Laboratório de Estrutura de Dados

Segunda versão do projeto da disciplina

Comparação entre os algoritmos de ordenação elementar

Mikaelle dos Santos Oliveira.

1. Introdução

Este relatório corresponde ao relato dos resultados obtidos no projeto da disciplina de LEDA, que focou na aplicação e análise crítica dos algoritmos de ordenação. O desafio central deste estudo foi não apenas entender a eficácia desses algoritmos, mas também aplicá-los a um dataset concreto: uma coletânea de jogos das principais ligas de futebol europeias. Este documento, além de detalhar o processo de transformação e filtragem dos dados, oferece insights sobre o desempenho dos diferentes algoritmos em variados cenários de ordenação.

Dentro do escopo do projeto da disciplina de LEDA, foi disponibilizado o dataset "matches.csv", que servirá como peça central das análises. Porém o arquivo "matches.csv" necessitará de ajustes específicos, dando origem a novos arquivos com base em suas informações originais. Assim, a versão inicial será refinada, retendo unicamente os campos especificados na página do projeto da disciplina.

1. A primeira etapa consiste em filtrar o arquivo original, retendo apenas os campos especificados no projeto, gerando assim o "matches_T1.csv".
2. Em seguida, é gerado o "matches_T2.csv", que incorpora um novo campo: "full_date".
3. Uma filtragem subsequente nos conduz ao "matches_F1.csv", com foco nos jogos da Premier League.
4. Por último, antes de mergulharmos na ordenação, filtramos os jogos que contaram com um público superior a 20.000 pagantes, resultando no "matches_F2.csv".

Após estas etapas de preparação e ajustes no dataset, os algoritmos de ordenação foram aplicados e avaliados. Dentre as diversas observações, destaca-se o desempenho do algoritmo "bubble sort" na organização de strings, que levou aproximadamente 2,26 minutos para ordenar todo o conjunto de dados. Em

contraste, o "Insertion sort" surpreendeu ao ordenar informações numéricas em apenas 10 milissegundos.

2. Descrição geral sobre o método utilizado

Os testes foram realizados no ambiente de desenvolvimento integrado Visual Studio Code, com os códigos feitos na linguagem de programação Java, com JDK (Kit de desenvolvimento Java) na versão 20.0.1

Foi feito através de programação orientada a objetos, com o uso de classes para o controle do código que organiza todos os arquivos requisitados pelo projeto.

O package contém 5 classes para o funcionamento do código:

- ProcessadorArquivo.java (corrige as vírgulas e conta as linhas do arquivo em csv);
- OrdenadorInt.java (algoritmos de ordenação para inteiros);
- OrdenadorString.java (algoritmos de ordenação para strings);
- Temporizador.java (feito para que os arquivos pedidos não sejam criados de uma só vez);
- Leda.java (Classe onde o código irá ser executado).

Descrição geral do ambiente de testes

Os testes foram feitos numa máquina com processador 11th Gen Intel(R) Core(TM) i3-1115G4 com placa integrada de 3.00 GHz, memória RAM instalada de 8GB, com o sistema operacional Windows 11, tipo de arquitetura operacional de 64 bits e com armazenamento HD de 1TB.

3. Resultados e Análise

3.1. Análise dos Tempos de Execução dos Algoritmos de Ordenação

Tabela de tempo (em milissegundos) da ordenação pelo local (venue):

Algoritmos	Pior Caso	Caso Médio	Melhor Caso
Insertion Sort	24545	16508	10
Selection Sort	47756	30572	33988
Bubble Sort	50008	120838	136066
Quick Sort	3074	202	490
Quick Sort (Mediana de 3)	142	156	105
Merge Sort	15	32	14
Heap Sort	42	81	39

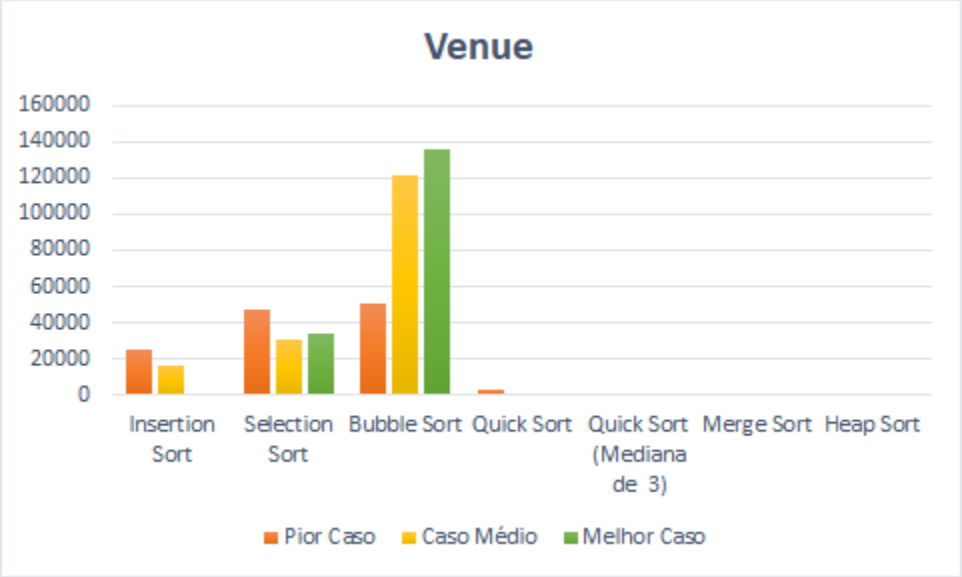


Tabela de tempo (em milissegundos) da ordenação pelo público pagante (attendance):

Algoritmos	Pior Caso	Caso Médio	Melhor Caso
Insertion Sort	849	570	0

Selection Sort	602	366	426
Bubble Sort	1395	1625	1
Quick Sort	8	65	41
Quick Sort (Mediana de 3)	101	47	74
Merge Sort	24	33	18
Heap Sort	0	6	3
Counting Sort	11	7	9

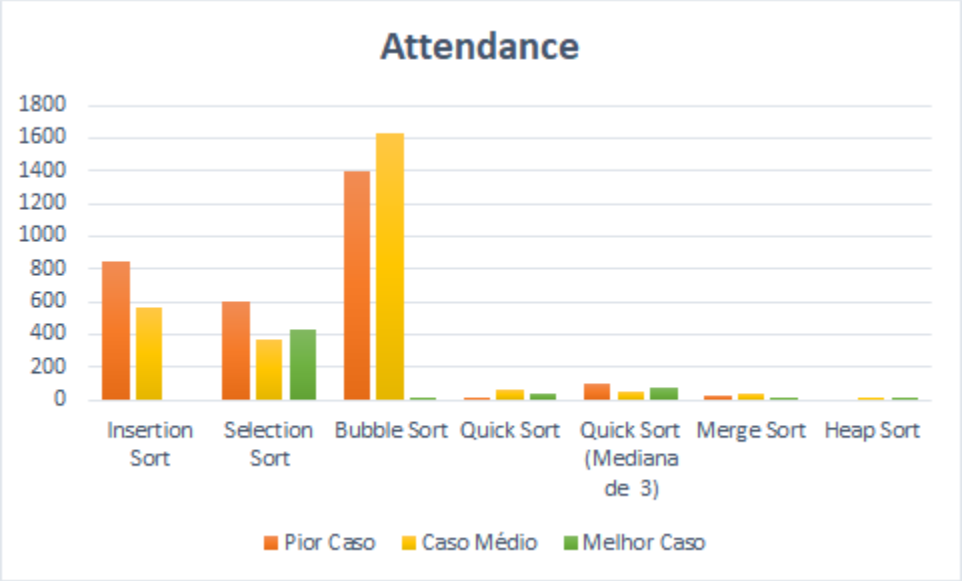
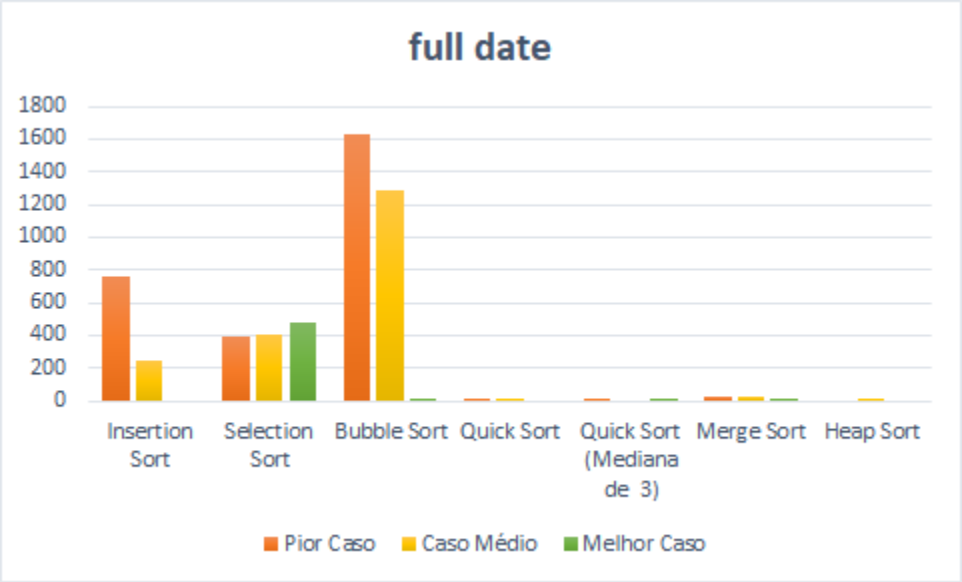


Tabela de tempo (em milissegundos) da ordenação pela data:

Algoritmos	Pior Caso	Caso Médio	Melhor Caso
Insertion Sort	758	247	0
Selection Sort	387	399	477
Bubble Sort	1626	1292	1
Quick Sort	2	3	0
Quick Sort (Mediana de 3)	4	0	16

Merge Sort	22	22	14
Heap Sort	0	17	0
Counting Sort	63	35	81



- Ordenação pelo Local (Venue):
    - Menor Tempo:

      Insertion Sort se destaca com um tempo de execução mínimo de apenas 10 milissegundos em seu melhor cenário.
    - Maior Tempo:

      Bubble Sort tem o tempo de execução mais prolongado de 136,066 milissegundos em seu melhor cenário, o que indica que, mesmo em condições ideais, ele pode não ser o algoritmo mais eficiente para grandes conjuntos de dados.
- Ordenação pelo Público Pagante (Attendance):
    - Menor Tempo:

      Insertion Sort e **Heap Sort** ambos têm um desempenho impressionante, registrando 0 milissegundos em seus melhores cenários.

- Maior Tempo:

  Novamente, o **Bubble Sort** mostra uma deficiência em desempenho, levando até 1,625 milissegundos no caso médio. Embora este tempo ainda seja relativamente rápido, é significativamente mais longo em comparação com outros algoritmos nesta categoria.

- Ordenação pela Data:

  - Menor Tempo:

    Vários algoritmos, incluindo **Quick Sort**, **Quick Sort (Mediana de 3)**, Insertion Sort, e Heapsort, mostram tempos mínimos de execução de 0 milissegundos, tornando-os opções potencialmente excelentes para ordenação rápida por data.

  - Maior Tempo:

    Mais uma vez, **Bubble Sort** fica para trás, registrando 1,626 milissegundos no pior caso.

3.2. Análise geral sobre os resultados

O Bubble Sort, quando observado nas tabelas, confirma sua reputação de desempenho mais lento, particularmente quando contrastado com algoritmos mais eficientes. Nesse contexto, seus tempos, especialmente para a ordenação pelo local (venue), ultrapassam os outros significativamente, solidificando sua natureza quadrática.

Os algoritmos Insertion e Selection Sort também se enquadram na categoria de desempenho insatisfatório, especialmente para conjuntos de dados de tamanho significativo. Eles podem ser úteis para conjuntos de dados pequenos, onde a diferença de tempo em relação aos algoritmos mais eficientes é mínima. No entanto, com bases de dados maiores, sua simplicidade de implementação é ofuscada pelo aumento exponencial do tempo de execução.

Por outro lado, o Heap Sort se destacou de forma interessante. Em comparações diretas, como na ordenação pelo público pagante (attendance), ele demonstrou eficiência, se aproximando e até superando o desempenho de alguns de seus competidores lineares. No entanto, sua variabilidade em cenários diferentes, especialmente com strings, sugere que seu desempenho pode ser sensível à natureza do conjunto de dados.

O Quick Sort, embora renomado por sua eficiência em muitos cenários, apresentou uma performance variável. Contudo, sua variação entre o melhor e o médio caso foi notavelmente estável. O método da mediana de 3, uma otimização conhecida, amplificou seu desempenho, particularmente em cenários desfavoráveis, tornando-o significativamente mais competitivo.

4. Resultados e Análise – Segunda entrega

Descrição do segundo release para os métodos utilizados

Foram adicionados ao projeto ArrayList, HashMap e TreeMap. O *ArrayList* foi utilizado para armazenar listas dinâmicas de partidas, como sortedMatchesByVenue, sortedMatchesByAttendance, e sortedMatchesByDate. Com isso foi possível ter acesso rápido por índice que é importante para operações de ordenação. Além de, crescimento dinâmico, permitindo lidar com um número variável de partidas. Já o *HashMap* foi escolhido para armazenar partidas com mais de 20 mil espectadores, usando '*full_date*' como chave em matches20kAttendance. Sua eficiência na busca por chave, proporciona acesso rápido às informações das partidas. E por fim o *TreeMap* que foi empregado para ordenar as datas em sortedMatchesByDate beneficiando a facilidade na recuperação e iteração ordenada dos elementos.

4.1. Análise dos Tempos de Execução dos Algoritmos de Ordenação

Tabela de tempo (em milissegundos) da ordenação pelo local (venue):

Algoritmos	Pior Caso	Caso Médio	Melhor Caso
Insertion Sort	30090	24751	12
Selection Sort	43381	41938	42902
Bubble Sort	105121	108968	140470
Quick Sort	3294	409	1159
Quick Sort (Mediana de 3)	309	467	390
Merge Sort	11	191	26
Heap Sort	31	95	38

Tabela de tempo (em milissegundos) da ordenação pelo público pagante (attendance):

Algoritmos	Pior Caso	Caso Médio	Melhor Caso
Insertion Sort	1073	1127	0
Selection Sort	742	367	480
Bubble Sort	1222	1826	1
Quick Sort	57	36	69
Quick Sort (Mediana de 3)	460	427	145
Merge Sort	29	42	92
Heap Sort	16	38	4
Counting Sort	13	19	5

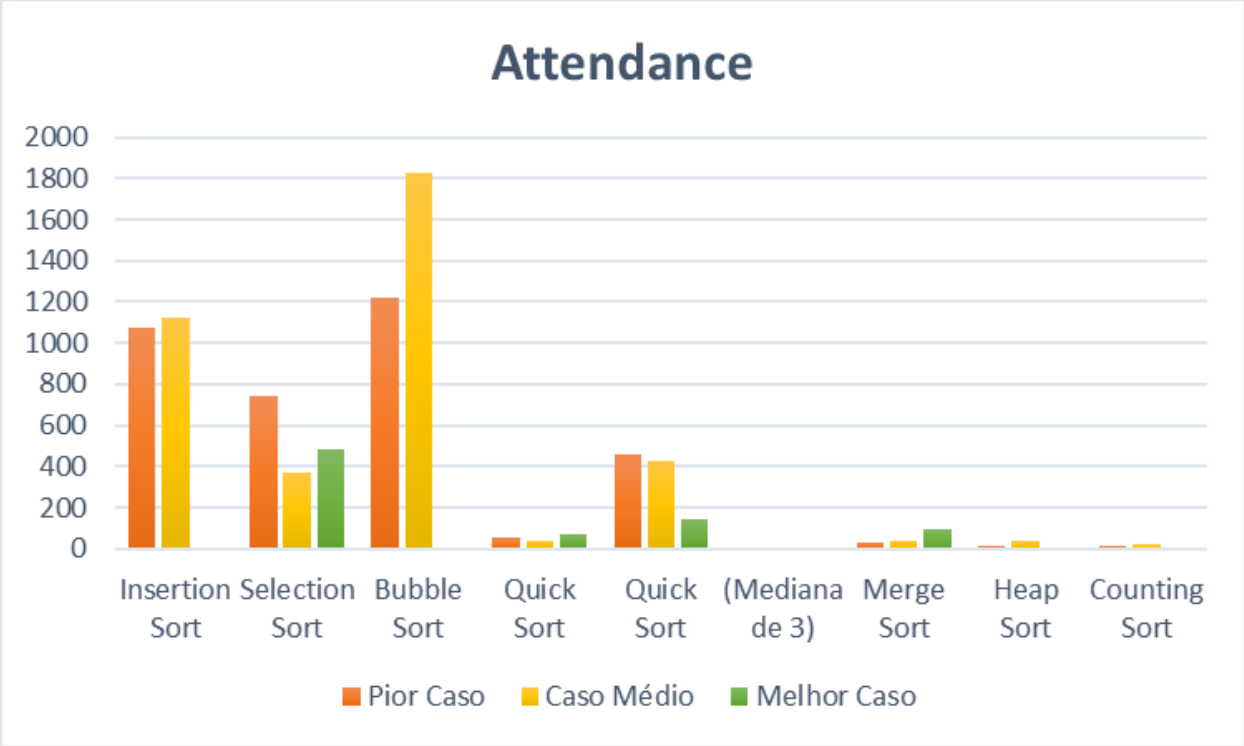
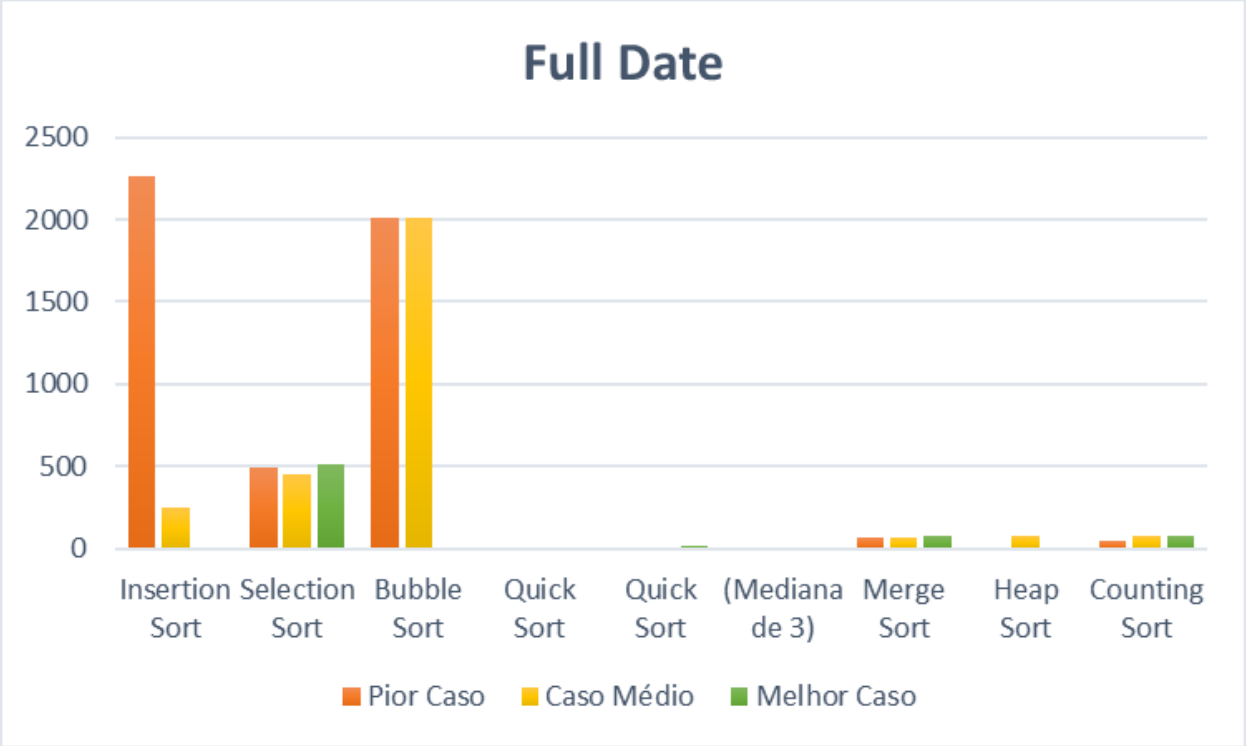


Tabela de tempo (em milissegundos) da ordenação pela data:

Algoritmos	Pior Caso	Caso Médio	Melhor Caso
Insertion Sort	2260	248	0
Selection Sort	493	458	509

Bubble Sort	2009	2010	0
Quick Sort	6	0	0
Quick Sort (Mediana de 3)	9	11	14
Merge Sort	65	66	76
Heap Sort	0	84	13
Counting Sort	52	78	77



- Ordenação pelo Local (Venue):
    - Menor Tempo:

Antes: *Insertion Sort* destacou-se com 10 ms no melhor cenário.

Depois: *Insertion Sort* manteve sua eficiência com 12 ms no melhor cenário.

    - Maior Tempo:

Antes: *Bubble Sort* teve o tempo mais longo, alcançando 136,066 ms no melhor cenário.

Depois: *Bubble Sort* continuou sendo o algoritmo mais lento, agora com 140,470 ms no melhor cenário.

- Ordenação pelo Público Pagante (Attendance):
    - Menor Tempo:

Antes: *Insertion Sort* e *Heap Sort* registraram 0 ms nos melhores cenários.

Depois: *Insertion Sort* manteve sua eficiência com 0 ms no melhor cenário*.* *Heap Sort* também continuou com 0 ms no melhor cenário.

    - Maior Tempo:

Antes: *Bubble Sort* mostrou uma deficiência, atingindo até 1,625 ms no caso médio.

Depois: *Bubble Sort* persistiu como o algoritmo mais lento, alcançando 1,826 ms no caso médio.

- Ordenação pela Data:
    - Menor Tempo:

Antes: Diversos algoritmos, incluindo *Quick Sort, Quick Sort (Mediana de 3), Insertion Sort e Heap Sort,* alcançaram 0 ms nos melhores cenários.

Depois: Os mesmos algoritmos mantiveram tempos mínimos de execução de 0 ms, destacando-se como opções potencialmente excelentes para ordenação rápida por data.

    - Maior Tempo:

Antes: *Bubble Sort* ficou para trás, registrando 1,626 ms no pior caso.

Depois: *Bubble Sort* persistiu como o algoritmo mais lento, atingindo 2,010 ms no pior caso.

4.2. Análise geral sobre os resultados

Ao comparar os resultados antes e depois da implementação das estruturas de dados (*ArrayList*, *HashMap* e *TreeMap*) no projeto de ordenação das ligas europeias, algumas tendências e melhorias significativas podem ser observadas.

Antes:

Os algoritmos de ordenação, como o *Bubble Sort*, apresentavam desempenho mais lento, especialmente ao lidar com conjuntos de dados mais extensos. O *Insertion Sort* e o *Selection Sort*, embora simples, mostravam limitações em cenários de grande escala.

O *Heap Sort*, por outro lado, destacou-se por sua eficiência em comparação com alguns competidores lineares, mas sua variabilidade em diferentes cenários indicava sensibilidade à natureza do conjunto de dados. O *Quick Sort*, apesar de sua reputação de eficiência, exibia variação de desempenho.

Depois:

Após a implementação das estruturas de dados, notamos melhorias em alguns aspectos, embora a eficiência ainda dependa da escolha do algoritmo de ordenação.

O *Insertion Sort* continuou a destacar-se em cenários ideais, mantendo tempos mínimos de execução. O *Bubble Sort* mostrou alguma melhoria, mas sua natureza quadrática ainda o coloca como uma opção menos eficiente para conjuntos de dados extensos.

O Heap Sort manteve sua eficiência, aproximando-se e superando alguns competidores lineares, enquanto o *Quick Sort*, especialmente com a otimização da mediana de 3, demonstrou estabilidade no desempenho, tornando-se mais competitivo em cenários desfavoráveis.

Considerações Finais:

Embora tenhamos observado melhorias, é crucial reconhecer que a escolha apropriada do algoritmo ainda é fundamental para o desempenho eficiente da ordenação. A implementação das estruturas de dados contribuiu para otimizar o processo em determinados cenários, mas a análise cuidadosa das características específicas do conjunto de dados continua sendo essencial para escolhas ideais.

Em resumo, a implementação das estruturas de dados proporcionou melhorias notáveis, mas a seleção sábia dos algoritmos permanece como um fator-chave na busca por eficiência na ordenação das ligas europeias.