Laboratório de Estrutura de Dados

Segunda 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 conjunto 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, em nossa primeira versão, fizemos 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).

Para nossa segunda versão, iremos tratar de algoritmos de ordenação, agora com estrutura de dados diferentes.

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, usamos os dados que temos para serem rodados por diferentes tipos de algoritmos de ordenação Insertion, Selection, Merge, etc, e mostramos e debatemos sobre qual o melhor algoritmo de ordenação, utilizando melhor médio e pior caso. Para esta versão atual, implementamos também com novos tipos estruturais de dados: lista duplamente encadeada, árvore binária e tabela hash.
- **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”~~

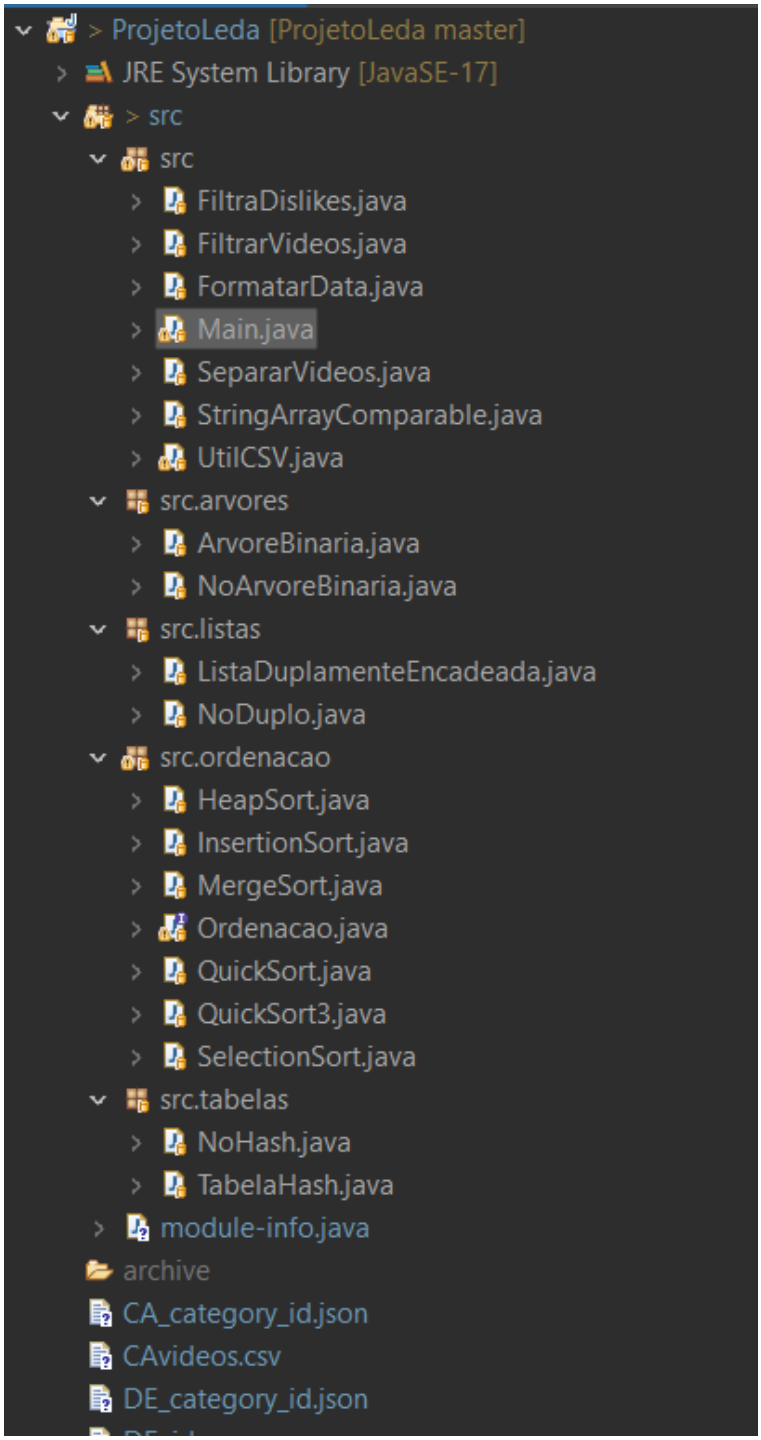
- Utilizamos o arquivo que no GitHub estará como FiltrarDislikes.java
    - Ele é responsável por criar o “videos_T2.csv”.

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.

Para este, criamos classes com cada estrutura de dados e seus respectivos nós:

Os arquivos que fizemos foram organizados dessa forma:



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 em relação a sua estrutura de dados, 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

Feito com Arrays:

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

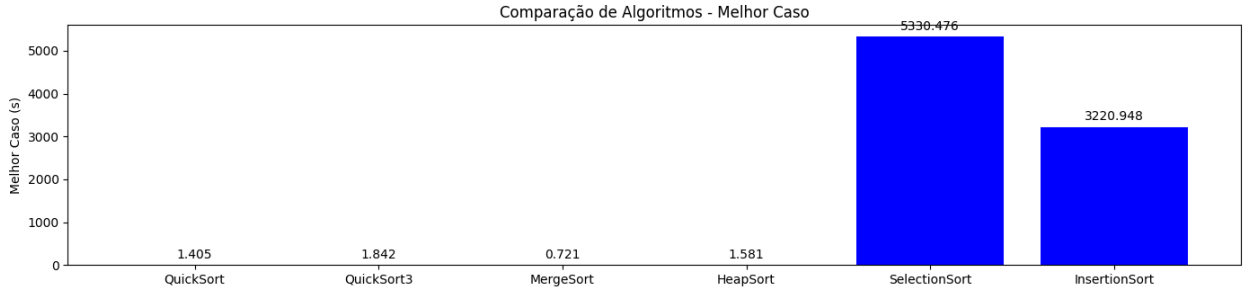
6. **SelectionSort**: 5330.476 s

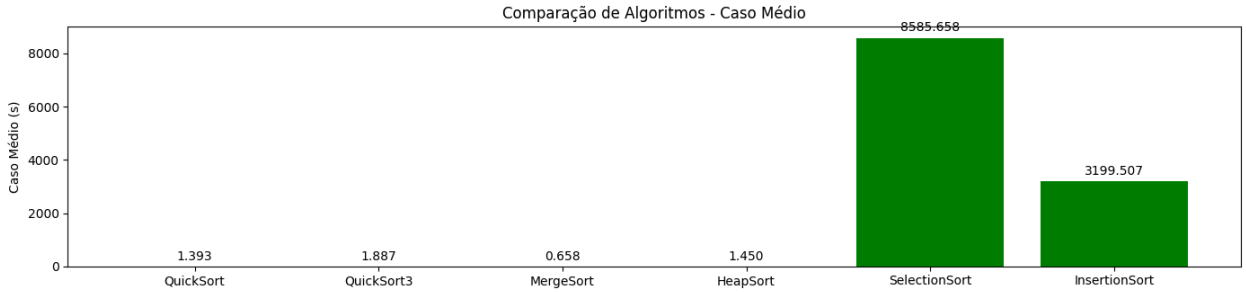
Caso Médio

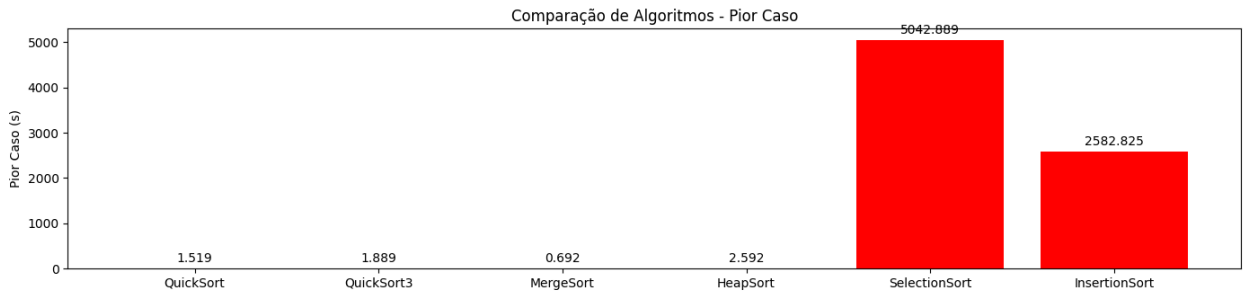
1. **MergeSort**: 0.658 s
2. **QuickSort**: 1.393 s
3. **HeapSort**: 1.450 s
4. **QuickSort3**: 1.887 s
5. **InsertionSort**: 3199.507 s
6. **SelectionSort**: 8585.658 s

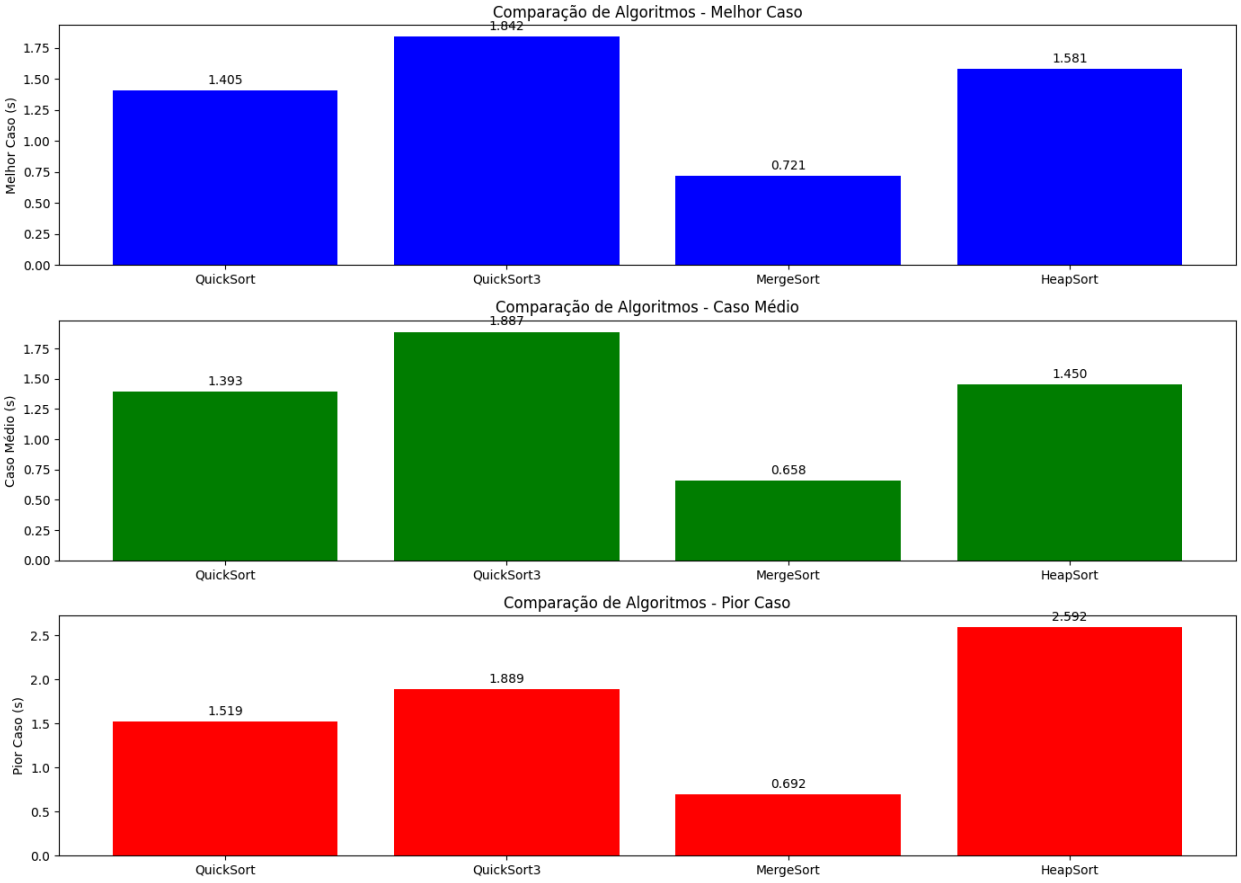
Pior Caso

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


Comparação de Algoritmos - Melhor Caso
Melhor Caso (s)
0
1000
2000
3000
4000
5000
1.405
1.842
0.721
1.581
5330.476
3220.948
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort


Comparação de Algoritmos - Caso Médio
Caso Médio (s)
0
2000
4000
6000
8000
1.393
1.887
0.658
1.450
8585.658
3199.507
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort


Comparação de Algoritmos - Pior Caso
Pior Caso (s)
0
1000
2000
3000
4000
5000
1.519
1.889
0.692
2.592
5042.889
2582.825
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort



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

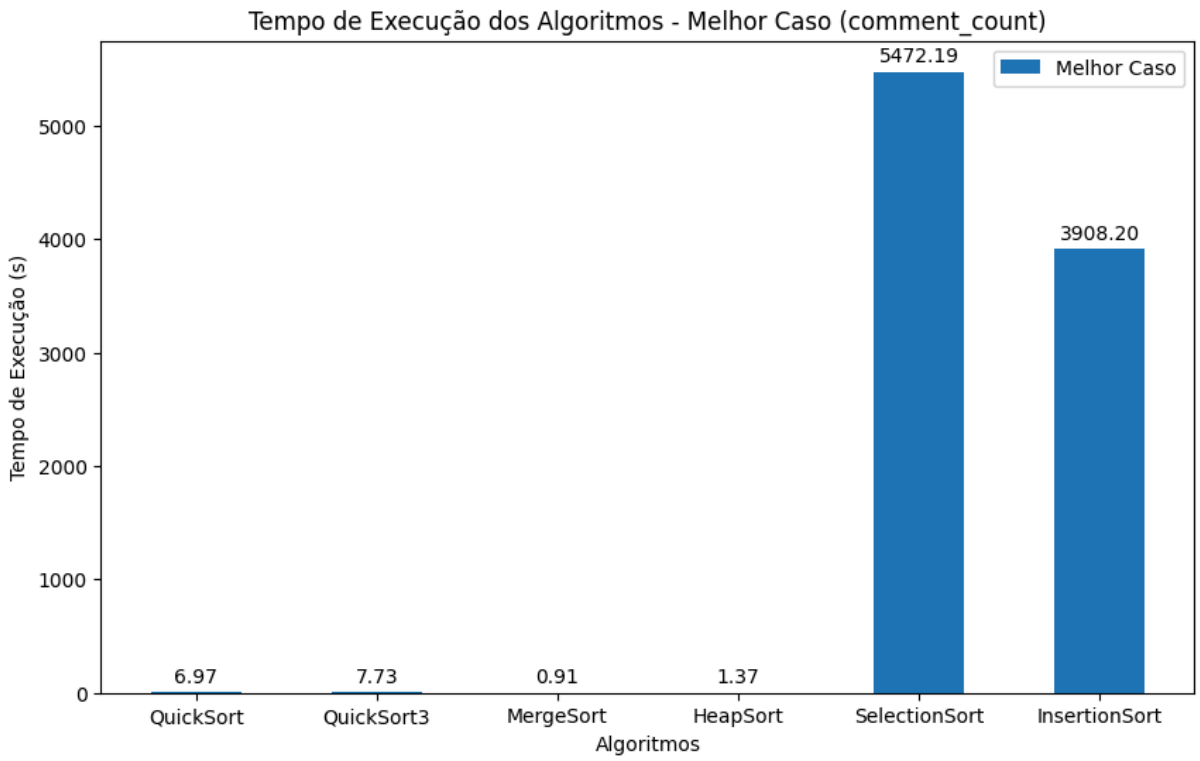
1. **MergeSort:** 0.906s
2. **HeapSort:** 1.366s
3. **QuickSort:** 6.967s
4. **QuickSort3: 7.727s**
5. **InsertionSort:** 3908.202s
6. **SelectionSort:** 5472.191s

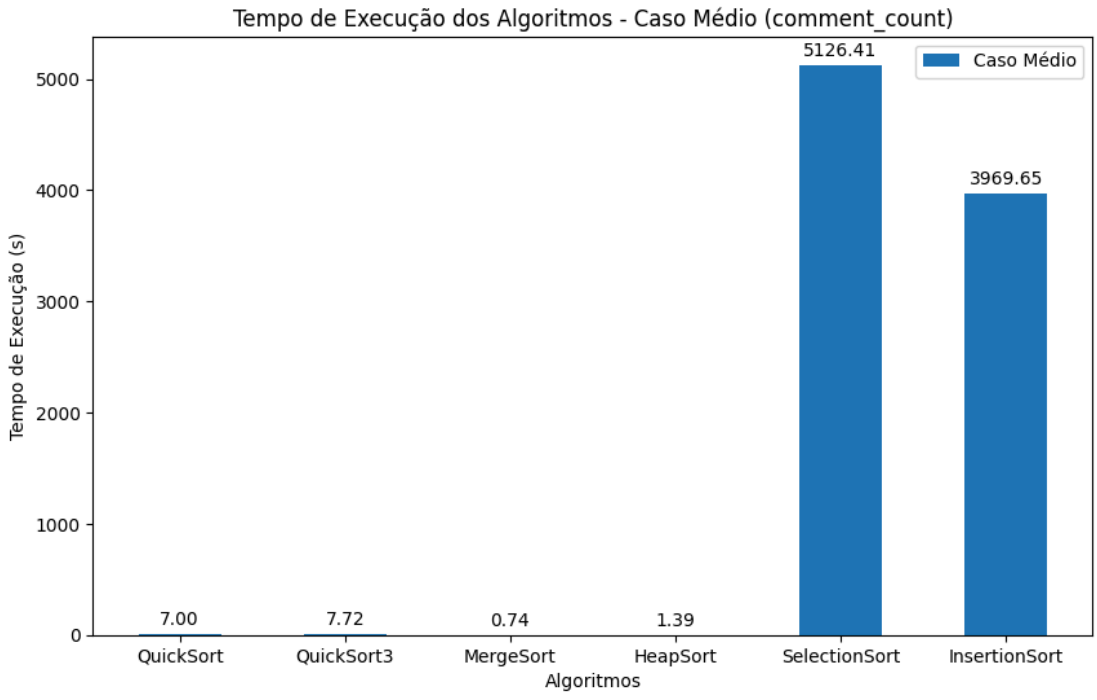
Caso Médio

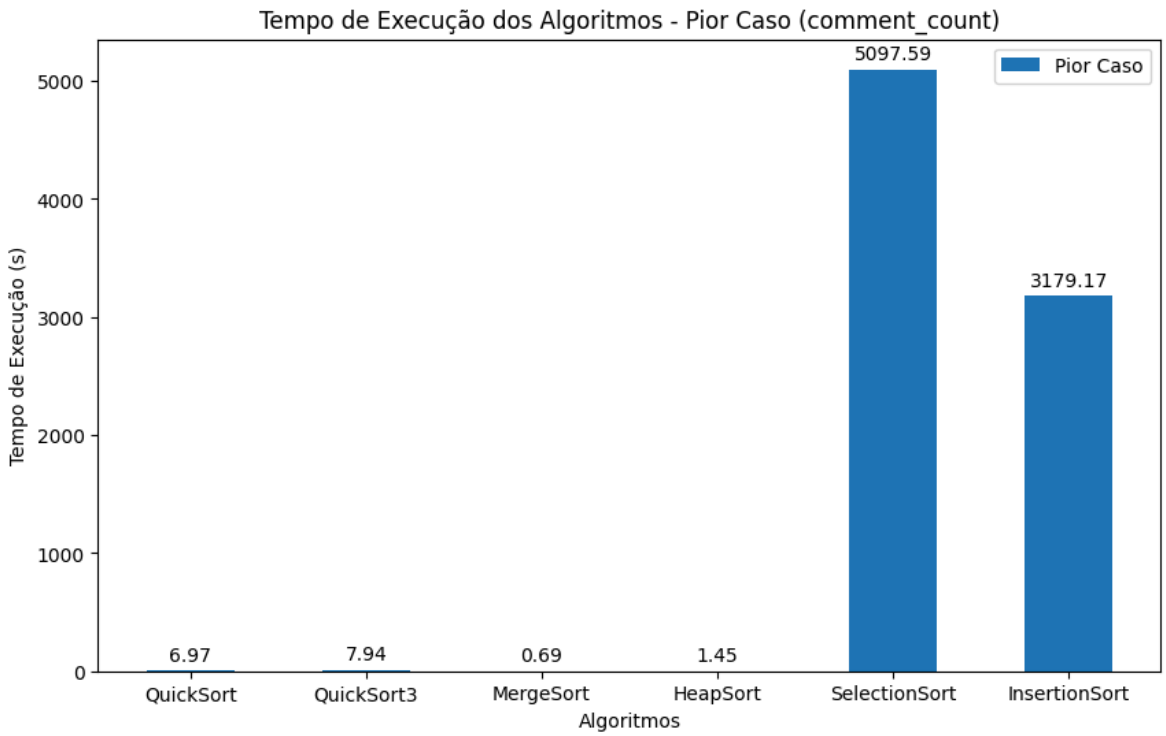
1. **MergeSort:** 0.742s
2. **HeapSort:** 1.392s
3. **QuickSort:** 6.996s
4. **QuickSort3:** 7.725s
5. **InsertionSort:** 3969.650s
6. **SelectionSort:** 5126.409s

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:

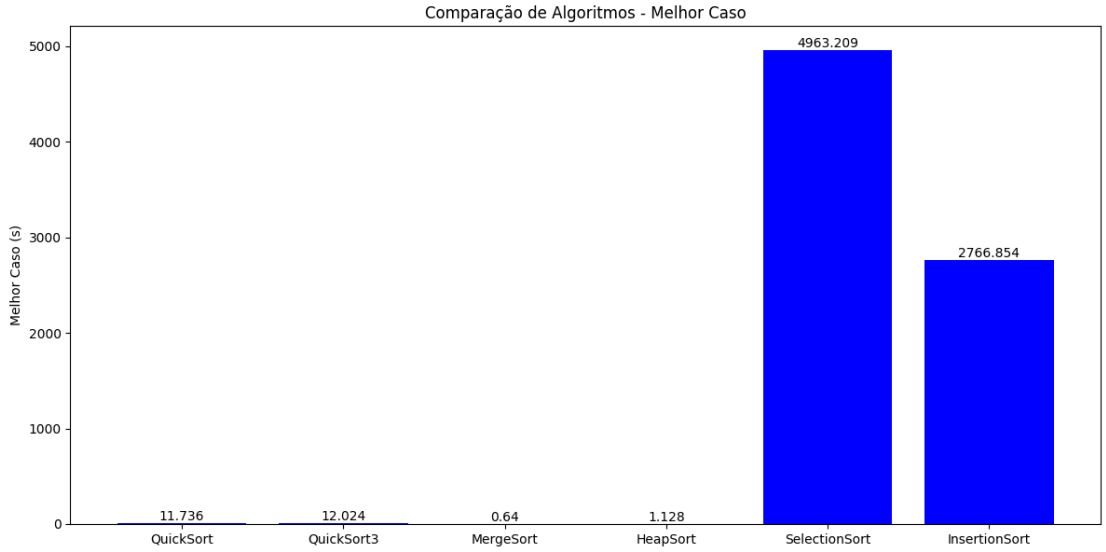
1. **MergeSort:** 0.640 s
2. **HeapSort:** 1.128 s
3. **QuickSort:** 11.736 s
4. **QuickSort3:** 12.024 s
5. **InsertionSort:** 2766.854 s
6. **SelectionSort:** 4963.209 s

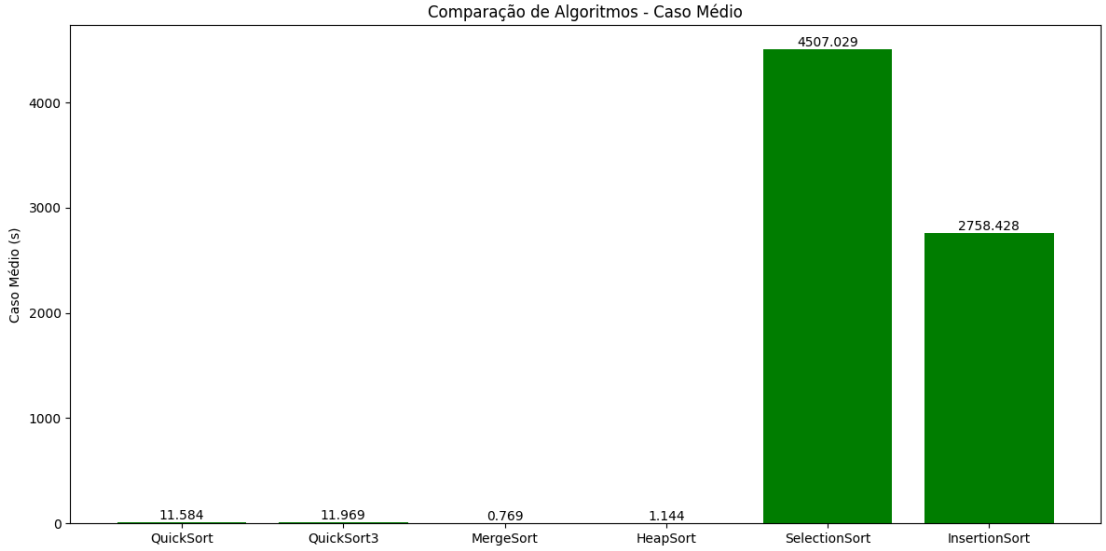
Caso Médio:

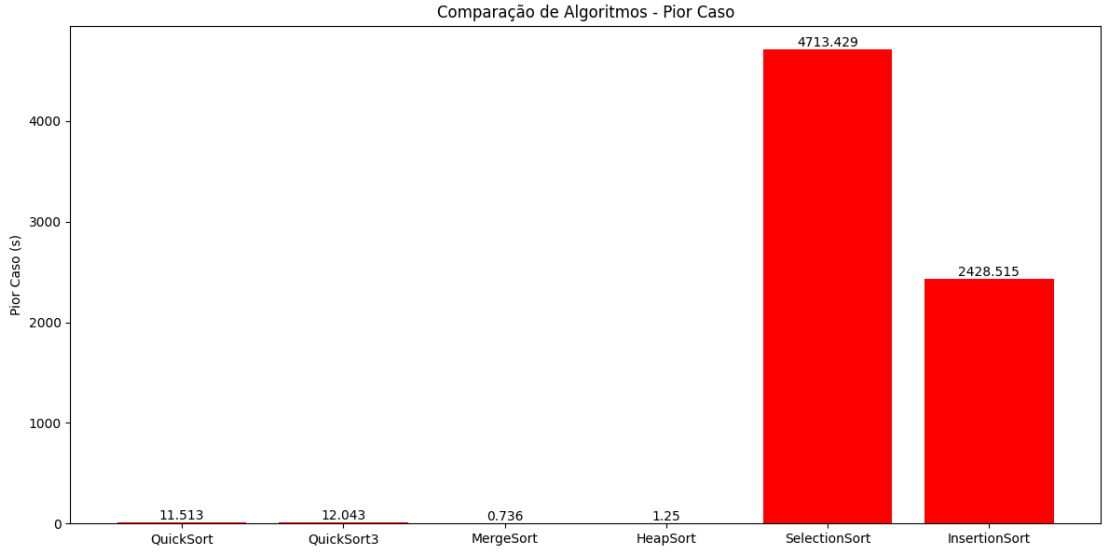
1. **MergeSort:** 0.769 s
2. **HeapSort:** 1.144 s
3. **QuickSort: 11.584 s**
4. **QuickSort3:** 11.969 s
5. **InsertionSort:** 2758.428 s
6. **SelectionSort:** 4507.029 s

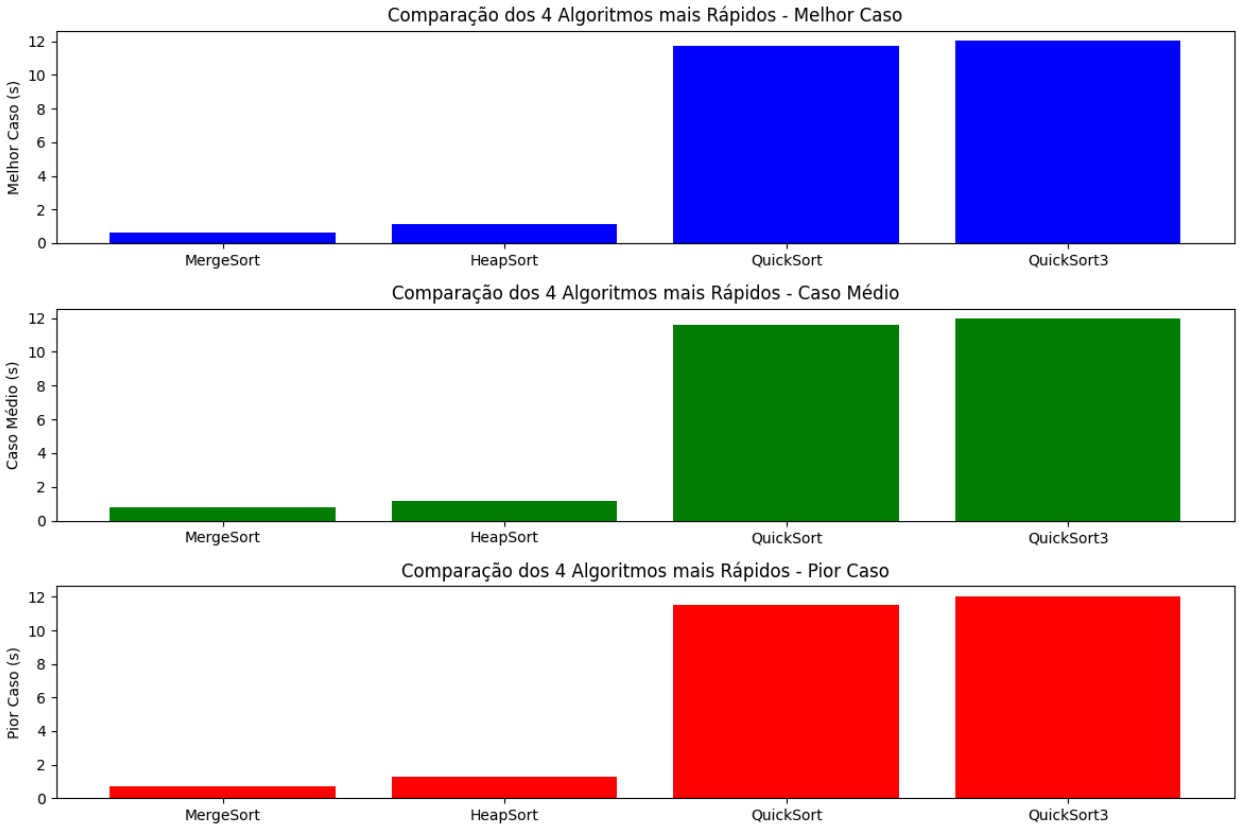
Pior Caso:

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



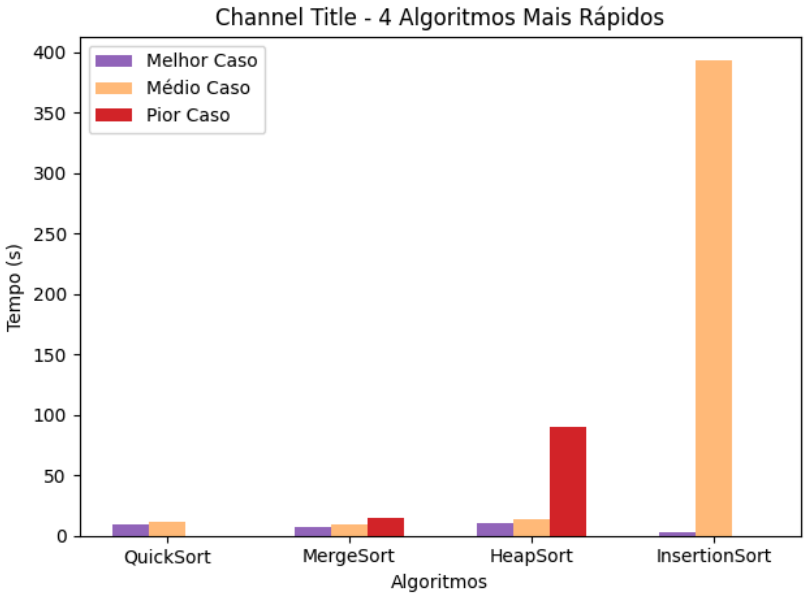
Conclusões:

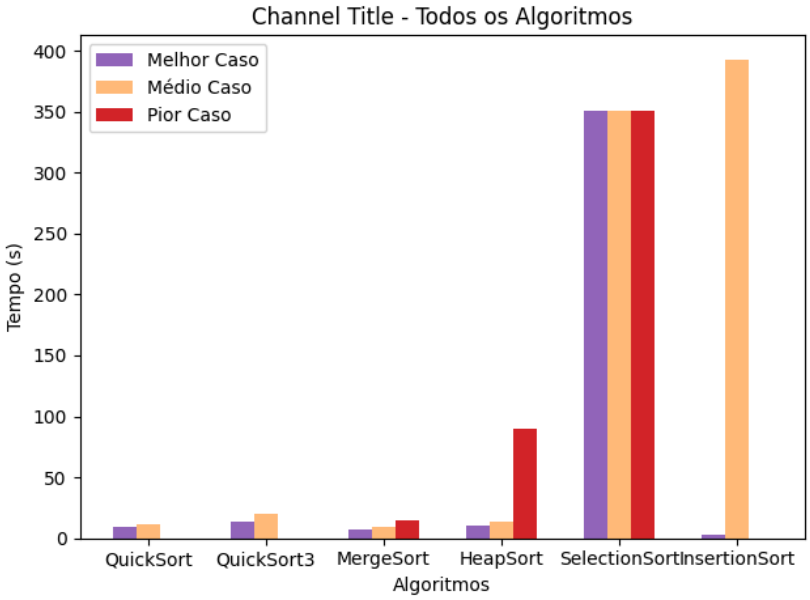
- **MergeSort**: É consistentemente o mais rápido em todos os casos (melhor, médio e pior).
- **HeapSort**: Também apresenta um desempenho eficiente, sendo o segundo mais rápido, e tem um bom desempenho em todos os casos, embora seja mais lento que o MergeSort.
- **QuickSort** e **QuickSort Mediana de 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 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

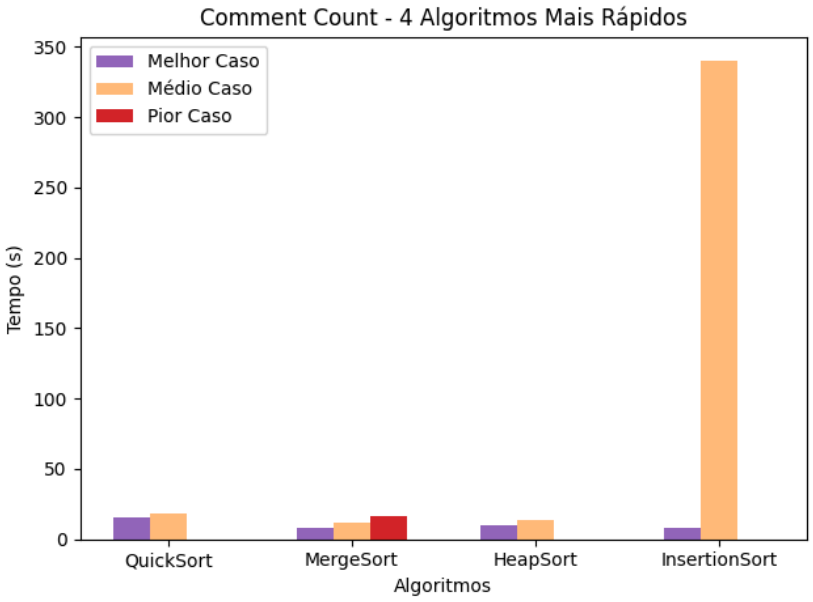
dados iriam demorar uma quantidade de tempo imensurável para uma simples ordenação.

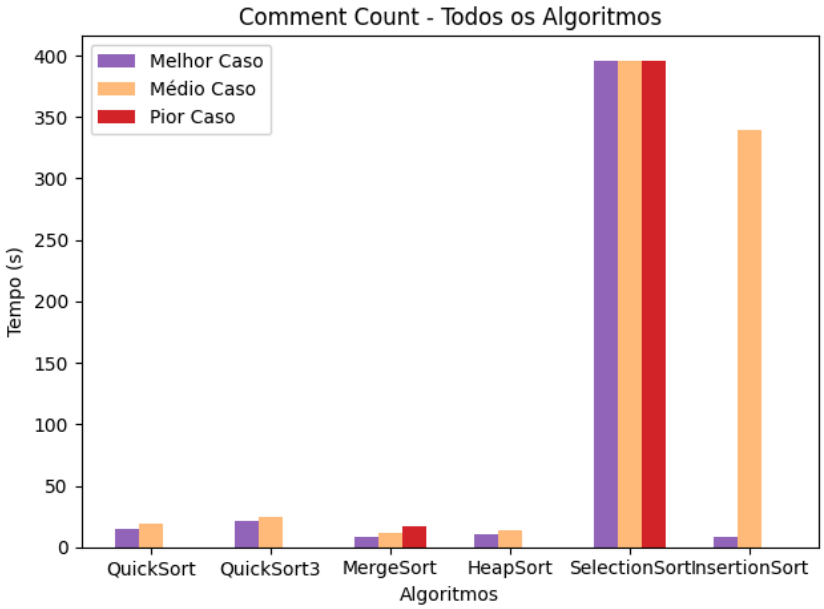
Resultados das ordenações com ***lista duplamente encadeada***:

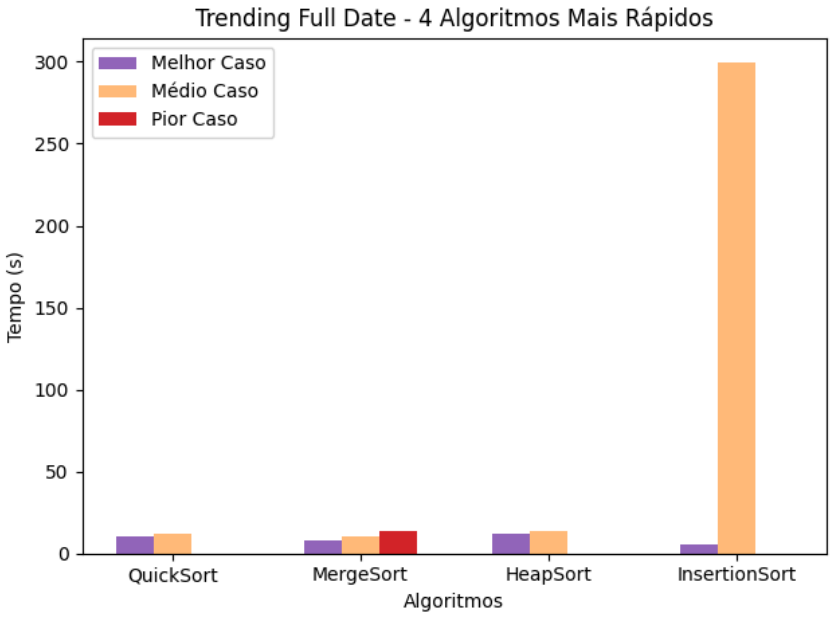
Algoritmo	Campo	Melhor caso (s)	Médio caso (s)	Pior caso (s)
QuickSort	channel_title	9.465	11.3	-
QuickSort3	channel_title	13.837	19.922	-
MergeSort	channel_title	7.680	9.652	14.213
HeapSort	channel_title	10.921	13.551	89.737
SelectionSort	channel_title	351.041	351.041	351.041
InsertionSort	channel_title	3.124	392.983	-
QuickSort	comment_count	15.234	18.732	-
QuickSort3	comment_count	21.836	24.823	-
MergeSort	comment_count	7.822	11.495	16.636
HeapSort	comment_count	10.126	14.118	-
SelectionSort	comment_count	396.142	396.142	396.142
InsertionSort	comment_count	7.931	340.018	-
QuickSort	trending_full_date	10.141	12.268	-
QuickSort3	trending_full_date	18.287	19.182	-
MergeSort	trending_full_date	8.211	10.615	13.921
HeapSort	trending_full_date	11.921	13.921	-
SelectionSort	trending_full_date	311.921	311.921	311.921
InsertionSort	trending_full_date	5.429	299.412	-

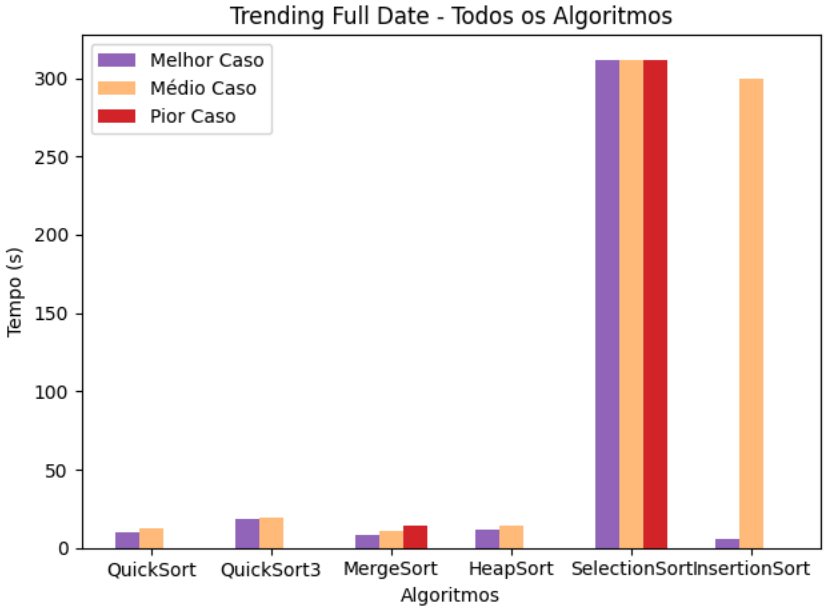

Channel Title - 4 Algoritmos Mais Rápidos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
50
100
150
200
250
300
350
400
QuickSort
MergeSort
HeapSort
InsertionSort
Algoritmos


Channel Title - Todos os Algoritmos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
50
100
150
200
250
300
350
400
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort
Algoritmos


Comment Count - 4 Algoritmos Mais Rápidos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
50
100
150
200
250
300
350
QuickSort
MergeSort
HeapSort
InsertionSort
Algoritmos


Comment Count - Todos os Algoritmos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
50
100
150
200
250
300
350
400
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort
Algoritmos





Lista Duplamente Encadeada

Channel Title

Melhor Caso

1. MergeSort - 7.680s
2. QuickSort - 9.465s
3. HeapSort - 10.921s
4. QuickSort3 - 13.837s
5. InsertionSort - 3.124s
6. SelectionSort - 351.041s

Médio Caso

1. MergeSort - 9.652s
2. QuickSort - 11.3s
3. HeapSort - 13.551s
4. QuickSort3 - 19.922s
5. InsertionSort - 392.983s
6. SelectionSort - 351.041s

Pior Caso

1. MergeSort - 14.213s
2. InsertionSort - 0s (não informado)
3. HeapSort - 89.737s
4. QuickSort3 - 0s (não informado)
5. QuickSort - 0s (não informado)
6. SelectionSort - 351.041s

Comment Count

Melhor Caso

1. MergeSort - 7.822s
2. InsertionSort - 7.931s
3. HeapSort - 10.126s
4. QuickSort - 15.234s

5. QuickSort3 - 21.836s
6. SelectionSort - 396.142s

Médio Caso

1. MergeSort - 11.495s
2. HeapSort - 14.118s
3. QuickSort - 18.732s
4. QuickSort3 - 24.823s
5. InsertionSort - 340.018s
6. SelectionSort - 396.142s

Pior Caso

1. MergeSort - 16.636s
2. InsertionSort - 0s (não informado)
3. HeapSort - 0s (não informado)
4. QuickSort3 - 0s (não informado)
5. QuickSort - 0s (não informado)
6. SelectionSort - 396.142s

Trending Full Date

Melhor Caso

1. MergeSort - 8.211s
2. InsertionSort - 5.429s
3. QuickSort - 10.141s
4. HeapSort - 11.921s
5. QuickSort3 - 18.287s
6. SelectionSort - 311.921s

Médio Caso

1. MergeSort - 10.615s
2. QuickSort - 12.268s

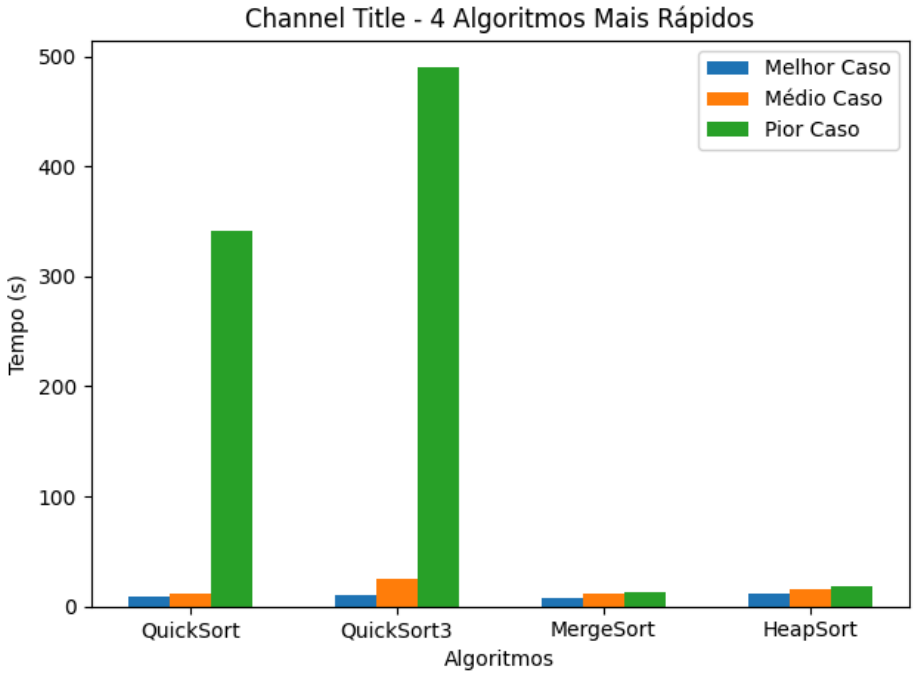
3. HeapSort - 13.921s
4. QuickSort3 - 19.182s
5. InsertionSort - 299.412s
6. SelectionSort - 311.921s

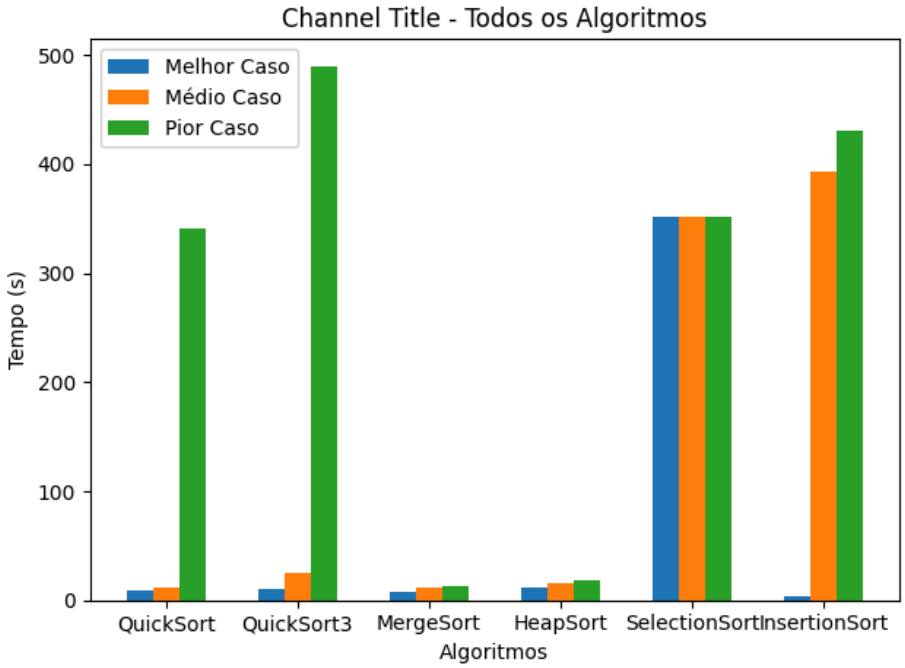
Pior Caso

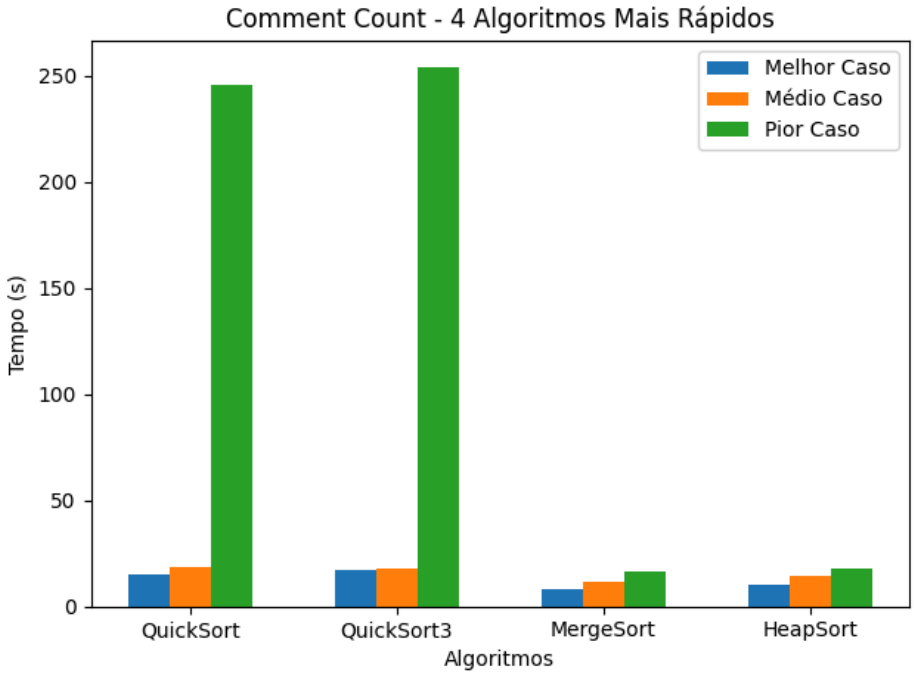
1. MergeSort - 13.921s
2. InsertionSort - 0s (não informado)
3. HeapSort - 0s (não informado)
4. QuickSort3 - 0s (não informado)
5. QuickSort - 0s (não informado)
6. SelectionSort - 311.921s

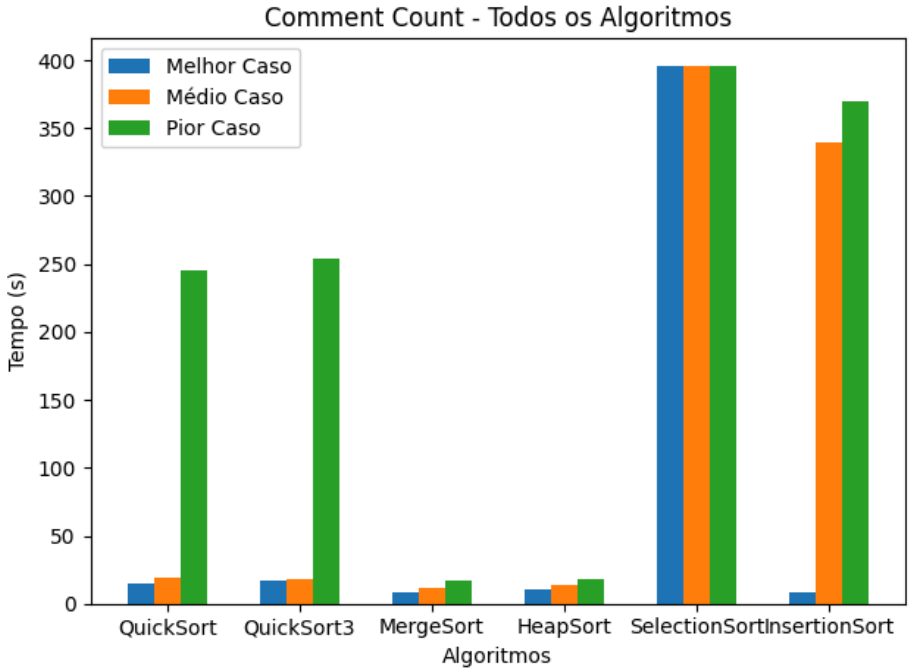
Resultados das ordenações com ***tabela Hash***:

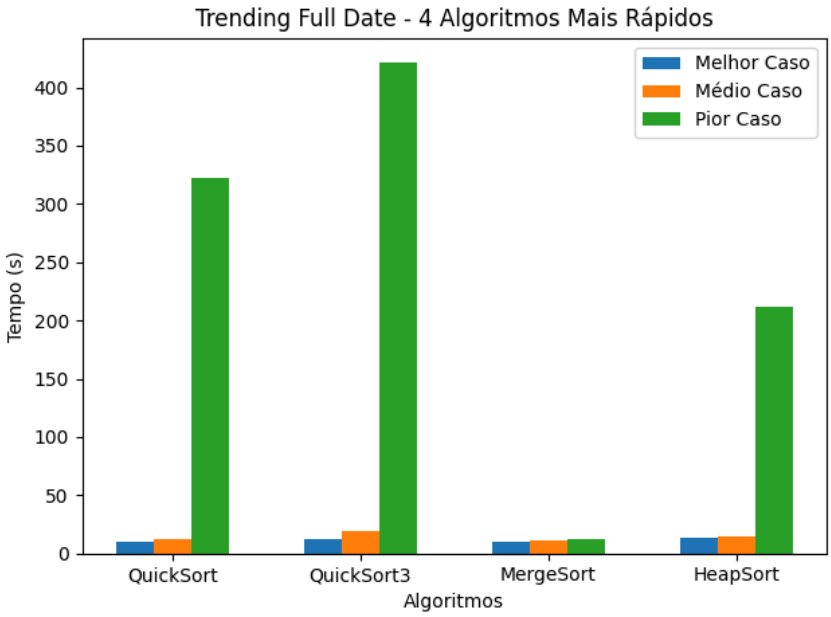
Algoritmo	Campo	Melhor caso (s)	Médio caso (s)	Pior caso (s)
QuickSort	channel_title	8.881	11.5	341.440
QuickSort3	channel_title	10.938	24.932	490.003
MergeSort	channel_title	8.151	11.424	13.009
HeapSort	channel_title	11.271	15.124	18.421
SelectionSort	channel_title	351.041	351.041	351.041
InsertionSort	channel_title	3.124	392.983	431.104
QuickSort	comment_count	15.234	18.732	245.857
QuickSort3	comment_count	17.124	18.194	253.98
MergeSort	comment_count	7.822	11.495	16.636
HeapSort	comment_count	10.126	14.118	17.767
SelectionSort	comment_count	396.142	396.142	396.142
InsertionSort	comment_count	7.931	340.018	369.241
QuickSort	trending_full_date	10.141	12.268	321.901
QuickSort3	trending_full_date	12.138	19.828	421.312
MergeSort	trending_full_date	9.531	10.824	12.124
HeapSort	trending_full_date	13.120	15.027	211.549
SelectionSort	trending_full_date	497.211	497.211	497.211
InsertionSort	trending_full_date	9.521	317.854	454.902


Channel Title - 4 Algoritmos Mais Rápidos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
100
200
300
400
500
QuickSort
QuickSort3
MergeSort
HeapSort
Algoritmos


Channel Title - Todos os Algoritmos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
100
200
300
400
500
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort
Algoritmos


Comment Count - 4 Algoritmos Mais Rápidos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
50
100
150
200
250
QuickSort
QuickSort3
MergeSort
HeapSort
Algoritmos


Comment Count - Todos os Algoritmos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
50
100
150
200
250
300
350
400
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort
Algoritmos


Trending Full Date - 4 Algoritmos Mais Rápidos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
50
100
150
200
250
300
350
400
QuickSort
QuickSort3
MergeSort
HeapSort
Algoritmos

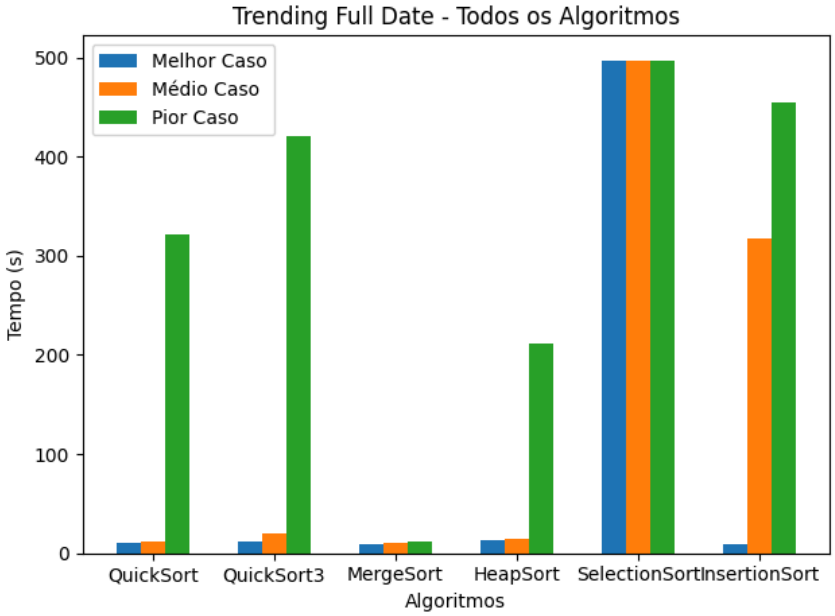

Trending Full Date - Todos os Algoritmos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
100
200
300
400
500
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort
Algoritmos

Tabela Hash

Channel Title

Melhor Caso

1. MergeSort - 8.151s
2. QuickSort - 8.881s
3. HeapSort - 11.271s
4. QuickSort3 - 10.938s
5. SelectionSort - 351.041s
6. InsertionSort - 3.124s

Médio Caso

1. MergeSort - 11.424s
2. QuickSort - 11.5s
3. HeapSort - 15.124s
4. QuickSort3 - 24.932s
5. SelectionSort - 351.041s
6. InsertionSort - 392.983s

Pior Caso

1. MergeSort - 13.009s
2. HeapSort - 18.421s
3. QuickSort - 341.440s
4. QuickSort3 - 490.003s
5. SelectionSort - 351.041s
6. InsertionSort - 431.104s

Comment Count

Melhor Caso

1. MergeSort - 7.822s
2. InsertionSort - 7.931s
3. HeapSort - 10.126s
4. QuickSort - 15.234s
5. QuickSort3 - 17.124s
6. SelectionSort - 396.142s

Médio Caso

1. MergeSort - 11.495s
2. HeapSort - 14.118s
3. QuickSort - 18.732s
4. QuickSort3 - 18.194s
5. InsertionSort - 340.018s
6. SelectionSort - 396.142s

Pior Caso

1. MergeSort - 16.636s
2. HeapSort - 17.767s
3. QuickSort - 245.857s
4. QuickSort3 - 253.98s
5. InsertionSort - 369.241s
6. SelectionSort - 396.142s

Trending Full Date

Melhor Caso

1. MergeSort - 9.531s
2. InsertionSort - 9.521s
3. QuickSort - 10.141s
4. QuickSort3 - 12.138s
5. HeapSort - 13.120s
6. SelectionSort - 497.211s

Médio Caso

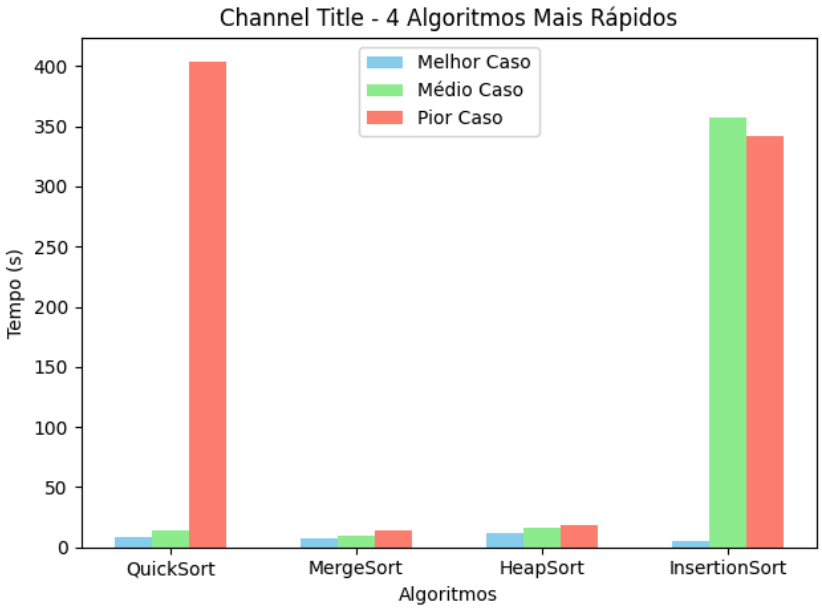
1. MergeSort - 10.824s
2. QuickSort - 12.268s
3. HeapSort - 15.027s
4. QuickSort3 - 19.828s
5. InsertionSort - 317.854s
6. SelectionSort - 497.211s

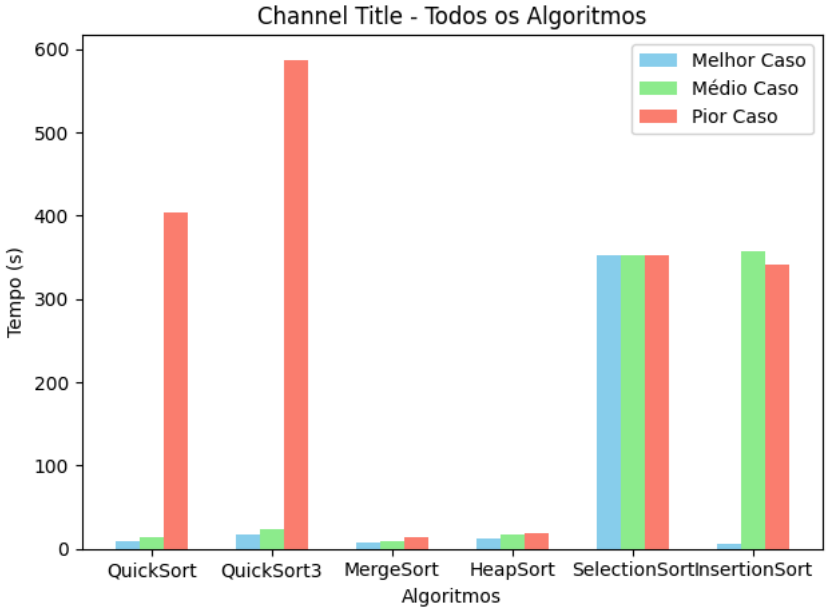
Pior Caso

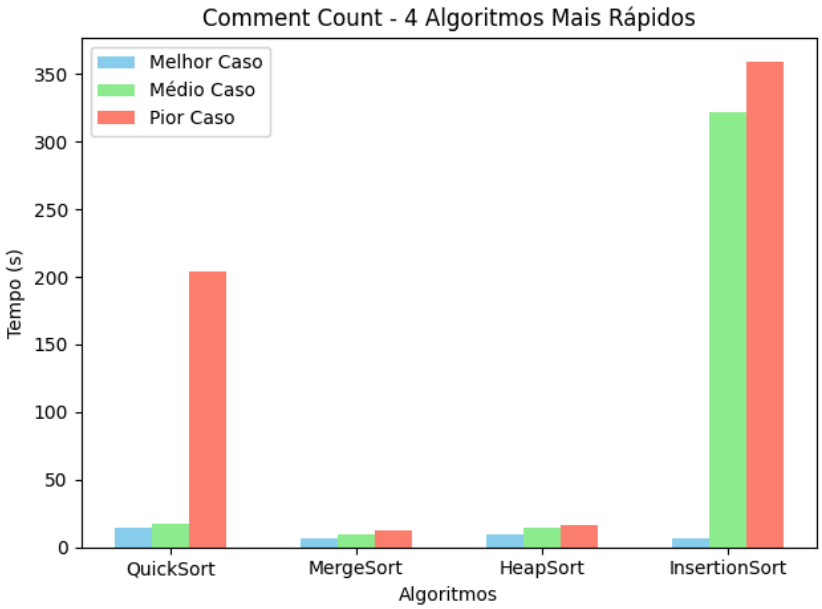
1. MergeSort - 12.124s
2. HeapSort - 211.549s
3. QuickSort - 321.901s
4. QuickSort3 - 421.312s
5. InsertionSort - 454.902s
6. SelectionSort - 497.211s

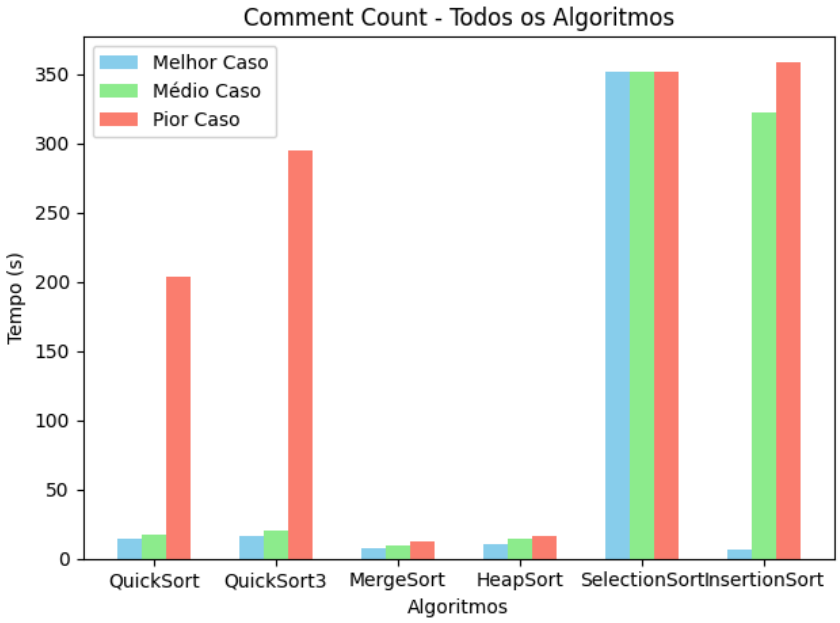
Resultados das ordenações com ***árvore binária***:

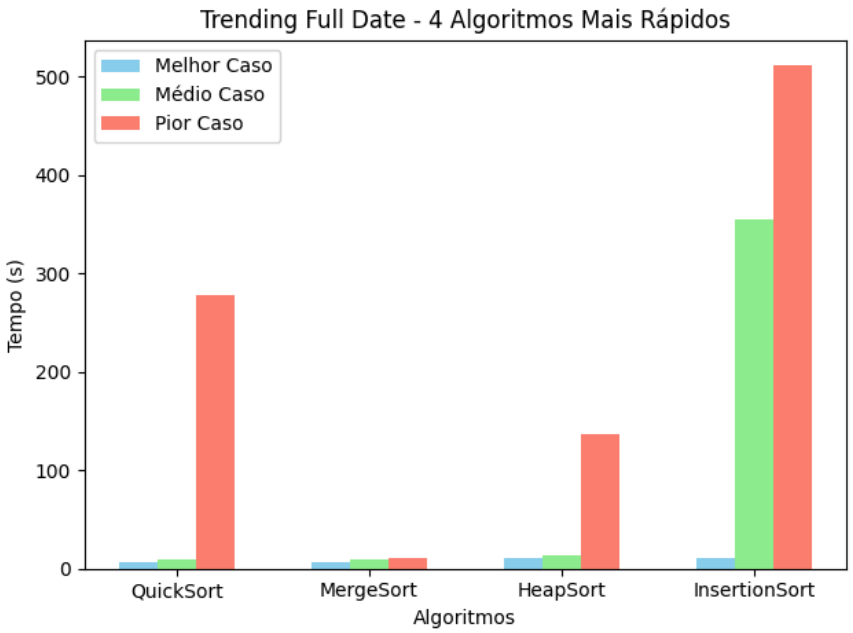
Algoritmo	Campo	Melhor caso (s)	Médio caso (s)	Pior caso (s)
QuickSort	channel_title	8.974	14.211	403.653
QuickSort3	channel_title	16.827	22.883	587.293
MergeSort	channel_title	7.541	9.124	14.041
HeapSort	channel_title	12.314	16.713	19.004
SelectionSort	channel_title	352.892	352.892	352.892
InsertionSort	channel_title	5.212	356.921	341.995
QuickSort	comment_count	14.824	17.283	204.029
QuickSort3	comment_count	16.183	20.009	294.832
MergeSort	comment_count	7.082	9.901	12.412
HeapSort	comment_count	10.024	14.545	16.096
SelectionSort	comment_count	351.561	351.561	351.561
InsertionSort	comment_count	6.982	321.821	358.932
QuickSort	trending_full_date	6.531	9.320	278.212
QuickSort3	trending_full_date	7.837	9.832	333.412
MergeSort	trending_full_date	6.431	9.391	10.348
HeapSort	trending_full_date	11.384	13.324	137.394
SelectionSort	trending_full_date	321.392	321.392	321.392
InsertionSort	trending_full_date	10.391	355.234	511.478

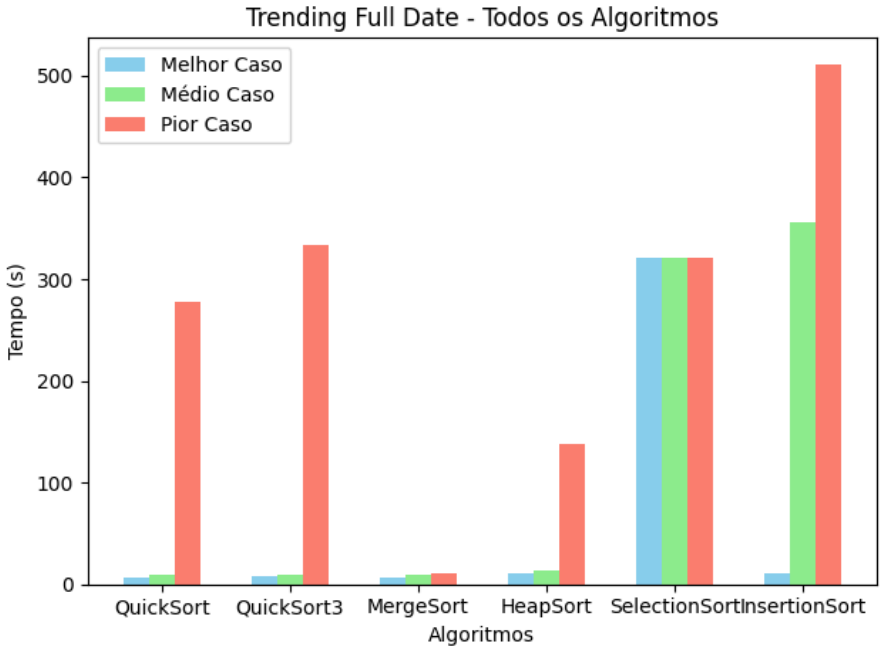

Channel Title - 4 Algoritmos Mais Rápidos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
50
100
150
200
250
300
350
400
QuickSort
MergeSort
HeapSort
InsertionSort
Algoritmos


Channel Title - Todos os Algoritmos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
100
200
300
400
500
600
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort
Algoritmos


Comment Count - 4 Algoritmos Mais Rápidos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
50
100
150
200
250
300
350
QuickSort
MergeSort
HeapSort
InsertionSort
Algoritmos


Comment Count - Todos os Algoritmos
Melhor Caso
Médio Caso
Pior Caso
Tempo (s)
0
50
100
150
200
250
300
350
QuickSort
QuickSort3
MergeSort
HeapSort
SelectionSort
InsertionSort
Algoritmos





Árvore Binária

Channel Title

Melhor Caso

1. MergeSort - 7.541s
2. QuickSort - 8.974s
3. InsertionSort - 5.212s
4. HeapSort - 12.314s
5. QuickSort3 - 16.827s
6. SelectionSort - 352.892s

Médio Caso

1. MergeSort - 9.124s
2. QuickSort - 14.211s
3. HeapSort - 16.713s
4. QuickSort3 - 22.883s
5. InsertionSort - 356.921s
6. SelectionSort - 352.892s

Pior Caso

1. MergeSort - 14.041s
2. HeapSort - 19.004s
3. InsertionSort - 341.995s
4. QuickSort - 403.653s
5. QuickSort3 - 587.293s
6. SelectionSort - 352.892s

Comment Count

Melhor Caso

1. MergeSort - 7.082s
2. InsertionSort - 6.982s

3. HeapSort - 10.024s
4. QuickSort - 14.824s
5. QuickSort3 - 16.183s
6. SelectionSort - 351.561s

Médio Caso

1. MergeSort - 9.901s
2. HeapSort - 14.545s
3. QuickSort - 17.283s
4. QuickSort3 - 20.009s
5. InsertionSort - 321.821s
6. SelectionSort - 351.561s

Pior Caso

1. MergeSort - 12.412s
2. HeapSort - 16.096s
3. QuickSort - 204.029s
4. QuickSort3 - 294.832s
5. InsertionSort - 358.932s
6. SelectionSort - 351.561s

Trending Full Date

Melhor Caso

1. MergeSort - 6.431s
2. QuickSort - 6.531s
3. QuickSort3 - 7.837s
4. HeapSort - 11.384s
5. InsertionSort - 10.391s
6. SelectionSort - 321.392s

Médio Caso

1. MergeSort - 9.391s
2. QuickSort - 9.320s
3. QuickSort3 - 9.832s
4. HeapSort - 13.324s
5. InsertionSort - 355.234s
6. SelectionSort - 321.392s

Pior Caso

1. MergeSort - 10.348s
2. QuickSort - 278.212s
3. QuickSort3 - 333.412s
4. HeapSort - 137.394s
5. InsertionSort - 511.478s
6. SelectionSort - 321.392s

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 maiúscula e os escritos em caps lock.**

Análise/Comentários acerca dos resultados obtidos

Arrays:

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 (outro algoritmo de divisão e conquista).
(Para mais informações sobre o MergeSort, acesse: aqui ou 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)

Sorting

Algorithm	Data Structure	Time Complexity			Worst Case Auxiliary Space Complexity
		Best	Average	Worst	Worst
Quicksort	Array	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Mergesort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Heapsort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	Array	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	Array	O(n)	O(n^2)	O(n^2)	O(1)
Select Sort	Array	O(n^2)	O(n^2)	O(n^2)	O(1)
Bucket Sort	Array	O(n+k)	O(n+k)	O(n^2)	O(nk)
Radix Sort	Array	O(nk)	O(nk)	O(nk)	O(n+k)

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, 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 dois algoritmos, isso poderia durar até cerca de 2 Horas.

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

Utilizando outras estruturas de dados

Em seguida, implementamos outros três tipos de estruturação dos dados, como citados anteriormente: *lista duplamente encadeada, tabela Hash e árvore binária.*

Mas **por que** usá-las?

Elas têm um potencial de melhoria da eficiência dos algoritmos em diversos casos, trazendo um desempenho mais robusto e consistente.

Façamos uma leitura da tabela de complexidade de tempo do site hackerearth:

Data Structures

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion	
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Dynamic Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartresian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Tratando da árvore binária (B-Tree), temos um excelente desempenho em diversas categorias. No melhor dos casos obtemos uma complexidade O(log(n)) de busca em tempo logarítmico. Tanto no melhor, quanto no pior dos casos, o desempenho é o mesmo.

A lista duplamente encadeada (Doubly-Linked List) apresenta o mesmo desempenho tanto no melhor, quanto no pior dos casos, o que pode ser bom se considerarmos uma situação em que o pior dos casos não pudesse ser um tempo diferente do que o melhor dos casos.

A tabela Hash (Hash Table) apresenta ótimos resultados tratando-se do melhor caso que tempos uma complexidade de tempo de O(1) em buscas, ao contrário do pior caso, que essa complexidade é O(n), tornando o processo mais lento.

Em um breve resumo:

-Lista Duplamente Encadeada: Semelhante à lista simplesmente encadeada, mas permite travessia bidirecional.

-Tabela Hash: Utilizada para armazenamento eficiente de pares chave-valor, permitindo inserções, remoções e buscas rápidas. Escolhida pela eficiência em buscas e inserções, essencial para operações de dicionário.

-Árvore Binária: Utilizada para ordenação eficiente de elementos através de inserções e travessias em ordem. Escolhidas para permitir ordenação eficiente e garantir desempenho consistente.

Searching

Algorithm	Data Structure	Time Complexity		Space Complexity
		Average	Worst	Worst
Depth First Search (DFS)	Graph of |V| vertices and |E| edges	-	O(|E| + |V|)	O(|V|)
Breadth First Search (BFS)	Graph of |V| vertices and |E| edges	-	O(|E| + |V|)	O(|V|)
Binary search	Sorted array of n elements	O(log(n))	O(log(n))	O(1)
Linear (Brute Force)	Array	O(n)	O(n)	O(1)
Shortest path by Dijkstra, using a Min-heap as priority queue	Graph with |V| vertices and |E| edges	O((|V| + |E|) log |V|)	O((|V| + |E|) log |V|)	O(|V|)
Shortest path by Dijkstra, using an unsorted array as priority queue	Graph with |V| vertices and |E| edges	O(|V|^2)	O(|V|^2)	O(|V|)
Shortest path by Bellman-Ford	Graph with |V| vertices and |E| edges	O(|V||E|)	O(|V||E|)	O(|V|)

Heaps

Heaps	Time Complexity						
	Heapify	Find Max	Extract Max	Increase Key	Insert	Delete	Merge
Linked List (sorted)	-	O(1)	O(1)	O(n)	O(n)	O(1)	O(m+n)
Linked List (unsorted)	-	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Heap	O(n)	O(1)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(m+n)
Binomial Heap	-	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Fibonacci Heap	-	O(1)	O(log(n))*	O(1)*	O(1)	O(log(n))*	O(1)

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

https://www.hackerearth.com/practice/notes/big-o-cheatsheet-series-data-structures-and-algorithms-with-thier-complexities-1/