Laboratório de Estrutura de Dados

**Primeira versão do projeto da disciplina**

Comparação entre os algoritmos de ordenação elementar

horizontal line

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# Vinícius Souto

# Gabriel Levy

# Introdução

Este relatório corresponde ao relato dos resultados obtidos no projeto da disciplina de LEDA que objetivamente buscou implementar e comparar diversos algoritmos de ordenação a um dataset de estatísticas de vídeos da plataforma Youtube, vale ressaltar que este projeto se alinha com os conteúdos apresentados nesta disciplina.

O projeto foi dividido em 2 partes. A primeira parte condiz com a preparação da base de dados original, a qual inicialmente foi feita a atribuição de uma coluna chamada “Countries” referente ao país em que o vídeo foi publicado (No casoEUA, GB, DE, CA, FR, RU, MX, KR e JP), assim gerando um arquivo chamado “videos.csv”, em seguida foi feita mais uma transformação, esta foi criar um campo para as datas “trending\_full\_date” e converter as datas do formato AA.DD.MM para DD/MM/AAAA gerando um arquivo “videos \_T1.csv”, a transformação seguinte foi filtrar o arquivo gerado “videos\_T1.csv” cujo campo “title” dos objetos json sejam “Trailers”, “Shows” e “Shorts”, desta forma gerando um arquivo “videos\_TSS.csv”, em seguida foi realizada mais uma filtragem em todos o arquivo “videos\_T1.csv” em que os números de “dislikes” são maiores que “likes”, gerando o arquivo “videos\_T2.csv”.

Na segunda parte do projeto, utilizando como entrada os dados da primeira transformação “videos\_T1.csv”, foram aplicados sete algoritmos de ordenação: Selection sort, Insertion Sort, Merge Sort, Quick Sort, Quick sort com mediana de três, Counting sort e Heap Sort, em três atributos dos vídeos - título dos canais, contagem de comentários e data completa de tendencia – estes algoritmos de ordenação foram avaliados em 3 critérios: entrada desordenada (caso médio), entrada ordenada (melhor caso) e entrada ordenada em ordem inversa (pior caso).

No geral, observou-se que o Merge Sort (algoritmo de complexidade O(n log n)) e o Counting Sort(algoritmo de complexidade (O)n) possuem os resultados mais estáveis (caso médio, pior e melhor) e eficientes para grandes volumes de dados, diferentemente dos algoritmos de complexidade quadráticas, dessa forma, mostrando-se menos viáveis.

# Descrição geral sobre o método utilizado

**2.1 Como os testes foram realizados**

* **Caso Médio:**

Utiliza-se o arquivo gerado anteriormente “vídeos\_T1.csv” que está ordenado aleatoriamente.

* **Pior Caso:**

Para o pior caso utilizei um código a parte para que já fosse gerado antes de entrar a fase de testes dos algoritmos, abaixo colocarei o nome de cada código utilizado para ordenar e seus arquivos gerados.  
**ordenationChannel.java -> videos\_T1\_channel\_title\_decrescente.csv**

**ordenationComments.java -> videos\_T1\_comment\_count\_decrescente.csv**

**ordenationTrendingDate.java -> videos\_T1\_trending\_full\_date\_decrescente.csv**

* **Melhor Caso:**

Para o melhor caso utilizei um código a parte para que já fosse gerado antes de entrar a fase de testes dos algoritmos, abaixo colocarei o nome de cada código utilizado para ordenar e seus arquivos gerados.

**ordenationChannel.java -> videos\_T1\_channel\_title\_crescente.csv**

**ordenationComments.java -> videos\_T1\_comment\_count\_crescente.csv**

**ordenationTrendingDate.java -> videos\_T1\_trending\_full\_date\_crescente.csv**

**2.2 Dados coletados**

## Tempo de Execução:

## Medido em milissegundos pela diferença entre duas chamadas a System.currentTimeMillis(), realizadas imediatamente antes e após cada algoritmo de ordenação.

**2.3 Observações Sobre os algoritmos**

Para o campo de ordenação “Channel\_title” não foi possível utilizar todos os algoritmos estudados, o counting ficou de fora desse caso. Para os demais campos foi possível aplicar todos, entretanto, algoritmos como selection, insertion e counting sort necessitaram de um longo tempo para serem executados. Recomendo que deixe comentadas as chamadas desses algoritmos pois caso execute eles possivelmente vai levar mais de dias para sua conclusão. Deixei comentários bem detalhados para cada etapa do código.

1. **Implementação da Ferramenta**

**3.1. Linguagem e Bibliotecas**

A ferramenta foi desenvolvida em Java 11, utilizando apenas arrays como solicitado. Utilizamos java.io.\* para leitura e escrita de arquivos, java.util.\* para manipulação de arrays e utilitários e também *usamos parseCSVLine para separar os campos na leitura dos arquivos.* Para Leitura de arquivos CSV: o programa faz parsing manual dos CSVs usando split com expressão regular para tratar aspas (").

**3.2. Estrutura de Pacotes**

Dentro da pasta **“ordenation”** tenho todos os algoritmos solicitados. Estão funcionando da seguinte maneira:  
**ordenation\OrdenacaoChannel.java** Responsável por ordenar os arquivos pelo campo “channel\_title”. Os arquivos gerados vão para uma pasta chamada “Ordenação Canais”.

ordenation\OrdenacaoComments.java Responsável por ordenar os arquivos pelo campo “comment\_count”. Os arquivos gerados vão para uma pasta chamada “Ordenação Comments”.

ordenation\OrdenacaoTrendingDate.java Responsável por ordenar os arquivos pelo campo “trending\_full\_date”. Os arquivos gerados vão para uma pasta chamada “Ordenação Datas”.

**3.3. Descrição geral do ambiente de testes**

* + **Processador: Intel® Core™ i3-10100F**
  + **Memória RAM: 32 GB DDR4**
  + **Armazenamento: Ssd Nvme 1tb gen 4**
  + **Placa de vídeo: NVIDIA GeForce RTX 3060 (12 GB VRAM)**
  + **Sistema Operacional: Winows 11 64-bits, processador baseado em x64**
  + **Java: OpenJDK 11.0.x**
  + **Ambiente de execução: testes conduzidos pela IDE Visual Studio Code**

# 4.Resultados e Análise

## 3.1 Desempenho por Atributo de Ordenação

* **TRENDING\_DATE**

Algoritmos destacados: Quick Sort com mediana de três ("Mediana de 3") e Merge Sort. Observação: O Quick Sort com mediana de três mostrou-se eficiente ao evitar o pior caso (O(n²)), enquanto o Merge Sort manteve tempos estáveis (O(n log n)). Algoritmos quadráticos como Selection e Insertion foram excluídos dos gráficos por lentidão.

* **COMMENT\_COUNT**

Algoritmos destacados: Counting Sort e Heap Sort ("Hedge").

Justificativa: O Counting Sort (complexidade O(n + k)) é ideal para dados com chaves limitadas (ex: contagem de comentários em faixas específicas). O Heap Sort (O(n log n)) também apresentou bom desempenho, enquanto Insertion e Selection tiveram tempos em segundos, inviabilizando uso em grandes volumes.

* **CHANNEL\_TITLE**

Destaques: Merge Sort e Quick Sort.

Contexto: Para ordenar strings (títulos de canais), o Merge Sort mostrou-se superior devido à estabilidade e desempenho consistente. O Quick Sort teve variações, mas a versão com mediana de três melhorou a robustez. Algoritmos quadráticos (Insertion, Selection) foram descartados do gráfico, com tempos acima de 10.000 ms (ex: 60.000 ms para grandes entradas).

## 3.2 Comparação entre Algoritmos

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algoritmo | Complexidade | Estabilidade | Melhor Caso | Pior Caso |
| Merge Sort | O(n log n) | Estável | Todos os casos | Consistente |
| Counting Sort | O(n + k) | Estável | Chaves limitadas | Inviável para k grande |
| Quick Sort | O(n log n) | Instável | Mediana de 3 | Pior caso (raro) |
| Heap Sort | O(n log n) | Instável | Grande n | Consistente |
| Selection/Insertion | O(n²) | - | Pequenos n | Inviável para grandes n |

Merge Sort: Destacou-se em todos os atributos, especialmente em CHANNEL\_TITLE, onde a estabilidade é crucial para ordenar strings sem perder a ordem relativa.

Counting Sort: Excelente para COMMENT\_COUNT (valores inteiros em faixa controlada), mas limitado a dados com chaves numéricas e não repetitivas.

Algoritmos Quadráticos: Selection e Insertion foram ineficientes para grandes volumes (ex: 60.000 entradas em CHANNEL\_TITLE), com tempos até 60x maiores que os O(n log n).

## 3.3 Impacto dos Casos de Teste

**Caso Médio** **(Entrada Desordenada):** Merge Sort e Quick Sort mantiveram desempenho próximo, com leve vantagem para o Merge Sort em dados heterogêneos (ex: strings). Counting Sort só se destacou em COMMENT\_COUNT devido à natureza dos dados.

**Melhor Caso (Entrada Ordenada):** Insertion Sort teria tempo O(n), mas sua implementação prática ainda foi lenta comparada a algoritmos O(n log n). Quick Sort com mediana de três mostrou adaptação rápida a dados pré-ordenados.

**Pior Caso (Entrada Inversa):** Quick Sort sem mediana de três teve desempenho degradado, enquanto a versão com mediana de três evitou isso. Merge Sort manteve a mesma eficiência, confirmando sua independência da disposição inicial dos dados.

## 3.4 Conclusão

**Para grandes volumes de dados:** Merge Sort é a escolha padrão devido à estabilidade e desempenho consistente. Counting Sort deve ser usado apenas quando as chaves são limitadas e conhecidas (ex: contagens, IDs numéricos).

**Para dados sensíveis a instabilidade:** Evitar Quick Sort e Heap Sort se a ordem relativa for crítica (ex: ordenar por CHANNEL\_TITLE e manter a ordem de vídeos do mesmo canal).

**Otimizações:** A mediana de três no Quick Sort reduziu o risco do pior caso, tornando-o competitivo com Merge Sort em certos cenários. Algoritmos quadráticos devem ser restritos a entradas pequenas (ex: n < 1.000).

**Limitações:** O Counting Sort não foi aplicável a CHANNEL\_TITLE (strings), enquanto o Heap Sort mostrou-se menos estável que o Merge Sort.

