Laboratório de Estrutura de Dados

Primeira versão do projeto da disciplina

Comparação entre os algoritmos de ordenação elementar

Aluno: Luiz José Mendonça Duarte e Raquel Melo de Queiroz

Disciplina: Laboratório de Estrutura de Dados

Professor: Fábio Luiz Leite Junior

2024

1. Introdução

Este relatório corresponde ao relato dos resultados obtidos no projeto da disciplina de Laboratório de estrutura de dados, cujos objetivos são estudar e analisar o desempenho dos algoritmos de ordenação utilizando dados de projetos voltados para Data Science.

No projeto especificamente é sobre o dataset do Youtube Video stats, nele faremos ordenações e manipulações com os dados como forma de mostrar o desempenho dos diferentes algoritmos de ordenação. (Caso não tenha conhecimento sobre algoritmos de ordenação clique aqui, para ir para as referências onde colocamos links para aprender mais sobre o assunto).

Visão geral do projeto:

- **Introdução:** Manipulação de dados do Youtube Video Stats, com uso de algoritmos de ordenação para visualização do desempenho.
- **Descrição geral sobre o método utilizado:** No nosso projeto, iremos usar os dados que temos para serem rodados por diferentes tipos de algoritmos de ordenação Insertion, Selection, Merge, etc, e iremos mostrar e debater sobre qual o melhor algoritmo de ordenação, utilizando melhor médio e pior caso.
- **Resultado e análise:** Os resultados e a análise serão disponibilizados no fim do nosso projeto, para acessá-los diretamente clique aqui.

2. Descrição geral sobre o método utilizado

Inicialmente, baixamos o dataset do projeto Trending Youtube Video Stats. Depois fizemos as transformações que foram requisitadas em “tasks” (tarefas) a se fazer.

1. Crie um arquivo chamado “videos.csv” juntando todos os arquivos que possuem a extensão “.csv”. O arquivo “videos.csv” deve ter uma coluna “countries”, referente ao país em que o vídeo foi publicado. (no caso, EUA, GB, DE, CA, FR, RU, MX, KR e JP)

Arquivos: GBvideos.csv, CAvideos.csv, MXvideos.csv, INvideos.csv, DEvideos.csv, RUvideos.csv, JPvideos.csv, FRvideos.csv, USvideos.csv e KRvideos.csv.

☑ ~~● Gerar um arquivo chamado “videos.csv”.~~

- Utilizamos o arquivo que no GitHub estará como SepararVideos.java
  - Ele é responsável por criar o “videos.csv”.

2. Considerando o arquivo gerado na primeira transformação (videos.csv). Crie um novo campo para as datas chamado “trending_full_date” no formato DD/MM/AAAA de acordo com a coluna “trending_date”, as datas da coluna “trending_date” estão no formato AA.DD.MM, você deve convertê-las para o formato DD/MM/AAAA.

Ex: 17.14.11 -> 14/11/2017.

☑ ~~● Gere um arquivo chamado “videos_T1.csv”~~

- Utilizamos o arquivo que no GitHub estará como FormatarData.java
  - Ele é responsável por criar o “videos_T1.csv”.

3. Utilize os arquivos “.json” para filtrar os vídeos do arquivo “videos_T1.csv” cuja o campo “title” dos objetos json sejam “Trailers”, “Shows" e "Shorts".

☑ ~~Gerar um arquivo chamado “videos_TSS.csv”~~

- Utilizamos o arquivo que no GitHub estará como FiltrarVideos.java
  - Ele é responsável por criar o “videos_TSS.csv”.

4. Filtre todos os vídeos do arquivo “videos_T1.csv” em que os números de “dislikes” são maiores que os números de “likes”.

- ☑ ~~● Gere um arquivo chamado “videos_T2.csv”~~
    - Utilizamos o arquivo que no GitHub estará como FiltrarDislikes.java
        - Ele é responsável por criar o “videos_T2.csv”.

Após as transformações feitas, trabalhamos fazendo as ordenações nos três casos especificados: Ordenar o arquivo completo pelo nomes dos canais (campo channel_title) em ordem alfabética; Ordenar o arquivo videos_T1.csv pelo número de comentários (campo comment_count) em ordem crescente, do menor para o maior; Ordenar o arquivo videos_T1.csv pela data completa em que o vídeo ficou em alta (campo trending_full_date) em ordem decrescente, da mais recente para a mais antiga. Aplicando os algoritmos de ordenação: selection sort, bubble sort, insertion sort, merge sort, quick sort, heap sort e counting sort.

Dessa forma, tivemos que implementar os algoritmos de ordenação, antes de implementá los, fizemos uma análise de quais seriam possíveis implementar, o único que não podem ser 100% bem implementado foi o Counting sort que não pode ser implementado no channel_tittle e no trending full date, devido as strings, e o comment_count não foi implementado devido a erros na memória sempre que o mesmo foi executado.

Os arquivos que fizemos foram organizados dessa forma:

```
src/
├── main/
│   ├── java/
│   │   ├── Main.java
│   │   ├── PreparacaoDados.java
│   │   ├── ordenacao/
│   │   │   ├── Ordenacao.java
│   │   │   ├── SelectionSort.java
│   │   │   ├── InsertionSort.java
│   │   │   ├── MergeSort.java
│   │   │   ├── QuickSort.java
│   │   │   ├── QuickSortMediana3.java
│   │   │   ├── CountingSort.java
│   │   │   ├── HeapSort.java
```

```
│   │   ├── performance/
│   │   │   └── AnalisePerformance.java
│   ├── resources/
│   │   ├── GBvideos.csv
│   │   ├── CAvideos.csv
│   │   ├── MXvideos.csv
│   │   ├── INvideos.csv
│   │   ├── DEvideos.csv
│   │   ├── RUvideos.csv
│   │   ├── JPvideos.csv
│   │   ├── FRvideos.csv
│   │   ├── USvideos.csv
│   │   ├── KRvideos.csv
│   │   ├── videos.csv
│   │   ├── videos_T1.csv
│   │   ├── videos_TSS.csv
│   │   ├── videos_T2.csv
│   │   ├── ...
```

Descrição da implementação da ferramenta (IDE) utilizada:

Utilizamos a ferramenta Eclipse IDE (Luiz)

Ferramenta recomendada para utilização: Eclipse IDE

Descrição geral do ambiente de testes:

Processador: Intel ® Corde (™) i5 - i5-1135G7 @ 2.40GHz 2.42 GHz

Memória RAM 8,00GB

Tipo de sistema operacional - Sistema operacional de 64 bits, processador baseado em x64.

Especificações do Windows - Windows 11 Home Single Language. Versão 23H2.

3. Resultados e Análise

Foi elaborada uma tabela comparando o tempo de execução dos algoritmos utilizados, para cada tipo de ordenação. Nas tabelas abaixo é possível ver os algoritmos e

seus respectivos tempos de execução, tudo isso comparando com os tipos de caso (médio caso, melhor caso e pior caso).

Elaborar os resultados dos testes usando tabelas e gráficos

Tabela geral de tempos de execução(em segundos(s)):

Caso ache que está faltando algum tempo, aperte aqui e entenda porquê.

Algoritmo	Campo	Melhor caso (s)	Caso médio (s)	Pior caso (s)
QuickSort	channel_title	1.405	1.393	1.519
QuickSort3	channel_title	1.842	1.887	1.889
MergeSort	channel_title	0.721	0.658	0.692
HeapSort	channel_title	1.581	1.45	2.592
SelectionSort	channel_title	5330.476	8585.658	5042.889
InsertionSort	channel_title	3220.948	3199.507	2582.825
QuickSort	comment_count	6.967	6.996	6.967
QuickSort3	comment_count	7.727	7.725	7.938
MergeSort	comment_count	0.906	0.742	0.694
HeapSort	comment_count	1.366	1.392	1.453
SelectionSort	comment_count	5472.191	5126.409	5097.589
InsertionSort	comment_count	3908.202	3969.65	3179.169
QuickSort	trending_full_date	11.736	11.584	11.513
QuickSort3	trending_full_date	12.024	11.969	12.043
MergeSort	trending_full_date	0.64	0.769	0.736
HeapSort	trending_full_date	1.128	1.144	1.25
SelectionSort	trending_full_date	4963.209	4507.029	4713.429
InsertionSort	trending_full_date	2766.854	2758.428	2428.515

Análise algoritmos mais rápidos ao mais lentos(channel title):

Fazendo uma análise, podemos perceber que em diversas situações (mesmo as críticas) os algoritmos que obtiveram uma melhor performance foram(ordem do maior para o menor de cima para baixo):

Melhor Caso

1. **MergeSort**: 0.721 s
2. **QuickSort**: 1.405 s
3. **HeapSort**: 1.581 s
4. **QuickSort3**: 1.842 s
5. **InsertionSort**: 3220.948 s
6. **SelectionSort**: 5330.476 s

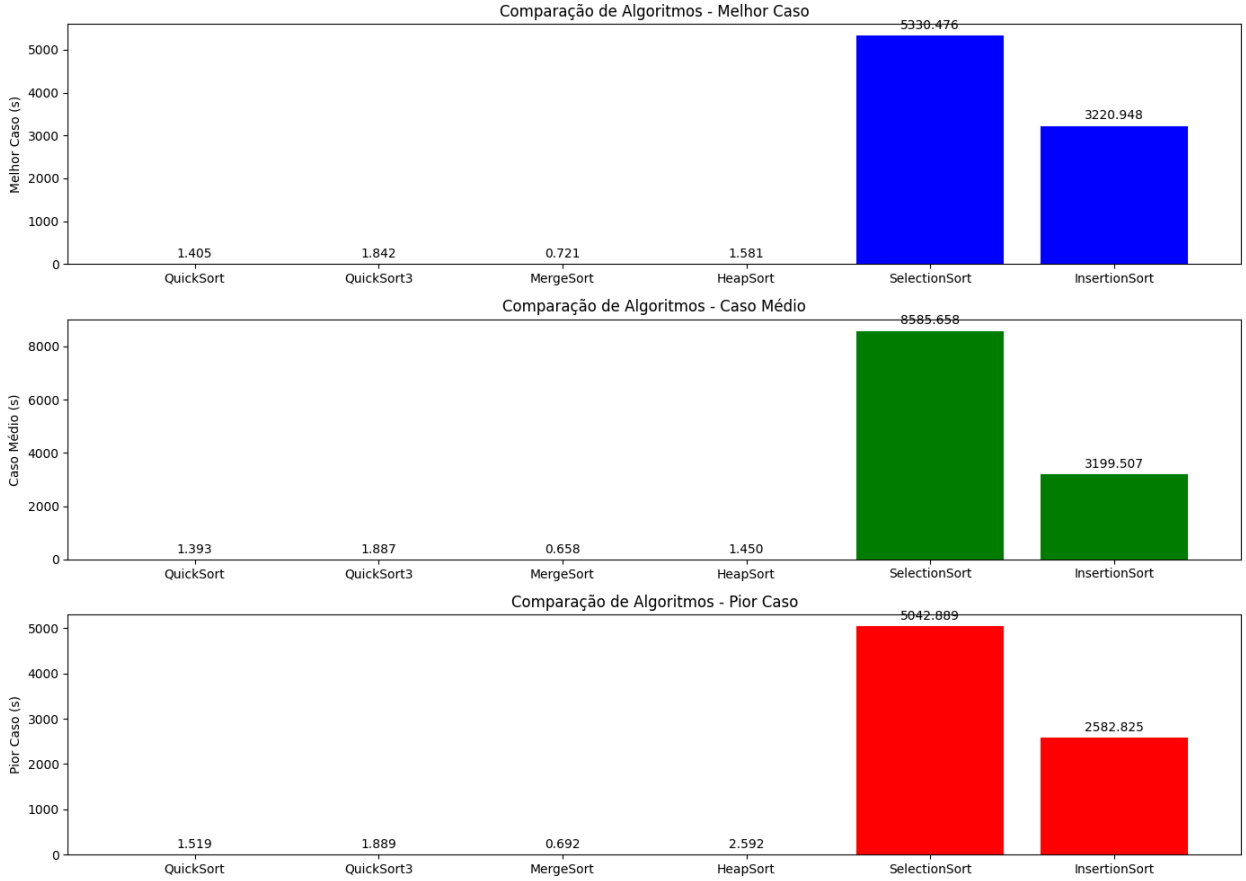
Caso Médio

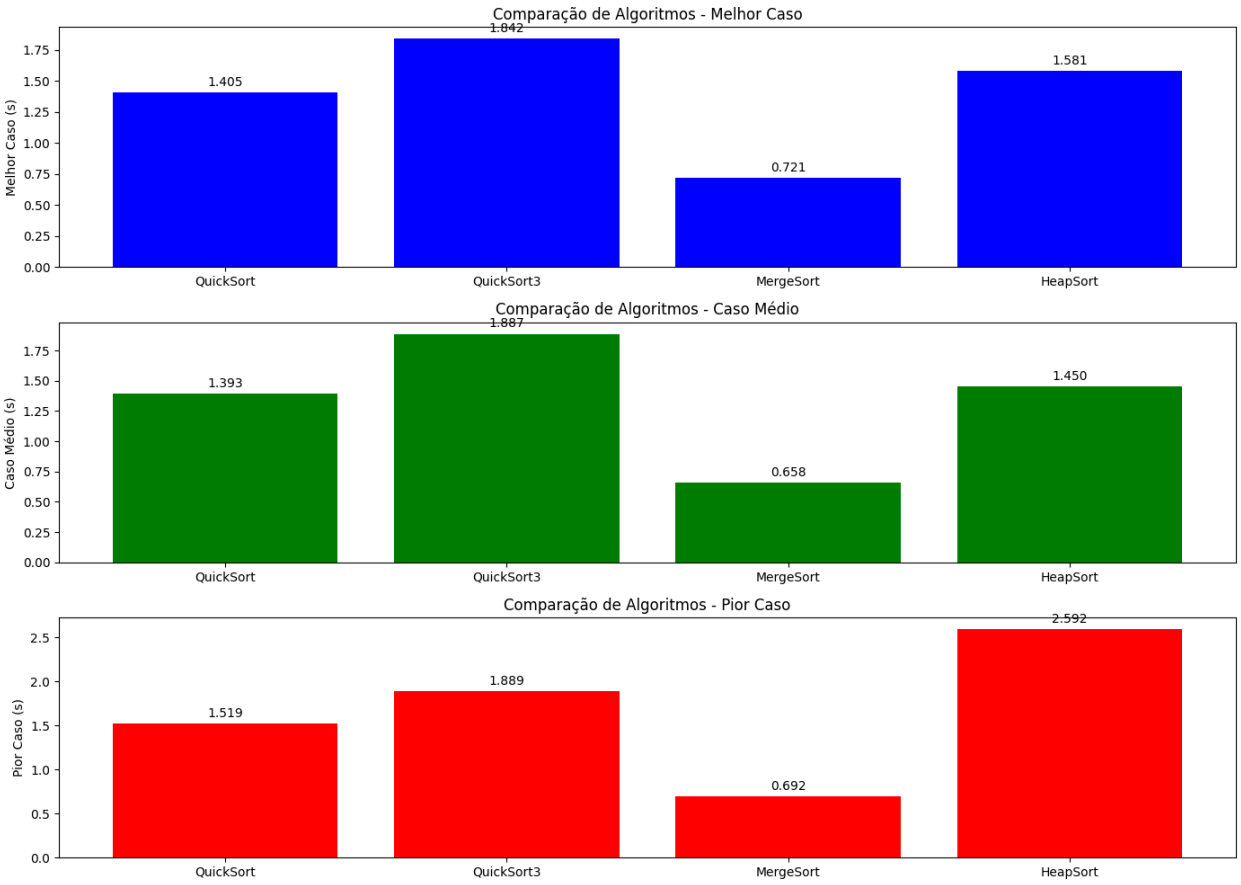
1. **MergeSort**: 0.658 s
2. **QuickSort**: 1.393 s
3. **HeapSort**: 1.450 s
4. **QuickSort3**: 1.887 s
5. **InsertionSort**: 3199.507 s
6. **SelectionSort**: 8585.658 s

Pior Caso

1. **MergeSort**: 0.692 s
2. **QuickSort**: 1.519 s
3. **QuickSort3**: 1.889 s
4. **HeapSort**: 2.592 s
5. **InsertionSort**: 2582.825 s

6. **SelectionSort**: 5042.889 s





Conclusões:

- **MergeSort** é consistentemente o mais rápido em todos os casos, por uma diferença de tempo considerável.
- **QuickSort**, **QuickSort Mediana de 3** e **HeapSort** são geralmente rápidos, mas o HeapSort tem uma pior performance no pior caso entre eles.
- **SelectionSort** é o mais lento em todos os casos.
- **InsertionSort** tem uma performance meio termo, mas ainda significativamente mais lenta que os algoritmos mais rápidos.

Análise algoritmos mais rápidos ao mais lentos(comment_count):

Fazendo uma análise, podemos perceber que em diversas situações (mesmo as críticas) os algoritmos que obtiveram uma melhor performance foram(ordem do maior para o menor de cima para baixo):

Melhor Caso

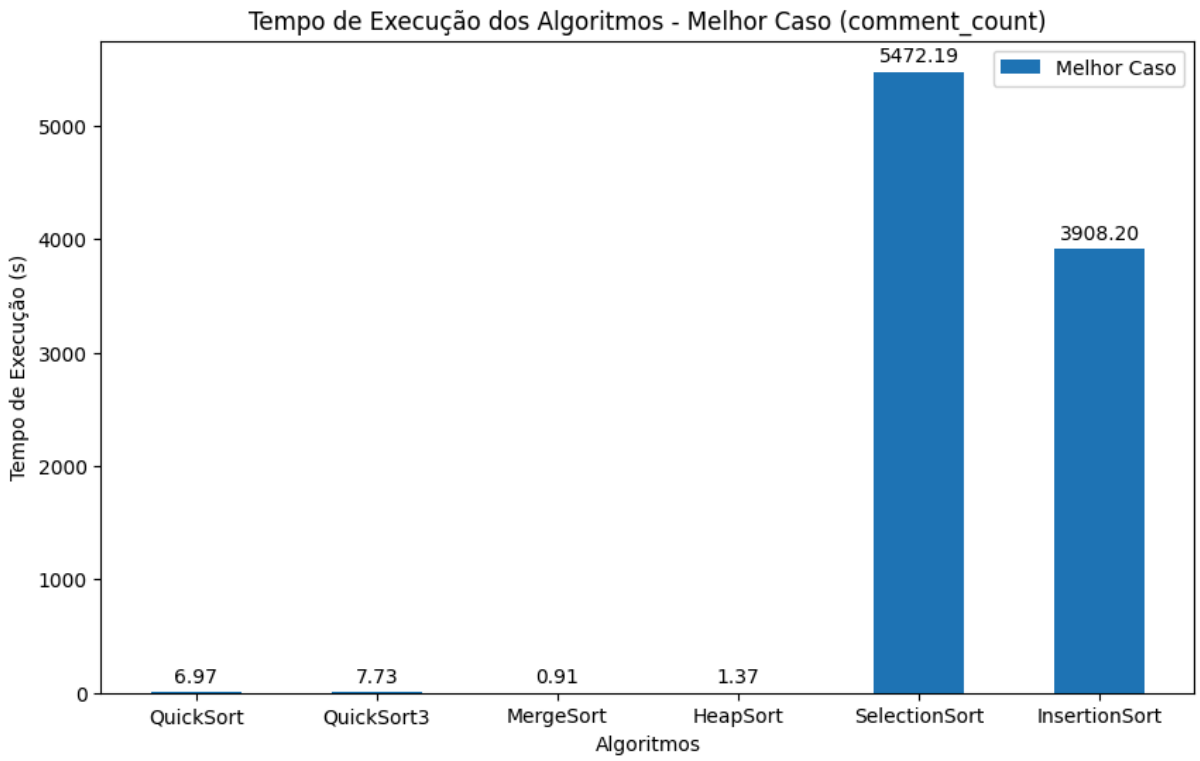
1. **MergeSort:** 0.906s
2. **HeapSort:** 1.366s
3. **QuickSort:** 6.967s
4. **QuickSort3: 7.727s**
5. **InsertionSort:** 3908.202s
6. **SelectionSort:** 5472.191s

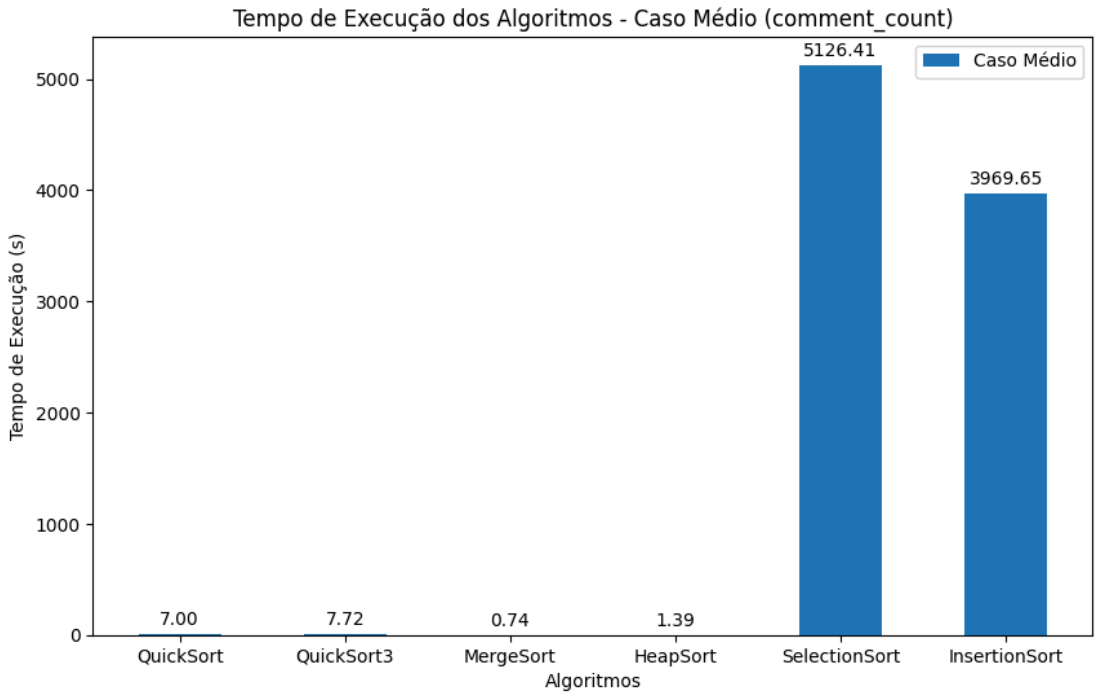
Caso Médio

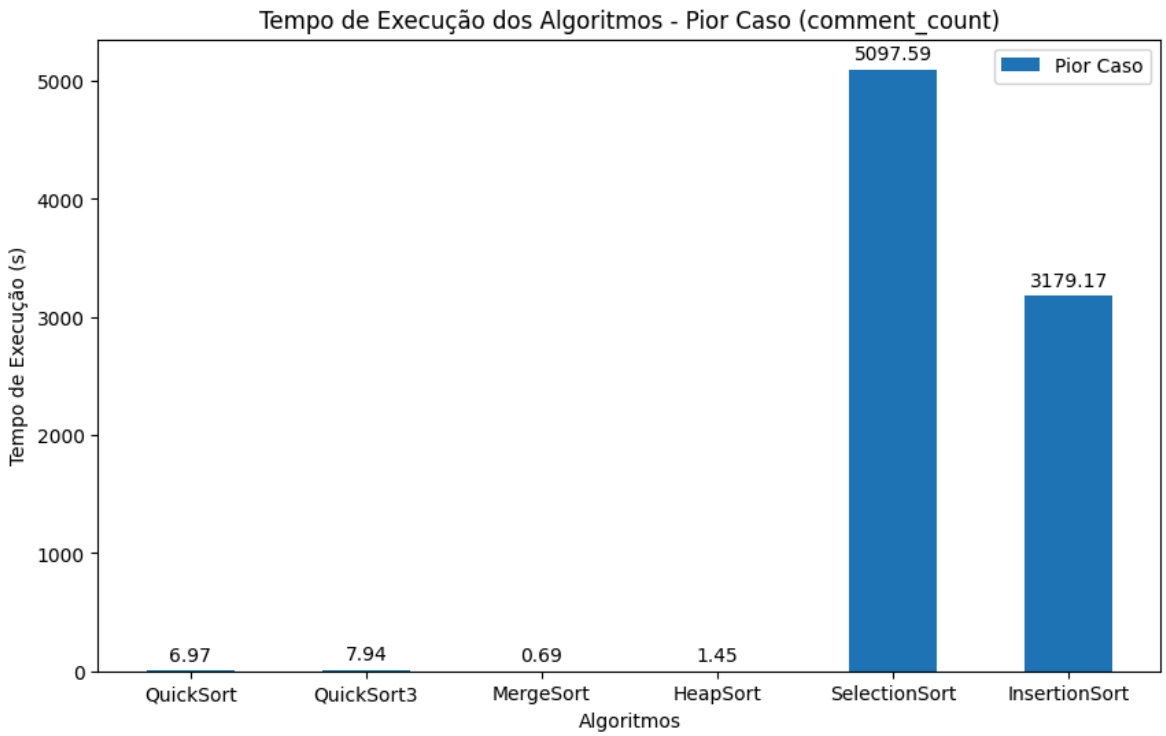
1. **MergeSort:** 0.742s
2. **HeapSort:** 1.392s
3. **QuickSort:** 6.996s
4. **QuickSort3:** 7.725s
5. **InsertionSort:** 3969.650s
6. **SelectionSort:** 5126.409s

Pior Caso

1. **MergeSort:** 0.694s
2. **HeapSort:** 1.453s
3. **QuickSort:** 6.967s
4. **QuickSort3:** 7.938s
5. **InsertionSort:** 3179.169s
6. **SelectionSort:** 5097.589s


Tempo de Execução dos Algoritmos - Melhor Caso (comment_count)
Melhor Caso
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6.97
7.73
0.91
1.37
5472.19
3908.20
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort
Algoritmos


Tempo de Execução dos Algoritmos - Caso Médio (comment_count)
Caso Médio
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
7.00
7.72
0.74
1.39
5126.41
3969.65
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort
Algoritmos



Conclusões:

- **MergeSort** é consistentemente o mais rápido em todos os casos, por uma diferença de tempo considerável. Logo atrás dele tem o **Heap Sort**, com um bom desempenho também.
- **QuickSort** e **QuickSort Mediana de 3** são geralmente rápidos, mas o Quick Mediana de 3 tem uma pior performance no pior caso entre eles.
- **Selection** e o **Insertion** tem o pior tempo de execução de longe.

Análise algoritmos mais rápidos ao mais lentos(trending full_date):

Fazendo uma análise, podemos perceber que em diversas situações (mesmo as críticas) os algoritmos que obtiveram uma melhor performance foram(ordem do maior para o menor de cima para baixo):

Melhor Caso:

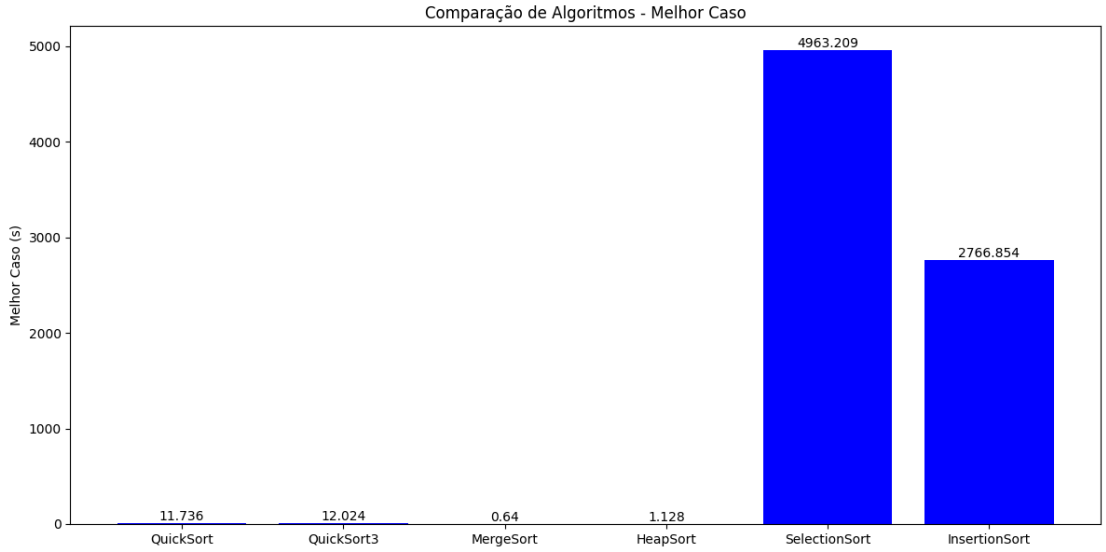
1. **MergeSort:** 0.640 s
2. **HeapSort:** 1.128 s
3. **QuickSort:** 11.736 s
4. **QuickSort3:** 12.024 s
5. **InsertionSort:** 2766.854 s
6. **SelectionSort:** 4963.209 s

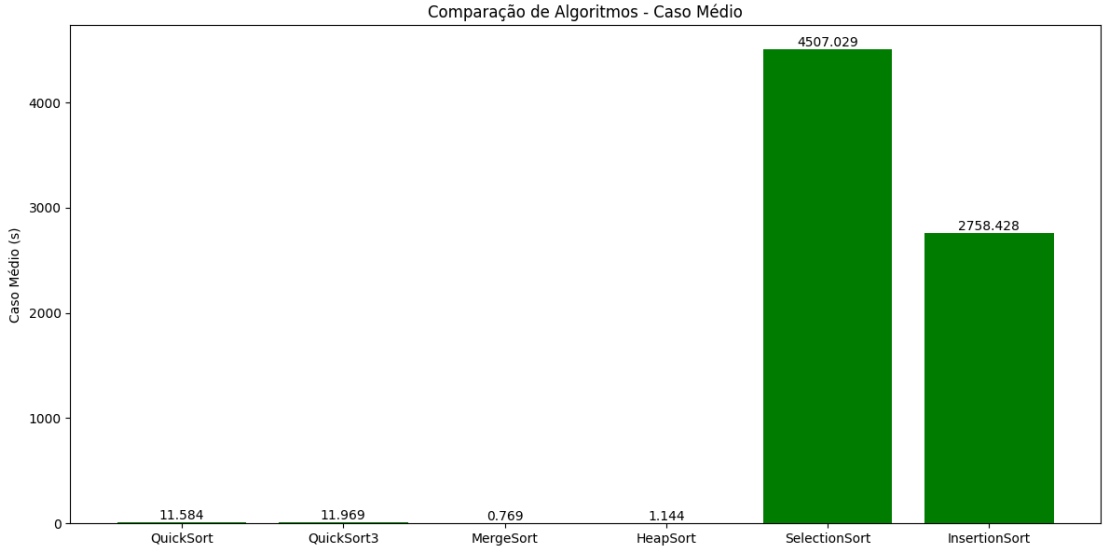
Caso Médio:

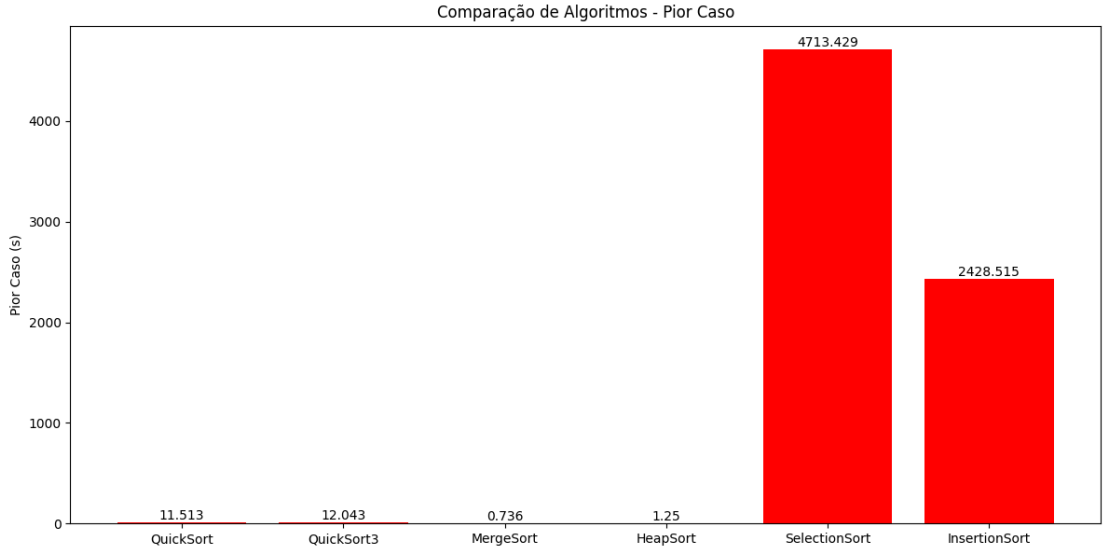
1. **MergeSort:** 0.769 s
2. **HeapSort:** 1.144 s
3. **QuickSort: 11.584 s**
4. **QuickSort3:** 11.969 s
5. **InsertionSort:** 2758.428 s
6. **SelectionSort:** 4507.029 s

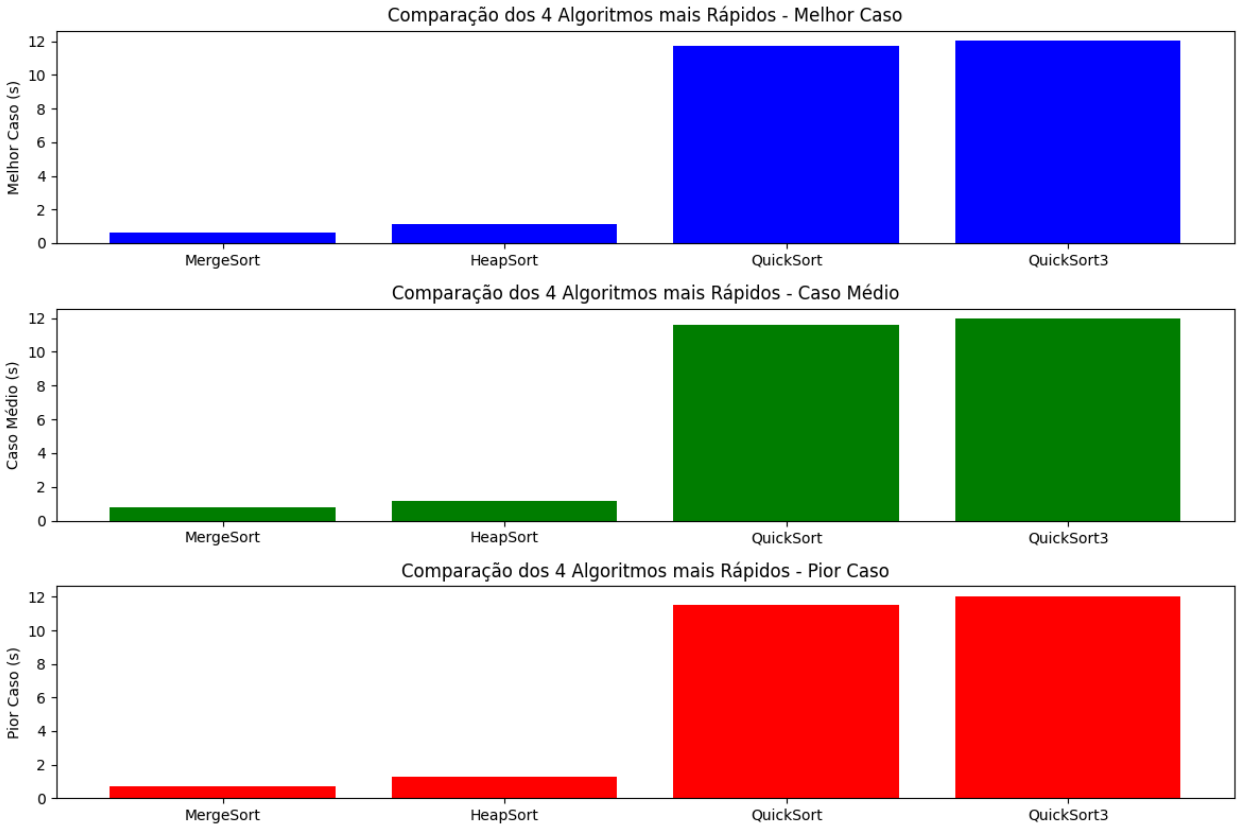
Pior Caso:

1. **MergeSort:** 0.736 s
2. **HeapSort:** 1.250 s
3. **QuickSort:** 11.513 s
4. **QuickSort3:** 12.043 s
5. **InsertionSort:** 2428.515 s
6. **SelectionSort:** 4713.429 s


Comparação de Algoritmos - Melhor Caso
Melhor Caso (s)
0
1000
2000
3000
4000
5000
11.736
12.024
0.64
1.128
4963.209
2766.854
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort


Comparação de Algoritmos - Caso Médio
Caso Médio (s)
0
1000
2000
3000
4000
11.584
11.969
0.769
1.144
4507.029
2758.428
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort


Comparação de Algoritmos - Pior Caso
Pior Caso (s)
0
1000
2000
3000
4000
11.513
12.043
0.736
1.25
4713.429
2428.515
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort



Conclusões:

- **MergeSort**: É consistentemente o mais rápido em todos os casos (melhor, médio e pior).
- **HeapSort**: Também apresenta um desempenho eficiente, sendo o segundo mais rápido, e tem um bom desempenho em todos os casos, embora seja mais lento que o MergeSort.
- **QuickSort** e **QuickSort Mediana de 3**: Ambos são rápidos, mas significativamente mais lentos que MergeSort e HeapSort, e mais rápidos que o Selection e insertion sort, QuickSort mediana de 3 tem um desempenho ligeiramente pior que QuickSort.
- **InsertionSort** e **SelectionSort**: São significativamente muito mais lentos em comparação com os outros algoritmos. Seus tempos de execução são impraticáveis para grandes conjuntos de dados, especialmente em casos piores, devido ao fato que grandes dados demorariam uma eternidade para uma simples ordenação.

OBSERVAÇÕES:

- **OBS: Não levamos em consideração o Counting sort pois não pudemos trabalhar na maioria das ordenações.**
- **OBS 2: Os algoritmos de ordenação gerados não ficaram 100% ordenados pois alguns strings e valores Number não podem ser ordenados devido ao caracteres especiais ou ao seu alto valor, dessa forma esse valores desordenados ficam logo pro inicio do arquivo de ordenação, e mais para o meio temos os arquivos ordenados e tudo mais.**
    - **No caso, do channel_title ele separa tudo em escrito só com a primeira letra maiscula e os escritos em caps lock.**

Análise/Comentários acerca dos resultados obtidos

Com base nos gráficos, na tabela e nas conclusões de cada um dos casos de arquivos, percebemos que o algoritmo que se saiu melhor foi o Merge Sort, muito disso se deve ao bom desempenho que o mesmo tem em ordenação de grandes dados, principalmente em comparação com o QuickSort (outroa algoritmo de divisão e conquista).
(Para mais informações sobre o MergeSort, acesse: aqui ou/e aqui.)

Nesses links, temos mais alguns exemplos de implementação e eficiência do mesmo.

Após o Merge, percebemos que o Algoritmo de ordenação mais eficiente foi o Heap, talvez ele tome grande vantagem do Merge devido ao alocamento de memória, nesse relatório não fizemos uma análise do alocamento de memória, mas no próximo iremos tentar colocar em prática, mas a figura a seguir mostra bem sobre isso:

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)

Percebe-se que o tempo do Merge e Heap aparece como igual, por isso eles são tão parecidos em tempos bons de execução, mas o Heap toma vantagem no alocamento de espaço.

Os algoritmos QuickSort e QuickSort Mediana de 3, tem um desempenho razoável, muito devido ao alocamento de memória também, além dos mesmos terem um desempenho parecido no melhor e médio caso.

Já os algoritmos Selection e Insertion sort, mostram que definitivamente só devem ser usados em grandes dados, como o do projeto, no pior caso possível, já que o tempo dos mesmos é altíssimo, o que é péssimo em questão de desempenho. Imagine que você quer fazer uma ordenação simples de dados rápida, usando esses 2, isso poderia durar até certa de 2 Horas.

Enfim, dessa forma, tiramos a conclusão que o **melhor algoritmo de ordenação** é o Merge Sort, Seguido o Heap e Quick. E de fora deixamos Selection e Insertion, devido ao péssimo desempenho deles.

Referências:

https://www.freecodecamp.org/news/sorting-algorithms-explained-with-examples-in-python-java-and-c/#:~:text=What%20is%20a%20Sorting%20Algorithm,%2C%209-0)%20order.

https://en.wikipedia.org/wiki/Sorting_algorithm

https://www.geeksforgeeks.org/sorting-algorithms/

https://visualgo.net/en/sorting?slide=1