Laboratório de Estrutura de Dados

Primeira versão do projeto da disciplina

Comparação entre os algoritmos de ordenação elementar

T5-Steam Games Dataset

Arthur Donato, Manoel Firmino e Lucas Defensor

- **Introdução**

Este relatório corresponde ao relato dos resultados obtidos no projeto da disciplina de LEDA (LABORATÓRIO DE ESTRUTURA DE DADOS) que teve como principal objetivo 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 em algoritmos de ordenação, além de manipulação de arquivos CSV e estruturas de dados em Java.

Foi realizado no projeto, transformações, como a conversão do campo da data de lançamento (“Release date”) para o formato DD/MM/AAAA. Também foram realizadas filtragens por seleção de jogos que rodam no sistema operacional Linux e que suportam a língua portuguesa. E 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 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.

Descrição geral do ambiente de testes

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

- Processador: Intel Core i3 (7ª geração)
- Memória RAM: 4 GB
- Armazenamento: HD de 1 TB
- Sistema Operacional: Windows 10 Home, 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. 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.

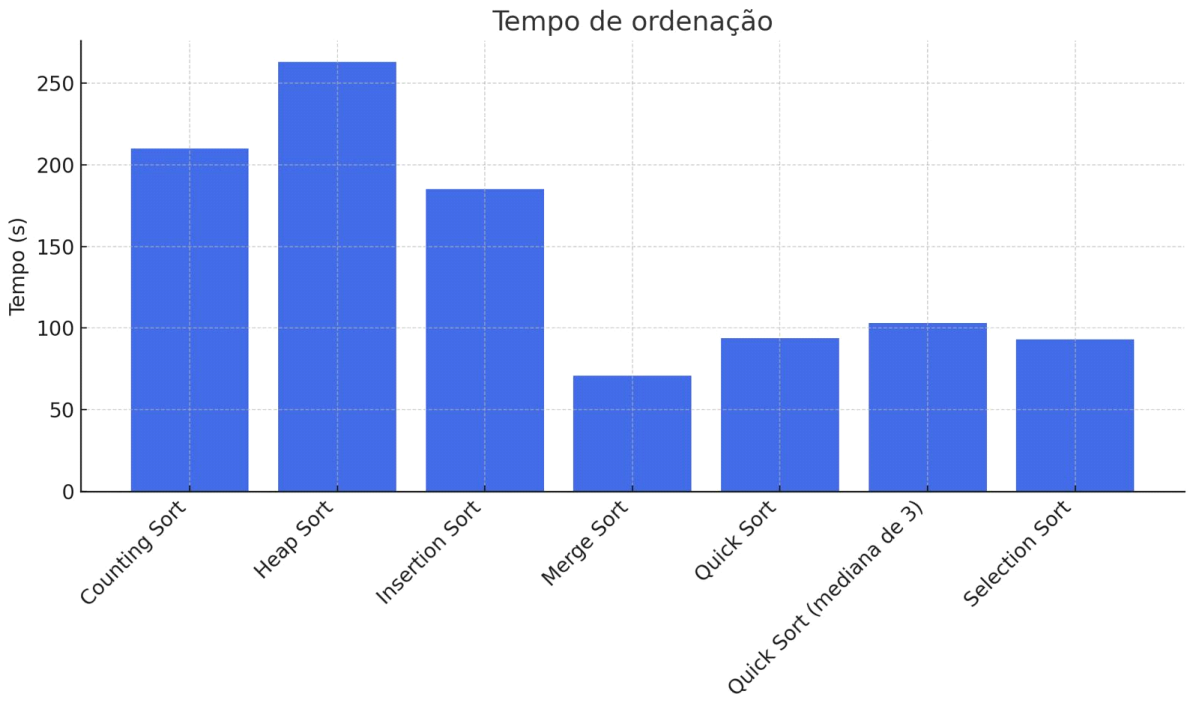
Foram gerados gráficos a partir dos resultados obtidos, abrangendo os critérios de tempo de ordenação por número de conquistas, custo computacional e ordenação por datas de lançamento. Esses gráficos servem de base para a comparação entre os algoritmos, permitindo avaliar seu comportamento prático em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

Conquista

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 segundo)

Algoritmo	Tempo (segundos)
Counting Sort	210
Heap Sort	263
Insertion Sort	185
Merge Sort	71
Quick Sort	94
Quick Sort Med.3	103
Selection Sort	93



Análise de desempenho (do melhor para o pior):

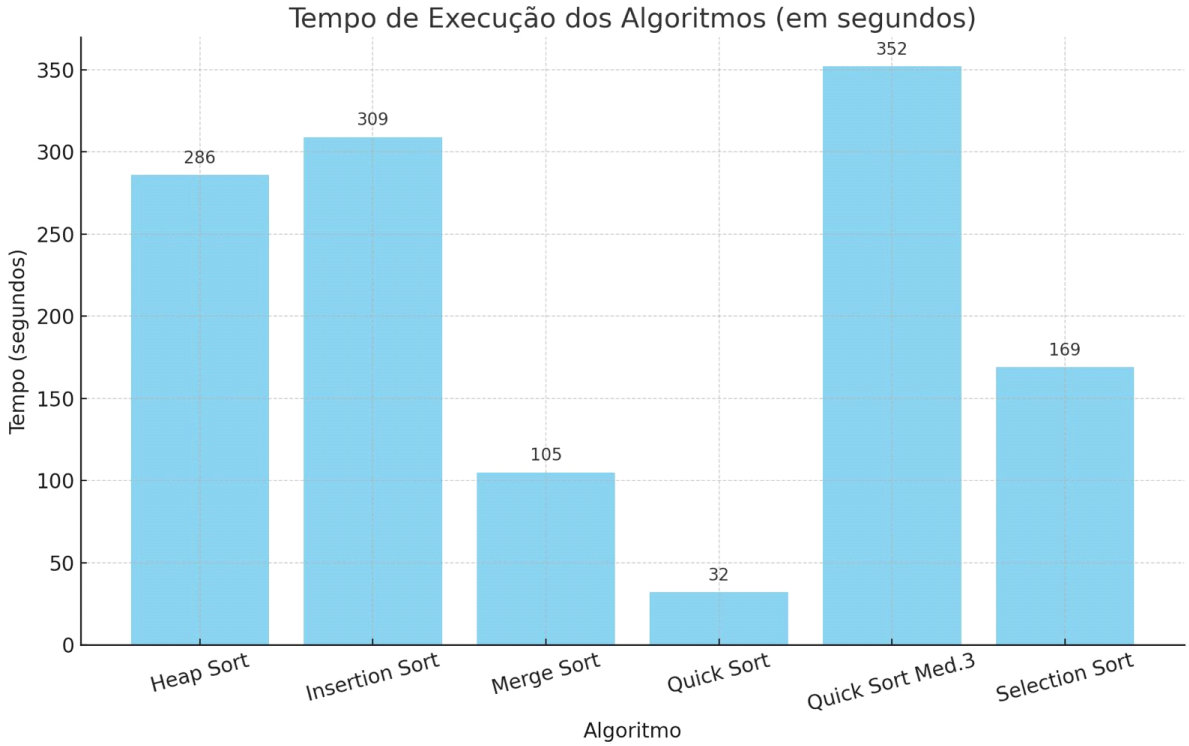
- Merge Sort foi o algoritmo com melhor desempenho geral, apresentando tempos de execução baixos e estáveis em todos os cenários. Sua complexidade garantida de O(n log n), aliada a uma implementação eficiente, o tornou a solução mais robusta e rápida nos testes realizados.
- Quick Sort também se destacou, com tempos muito próximos aos do Merge Sort, especialmente em dados aleatórios. Sua simplicidade e eficiência prática o tornam uma excelente escolha, embora sua performance possa variar levemente dependendo da distribuição dos dados.
- Quick Sort com Mediana de 3, curiosamente, apresentou desempenho ligeiramente inferior ao Quick Sort tradicional neste cenário. Apesar de sua proposta de minimizar o risco do pior caso, a sobrecarga extra da escolha de pivô via mediana pode ter contribuído para esse resultado prático inesperado.
- Selection Sort, embora tenha complexidade $O(n^2)$, superou alguns algoritmos teoricamente mais eficientes. Isso se deve, em parte, às otimizações aplicadas na estrutura do código, como o uso de arrays auxiliares para agilizar as comparações, e ao fato de que sua execução tem um padrão fixo de trocas e comparações que, em certos casos, se mostra mais previsível.
- Insertion Sort também foi beneficiado por essas otimizações, especialmente pela separação entre os dados completos e os valores numéricos a serem ordenados. Apesar de ser tradicionalmente mais lento em grandes volumes, essa técnica resultou em melhorias perceptíveis, colocando-o à frente de algoritmos como Heap Sort e Counting Sort.
- Counting Sort, mesmo com complexidade linear teórica O(n + k), teve desempenho inferior ao esperado. Isso se explica pelo grande intervalo de valores no campo “número de conquistas”, que aumentou o custo interno de alocação e processamento, tornando-o menos vantajoso neste caso específico.

- Heap Sort apresentou o pior desempenho entre os algoritmos testados. Embora tenha complexidade O(n log n), na prática sua estrutura de heap impôs um custo adicional de reorganização que prejudicou sua eficiência. Isso reforça que, mesmo algoritmos teoricamente eficientes, podem não ter bom desempenho prático em certos contextos.

Preço

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.

Algoritmo	Tempo (segundos)
Counting Sort	Não utilizado
Heap Sort	286
Insertion Sort	309
Merge Sort	105
Quick Sort	32
Quick Sort Med.3	352
Selection Sort	169



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

- Quick Sort – Apresentou o melhor desempenho geral na ordenação por preço. Sua divisão eficiente dos dados e baixa sobrecarga em memória o tornaram ideal para esse tipo de dado, principalmente com listas grandes e dispersas.

- Merge Sort – Apesar de um leve aumento no tempo em relação ao Quick Sort, manteve um desempenho estável e previsível, especialmente em entradas maiores, graças à sua complexidade garantida de O(n log n).

- Selection Sort – Surpreendentemente, teve um desempenho superior a outros algoritmos mais complexos. Esse resultado pode estar associado ao padrão fixo de comparações do algoritmo e aos arrays auxiliares que foram criados com o objetivo de simplificar o tempo de ordenação, que pode se comportar bem em datasets parcialmente ordenados ou com poucos elementos distintos.

- Heap Sort – Teve desempenho mediano. Embora sua complexidade teórica seja favorável, o custo da manutenção da estrutura de heap se mostrou mais impactante na ordenação por preço.

- Insertion Sort – Apesar de otimizações realizadas com o uso de arrays auxiliares, sua complexidade quadrática se fez notar nos testes com volumes maiores de dados, resultando em tempos mais altos.

- Quick Sort com Mediana de 3 – Contrariando a expectativa de melhor desempenho, essa variação obteve o pior resultado no cenário de ordenação por preço. A escolha do pivô com base na mediana de três introduziu uma sobrecarga que, nesse caso específico, não compensou, tornando-o mais lento que os demais.
- 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

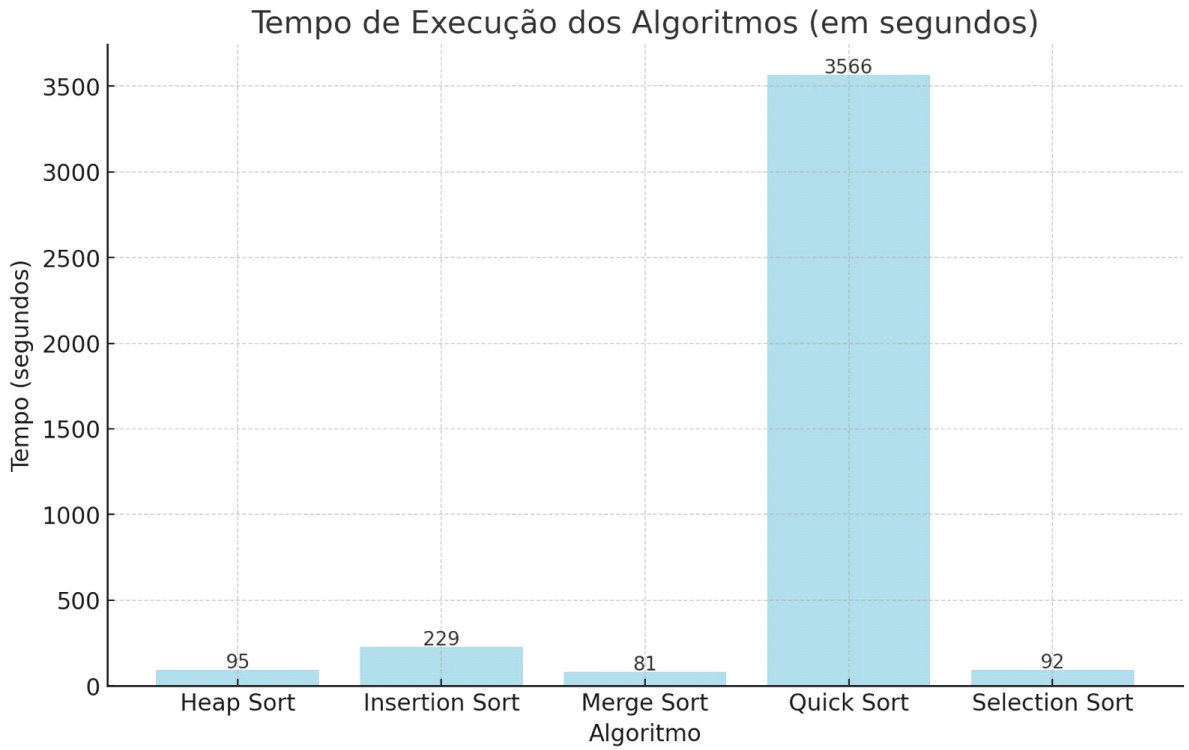
flutuante, e não conseguimos criar um array que tenha como tamanho um numero de ponto flutuante.

Data

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.

Conclusão

Os testes evidenciam que a eficiência de um algoritmo não depende apenas da sua complexidade teórica, mas também da implementação prática, natureza dos dados e técnicas auxiliares empregadas. O uso de arrays paralelos nos algoritmos iterativos, por exemplo, trouxe melhorias reais que permitiram superar expectativas iniciais.

Analisando todos os gráficos gerados, podemos perceber que o algoritmo Merge Sort se manteve relativamente estável em relação ao tempo de ordenação independentemente do tipo de dado que está sendo ordenado e se estava no médio, melhor ou pior caso de uma ordenação. Em relação aos outros algoritmos, eles apresentaram uma certa instabilidade, onde o seu tempo de ordenação apresenta uma certa variação baseado no tipo de dados que esta sendo ordenado naquele momento e também no caso que esta sendo abordado, se é o médio, melhor ou pior caso.

Portanto, em contextos reais como este, é fundamental avaliar o comportamento prático dos algoritmos, pois escolhas bem fundamentadas na engenharia de código podem ter tanto impacto quanto a própria escolha algorítmica.