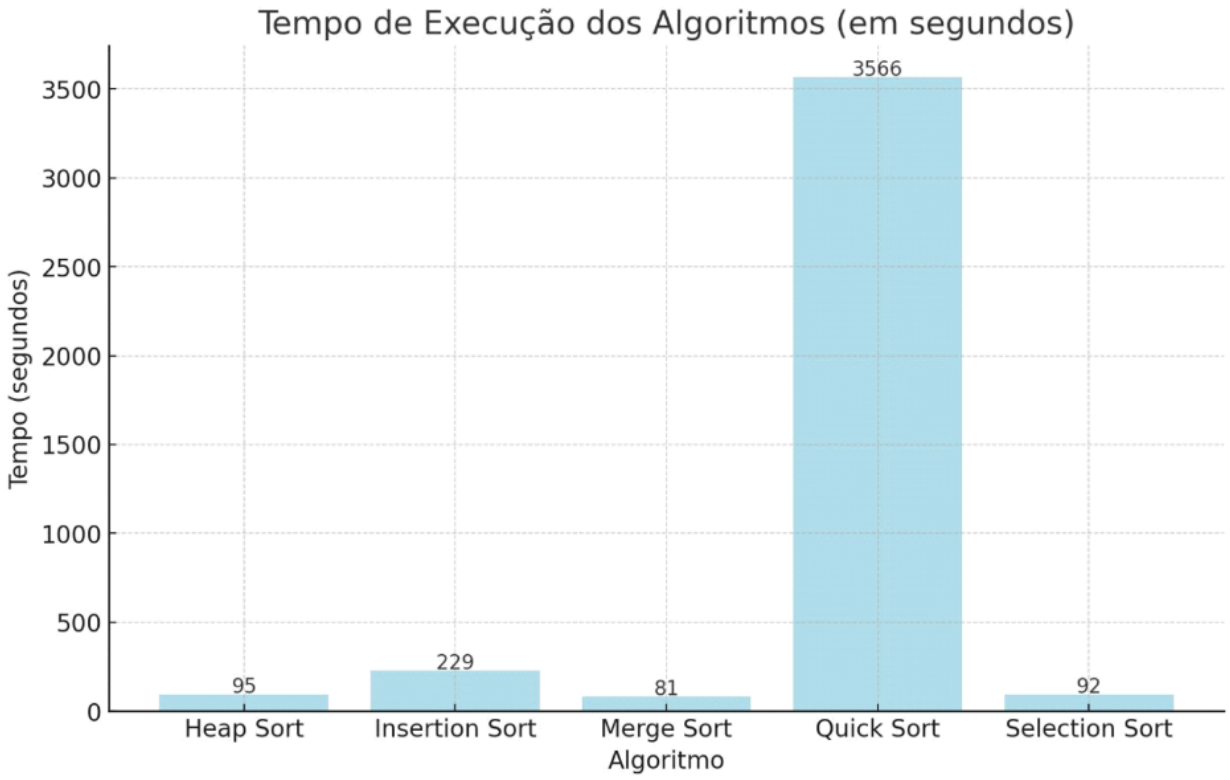
• **Counting Sort** – Esse algoritmo não conseguiu ser aplicado para ordenação do campo preço devido sua lógica. O tamanho do array auxiliar do counting sort depende diretamente do maior número que foi encontrado dentro do array a ser ordenado, que por sua vez trata os preços dos jogos como um número de ponto.

• Datas

Nesta seção, analisamos o desempenho dos algoritmos aplicados à ordenação do campo "data de lançamento" dos jogos, previamente transformado para o formato DD/MM/AAAA. Por se tratar de um campo originalmente textual, a ordenação exigiu tratamento prévio e conversão para formatos comparáveis, o que impactou diretamente o tempo de execução dos algoritmos.

Tempos de Ordenação (em segundos)

Algoritmo	Tempo (segundos)
Counting Sort	Não utilizado
Heap Sort	95
Insertion Sort	229
Merge Sort	81
Quick Sort	3,566
Quick Sort Med.3	Não utilizado
Selection Sort	92



A ordem de desempenho observada, do mais eficiente ao menos eficiente, foi:

• Merge Sort – Mais uma vez, destacou-se como o algoritmo mais eficiente, apresentando tempos baixos e consistentes. Sua natureza estável e capacidade de lidar bem com grandes volumes de dados organizados em estruturas complexas, como datas, o tornou ideal para essa tarefa.

• Selection Sort – Apesar de sua complexidade quadrática, teve desempenho superior ao esperado. Isso pode ser explicado pela presença de muitos registros com datas semelhantes ou pela previsibilidade do algoritmo, que realiza comparações fixas sem depender da posição relativa dos elementos.

• Heap Sort – Obteve um desempenho intermediário, superior apenas ao Insertion Sort. A sobrecarga de reestruturação da heap pesou mais neste cenário, mas ainda manteve um tempo razoável.

• Insertion Sort – Apesar das otimizações aplicadas com o uso de arrays auxiliares, o algoritmo teve baixa eficiência diante de grandes volumes de dados.

• Quick Sort – Apresentou o pior desempenho de todos os algoritmos aplicados à ordenação por data. A principal causa está na estratégia de escolha do pivô, que nesta implementação foi sempre o primeiro elemento do array. Essa abordagem, quando aplicada a dados parcialmente ordenados ou com distribuição desigual (como no caso de datas), resulta consistentemente em seu pior caso de complexidade $O(n^2)$. Além disso, o próprio tipo de dado, datas em formato textual convertidas para o padrão DD/MM/AAAA, aumentou a complexidade das comparações, tornando cada operação mais custosa em termos computacionais. Como resultado, o algoritmo levou impressionantes 3.566 segundos para completar a ordenação, em contraste com os 299 segundos do Insertion Sort, que teoricamente possui complexidade semelhante. Essa diferença expressiva evidencia como uma escolha inadequada de estratégia de pivô e a natureza dos dados podem comprometer drasticamente o desempenho de algoritmos eficientes em outros contextos.

• Counting Sort – Utilizando da mesma lógica que foi abordada na ordenação dos preços dos jogos, também não conseguimos aplicar o counting sort para ordenar os jogos baseado em suas datas de lançamento, devido ao tipo de dado que ta sendo tratado.

• Quick Sort Mediana de 3 – Ao analisar a lógica de implementação desse quick sort, que defini o pivo como sendo a mediana de 3 valores, percebemos que não conseguimos aplicar esse algoritmo para ordenar os jogos baseado em sua data de lançamento, já que não conseguimos calcular a mediana de 3 datas.

Análise comparativa dos tempos do projeto UT1 com UT2 na ordenação de conquistas:

Tempos de ordenação (em segundos) Conquistas(UT1)

Algoritmo	Tempo (segundos)
Counting Sort	10
Heap Sort	10
Insertion Sort	53
Merge Sort	10
Quick Sort	17
Quick Sort Mediana 3	20
Selection Sort	20

Tempos de ordenação (em segundos)

Algoritmo	Tempo (segundos)
Counting Sort	19
Heap Sort	11
Insertion Sort	19
Merge Sort	11
Quick Sort	10
Quick Sort Mediana 3	10
Selection Sort	25

UT1 **UT2**

Counting Sort teve um aumento significativo no tempo (de 10s para 19s). A diferença de desempenho entre a UT1 (10 segundos) e a UT2 (19 segundos) ocorre porque a UT2 introduziu estruturas adicionais que, apesar de organizarem melhor o código, tornaram um pouco mais lento que na UT1.

Heap Sort e **Merge Sort** mantiveram-se relativamente estáveis, o que mostra sua robustez independente da entrada e de outros tratamentos adicionais.

A queda significativa no tempo do **Insertion Sort** indica uma entrada de dados mais favorável em UT2. O mesmo vale para o **Quick Sort** e **Quick Sort Mediana 3**, que

tiveram ganhos notáveis, assim, consagrando as estruturas de dados que foram implementadas nessa parte do projeto.

Análise comparativa dos tempos do projeto UT1 com UT2 na ordenação de preços:

Tempos de ordenação (em segundos) Preços(UT1)

Algoritmo	Tempo (segundos)
Counting Sort	Não utilizado
Heap Sort	10
Insertion Sort	122
Merge Sort	10
Quick Sort	04
Quick Sort Mediana 3	10
Selection Sort	31

UT1

Tempos de ordenação (em segundos)

Algoritmo	Tempo (segundos)
Heap Sort	11
Insertion Sort	35
Merge Sort	10
Quick Sort	19
Quick Sort Mediana 3	10
Selection Sort	42

UT2

Heap Sort, Merge Sort e Quick Sort Mediana de 3 Se mantiveram com tempos estáveis, não sofrendo com as alterações feitas no código.

O **Quick Sort** é mais lento no UT2 devido a overhead de estruturas e má escolha de pivô.

O **Insertion Sort** da UT2 foi otimizado para aproveitar dados parcialmente ordenados (early return), por isso a melhora no tempo de execução.

Como já era de se esperar, o **Selection Sort** foi o de pior desempenho, devido aos dados desordenados, e um overhead na UT2.

Conclusão

As avaliações feitas revelaram que as estruturas de dados selecionadas em alguns casos gerou um grande ganho ou não teve nenhuma mudança de performance, porém em outros casos se mostrou desvantajoso a utilização das mesmas. O método de organização selecionado também tem um grande efeito sobre a performance do

sistema, sobretudo com diferentes tipos de informação, como números, datas e textos. Métodos com complexidade O(n logn), como o Merge Sort e o Quick Sort, em geral, se mostraram excelentes com dados aleatórios, mas nem sempre foram os mais velozes, provando que a eficiência teórica nem sempre gera um desempenho prático ideal.

Além disso, a escolha das estruturas de dados que vão ser utilizadas no projeto também afeta diretamente o desempenho do sistema, provando que a combinação da estrutura de dados correta com um algoritmo de ordenação adequado leva a um tempo de execução do programa muito mais rápido, mas também mesmo que haja a escolha correta de um bom algoritmo de ordenação mas a estrutura de dados não seja a mais adequada irá resultar em uma perda significativa de desempenho, como foi mostrado anteriormente.

Esses resultados reforçam a necessidade de levar em consideração não só a complexidade assintótica, mas também a natureza dos dados, o contexto de uso na seleção do método de organização e na seleção das estruturas de dados que irão ser utilizadas. O uso cuidadoso e adaptado das técnicas e estruturas pode trazer grandes ganhos de performance e eficiência computacional.