Relatório de Implementação Algorítmica Atividade 1 Algoritmos de Ordenação

Fabio Ramos 202319060712
Gustavo Henrique Florentin 202319060232

Linguagem

Para implementação C + + e para os gráficos Python.

Introdução

O objetivo desta atividade, é implementar diferentes algoritmos de ordenação, e posteriormente analisar seu desempenho, realizando um comparativo entre o desempenho de ambos.

Os algoritmos estudados foram: Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, Quick Sort, Counting Sort.

A ordenação é uma das operações fundamentais da computação e desempenha um papel crucial em várias áreas da computação, existem algumas razões pela qual a ordenação seja importante, são elas: Eficiência de pesquisa e recuperação de dados, facilidade de análise e processamento, otimização de algoritmos, etc.

Metodologia

Valor dos parâmetros:

- **inc = 2000;**
- **fim = 30000;**
- **stp = 2000;**
- **rpt = 10.**

Resultados

1. Vetor aleatório

[[RANDOM]]

n	Bubble	Insertion	Merge	Heap	Quick	Counting
2000	0.013795	0.003694	0.000800	0.000703	0.000600	0.000100
4000	0.047376	0.011794	0.001396	0.001203	0.000700	0.000100
6000	0.100490	0.024286	0.001896	0.001901	0.001099	0.000198
8000	0.193904	0.045967	0.002304	0.002998	0.001399	0.000300
10000	0.303875	0.066964	0.003293	0.003204	0.001899	0.000300
12000	0.427006	0.097842	0.003698	0.003898	0.002298	0.000300
14000	0.576425	0.132753	0.004697	0.004896	0.002699	0.000200
16000	0.751487	0.172700	0.005198	0.005397	0.003398	0.000300
18000	0.974949	0.222720	0.006094	0.006099	0.003895	0.000100
20000	1.183456	0.268598	0.006798	0.006893	0.004001	0.000700
22000	1.445035	0.326434	0.007296	0.007793	0.004400	0.000699
24000	1.746933	0.389930	0.007796	0.008493	0.005100	0.000500
26000	2.014984	0.453914	0.008501	0.009295	0.005497	0.000600
28000	2.332726	0.527273	0.009695	0.009994	0.006097	0.000600
30000	2.713949	0.611820	0.010292	0.010897	0.006193	0.000400

Tabela 1: Conjunto Aleatório

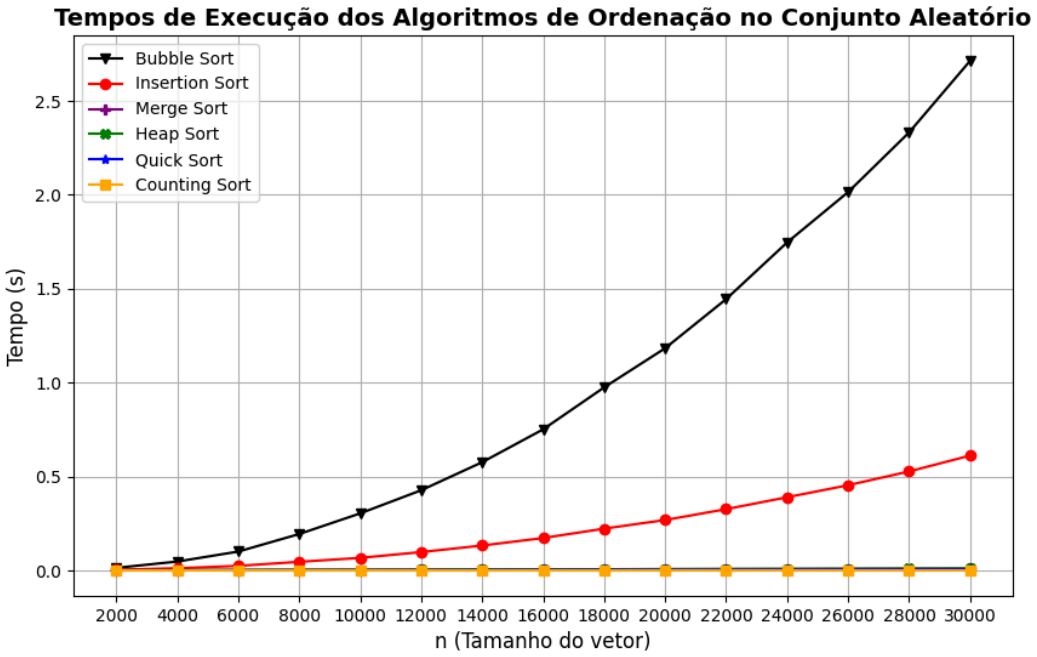

Figura 1: Gráfico com tempo de execução dos 6 algoritmos

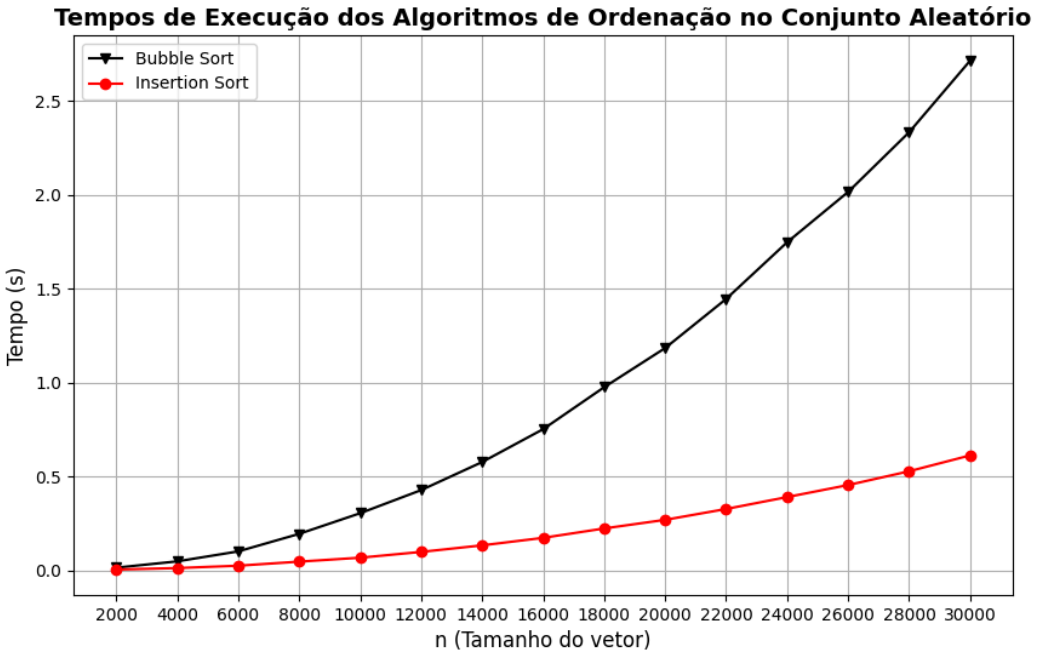

Figura 1.1: Algoritmos elementares

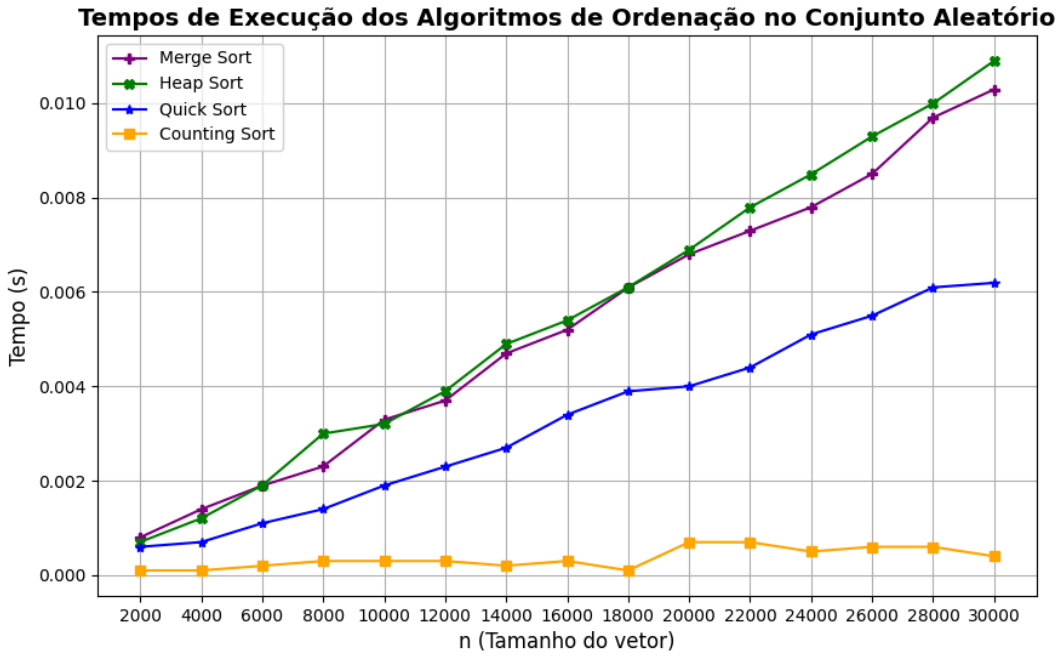

Figura 1.2: Algoritmos eficientes

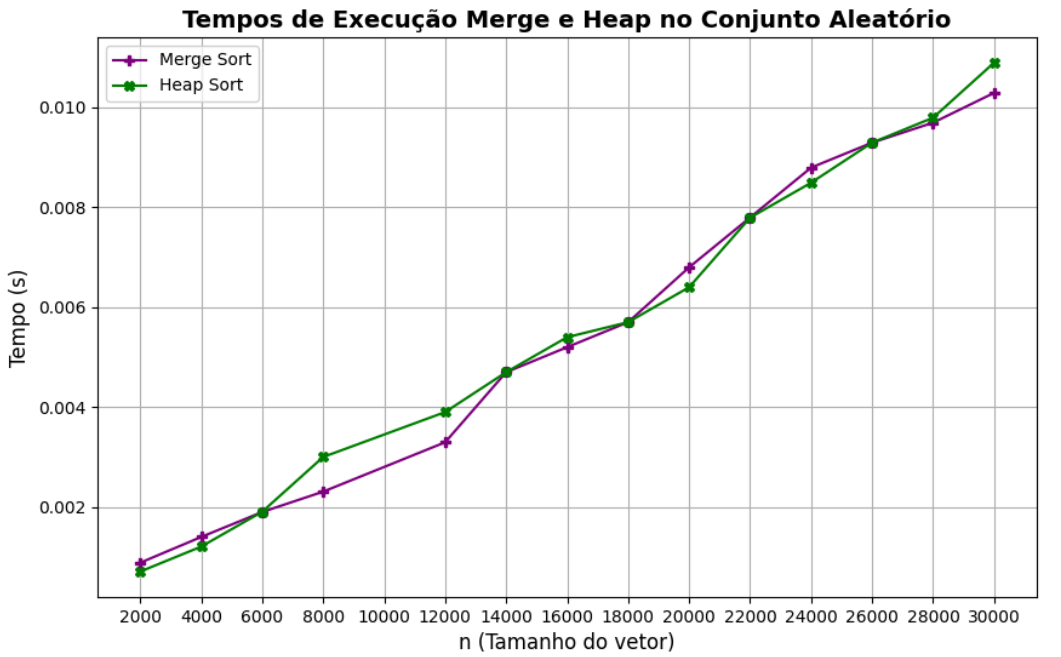

Figura 1.3: Comparação Heap e Merge

Análise dos Algoritmos de Ordenação no Conjunto Aleatório

Conforme apresentado na Figura 1, é possível observar o comportamento dos diferentes algoritmos de ordenação em termos de tempo de execução. Notamos que o algoritmo ***Bubble Sort*** apresentou o maior tempo de execução, seguido pelo ***Insertion Sort***. Ambos os algoritmos, sendo de complexidade quadrática $O(n^2)$, têm desempenho inferior em relação aos demais, o que dificulta a visualização clara de algoritmos mais eficientes no gráfico, como o ***Merge Sort***, ***Heap Sort***, ***Quick Sort*** e ***Counting Sort***, que possuem complexidades melhores para conjuntos maiores. Isso é particularmente evidente nos algoritmos lineares ou quase lineares, como o ***Counting Sort***, que demonstrou tempos de execução muito baixos em comparação aos demais.

Na **Figura 1.2**, observa-se que o algoritmo **Heap Sort** apresenta um comportamento que, embora estável, não é exatamente linear. Ele mantém uma execução consistente conforme o tamanho do vetor aumenta, mas não é tão eficiente quanto o **Counting Sort**, que apresenta tempos de execução consideravelmente menores e mais constantes, independentemente do aumento no tamanho do vetor. **Heap Sort** e **Merge Sort** mostraram performances semelhantes, com tempos de execução muito próximos ao longo dos testes, reforçando sua eficiência em cenários que demandam alto volume de dados. No entanto, o **Counting Sort** foi significativamente mais eficiente, especialmente para conjuntos maiores.

A Figura 1.3 apresenta a comparação direta entre os algoritmos **Heap Sort** e **Merge Sort** no conjunto de dados aleatórios. É possível observar que ambos os algoritmos mantêm uma performance estável à medida que o tamanho do vetor aumenta. Embora existam pequenas variações nos tempos de execução, os dois algoritmos possuem complexidades similares, o que garante que ambos sejam altamente eficientes em comparação com outros métodos de ordenação, como o **Bubble Sort** ou **Insertion Sort**.

2. Vetor reverso

[[REVERSE]]

n	Bubble	Insertion	Merge	Heap	Quick	Counting
2000	0.013518	0.003517	0.000118	0.000287	0.008368	0.000016
4000	0.061591	0.015150	0.000248	0.000666	0.039497	0.000045
6000	0.140848	0.035284	0.000425	0.001089	0.095013	0.000065
8000	0.274305	0.066118	0.000569	0.001587	0.152946	0.000063
10000	0.416473	0.101175	0.000769	0.001783	0.236929	0.000079
12000	0.572774	0.138009	0.000825	0.002225	0.311988	0.000147
14000	0.739665	0.187767	0.001049	0.002468	0.433178	0.000128
16000	0.988916	0.239261	0.001124	0.002943	0.616407	0.000198
18000	1.579272	0.398707	0.001301	0.003496	0.807343	0.000222
20000	1.654841	0.365285	0.001402	0.003675	0.881161	0.000189
22000	2.064549	0.493375	0.001604	0.004308	1.221706	0.000218
24000	2.331638	0.667255	0.001770	0.004796	1.783061	0.000269
26000	2.804673	0.647580	0.001928	0.005088	1.536190	0.000234
28000	2.994502	0.803030	0.002147	0.005431	1.907461	0.000256
30000	3.406796	0.863047	0.002265	0.006443	2.268131	0.000370

Tabela 2: Conjunto Reverso

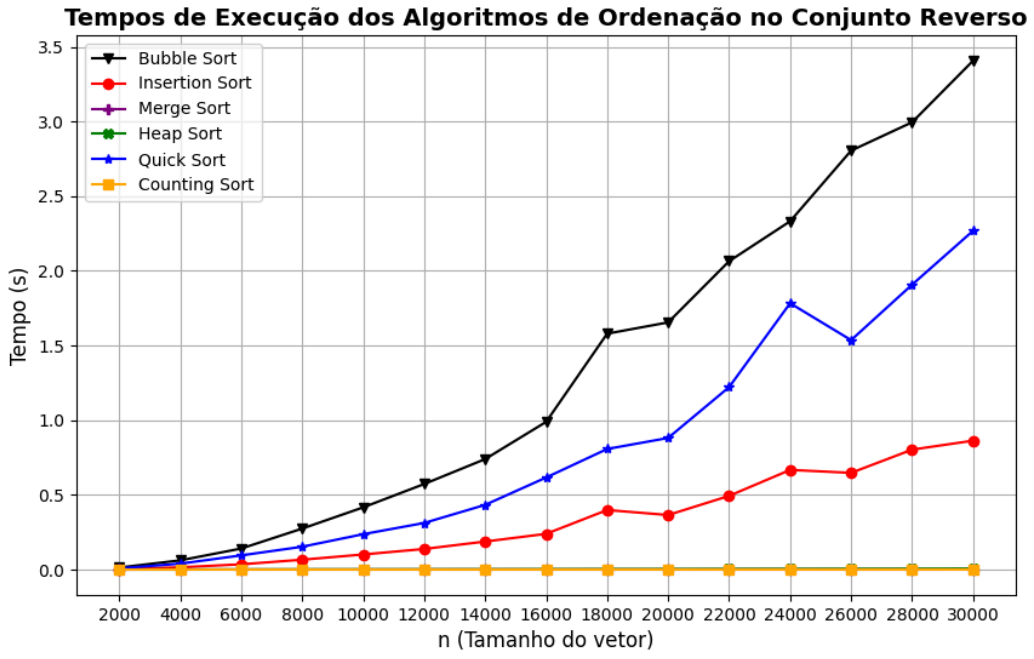

Figura 2: Gráfico com tempo de execução dos 6 algoritmos

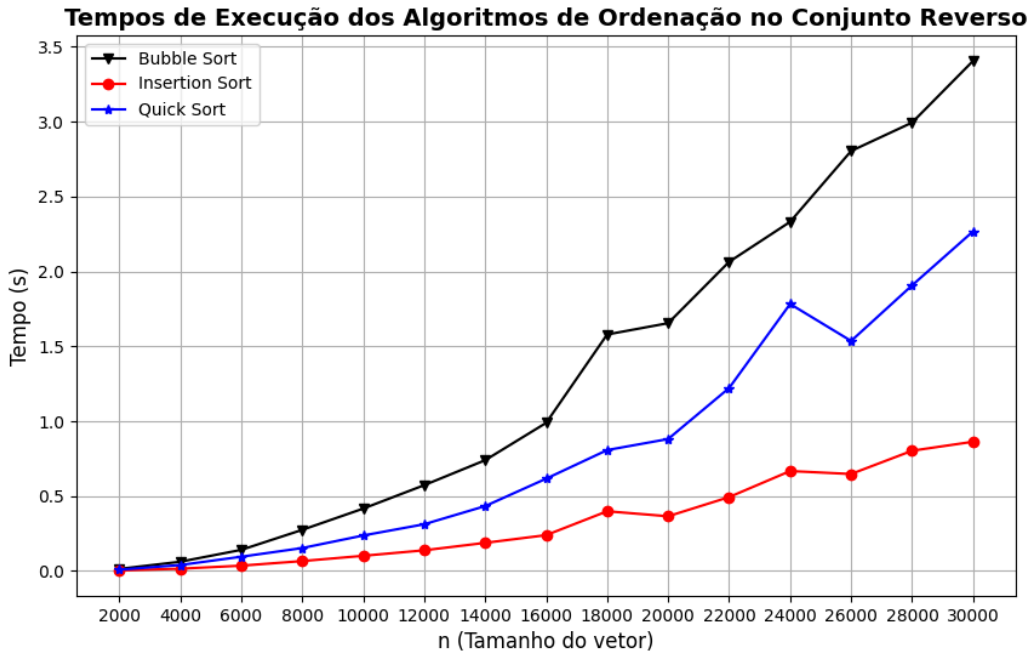

Figura 2-1: Gráfico com tempo dos algoritmos Elementares

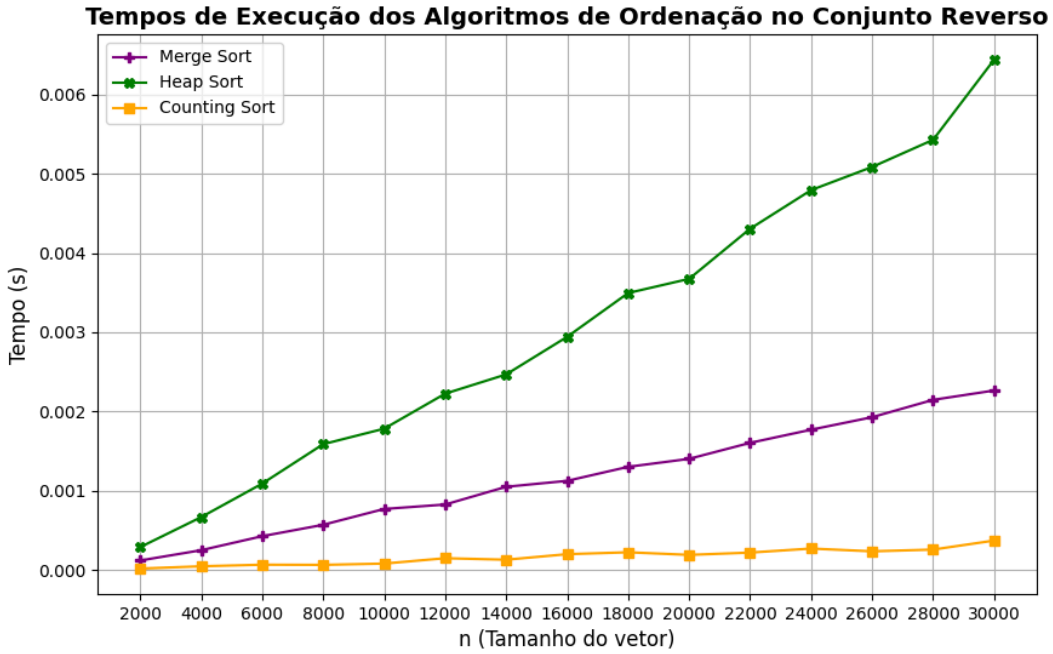

Figura 2-2: Gráfico com tempo dos algoritmos Eficientes

Análise dos Algoritmos de Ordenação no Conjunto Reverso

Nesta seção, analisamos o comportamento dos algoritmos de ordenação ao serem aplicados a um vetor com elementos em ordem reversa. Os tempos de execução dos algoritmos foram registrados e comparados, conforme ilustrado na Figura 2 e nos dados apresentados na Tabela 2.

A tabela mostra o tempo de execução de cada algoritmo conforme o tamanho do vetor aumenta. O ***Bubble Sort*** apresentou o maior tempo de execução, alcançando mais de 3,4 segundos para um vetor com 30.000 elementos. Seguido do **Quick Sort** e o **Insertion**. Na Figura 2-1 O ***Quick Sort***, em particular, apresentou um comportamento interessante: enquanto seu tempo de execução geral segue a tendência de crescimento esperado, houve uma anomalia observada no vetor de 24.000 elementos, onde o tempo de execução foi maior do que para o vetor de 26.000 elementos, essa anomalia pode ser atribuída tanto a uma escolha desfavorável do pivô na implementação do **Quick Sort**, quanto a variações na performance da máquina durante a execução dos testes. O tempo dos demais algoritmos não ficou perceptível na Figura 2.

Na Figura 2-2, os algoritmos mais eficientes foram analisados separadamente dos outros. O ***Counting Sort*** se destacou como o mais eficiente, mantendo um tempo de execução praticamente constante, independentemente do aumento do tamanho do vetor. O **Merge Sort** e o **Heap Sort** também demonstraram desempenho consistente, conforme esperado pela sua complexidade O (n log n), tornando-os adequados para vetores grandes e mantendo tempos de execução estáveis durante os testes. Isso reforça a eficiência desses algoritmos em situações que exigem maior volume de dados. Fica evidente que, apesar do crescimento do vetor, o tempo de execução do Merge Sort permanece estável, mesmo no caso de um conjunto reverso. Essa característica o torna altamente eficaz em situações onde o conjunto de dados pode estar organizado de maneira desfavorável, como no cenário apresentado.

3. Vetor ordenado

[[SORTED]]

n	Bubble	Insertion	Merge	Heap	Quick	Counting
2000	0.006000	0.000006	0.000129	0.000338	0.025228	0.000023
4000	0.023108	0.000010	0.000254	0.000628	0.097944	0.000033
6000	0.034514	0.000014	0.000393	0.001144	0.179487	0.000077
8000	0.121696	0.000022	0.000626	0.001956	0.280260	0.000063
10000	0.106391	0.000035	0.000875	0.002495	0.403630	0.000107
12000	0.147094	0.000028	0.000867	0.002184	0.527747	0.000115
14000	0.168588	0.000036	0.000991	0.002678	0.778940	0.000175
16000	0.231165	0.000052	0.001262	0.003585	0.966835	0.000162
18000	0.270232	0.000046	0.001216	0.003473	1.375635	0.000225
20000	0.328672	0.000065	0.001759	0.004012	1.446847	0.000280
22000	0.448689	0.000053	0.001612	0.004322	1.615326	0.000216
24000	0.470239	0.000057	0.002106	0.005150	1.965018	0.000217
26000	0.630184	0.000064	0.001888	0.008993	2.578122	0.000283
28000	0.675118	0.000066	0.001956	0.005920	3.023645	0.000360
30000	0.877202	0.000073	0.002358	0.006242	3.228158	0.000282

Tabela 3: Conjunto Ordenado

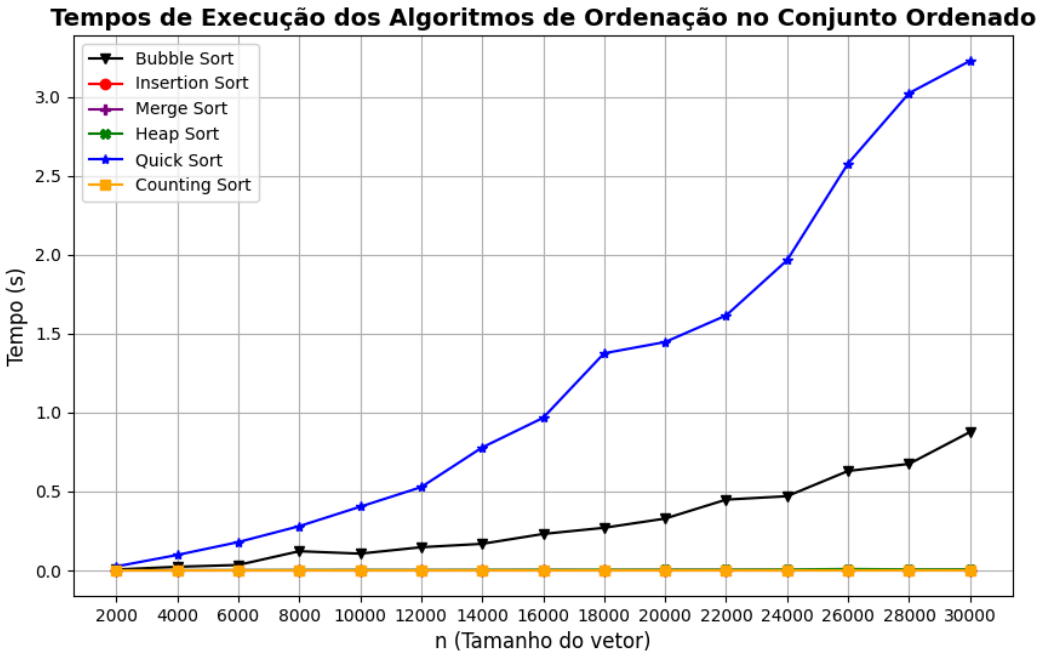

Figura 3: Gráfico com tempo de execução dos 6 algoritmos

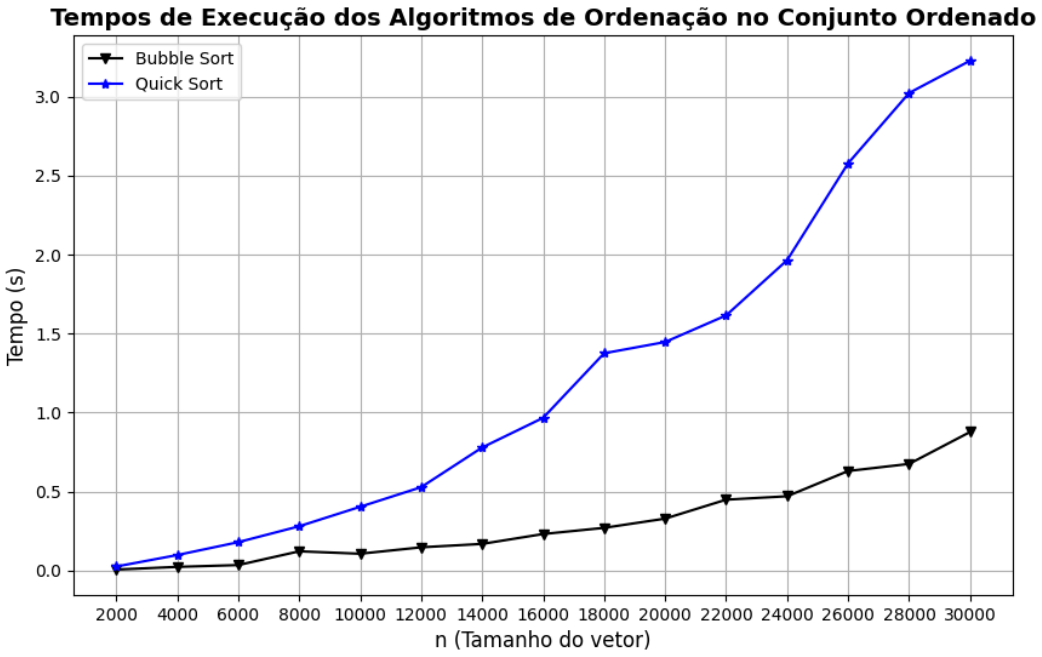

Figura 3-1: Gráfico com tempo de execução dos algoritmos Elementares

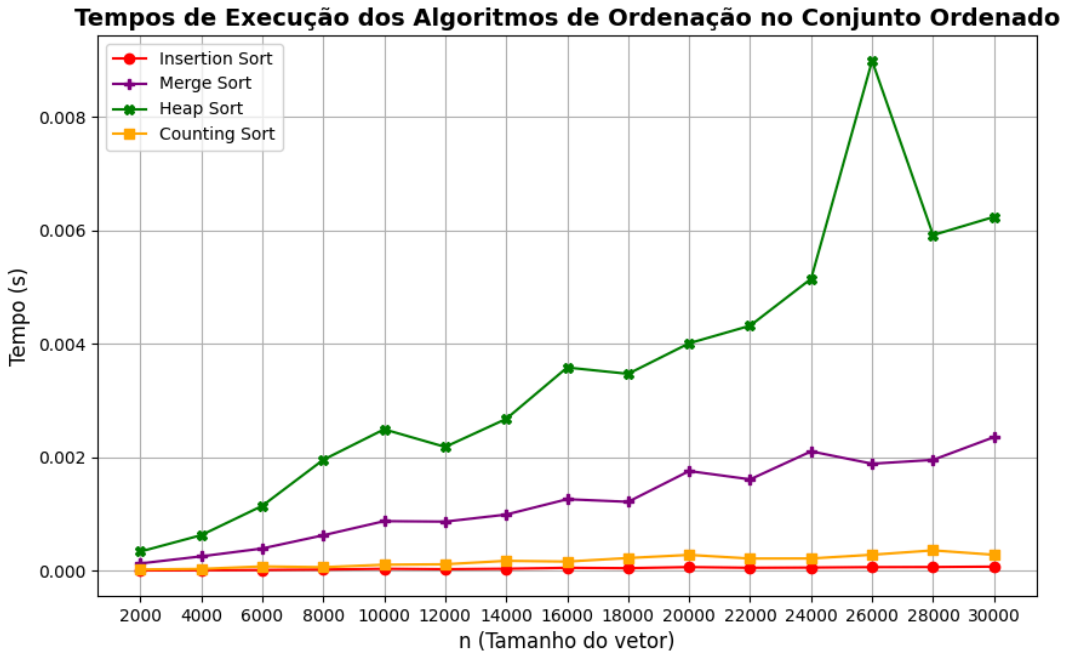

Figura 3-2: Gráfico com tempo de execução dos algoritmos Eficientes

Análise dos Algoritmos de Ordenação no Conjunto Ordenado

Ao aplicar os algoritmos de ordenação a um vetor já ordenado, observamos comportamentos esperados, conforme mostrado na Figura 3.

Na Figura 3-1, o **Quick Sort** se mostrou o menos eficiente para esse tipo de conjunto, seguido pelo **Bubble Sort**. Devido ao tempo de execução desses dois algoritmos ser consideravelmente superior, a visualização dos demais algoritmos foi prejudicada, não permitindo uma análise clara de suas performances.

O gráfico da Figura 3-2 mostra o desempenho dos algoritmos eficientes (***Insertion Sort***, ***Merge Sort***, ***Heap Sort***, ***Counting Sort***) aplicados a um vetor já ordenado.

O ***Insertion Sort*** apresentou excelente desempenho nesse cenário, uma vez que, em um vetor já ordenado, ele não realiza nenhuma troca de posições. Isso resulta em um tempo de execução praticamente constante, pois o algoritmo apenas percorre o vetor sem necessidade de reorganizar os elementos. Sua complexidade

no melhor caso é O(n), o que o torna muito eficiente para dados já ordenados. O **Merge Sort** e o **Heap Sort**, ambos com complexidade O(n log n), apresentaram tempos de execução similares, como esperado. Esses algoritmos, apesar de serem extremamente eficientes para grandes volumes de dados e casos não ordenados, não conseguem superar o desempenho do Insertion Sort em vetores já ordenados devido à necessidade de realizar operações adicionais para manter suas estruturas internas, mesmo quando não há necessidade de reorganização dos elementos.

Dando continuidade à análise do comportamento do **Merge Sort** e do **Heap Sort**, é importante destacar a forma como esses algoritmos operam e como isso impacta o seu desempenho, especialmente em cenários de vetores já ordenados.

O **Merge Sort**, por exemplo, segue uma abordagem **dividir-para-conquistar**, quebrando o vetor em duas metades de maneira recursiva, até chegar a subvetores de tamanho 1. Embora essa abordagem seja eficiente em termos de complexidade O(n log n), ela requer uma quantidade significativa de operações para reorganizar as partes do vetor, mesmo que os elementos já estejam ordenados. Isso gera uma sobrecarga desnecessária quando comparado ao **Insertion Sort**, que simplesmente percorre o vetor sem realizar trocas. No caso do Merge Sort, mesmo em vetores já ordenados, o processo de divisão e a subsequente fusão dos subvetores ainda ocorrem, o que explica seu tempo de execução ligeiramente mais alto.

Quanto ao **Heap Sort**, ele também mantém uma complexidade O (n log n), mas seu funcionamento envolve a construção e o ajuste de uma estrutura de **heap** (árvore binária), o que adiciona operações de reorganização interna, mesmo quando os dados já estão em ordem. Assim como o **Merge Sort**, o Heap Sort sofre de uma ineficiência em vetores já ordenados, pois ainda precisa construir e ajustar a estrutura do heap, o que leva a um tempo de execução maior em relação ao **Insertion Sort**.

Além disso, observou-se uma **anomalia** específica no comportamento do **Heap Sort** com o vetor de 26.000 elementos. Essa irregularidade pode estar associada à **gestão da memória** ou a detalhes da **implementação**.

4. Vetor quase ordenado

[[[NEARLY SORTED]]

n	Bubble	Insertion	Merge	Heap	Quick	Counting
2000	0.011164	0.001002	0.000000	0.000000	0.001001	0.000000
4000	0.040894	0.003291	0.000998	0.001000	0.002000	0.000000
6000	0.108530	0.008002	0.000998	0.002084	0.001998	0.000000
8000	0.170958	0.013999	0.001000	0.003000	0.003189	0.000000
10000	0.287427	0.019459	0.001000	0.002999	0.001963	0.001039
12000	0.382537	0.022001	0.001000	0.003964	0.003043	0.000000
14000	0.403288	0.032001	0.001998	0.004000	0.003001	0.000000
16000	0.529833	0.071001	0.003001	0.005997	0.005001	0.000000
18000	0.757163	0.047999	0.002000	0.005001	0.009000	0.000000
20000	0.882821	0.057698	0.002054	0.006001	0.004999	0.001001
22000	0.978829	0.070220	0.000000	0.000000	0.015870	0.001033
24000	1.255112	0.088518	0.002999	0.007969	0.006362	0.000000
26000	1.497079	0.107962	0.003037	0.008001	0.005999	0.000000
28000	1.668077	0.132339	0.004001	0.007999	0.005318	0.000000
30000	1.921021	0.146641	0.003033	0.009174	0.007999	0.000000

Tabela 4: Conjunto Quase Ordenado

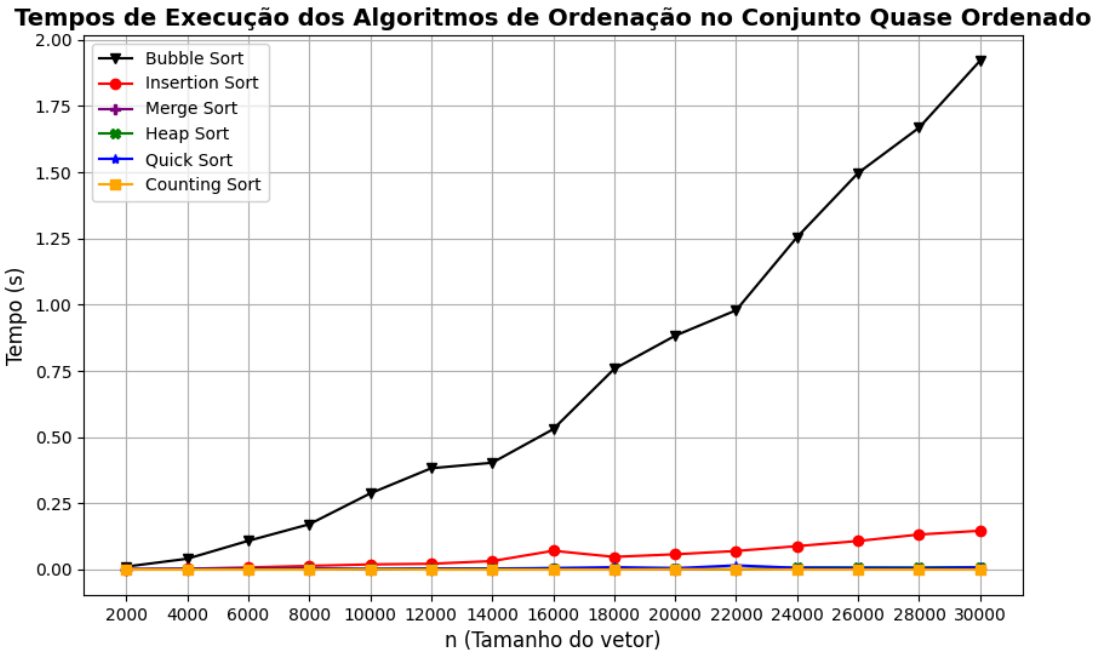

Figura 4: Gráfico com tempo de execução dos 6 algoritmos

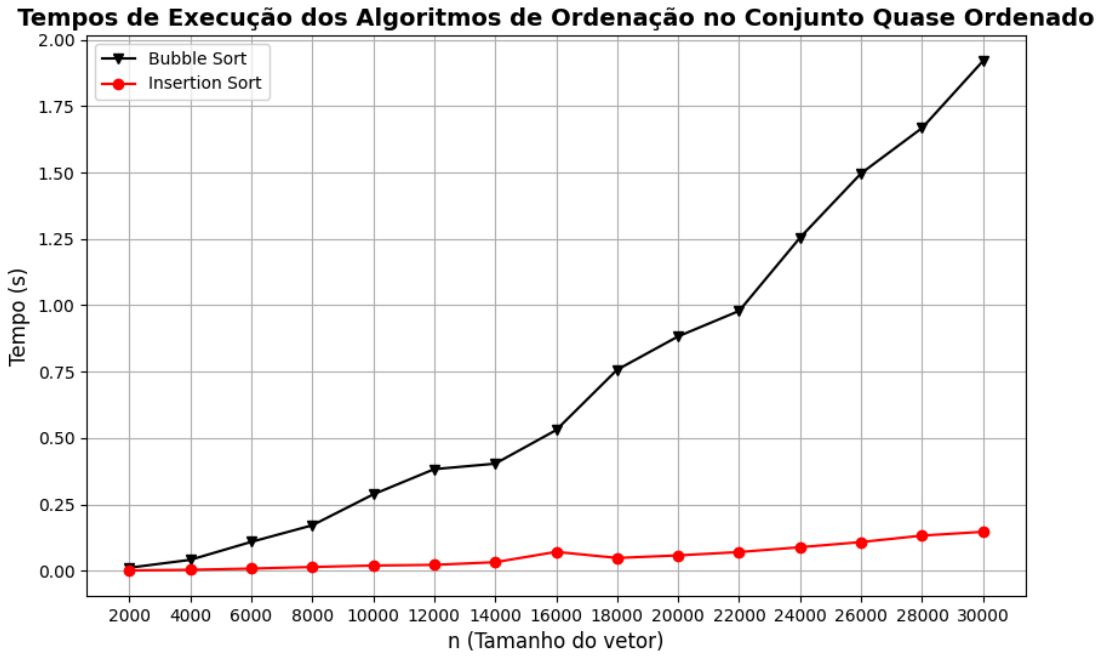

Figura 4-1: Gráfico com tempo de execução dos algoritmos Elementares

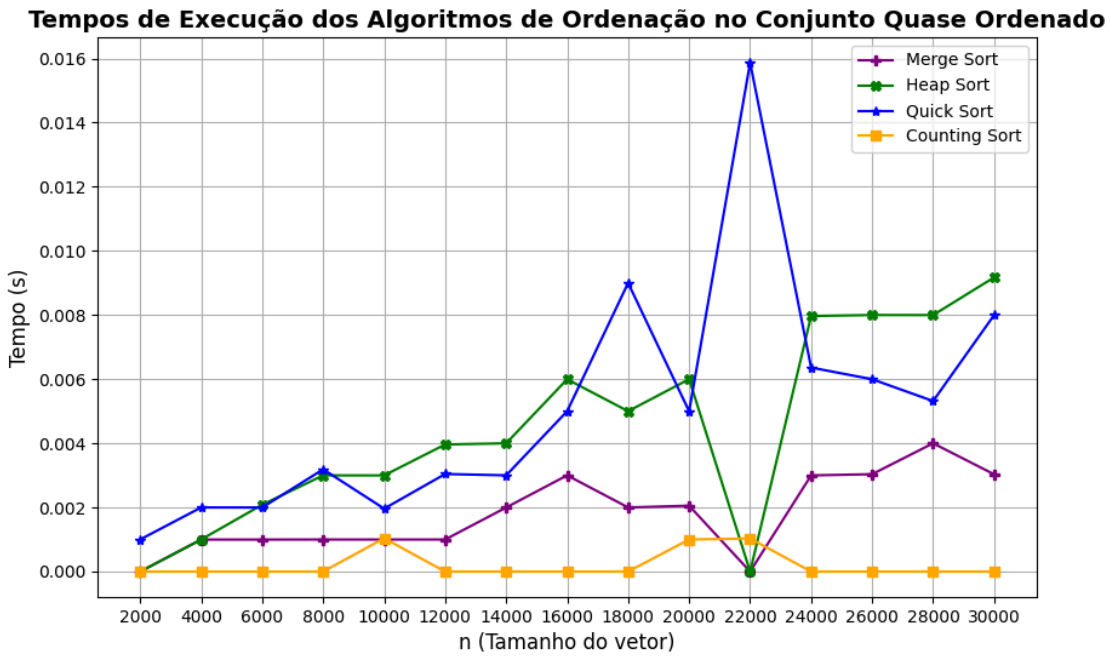

Figura 4-2: Gráfico com tempo de execução dos algoritmos Eficientes

Análise dos Algoritmos de Ordenação no Conjunto Quase Ordenado

Ao implementar os algoritmos de ordenação no conjunto quase ordenado, é possível identificar os algoritmos mais eficientes e os menos eficientes. Observando a Figura 4 nota-se que os algoritmos mais eficientes são, o **Merge Sort, Heap Sort**, **Quick Sort e Counting Sort**, com algumas anomalias de medição de tempo devido a máquina utilizada na implementação dos algoritmos, como também alguma questão de implementação, como exemplo o pivô no **Quick Sort** ser colocado na última posição, o que pode ocasionar do algoritmo cair no pior caso (O n^2), todas essas anomalias podem ser observadas na Figura 4-2. Ao continuar analisando o tempo de cada implementação dos algoritmos, nota-se na Figura 4-1 que os algoritmos, **Bubble Sort e Insertion Sort**, são os algoritmos menos eficientes, sendo o **Bubble Sort** o 1° e o **Insertion Sort** o 2° em eficiência, em que no pior caso seu tempo seja O n^2 e seu melhor caso N log n.

Os algoritmos mais eficientes para esse conjunto são **Counting Sort**, **Merge Sort**, e **Heap Sort**, seguidos de perto pelo **Quick Sort**. No entanto, é importante observar alguns detalhes específicos:

- **Counting Sort** mantém sua eficiência impressionante, com tempos de execução muito baixos. Este comportamento linear é esperado devido à sua complexidade **O(n + k)**, que é particularmente adequada para cenários onde os dados já estão parcialmente ordenados.
- **Merge Sort** também apresenta um desempenho estável, mantendo um crescimento gradual à medida que o tamanho do vetor aumenta. Como esperado, sua complexidade **O(n log n)** faz com que ele seja eficiente para grandes volumes de dados, mesmo que os dados estejam quase ordenados.
- **Heap Sort** segue uma tendência semelhante, com um comportamento estável, mas com variações mais notáveis do que o Merge Sort. Observamos algumas pequenas oscilações no tempo de execução, possivelmente devido à necessidade de manter a estrutura de heap, que exige operações adicionais de reorganização, mesmo em dados quase ordenados.

Anomalia encontrada com n = 22000

O **Quick Sort** apresenta uma **anomalia notável** em torno do tamanho do vetor de **22.000 elementos**, onde seu tempo de execução aumenta drasticamente.

Após essa anomalia, o **Quick Sort** retorna ao seu comportamento mais eficiente, com tempos de execução próximos aos dos outros algoritmos eficientes, mas o pico no tempo de execução ilustra a vulnerabilidade talvez pela implementação ou questão da máquina utilizada no teste.

O gráfico também revela uma anomalia significativa no comportamento do **Heap Sort** em torno de um vetor de tamanho específico, mais notavelmente em torno de 22.000 elementos. Assim como o **Quick Sort**, que sofreu com a escolha inadequada do pivô, o **Heap Sort** apresentou uma oscilação acentuada no tempo de execução. Essa irregularidade pode ser atribuída a fatores como a estrutura interna do heap, que exige uma série de reorganizações para manter suas propriedades de árvore binária, especialmente em conjuntos quase ordenados. O gerenciamento de memória e os custos associados à manutenção da estrutura do heap podem ter contribuído para esse aumento no tempo de execução.

No entanto, após essa anomalia, o **Heap Sort** retorna ao seu comportamento estável e previsível com complexidade O(n log n).

Embora o **Merge Sort** geralmente apresente uma performance estável e previsível com complexidade **O(n log n)**, uma anomalia foi observada durante a implementação no conjunto quase ordenado, especificamente ao lidar com vetores de tamanho maior, como em torno de **22.000 elementos**.

Após essa anomalia, o **Merge Sort** retorna ao seu comportamento eficiente, mantendo sua complexidade **O(n log n)** e tempos de execução comparáveis aos outros algoritmos eficientes.

Resultados

Com base nas análises anteriores e nos gráficos gerados para cada cenário, é possível identificar qual algoritmo se destacou em termos de desempenho para cada conjunto de dados testados. A seguir, destacamos os melhores algoritmos para os diferentes conjuntos:

1. **Vetor Aleatório:** O **Counting Sort** foi o mais eficiente neste cenário. Com sua complexidade **O(n + k)**, ele se mostrou extremamente rápido para vetores com muitos elementos, independentemente da ordem inicial dos dados. O **Merge Sort** e o **Heap Sort** também apresentaram bons resultados, especialmente para grandes volumes de dados por ter uma consistência, mas foram superados pelo Counting Sort em termos de tempo de execução.
2. **Vetor Reverso:** Novamente, o **Counting Sort** se destacou como o algoritmo mais eficiente, com tempos de execução consistentemente baixos, mesmo em um cenário desfavorável como o vetor em ordem reversa. O **Merge Sort** e o **Heap Sort** mantiveram uma performance estável, enquanto o **Quick Sort** apresentou algumas oscilações devido à escolha do pivô.
3. **Vetor Ordenado:** O **Insertion Sort** foi o algoritmo mais eficiente para o vetor já ordenado, apresentando uma complexidade **O(n)** no melhor caso. Ele apenas percorreu os elementos sem realizar trocas, resultando em um tempo de execução praticamente constante. O **Merge Sort** e o **Heap Sort** tiveram uma performance estável, mas não foram capazes de superar o Insertion Sort nesse cenário específico.
4. **Vetor Quase Ordenado:** O **Counting Sort** foi novamente o mais eficiente, mantendo seu padrão de tempo de execução constante. O **Merge Sort** e o **Heap Sort** também mostraram bom desempenho, mas o **Quick Sort** sofreu uma queda de performance devido à escolha inadequada do pivô, resultando em um pico de tempo de execução, conforme observado no gráfico.

Discussão

Cada algoritmo foi avaliado em diferentes cenários de ordenação, incluindo vetores aleatórios, reversos, ordenados e quase ordenados.

- **Bubble Sort**
- **Insertion Sort**
- **Merge Sort**
- **Heap Sort**
- **Quick Sort**,
- **Counting Sort**

Em resumo, **não existe um único "melhor" algoritmo de ordenação**. A escolha do algoritmo mais adequado depende fortemente do contexto, como o tamanho dos dados, sua distribuição, e as restrições de tempo e espaço. Cada algoritmo tem suas forças e fraquezas, e a decisão sobre qual usar deve ser baseada em uma análise cuidadosa do cenário específico.

Aprendizado

A realização desta atividade proporcionou valiosos aprendizados tanto técnicos quanto conceituais:

- **Entendimento Profundo dos Algoritmos:** A implementação prática dos algoritmos de ordenação permitiu um entendimento mais profundo de suas operações internas, eficiência, e limitações. Comparar o desempenho de cada algoritmo em diferentes cenários revelou como fatores como escolha do pivô e estrutura de dados impactam diretamente a eficiência.
- **Importância do Contexto:** Como destacado, aprendemos que **não existe um melhor algoritmo de ordenação em termos absolutos**. Em vez disso, a eficácia de um algoritmo depende do contexto no qual ele é aplicado. Esta compreensão é crucial para a tomada de decisões informadas no desenvolvimento de software e na otimização de sistemas.
- **Análise Crítica e Depuração:** Identificar e investigar a anomalia observada no desempenho do **Quick Sort** reforçou a importância da análise crítica e da depuração cuidadosa no processo de desenvolvimento. O comportamento inesperado nos lembrou que a implementação e as suposições subjacentes a um algoritmo devem ser constantemente revisadas e validadas.

Essa atividade foi uma oportunidade de consolidar conhecimentos sobre algoritmos de ordenação, ao mesmo tempo que nos proporcionou uma compreensão mais ampla de como escolher e aplicar essas técnicas em cenários reais.