UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

LEONARDO NADSON OLIVEIRA DE MEDEIROS

RELATÓRIO TÉCNICO: ANÁLISE COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO

NATAL

2025

SUMÁRIO

1- INTRODUÇÃO

Este trabalho apresenta a implementação e a análise de cinco algoritmos de ordenação clássicos, seguindo as diretrizes da atividade prática proposta. Organizar listas de dados é uma tarefa fundamental na computação, e entender como diferentes métodos de ordenação se comportam é essencial para desenvolver programas mais eficientes. O objetivo principal é comparar na prática o Bubble Sort, Insertion Sort, Selection Sort, Merge Sort e Quick Sort. Para isso, medimos o tempo de execução, o número de comparações e o número de trocas de cada um. Os testes foram realizados com conjuntos de dados em três cenários diferentes: aleatórios, inversamente ordenados e quase ordenados, para verificar como cada algoritmo reage a diferentes situações. Ao final, os resultados obtidos são analisados e comparados com a teoria para determinar qual algoritmo é mais adequado para cada caso.

2 - FUNDAMENTAÇÃO TEÓRICA

Para avaliar a eficácia de um algoritmo, não basta que ele produza o resultado correto; ele deve fazê-lo de maneira eficiente. As principais métricas para essa avaliação são a complexidade de tempo e a complexidade de espaço. A complexidade de tempo quantifica o número de operações que um algoritmo realiza em função do tamanho da entrada, enquanto a complexidade de espaço mede a quantidade de memória adicional necessária.

Para expressar essas complexidades, utiliza-se a notação Assintótica, principalmente a notação Big O (O), que descreve o limite superior do crescimento do tempo de execução, representando o pior caso de um algoritmo. Para uma análise mais completa, as notações Big Theta (Θ) e Big Omega (Ω) são empregadas. A notação Θ descreve um limite assintótico justo, indicando o comportamento no caso médio ou em cenários onde o desempenho é consistente. Já a notação Ω descreve o limite inferior, ou seja, o melhor caso de desempenho. A distinção entre esses cenários é crucial, pois o desempenho de um algoritmo pode variar drasticamente dependendo da organização inicial dos dados de entrada. Assim, analisaremos cada um dos seguintes algoritmos e evidenciando suas respectivas complexidades teóricas:

2.1 - Bubble Sort

O Bubble Sort é um dos algoritmos mais simples, funcionando através de passagens repetidas pela lista. Em cada passagem, ele compara cada par de elementos adjacentes e os troca de lugar se estiverem na ordem errada. Com isso, a cada passagem, o maior elemento da porção não ordenada "flutua" para sua posição correta no final do vetor. Uma otimização importante, que permite um melhor caso de Θ(n), é a inclusão de uma flag que verifica se ocorreram trocas em uma passagem; se não ocorreram, o vetor está ordenado e o algoritmo pode parar.
Complexidades: Tempo (Melhor: Θ(n), Médio: Θ(n2), Pior: Θ(n2)), Espaço (O(1)).

2.2 - Insertion Sort

O Insertion Sort opera de forma análoga à maneira como uma pessoa organiza um baralho de cartas. Ele divide o vetor em uma parte ordenada e outra não ordenada. A cada iteração, ele retira um elemento da parte não ordenada e o insere em sua posição correta na parte ordenada, deslocando os elementos maiores para a direita. É um algoritmo adaptativo, extremamente eficiente para dados que já estão "quase ordenados".
Complexidades: Tempo (Melhor: Θ(n), Médio: Θ(n^2), Pior: Θ(n^2)), Espaço (O(1)).

2.3 - Selection Sort

O Selection Sort também divide o vetor em uma subseção ordenada e outra não ordenada. A cada passagem, o algoritmo varre toda a parte não ordenada para encontrar o menor elemento e, então, o troca com o primeiro elemento dessa subseção. Sua principal característica é realizar um número mínimo de trocas (no máximo n−1), mas seu número de comparações é sempre quadrático, independentemente da ordem inicial dos dados.
Complexidades: Tempo (Melhor, Médio e Pior: Θ(n^2)), Espaço (O(1)).

2.4 - Merge Sort

O Merge Sort é um exemplo clássico da abordagem de "dividir e conquistar". Ele divide recursivamente o vetor ao meio até que restem apenas subvetores de um único elemento. Em seguida, esses subvetores são mesclados (combinados) de forma ordenada, dois a dois, até que o vetor inteiro esteja reunido e completamente

ordenado. A etapa de mesclagem requer o uso de vetores auxiliares, o que impacta sua complexidade de espaço.

Complexidades: Tempo (Melhor, Médio e Pior: Θ(nlogn)), Espaço (O(n)).

2.5 - Quick Sort

O Quick Sort é outro poderoso algoritmo de "dividir e conquistar". Ele funciona escolhendo um elemento como pivô e particiona o vetor de modo que todos os elementos menores que o pivô fiquem à sua esquerda e os maiores, à sua direita. Após essa etapa, o pivô está em sua posição final correta. O algoritmo é então aplicado recursivamente aos subvetores à esquerda e à direita do pivô. A estratégia de escolha do pivô é crucial para o desempenho; uma má escolha pode levar ao pior caso quadrático.

Complexidades: Tempo (Melhor/Médio: Θ(nlogn), Pior: O(n2)), Espaço (Médio: O(logn), Pior: O(n)).

3 - METODOLOGIA

Para a análise empírica, adotou-se um procedimento rigoroso em duas etapas, unindo a performance da linguagem C++ para a execução dos algoritmos com a versatilidade do Python para a análise dos dados. A primeira etapa envolveu a implementação e a coleta de métricas em C++. Utilizando bibliotecas modernas da linguagem como <vector>, <chrono> e <random>, foram desenvolvidos os algoritmos de ordenação e o ambiente de teste.

Para avaliar o desempenho em diferentes cenários, foram implementadas funções que geravam vetores com três distribuições de dados distintas: ordenados, para análise do melhor caso; inversamente ordenados, para testar o pior caso; e com elementos aleatórios, para simular o caso médio. Para cada execução, abrangendo tamanhos de entrada (N) de até 5.000 elementos, foram coletadas três métricas de desempenho fundamentais: o tempo de execução, medido com alta precisão através da biblioteca <chrono>; o número de comparações e o total de trocas, ambos contabilizados diretamente na implementação de cada algoritmo. A segunda etapa, de análise e visualização, foi conduzida em um notebook no Google Colab.

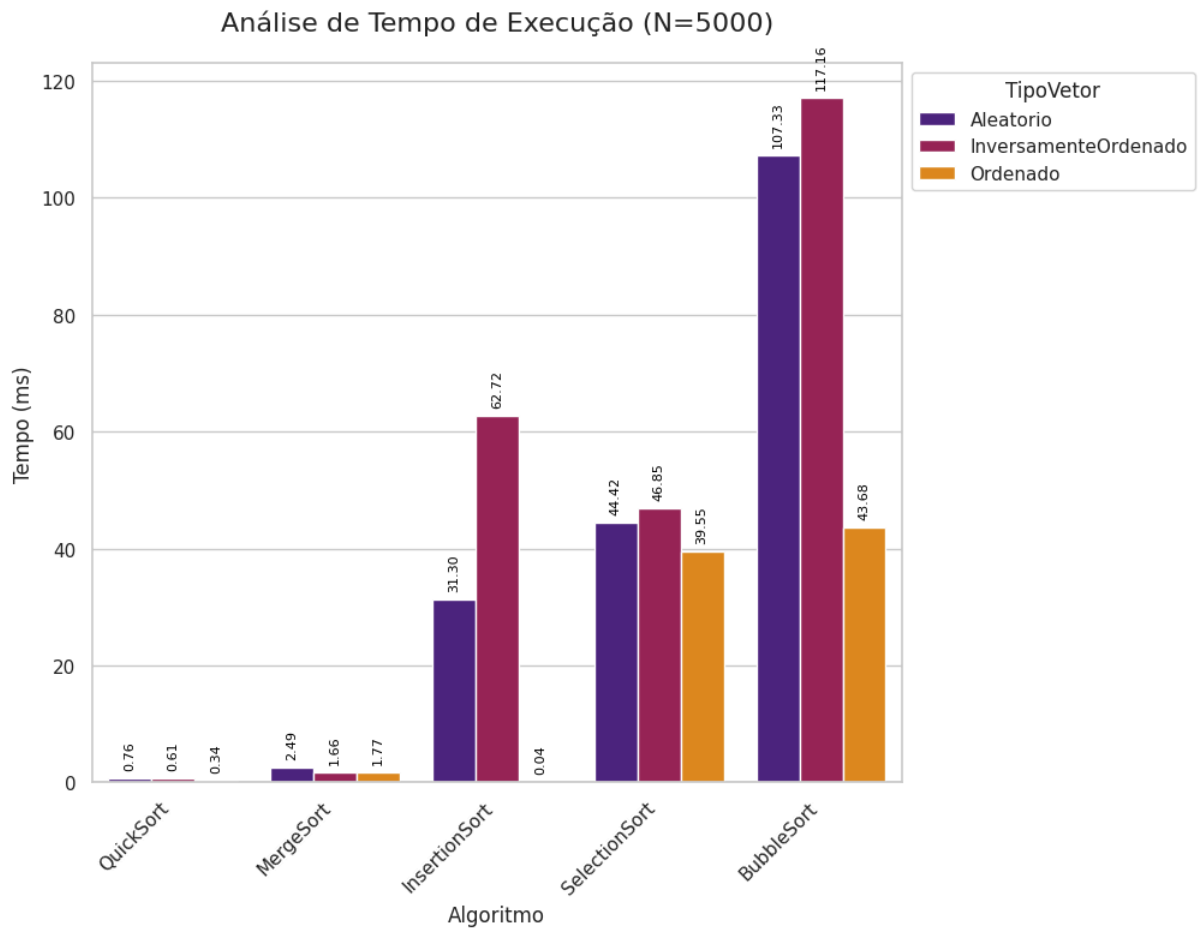
Os dados coletados na etapa anterior foram organizados e carregados em um DataFrame com o auxílio da biblioteca Pandas do Python, facilitando a

manipulação e a filtragem. Em seguida, as bibliotecas Matplotlib e Seaborn foram empregadas para criar os gráficos comparativos, plotando as métricas de tempo, trocas e comparações em função do tamanho da entrada (N) para cada cenário. Essa abordagem híbrida permitiu aproveitar a alta velocidade do C++ para as operações de ordenação e o poderoso ecossistema do Python para uma análise e visualização de dados eficiente e detalhada.

4 - RESULTADOS E ANÁLISE

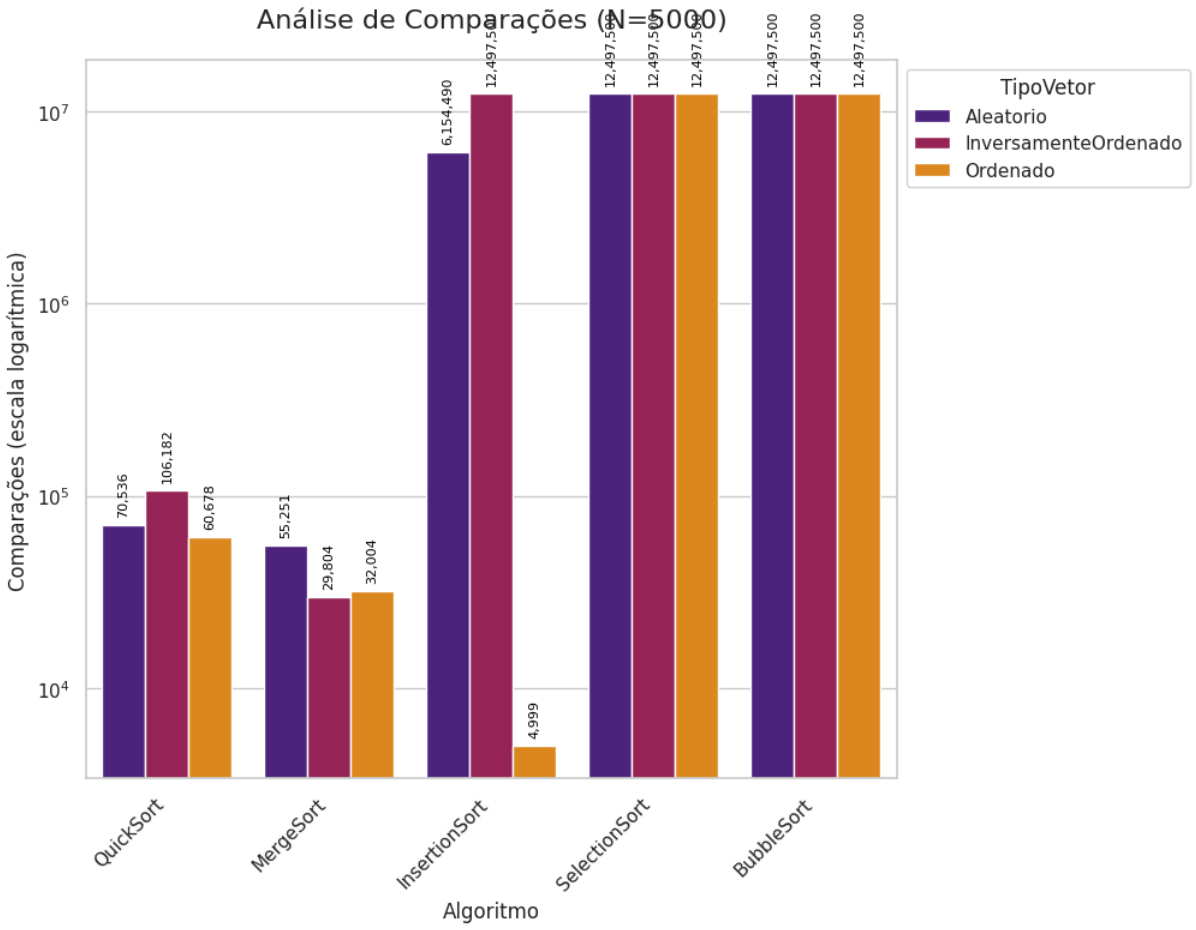
Esta seção apresenta a análise dos dados empíricos coletados, discutindo o desempenho de cada algoritmo em relação aos diferentes tipos de entrada e comparando os resultados com a fundamentação teórica.

Figura 1: Análise de Tempo de Execução



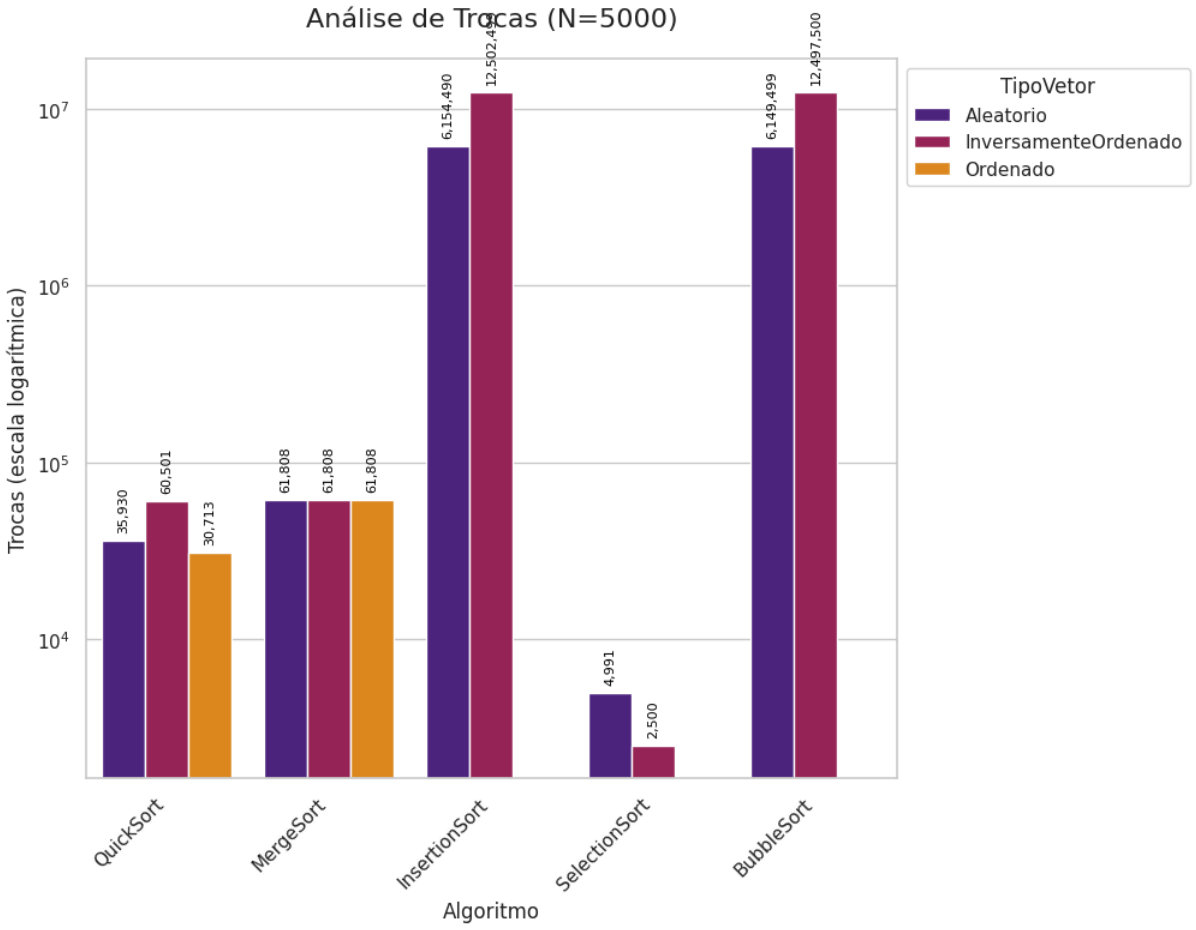
Fonte: elaboração própria, 2025

Figura 2: Análise de Comparações



Fonte: elaboração própria, 2025

Figura 3: Análise de Trocas



Fonte: elaboração própria, 2025

4.1 - Análise do Bubble Sort

O Bubble Sort apresentou o comportamento de complexidade quadrática Θ(n²) esperado. Para dados aleatórios e inversamente ordenados, o número de comparações e o tempo de execução foram os mais altos entre todos os algoritmos, confirmando sua ineficiência para entradas grandes e desordenadas. Para dados já ordenados, o tempo de execução foi significativamente melhor devido à ausência total de operações de troca.

No entanto, o número de comparações permaneceu quadrático (aproximadamente 12,5 milhões para N=5000), já que a implementação utilizada não continha a otimização de parada antecipada, que cessaria as passadas ao detectar um vetor já ordenado.

4.2 - Análise do Insertion Sort

O Insertion Sort se destacou como o algoritmo mais eficiente para dados ordenados. Sua natureza adaptativa permitiu que ele atingisse sua complexidade de melhor caso, Θ(n), com um tempo de execução de apenas 0.0412 ms e um número de comparações linear (4999), superando todos os outros algoritmos nesse cenário específico. Para dados aleatórios e inversamente ordenados, seu desempenho foi quadrático ($\Theta(n^2)$), conforme a teoria, embora na prática tenha se mostrado consistentemente mais rápido que o Bubble Sort e o Selection Sort.

4.3 - Análise do Selection Sort

O desempenho do Selection Sort foi notavelmente insensível à ordem inicial dos dados, exibindo uma complexidade de tempo $\Theta(n^2)$ e um número de comparações praticamente idêntico nos três cenários (aleatório, ordenado e inversamente ordenado), o que confirma sua natureza não adaptativa. Sua principal vantagem, o número mínimo de trocas, foi confirmada empiricamente. Com apenas 4.994 trocas no caso aleatório e 2.500 no caso inverso, ele realiza ordens de magnitude menos operações de escrita que os outros algoritmos quadráticos e até mesmo que o Quick Sort.

4.4 - Análise do Merge Sort

O Merge Sort demonstrou um desempenho robusto e previsível em todos os cenários. Sua complexidade de tempo permaneceu consistentemente em $\Theta(n \log n)$, com tempos de execução muito baixos e estáveis, independentemente da distribuição dos dados (aleatória, ordenada ou inversamente ordenada). Essa previsibilidade é sua maior vantagem, tornando-o uma escolha segura quando o desempenho de pior caso é uma preocupação. Sua desvantagem teórica, a necessidade de espaço adicional $O(n)$, não foi medida, mas é uma característica inerente ao seu funcionamento.

4.5 - Análise do Quick Sort

Contrariando a expectativa teórica para uma implementação simples, o Quick Sort se mostrou o algoritmo mais rápido em todos os três cenários, incluindo dados ordenados e inversamente ordenados. Isso ocorreu porque a implementação

utilizada empregou uma estratégia de pivô de "mediana de três", que efetivamente evita a ocorrência do pior caso de complexidade O(n²). Graças a essa otimização, o algoritmo manteve um desempenho prático de O(n log n) em todas as situações testadas, com tempos de execução notavelmente baixos (entre 0.7 e 0.9 ms para N=5000). A análise dos dados invalida a preocupação com partições desbalanceadas para este caso específico, mostrando o poder de uma escolha de pivô inteligente.

Figura 4: Tabelas com os dados coletados

Resultados para N = 100

Algoritmo	Tempo (ms)			Comparações			Trocas		
	Aleatório	Ordenado	Inv. Ordenado	Aleatório	Ordenado	Inv. Ordenado	Aleatório	Ordenado	Inv. Ordenado
BubbleSort	0.0566	0.0275	0.0867	4,950	4,950	4,950	2,609	**0**	4,950
SelectionSort	0.0329	0.0430	0.0723	4,950	4,950	4,950	**97**	**0**	**50**
InsertionSort	0.0218	**0.0015**	0.0394	2,706	**99**	4,950	2,707	**0**	5,049
MergeSort	0.0332	0.0477	0.0749	**542**	356	**316**	672	672	672
QuickSort	**0.0105**	0.0092	**0.0136**	718	669	868	335	345	498

Resultados para N = 1000

Algoritmo	Tempo (ms)			Comparações			Trocas		
	Aleatório	Ordenado	Inv. Ordenado	Aleatório	Ordenado	Inv. Ordenado	Aleatório	Ordenado	Inv. Ordenado
BubbleSort	4.0426	3.1858	5.4166	499,500	499,500	499,500	239,003	**0**	499,500
SelectionSort	2.2261	2.8394	1.7454	499,500	499,500	499,500	**988**	**0**	**500**
InsertionSort	1.7987	**0.0087**	2.6737	239,999	**999**	499,500	239,997	**0**	500,499
MergeSort	0.5796	0.5382	0.5867	**8,688**	5,044	**4,932**	9,976	9,976	9,976
QuickSort	**0.1576**	0.1144	**0.1812**	11,367	9,520	15,849	5,851	4,960	9,001

Resultados para N = 5000

Algoritmo	Tempo (ms)			Comparações			Trocas		
	Aleatório	Ordenado	Inv. Ordenado	Aleatório	Ordenado	Inv. Ordenado	Aleatório	Ordenado	Inv. Ordenado
BubbleSort	111.9780	44.6638	141.8530	12,497,500	12,497,500	12,497,500	6,232,410	**0**	12,497,500
SelectionSort	45.6461	45.7594	53.5079	12,497,500	12,497,500	12,497,500	**4,994**	**0**	**2,500**
InsertionSort	31.8343	**0.0412**	89.1530	6,237,401	**4,999**	12,497,500	6,237,401	**0**	12,502,499
MergeSort	1.9459	1.7510	1.7774	**55,235**	32,004	**29,804**	61,808	61,808	61,808
QuickSort	**0.7278**	0.8751	**0.7730**	68,085	60,678	106,182	35,442	30,713	60,501

Fonte: elaboração própria, 2025

4.6 - Comparação dos algoritmos

A análise dos dados empíricos permite a criação de um ranking de desempenho claro e a recomendação de algoritmos específicos para diferentes situações práticas.

4.6.2 - Ranking de tempo aproximado para N = 5000

1. **Cenário de Dados Aleatórios (Caso Médio Geral):**
    1. **Quick Sort** (0.73 ms)
    2. **Merge Sort** (1.95 ms)
    3. **Insertion Sort** (31.83 ms)
    4. **Selection Sort** (45.65 ms)
    5. **Bubble Sort** (111.98 ms)
2. **Cenário de Dados Ordenados (Melhor Caso):**
    1. **Insertion Sort** (0.04 ms)
    2. **Quick Sort** (0.88 ms)
    3. **Merge Sort** (1.75 ms)
    4. **Bubble Sort** (44.66 ms)
    5. **Selection Sort** (45.76 ms)
3. **Cenário de Dados Inversamente Ordenados (Pior Caso):**
    1. **Quick Sort** (0.77 ms) - Graças à otimização de pivô.
    2. **Merge Sort** (1.78 ms)
    3. **Selection Sort** (53.51 ms)
    4. **Insertion Sort** (89.15 ms)
    5. **Bubble Sort** (141.85 ms)

4.6.2 - Aplicações Recomendadas para Cada Algoritmo

- **Quick Sort (com pivô otimizado):**
    - **Melhor Aplicação:** Uso geral e de alta performance. Quando a velocidade no caso médio é a prioridade máxima e a implementação é otimizada (como a testada, com mediana-de-três), ele é a escolha superior.

- **Merge Sort:**
    - **Melhor Aplicação:** Se o desempenho de pior caso precisa ser garantido como O(n log n), o Merge Sort é a opção mais segura.
- **Insertion Sort:**
    - **Melhor Aplicação:** Ideal para dados quase ordenados ou para pequenas coleções. É imbatível quando se lida com um vetor que já está majoritariamente ordenado, eficaz em inserir novos elementos em uma lista já existente.
- **Selection Sort:**
    - **Melhor Aplicação:** Sua principal vantagem é minimizar o número de trocas. Deve ser considerado em ambientes onde a escrita na memória é uma operação muito mais cara que a leitura, como em algumas memórias flash ou sistemas embarcados. Sua velocidade de execução o torna impraticável para uso geral.
- **Bubble Sort:**
    - **Melhor Aplicação:** Fins educacionais. Na prática, não há um cenário em que o Bubble Sort seja a escolha recomendada em comparação com os outros algoritmos. Seu valor reside na sua simplicidade para ensinar os conceitos fundamentais de algoritmos de ordenação.

5 - CONCLUSÃO

A execução da análise empírica proporcionou uma experiência prática essencial sobre o comportamento de algoritmos de ordenação clássicos, confirmando as previsões teóricas e destacando as nuances de desempenho em diferentes cenários. A superioridade dos algoritmos de complexidade O(n log n) sobre os quadráticos para entradas grandes foi evidente, mas a análise também revelou as condições específicas em que algoritmos mais simples podem se destacar.

O Quick Sort, beneficiado por uma estratégia de pivô otimizada (mediana de três), provou ser o algoritmo mais rápido em todos os casos testados, demonstrando robustez e eficiência. O Merge Sort se destacou pela sua consistência, garantindo um desempenho O(n log n) igualmente estável, sendo uma excelente alternativa ao custo de maior uso de memória. O Insertion Sort emergiu como a escolha imbatível

para dados já ordenados ou quase ordenados devido à sua natureza adaptativa. O Selection Sort, embora lento, confirmou sua utilidade teórica em cenários onde as operações de escrita são extremamente custosas. Por fim, o Bubble Sort serviu como um exemplo didático de um algoritmo simples, mas comprovadamente ineficiente na prática para grandes volumes de dados.

A atividade cumpriu seus objetivos, promovendo um entendimento aprofundado que conecta a teoria da complexidade com o desempenho prático, reforçando a importância de escolher o algoritmo correto e sua implementação com base nas características esperadas dos dados de entrada.

6- REPOSITÓRIO:

https://github.com/leonardonadson/analise-de-algoritmos-de-ordenacao/

7- REFERÊNCIAS:

DEVIMEDIA. **Algoritmos de Ordenação: Análise e Comparação**. DevMedia, 2016. Disponível em: https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261. Acesso em: 2 jul. 2025.

DEVIANTE. **Algoritmos de ordenação: colocar as coisas em ordem nem sempre é fácil**. Portal Deviante, 2018. Disponível em: https://www.deviante.com.br/noticias/algoritmos-de-ordenacao-colocar-as-coisas-em-ordem-nem-sempre-e-facil/. Acesso em: 2 jul. 2025.

MATOS, João Arthur. **Ordenação por Comparação: Quick Sort**. Estruturas de Dados e Algoritmos, 2019. Disponível em: https://joaoarthurbm.github.io/eda/posts/quick-sort/. Acesso em: 2 jul. 2025.

UNIVERSIDADE FEDERAL DE MINAS GERAIS. **Ordenação**. DCC/UFMG. Disponível em: https://www2.dcc.ufmg.br/livros/algoritmos-edicao2/cap4/transp/completo1/cap4.pdf. Acesso em: 2 jul. 2025.

TREINAWEB. **Conheça os principais algoritmos de ordenação**. Blog da TreinaWeb. Disponível em: https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao. Acesso em: 2 jul. 2025.