Desempenho Comparativo de QuickSort, MergeSort e HeapSort em Cenários Variados de Entrada

Edinéia dos Santos Brizola Brum[1], **Jefferson Rodrigo Speck**[1], **Rafael Ferreira Lima**[1]

[1]Programa de Pós-Graduação em Ciência da Computação
Universidade Estadual do Oeste do Paraná - Unioeste
Cascavel, Paraná – Brasil

Abstract. *This study aimed to compare the performance of the classic sorting algorithms* `QuickSort`*,* `MergeSort`*, and* `HeapSort`*, analyzing their execution time efficiency across different input patterns and data volumes. To achieve this, empirical experiments were conducted in a controlled environment, measuring the execution time of the algorithms on arrays ranging from* 100 *to* $2{,}000{,}000$ *elements, distributed across four distinct scenarios: data sorted in ascending order, descending order, randomly shuffled, and partially sorted. The results revealed that* `QuickSort` *is sensitive to the initial ordering of the data, experiencing significant performance degradation in unfavorable cases;* `MergeSort` *exhibited stable behavior regardless of the input structure; and* `HeapSort` *showed consistent performance with good predictability. It is concluded that* `QuickSort`*, although efficient on average cases, requires attention regarding pivot selection;* `MergeSort` *stood out for its low variability and stability; and* `HeapSort` *demonstrated structural robustness and uniform behavior even with large datasets and varying distributions.*

Resumo. *Este estudo teve como objetivo comparar o desempenho dos algoritmos de ordenação* `QuickSort`*,* `MergeSort` *e* `HeapSort`*, analisando sua eficiência em tempo de execução frente a diferentes padrões e volumes de entrada. Para isso, foram realizados experimentos empíricos em ambiente controlado, mensurando o tempo de execução dos algoritmos sobre vetores com tamanhos variando entre* 100 *e* $2.000.000$ *elementos, distribuídos em quatro cenários distintos: dados ordenados crescentemente, ordenados decrescentemente, aleatórios e parcialmente ordenados. Os resultados revelaram que o* `QuickSort` *é sensível à ordenação inicial dos dados, sofrendo degradação significativa em casos desfavoráveis; o* `MergeSort` *demonstrou comportamento estável independentemente da estrutura dos dados de entrada; e o* `HeapSort` *apresentou desempenho consistente com boa previsibilidade. Conclui-se que o* `QuickSort`*, embora eficiente em casos médios, demanda atenção quanto à escolha do pivô; o* `MergeSort` *destacou-se pela baixa variabilidade e estabilidade; e o* `HeapSort` *mostrou robustez estrutural e comportamento uniforme mesmo com grandes volumes de dados e diferentes distribuições.*

1. Introdução

Os algoritmos de ordenação representam, historicamente, uma área central na Ciência da Computação, influenciando tanto a análise teórica quanto as aplicações práticas em diversos domínios. A modelagem, análise e comparação dessas técnicas fornecem *insights*

relevantes sobre a eficiência algorítmica, a complexidade computacional e o uso de recursos. Em muitos sistemas computacionais, uma ordenação eficiente é essencial para a gestão, recuperação e otimização de dados, impactando diretamente o desempenho geral das aplicações [Mohammadagha 2025].

Algoritmos clássicos como `QuickSort`, `MergeSort` e `HeapSort` são amplamente reconhecidos na literatura por sua importância teórica e aplicabilidade prática. Suas complexidades assintóticas são bem documentadas em obras fundamentais como [Cormen et al. 2012]. Os três algoritmos apresentam a mesma complexidade assintótica $\mathcal{O}(n \log n)$ no melhor caso e em casos médios. No pior caso, enquanto o `MergeSort` e o `HeapSort` continuam com uma complexidade assintótica $\mathcal{O}(n \log n)$, o `QuickSort` varia para uma complexidade assintótica $\mathcal{O}(n^2)$. Todavia, o desempenho real desses algoritmos depende da análise teórica e de fatores empíricos, como a implementação, o volume de dados e, sobretudo, a distribuição dos elementos na entrada [Mohammadagha 2025].

Diante desse panorama, este trabalho tem como objetivo realizar uma análise empírica comparativa dos algoritmos `QuickSort`, `MergeSort` e `HeapSort`, aplicados a diferentes configurações de entrada. São comparados os tempos de execução dos algoritmos em quatro cenários distintos: dados previamente ordenados, parcialmente ordenados, aleatórios e ordenados em ordem decrescente.

Este artigo está organizado da seguinte forma: a Seção 2 apresenta a fundamentação teórica dos algoritmos estudados e análise teórica dos algoritmos contruídos; a Seção 3 descreve a metodologia experimental adotada; a Seção 4 expõe e discute os resultados obtidos; e a Seção 5 apresenta as conclusões e sugestões para trabalhos futuros.

2. Construção e Análise Teórica dos Algoritmos de Ordenação

Esta seção apresenta a fundamentação teórica dos algoritmos de ordenação utilizados no presente estudo: `QuickSort`, `MergeSort` e `HeapSort`. São descritas suas estruturas lógicas, estratégias de ordenação, bem como a forma como foram implementados com instrumentação para coleta de métricas. Além da descrição funcional, são analisadas suas complexidades teóricas nos melhores, médios e piores casos, dos algoritmos desenvolvidos para este estudo, considerando critérios como número de comparações, consumo de memória e comportamento recursivo.

2.1. QuickSort

Desenvolvido por Tony Hoare em 1960, o algoritmo `QuickSort` pertence à classe dos métodos de ordenação baseados na estratégia de "dividir para conquistar", sendo amplamente reconhecido por sua eficiência média e por seu uso em bibliotecas padrão de diversas linguagens de programação [Wild et al. 2013].

Segundo [Cormen et al. 2009], sua lógica consiste na escolha de um elemento pivô para particionar o vetor: todos os elementos menores que o pivô são posicionados à esquerda, e os maiores, à direita. Esse processo ocorre *in-place*, ou seja, sem uso de estruturas auxiliares, por meio de trocas diretas no próprio vetor. A qualidade do particionamento é crucial, pois influencia diretamente a profundidade da recursão e, consequentemente, o tempo total de execução.

Após essa etapa, o algoritmo é aplicado recursivamente aos subvetores resultantes, até que todos possuam tamanho um ou zero — condição em que a ordenação já está garantida. O desempenho geral, portanto, depende fortemente do equilíbrio entre as partições formadas a cada chamada recursiva.

Nos casos em que o pivô gera divisões altamente desbalanceadas — como uma partição com $n-1$ elementos e outra vazia — o tempo de execução se deteriora. A função de tempo associada é $T(n) = T(n-1)+\Theta(n)$, cuja solução resulta em uma complexidade de $\Theta(n^2)$. Esse comportamento é comum quando os dados já se encontram ordenados ou em ordem reversa, tornando o método menos eficiente até mesmo que algoritmos mais simples, como a ordenação por inserção.

Por outro lado, na divisão mais equilibrada possível — com subproblemas de tamanho próximo a $n/2$ —, a complexidade cai para $\Theta(n \log n)$, segundo o caso 2 do Teorema Mestre. Nessa situação, o desempenho se aproxima do ideal, com partições que minimizam a profundidade da árvore de chamadas.

Na maioria dos casos práticos, especialmente com entradas aleatórias, é improvável que as divisões ocorram de forma consistentemente ruim. O particionamento tende a produzir uma combinação de divisões boas e ruins, distribuídas de maneira aleatória na árvore recursiva. Mesmo com essa alternância, o tempo médio de execução permanece assintoticamente eficiente, da ordem de $\mathcal{O}(n \log n)$, embora com constante oculta ligeiramente superior à do melhor caso.

Em síntese, o sucesso da abordagem está diretamente ligado à escolha do pivô e à estrutura dos dados de entrada. Estratégias como a seleção aleatória ou a mediana de três são frequentemente utilizadas para evitar partições desfavoráveis, assegurando uma performance mais próxima do comportamento médio mesmo em situações adversas [Backes 2023].

2.1.1. Análise teórica do algoritmo deste estudo

A seguir, apresentam-se as estimativas teóricas para o pior caso e o caso médio do algoritmo implementado neste estudo, com base na análise assintótica da estrutura recursiva e na contagem de comparações.

No pior cenário, o particionamento induzido pelo pivô resulta em divisões altamente desbalanceadas, como ocorre quando o pivô escolhido é sistematicamente o menor (ou maior) elemento da partição ativa. Nessa configuração, cada chamada recursiva atua sobre um subvetor de tamanho $n-1$, gerando uma cadeia de chamadas com profundidade máxima. A função de tempo associada é: $T(n) = T(n-1) + \Theta(n)$, cuja solução, por expansão da recorrência, resulta em uma complexidade quadrática: $T(n) = \frac{n(n-1)}{2}$.

Essa equação representa a soma dos termos de uma progressão aritmética, em que:

- Primeiro termo: $a_1 = 1$
- Último termo: $a_{n-1} = n - 1$
- Número de termos: $n - 1$
- Soma: $S = \frac{n(n-1)}{2}$

A Tabela 1 ilustra esse comportamento, simulando o número de comparações acumuladas em cada iteração para uma entrada de tamanho $n = 10$.

Tabela 1. Comparações realizadas no pior caso para diferentes tamanhos de entrada (`QuickSort`)

Tamanho do Subvetor (n_i)	Comparações na Iteração	Comparações Acumuladas
10	9	9
9	8	17
8	7	24
7	6	30
6	5	35
5	4	39
4	3	42
3	2	44
2	1	45
1	0	45
Total	–	$\frac{n(n-1)}{2} = 45$

No caso médio, assume-se que o particionamento tende a produzir subvetores aproximadamente equilibrados. Ainda que nem todas as divisões sejam ideais, espera-se uma alternância entre partições boas e ruins ao longo da árvore de chamadas, o que mantém a profundidade limitada e o custo da ordenação dentro de uma complexidade assintoticamente eficiente: $T(n) \approx n \log_2 n$.

Mesmo nesse cenário, o número de comparações realizadas em cada chamada da função `partition` é igual ao tamanho do segmento considerado, ou seja, `high` $-$ `low`. Ainda que neste estudo tenha sido adotado o pivô central como estratégia de particionamento, o que pode reduzir significativamente a chance de divisões desbalanceadas , entradas com ordenação crescente ou decrescente ainda podem induzir o comportamento de pior caso, sobretudo quando a estrutura dos dados favorece divisões assimétricas.

Portanto, mesmo com heurísticas aprimoradas, o número total de comparações pode crescer quadraticamente, conforme a expressão da progressão aritmética apresentada, evidenciando a sensibilidade do método à escolha do pivô e à distribuição dos dados [Cormen et al. 2009].

2.2. MergeSort

De acordo com [Cormen et al. 2009], o `MergeSort` é um algoritmo recursivo baseado na estratégia de divisão e conquista. Seu funcionamento consiste em dividir o vetor de entrada em duas partes, aplicar recursivamente o mesmo procedimento a cada subvetor até que se atinja o caso base (vetores unitários), e então combinar os resultados parciais por meio da intercalação ordenada dos elementos.

A etapa de combinação é realizada pela função `MERGE`, que compara os elementos de dois subvetores já ordenados, inserindo os menores, um a um, em um vetor auxiliar. Essa operação garante a preservação da ordenação global e assegura a estabilidade do algoritmo, ou seja, elementos iguais mantêm a mesma ordem relativa presente na entrada.

No pior caso, o algoritmo realiza todas as divisões e combinações independentemente da estrutura dos dados. O vetor é segmentado ao meio em cada chamada re-

cursiva, o que produz uma árvore de altura $\log_2 n$. Em cada nível dessa árvore, ocorre a intercalação dos elementos, com custo linear em relação ao tamanho total do vetor. Como essas operações são executadas em todos os níveis, a complexidade total resulta em $\mathcal{O}(n \log n)$. Esse comportamento permanece inalterado mesmo em situações desfavoráveis, como vetores ordenados em ordem inversa ou com muitos elementos duplicados.

No melhor caso, o comportamento do algoritmo também não se altera. Mesmo que os dados já estejam ordenados, todas as etapas de divisão e combinação são realizadas integralmente. Isso ocorre porque o `MergeSort` não possui mecanismos internos para detectar ordenação prévia ou interromper o processo de forma otimizada. Assim, o número de operações permanece constante em relação a qualquer entrada de mesmo tamanho, mantendo a complexidade no melhor caso também em $\mathcal{O}(n \log n)$.

No caso médio, espera-se o mesmo desempenho observado nos extremos. Como o algoritmo aplica exatamente a mesma sequência de operações recursivas e de mesclagem para qualquer distribuição dos dados, a complexidade esperada continua sendo $\mathcal{O}(n \log n)$. Essa regularidade torna o `MergeSort` particularmente valioso em contextos que exigem previsibilidade de tempo de execução, independentemente da estrutura da entrada.

Diferentemente de métodos como o `QuickSort`, cuja eficiência depende da escolha do pivô e da organização dos dados, o `MergeSort` oferece um desempenho mais uniforme. Entretanto, essa estabilidade tem como contrapartida o uso adicional de memória auxiliar proporcional a n, uma vez que a operação de mesclagem é realizada em vetores temporários [Backes 2023]. Ainda assim, em aplicações que priorizam robustez e previsibilidade, o algoritmo se destaca como uma das alternativas mais confiáveis para ordenação eficiente.

2.2.1. Análise teórica do algoritmo deste estudo

A eficiência do algoritmo `MergeSort` implementado neste estudo está diretamente associada à sua fase de mesclagem (`merge`), responsável por combinar recursivamente os subvetores previamente ordenados. Durante a execução, o vetor de entrada é segmentado repetidamente até que cada subvetor contenha apenas um elemento. A partir desse ponto, inicia-se o processo de recombinação, no qual os pares de subvetores são mesclados de forma ordenada. Em cada nível da árvore de recursão, o custo da etapa de intercalação é proporcional ao número total de elementos envolvidos, ou seja, no máximo n comparações por nível. Como a altura da árvore é $\log_2 n$, a estimativa superior para o número total de comparações é dada por: $T(n) \leq n \log_2 n$.

A estrutura da função `merge`, empregada na implementação, realiza comparações diretas entre os elementos dos subvetores auxiliares esquerdo (`L`) e direito (`R`). O trecho abaixo exemplifica o funcionamento do laço principal responsável por essa operação que foi implementado em `C`:

```
while (i < n1 && j < n2) {
    comparacoes++;
    arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
}
```

Esse laço garante que, a cada iteração, o menor elemento entre os dois subvetores seja transferido para a posição correta do vetor principal, mantendo a ordenação. Para cada chamada da função `merge`, o número de comparações é proporcional ao tamanho do segmento considerado, e o pior caso ocorre quando os elementos de ambos os lados se alternam, exigindo o número máximo possível de comparações.

A Tabela 2 ilustra uma simulação do comportamento do algoritmo para uma entrada de tamanho $n = 8$, detalhando o número estimado de comparações por nível da árvore de recursão, considerando um cenário de mesclagem plena em cada etapa.

Tabela 2. Estimativa de comparações por nível de recursão no `MergeSort` para $n = 8$

Nível da Recursão	Mesclas Realizadas	Comparações Estimadas
1 (tamanho 1)	4	4 (1 por mescla)
2 (tamanho 2)	2	4 (2 por mescla)
3 (tamanho 4)	1	7 (até $n - 1$)
Total	7	$\leq n \log_2 n = 8 \cdot 3 = 24$

Embora o número exato de comparações possa variar em função da distribuição dos dados, o comportamento assintótico permanece limitado superiormente por $\mathcal{O}(n \log n)$, independentemente da ordenação inicial. Essa estabilidade é uma das principais vantagens do algoritmo. Por outro lado, o uso de memória adicional é inevitável, pois a mesclagem requer vetores auxiliares temporários com espaço proporcional a n. Essa característica foi considerada nas medições deste estudo, embora não faça parte da análise teórica formal aqui apresentada.

2.3. HeapSort

Segundo [Cormen et al. 2009], o `HeapSort` realiza a ordenação com base em uma estrutura de dados chamada *heap* binário, mais especificamente um *max-heap*, no qual cada elemento interno é maior ou igual aos seus filhos. A execução ocorre em duas fases: primeiramente, o vetor é reorganizado para formar um *max-heap*, por meio da rotina `BUILD-MAX-HEAP`, que garante a propriedade de ordenação interna da estrutura. Em seguida, são feitas sucessivas chamadas à função `MAX-HEAPIFY`, que restaura a propriedade de *heap* após a troca do maior elemento (a raiz) com o último elemento da porção ativa do vetor.

O algoritmo opera diretamente sobre o vetor original, sem utilizar memória auxiliar, sendo caracterizado como um método *in-place*. A cada iteração, o maior valor é deslocado para a posição final disponível, enquanto o tamanho lógico do *heap* é reduzido. As operações internas de navegação entre elementos pais e filhos são baseadas em cálculos de índices, o que dispensa o uso de estruturas adicionais.

No pior caso, a complexidade do `HeapSort` é $\mathcal{O}(n \log n)$. A fase inicial de construção do *heap* apresenta custo linear, $\mathcal{O}(n)$, enquanto a segunda fase realiza $n - 1$ chamadas à rotina `MAX-HEAPIFY`, cada uma com custo proporcional à altura do *heap*, que é $\log n$. Assim, o tempo total de execução permanece limitado por $n \log n$, mesmo em cenários adversos.

No melhor caso, ainda que os elementos já estejam ordenados, o `HeapSort` não apresenta otimizações. A estrutura do *heap* é integralmente construída, e as operações de extração e reequilíbrio continuam sendo executadas. Isso mantém a complexidade em $\mathcal{O}(n \log n)$, sem ganhos de desempenho decorrentes da entrada inicial.

No caso médio, o comportamento do algoritmo não se altera significativamente. Como a rotina `MAX-HEAPIFY` é aplicada de forma sistemática e o número de chamadas não depende da distribuição dos dados, a complexidade esperada também permanece em $\mathcal{O}(n \log n)$.

2.3.1. Análise teórica do algoritmo deste estudo

O algoritmo implementado neste trabalho segue a abordagem clássica de construção e manutenção do *heap* binário máximo. A função `heapify` mantém a propriedade de ordenação a partir de um nó i até as folhas, realizando comparações para identificar o maior valor entre o nó pai e seus filhos. Em cada chamada, podem ocorrer até duas comparações, seguidas de uma troca e, se necessário, uma chamada recursiva para restabelecer a estrutura. A construção inicial do *heap* é realizada com chamadas de `heapify` para os índices da metade inferior do vetor, resultando em um custo linear: $T_{\text{build}}(n) = \mathcal{O}(n)$.

Após a construção, o algoritmo realiza trocas sucessivas entre o maior elemento (na raiz) e o último elemento disponível, reduzindo a área do *heap* e reaplicando `heapify`. Como o tempo de execução de `heapify` é limitado pela altura da árvore, temos: $T_{\text{heapify}}(n) = \mathcal{O}(\log n)$, e a fase de ordenação é repetida $n - 1$ vezes, resultando na complexidade total: $T(n) = \mathcal{O}(n \log n)$. A Tabela 3 apresenta uma estimativa do número de comparações realizadas durante a construção do *heap* e durante a fase de ordenação para uma entrada com $n = 7$.

Tabela 3. Estimativa de comparações no `HeapSort` para $n = 7$

Fase	Chamadas	Comparações Estimadas
Construção do Heap	3	$\leq 2 + 2 + 2 = 6$
Ordenação (6 extrações)	6	$\leq 6 \cdot \log_2 6 \approx 15.5$
Total Estimado	9	≤ 22

Em síntese, o `HeapSort` mantém a complexidade assintótica $\mathcal{O}(n \log n)$ independentemente do padrão de ordenação inicial. Contudo, tende a realizar um número maior de trocas e acessos à memória quando comparado a outros algoritmos eficientes, como o `MergeSort`, principalmente devido à necessidade de reequilíbrios frequentes na estrutura de *heap*.

3. Materiais e Métodos

O estudo foi conduzido por meio de uma análise experimental controlada, com foco na comparação empírica entre os algoritmos de ordenação `QuickSort`, `MergeSort` e `HeapSort` [Wohlin et al. 2012]. A infraestrutura computacional consistiu em um com-

putador pessoal com processador *Intel Core i7-9700K* (até 4,9 GHz, 12 MB de cache)[1], 32 GB de memória RAM $DDR4$ (4×8 GB a 3200 MHz) e unidade de armazenamento SSD NVMe de 512 GB (com velocidade de leitura de 2300 MB/s e gravação de 900 MB/s). O sistema operacional foi o Ubuntu 22.04 LTS, executado via *Windows Subsystem for Linux* (WSL)[2] sobre o Windows 11 (versão $24H2$, compilação 26100.3915).

As implementações dos algoritmos foram feitas na linguagem C, selecionada por sua proximidade com a literatura clássica [Cormen et al. 2009] e por permitir controle granular sobre tempo de execução, comparações e alocações dinâmicas de memória. O projeto foi modularizado, com cada algoritmo mantido em arquivos separados, além de um módulo principal responsável pela execução automatizada dos testes.

Os dados de entrada foram organizados em arquivos de texto com extensão `.txt`, contendo números inteiros, um por linha, sem cabeçalho. Cada nome de arquivo codifica o tipo e o tamanho do vetor, por exemplo, `a1000.txt` representa um vetor de 1000 elementos aleatórios. Os tipos considerados foram: aleatório, ordenado crescente, ordenado decrescente e parcialmente ordenado. Durante a execução, os vetores são carregados, copiados internamente e submetidos separadamente aos três algoritmos, garantindo que todos operem sobre os mesmos dados em condições idênticas.

Foram realizadas sete execuções para cada algoritmo e conjunto de dados. Com o objetivo de reduzir os efeitos de variações aleatórias e ruídos experimentais, os valores extremos (melhor e pior tempo de execução em cada grupo) foram descartados, permanecendo cinco execuções por combinação. A coleta de métricas compreendeu três frentes principais:

- Tempo de execução, medido em segundos com base na função `clock()` da biblioteca padrão `<time.h>`;
- Número de comparações, contabilizado manualmente nas estruturas de decisão de cada algoritmo;
- Memória utilizada, estimada a partir das alocações dinâmicas realizadas por meio das funções `malloc()` e `realloc()`.

Os resultados de cada execução foram salvos em arquivos no formato `.csv`, armazenados na pasta `output/`. Os dados tabulados incluem o nome do arquivo de origem, tipo e tamanho do vetor, algoritmo aplicado, tempo de execução, número de comparações e uso de memória. Para fins de comparação deste estudo foram utilizadas a métricas de média e desvio padrão.

3.1. Tratamento Estatístico e Visualização

Após a coleta dos dados experimentais, procedeu-se à etapa de tratamento estatístico utilizando a linguagem Python (versão 3.10 ou superior). Para isso, foram empregadas bibliotecas amplamente adotadas na comunidade científica para análise de dados. A biblioteca `pandas`[3] foi utilizada para a manipulação dos dados tabulares, permitindo a

[1]Detalhes do processador disponível em: `https://www.intel.com.br/content/www/br/pt/products/sku/186604/intel-core-i79700k-processor-12m-cache-up-to-4-90-ghz/specifications.html`

[2]Documentação disponível em: `https://learn.microsoft.com/pt-br/windows/wsl/`

[3]Documentação disponível em: `https://pandas.pydata.org/docs/` (versão mínima: 1.4)

leitura dos arquivos `.csv`, filtragem de dados e realização de operações de agregação por grupo. Para a construção dos gráficos, recorreu-se à biblioteca `matplotlib`[4], responsável pela geração de visualizações como gráficos de linha e barras. Complementarmente, empregou-se a biblioteca `seaborn`[5], que oferece recursos estatísticos avançados para visualização de dados, como o suporte a intervalos de confiança e a inclusão de barras de erro (± desvio padrão) nas representações gráficas. Essas ferramentas, em conjunto, proporcionaram uma análise visual e estatística robusta dos resultados obtidos nos experimentos.

Essas bibliotecas permitiram organizar os dados experimentais conforme o algoritmo utilizado, o tipo de dado de entrada e o tamanho do vetor, possibilitando o cálculo de médias e desvios padrão para as três métricas observadas: tempo de execução, número de comparações e uso de memória. Com base nesses dados agregados, foram gerados dois conjuntos principais de análises comparativas. O primeiro conjunto apresenta, para cada algoritmo, o desempenho observado frente a diferentes tipos de entrada, permitindo identificar padrões de variação associados às características dos dados. O segundo conjunto reúne gráficos comparativos entre os três algoritmos aplicados a cada tipo de dado, destacando suas diferenças de desempenho com o uso de barras de erro (± desvio padrão), as quais evidenciam a variabilidade entre execuções.

3.2. Reprodutibilidade e Organização do Projeto

O projeto foi organizado em uma estrutura modular e documentada, incluindo código-fonte em `C`, arquivos de entrada, *scripts* de análise estatística em Python e os resultados consolidados. Todo o conteúdo foi versionado via *Git* e está disponível publicamente em repositório no GitHub[6].

4. Resultados

Esta seção apresenta a análise empírica do desempenho dos algoritmos `QuickSort`, `MergeSort` e `HeapSort` frente a diferentes tipos de entrada e tamanhos de dados. Os experimentos foram conduzidos utilizando conjuntos com características distintas, com o objetivo de avaliar o comportamento dos algoritmos em cenários que representam casos médios, melhores e piores de desempenho. As métricas consideradas incluem o tempo médio de execução e o desvio padrão. Os resultados obtidos foram organizados em duas tabelas: a Tabela 4, que agrupa os dados para tamanhos de entrada até 50.000 elementos, e a Tabela 5, que apresenta os dados para conjuntos com 100.000 elementos ou mais. A análise está estruturada por algoritmo individualmente, seguida de uma comparação cruzada entre eles, com base nos valores consolidados nas referidas tabelas.

[4]Documentação disponível em: `https://matplotlib.org/stable/index.html` (versão mínima: 3.5)

[5]Documentação disponível em: `https://seaborn.pydata.org/whatsnew/` (versão mínima: 0.11)

[6]Arquivos e resultados do experimento estão disponíveis em: `https://github.com/jeffersonspeck/edaa_analise_2`

Tabela 4. Tempo médio de execução (em segundos) e desvio padrão para tamanhos até 50.000

Tipo de Dado	Algoritmo	100	1.000	10.000	50.000
Crescente	`HeapSort`	0,000009 ±0,000001	0,000111 ±0,000004	0,001362 ±0,000014	0,007048 ±0,000123
Crescente	`MergeSort`	0,000009 ±0,000001	0,000138 ±0,000003	0,001555 ±0,000032	0,007860 ±0,000100
Crescente	`QuickSort`	0,000007 ±0,000001	0,000103 ±0,000003	0,001138 ±0,000015	0,005635 ±0,000046
Decrescente	`HeapSort`	0,000010 ±0,000001	0,000130 ±0,000004	0,001508 ±0,000012	0,007753 ±0,000065
Decrescente	`MergeSort`	0,000011 ±0,000001	0,000161 ±0,000003	0,001783 ±0,000020	0,009015 ±0,000102
Decrescente	`QuickSort`	0,000012 ±0,000001	0,000153 ±0,000005	0,002216 ±0,000021	0,021992 ±0,000249
Parcialmente Ordenado	`HeapSort`	0,000009 ±0,000001	0,000113 ±0,000002	0,001354 ±0,000020	0,007055 ±0,000103
Parcialmente Ordenado	`MergeSort`	0,000010 ±0,000001	0,000146 ±0,000003	0,001670 ±0,000035	0,008498 ±0,000113
Parcialmente Ordenado	`QuickSort`	0,000008 ±0,000001	0,000109 ±0,000003	0,001222 ±0,000012	0,006019 ±0,000053
Aleatório	`HeapSort`	0,000010 ±0,000001	0,000125 ±0,000004	0,001468 ±0,000017	0,007548 ±0,000107
Aleatório	`MergeSort`	0,000010 ±0,000001	0,000151 ±0,000004	0,001707 ±0,000024	0,008647 ±0,000095
Aleatório	`QuickSort`	0,000009 ±0,000001	0,000122 ±0,000003	0,001366 ±0,000017	0,006868 ±0,000077

Tabela 5. Tempo médio de execução (em segundos) e desvio padrão para tamanhos a partir de 100.000

Tipo de Dado	Algoritmo	100.000	500.000	1.000.000	2.000.000
Crescente	`HeapSort`	0,014118 ±0,000143	0,073269 ±0,000324	0,151925 ±0,000958	0,306336 ±0,001390
Crescente	`MergeSort`	0,015715 ±0,000183	0,079506 ±0,000310	0,162192 ±0,000464	0,326406 ±0,001072
Crescente	`QuickSort`	0,011190 ±0,000123	0,057763 ±0,000247	0,116920 ±0,000744	0,234472 ±0,001040
Decrescente	`HeapSort`	0,015417 ±0,000089	0,079424 ±0,000235	0,164036 ±0,000568	0,330336 ±0,001058
Decrescente	`MergeSort`	0,018027 ±0,000225	0,091067 ±0,000386	0,185887 ±0,000520	0,375758 ±0,001151
Decrescente	`QuickSort`	0,050765 ±0,000540	0,269319 ±0,000817	0,561337 ±0,002187	1,127077 ±0,002951
Parcialmente Ordenado	`HeapSort`	0,014175 ±0,000118	0,073522 ±0,000274	0,152331 ±0,000388	0,306620 ±0,001134
Parcialmente Ordenado	`MergeSort`	0,017018 ±0,000189	0,085405 ±0,000286	0,174951 ±0,000616	0,353724 ±0,001216
Parcialmente Ordenado	`QuickSort`	0,011957 ±0,000122	0,059621 ±0,000167	0,120570 ±0,000781	0,241761 ±0,001356

Tipo de Dado	Algoritmo	100.000	500.000	1.000.000	2.000.000
Aleatório	`HeapSort`	0,015043 ±0,000151	0,077658 ±0,000314	0,159748 ±0,000503	0,319669 ±0,001123
Aleatório	`MergeSort`	0,017328 ±0,000252	0,087434 ±0,000447	0,178187 ±0,000643	0,360822 ±0,001227
Aleatório	`QuickSort`	0,013800 ±0,000160	0,068832 ±0,000267	0,139360 ±0,000837	0,279728 ±0,001204

4.1. Análise do Desempenho do `QuickSort`

Segundo [Cormen et al. 2009], o algoritmo `QuickSort` possui complexidade média e melhor caso igual a $\mathcal{O}(n \log n)$, podendo degradar para $\mathcal{O}(n^2)$ em seu pior caso, especialmente quando a escolha do pivô gera partições desbalanceadas. Para este experimento, adotou-se como pivô o elemento central do arranjo, uma heurística que torna o comportamento médio mais provável em alguns casos aleatórios, e pode ser ruim para outros.

As curvas apresentadas nas Figuras 1 e 2 demonstram esse comportamento, elas apresenta o tempo médio de execução (1) e o tempo médio com desvio padrão (2) do algoritmo aplicado aos diferentes conjuntos de dados: aleatórios, ordenados, decrescentes e parcialmente ordenados, com tamanhos variando de 100 a $2.000.000$ elementos.

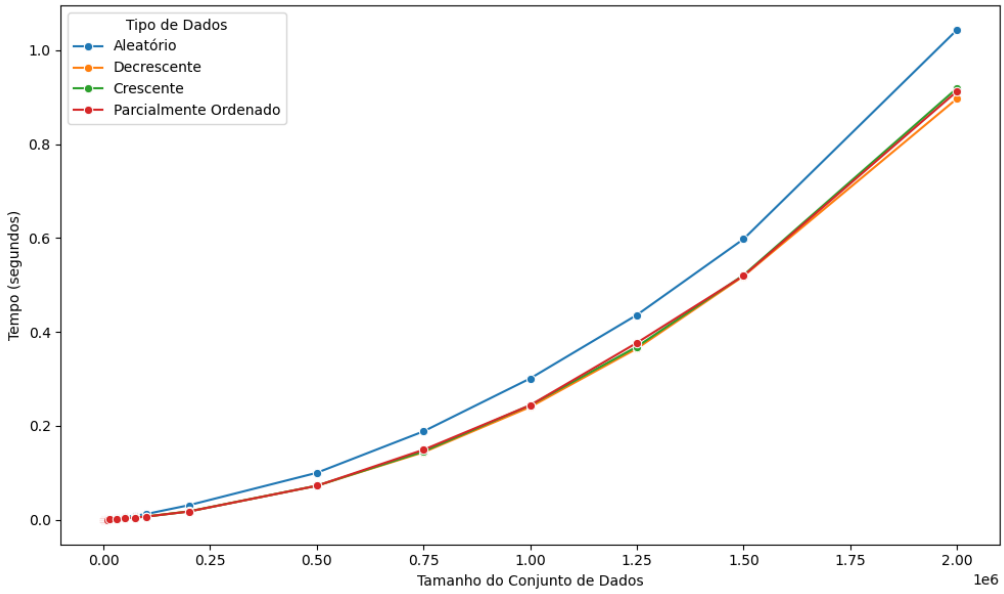

Figura 1. Tempo médio de execução do `QuickSort` para diferentes tipos de dados

Para dados aleatórios, o algoritmo mantém desempenho superior, como esperado. Contudo, diferentemente de implementações com pivô fixo, não se observa crescimento exponencial do tempo para dados ordenados ou decrescentes. Isso evidencia o impacto positivo da escolha do pivô central, que proporciona divisões mais equilibradas mesmo em entradas estruturadas.

Apesar disso, nota-se que os dados aleatórios ainda são processados mais rapidamente. Dados parcialmente ordenados também apresentam desempenho eficiente,

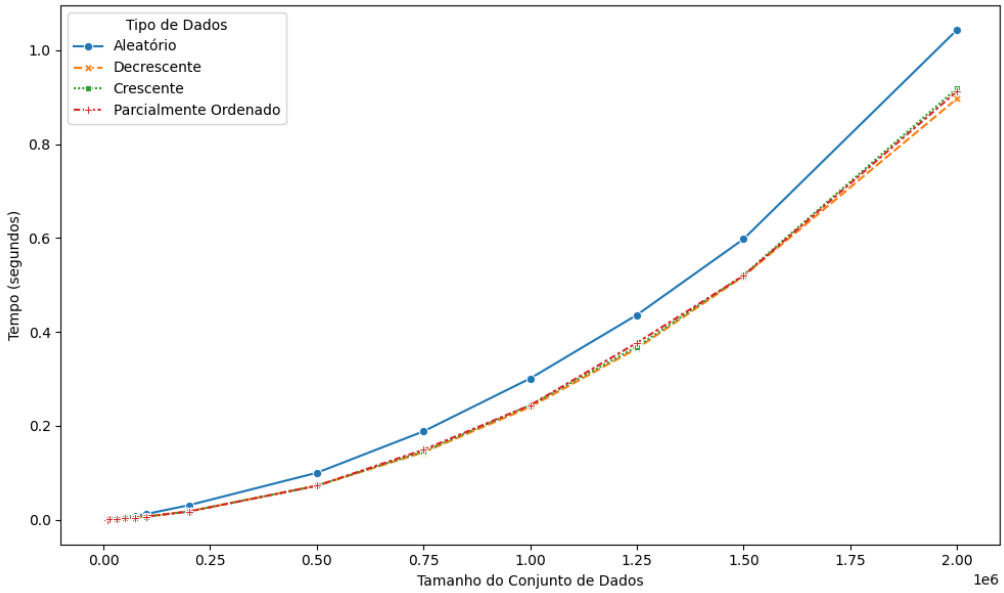

Figura 2. Tempo médio com desvio padrão do `QuickSort` considerando variações nos dados

com curvas similares às dos dados aleatórios, especialmente para tamanhos maiores. A diferença de comportamento mais relevante ocorre com os dados decrescentes, cujo tempo cresce de maneira mais acentuada, embora não exponencial.

Destaca-se uma inversão de desempenho entre os dados ordenados e parcialmente ordenados em torno de $n = 100{,}000$, sugerindo que, para conjuntos menores, a ordenação completa interfere menos na escolha do pivô. Já em volumes maiores, a aleatoriedade residual dos dados parcialmente ordenados favorece partições mais eficazes. A Figura 2 reforça essas observações ao incluir o desvio padrão das execuções. Dados aleatórios exibem baixa variabilidade, enquanto os decrescentes apresentam maior instabilidade, compatível com os desafios impostos à estratégia de particionamento mesmo com o pivô central.

Esses achados reiteram a importância da escolha adequada do pivô para o desempenho do `QuickSort`. Estratégias como a seleção do elemento central ou a mediana de três são recomendadas para evitar o comportamento quadrático do algoritmo em casos desfavoráveis, como defendido por [Cormen et al. 2009].

4.2. Análise do Desempenho do `MergeSort`

O MergeSort mantém desempenho estável em todos os casos, com complexidade $\mathcal{O}(n \log n)$, conforme já discutido na Seção 2. Os dados experimentais confirmam essa expectativa, especialmente pela regularidade das curvas para os dados ordenados, decrescentes e parcialmente ordenados, que crescem de forma bastante linear com relação ao tamanho da entrada. Isso demonstra que a ordem inicial dos dados não impacta significativamente o desempenho do algoritmo, evidenciando sua estabilidade.

As Figuras 3 e 4 apresentam o tempo médio de execução (3) e o tempo médio com desvio padrão (4) aplicado aos diferentes conjuntos de dados: aleatórios, ordenados, decrescentes e parcialmente ordenados, com tamanhos variando de 100 a $2.000.000$

elementos.

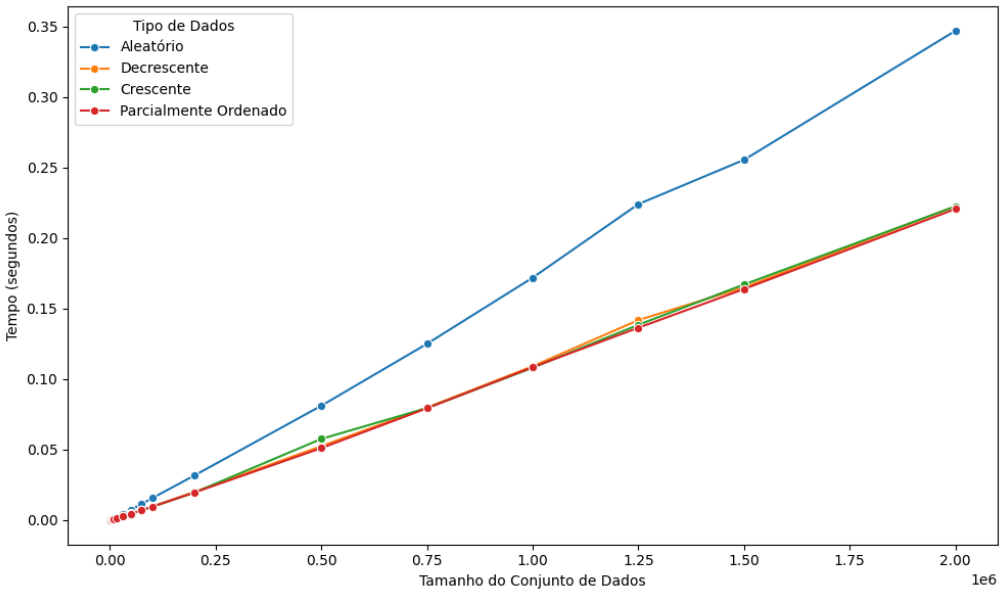

Figura 3. Tempo médio de execução do `MergeSort` para diferentes tipos de dados

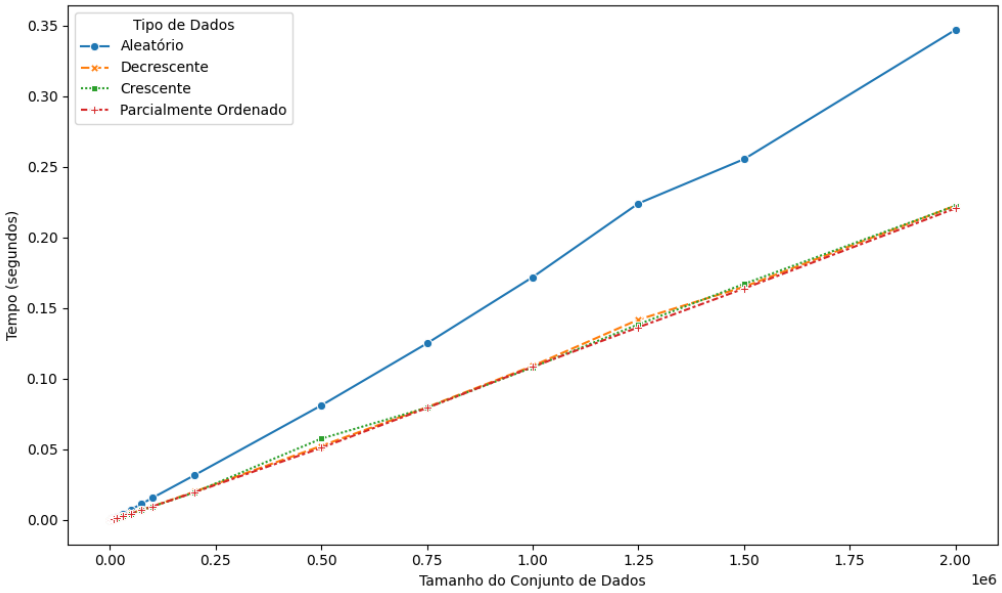

Figura 4. Tempo médio com desvio padrão do `MergeSort` considerando variações nos dados

A curva correspondente aos dados aleatórios apresenta tempo médio ligeiramente superior, mas ainda dentro do comportamento esperado para o algoritmo. Essa diferença pode ser atribuída à sobrecarga da manipulação de memória e à profundidade das chamadas recursivas em dados menos previsíveis. Ainda assim, a variação de tempo entre execuções permanece baixa, como indicado pelos pequenos desvios padrões na Figura 4, reforçando a previsibilidade do `MergeSort`.

Diferentemente do `QuickSort`, o `MergeSort` não depende da escolha de pivôs e realiza divisões fixas ao meio, evitando partições desbalanceadas. Assim, mesmo em entradas potencialmente desfavoráveis, como conjuntos decrescentes, o tempo de execução permanece estável. Isso o torna uma escolha adequada em contextos nos quais robustez e previsibilidade são essenciais, validando sua reputação como um dos algoritmos mais confiáveis para ordenação de grandes volumes de dados.

4.3. Análise do Desempenho do `HeapSort`

O comportamento do `HeapSort` ao longo dos experimentos revela um crescimento consistente do tempo de execução em função do tamanho das entradas, mantendo padrão compatível com sua complexidade assintótica teórica de $\mathcal{O}(n \log n)$ [Cormen et al. 2009]. As curvas para dados ordenados, parcialmente ordenados e decrescentes apresentam inclinações muito próximas, com tempos levemente inferiores ao caso aleatório, evidenciando que a ordenação prévia exerce influência limitada sobre o desempenho do algoritmo.

A exceção observada, como esperado, refere-se aos dados aleatórios, cujo tempo médio de execução foi significativamente superior ao dos demais tipos de entrada. Tal diferença decorre do maior número de operações de reconstrução do heap necessárias quando os dados não seguem qualquer padrão estrutural, o que impõe mais trabalho à fase de ordenação propriamente dita.

A Figura 5 ilustra o comportamento do algoritmo `HeapSort` frente a diferentes tipos de entrada, considerando o tempo médio de execução e a variabilidade estatística expressa pelo desvio padrão. A Figura 6 confirma a estabilidade do algoritmo, apresentando baixos desvios padrão em todas as execuções, mesmo em conjuntos com até dois milhões de elementos. Essa baixa variabilidade reforça a previsibilidade do `HeapSort`, característica fundamental em sistemas onde o tempo de execução precisa ser determinístico e pouco sensível à natureza da entrada.

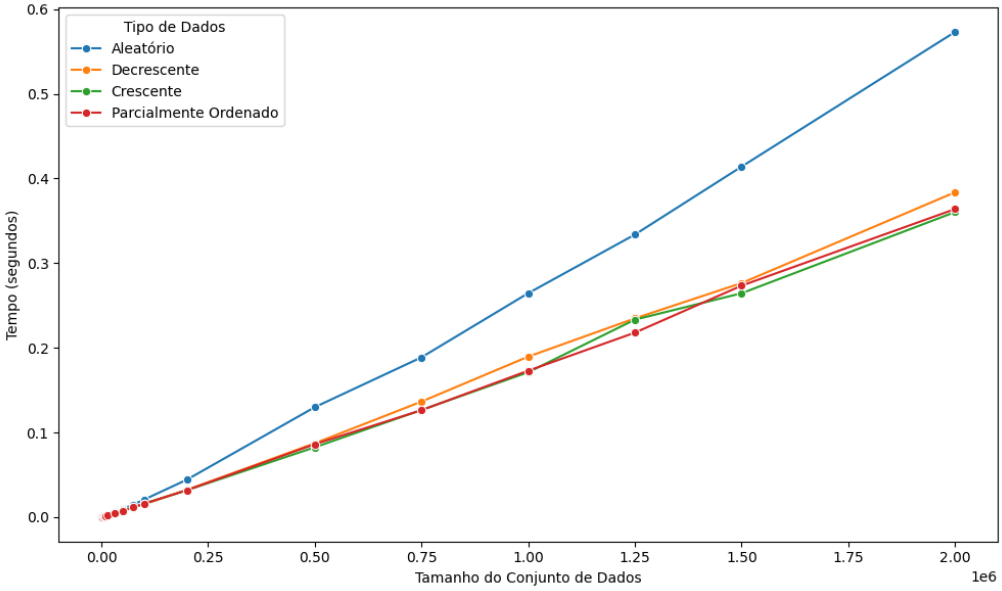

Figura 5. Tempo médio de execução do `HeapSort` para diferentes tipos de dados

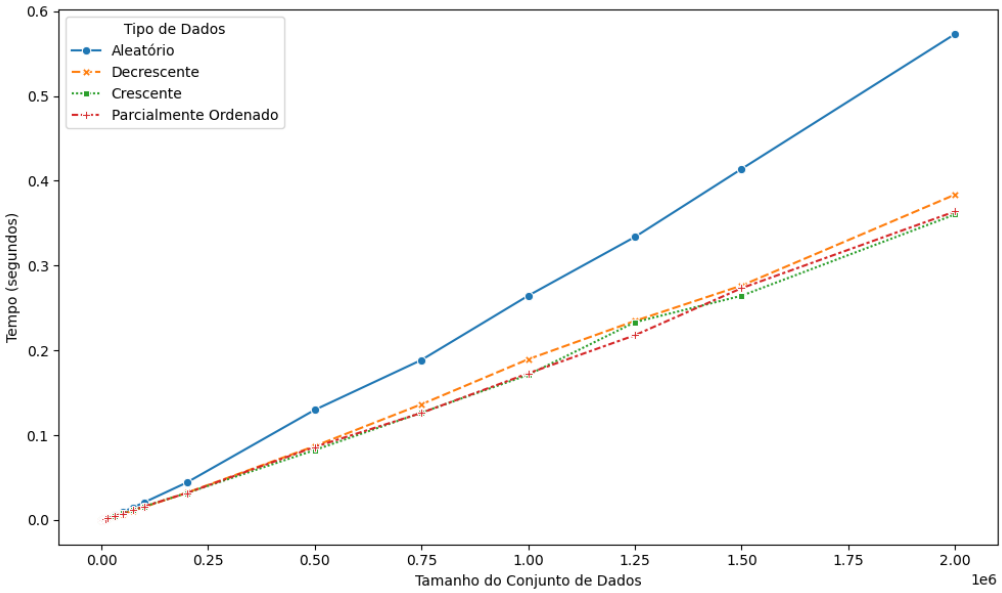

Figura 6. Tempo médio com desvio padrão do `HeapSort` considerando variações nos dados

Embora o `HeapSort` não tenha se destacado como o mais rápido entre os algoritmos testados, sua consistência, aliada à insensibilidade ao padrão de ordenação dos dados, o torna uma escolha segura em cenários críticos, especialmente quando a robustez é mais importante que o desempenho ótimo médio. Tais características o tornam útil em aplicações embarcadas, sistemas com restrições temporais rígidas e ambientes que exigem comportamento estável independentemente da entrada.

4.4. Análise Comparativa entre os Algoritmos

No cenário com dados aleatórios (Figura 7), o `QuickSort` inicialmente se destaca como o algoritmo mais eficiente, apresentando o menor tempo médio de execução até aproximadamente 750.000 elementos. No entanto, a partir desse ponto, observa-se uma clara troca de dominância: o tempo de execução do `QuickSort` passa a crescer mais rapidamente, sendo superado primeiramente pelo `HeapSort`. Esse comportamento evidencia a degradação progressiva do `QuickSort` em cenários de maior escala, mesmo com entradas aleatórias.

Embora o `QuickSort` tenha complexidade média esperada de $\mathcal{O}(n \log n)$, sua performance prática pode se aproximar de $\mathcal{O}(n^2)$ em partições desbalanceadas, especialmente quando a escolha do pivô é subótima. Por outro lado, tanto o `MergeSort` quanto o `HeapSort` mantêm um crescimento mais estável e condizente com o comportamento assintótico de $\mathcal{O}(n \log n)$, o que garante maior previsibilidade e eficiência em grandes volumes de dados. Assim, os resultados reforçam a importância de heurísticas de seleção de pivô para que o `QuickSort` mantenha sua eficiência teórica mesmo em situações práticas de escalabilidade.

Nos cenários com dados ordenados (Figura 8) e decrescentes (Figura 9), o `QuickSort` apresenta desempenho significativamente inferior, sendo consistentemente o algoritmo mais lento ao longo de todos os tamanhos de entrada. Esse comportamento

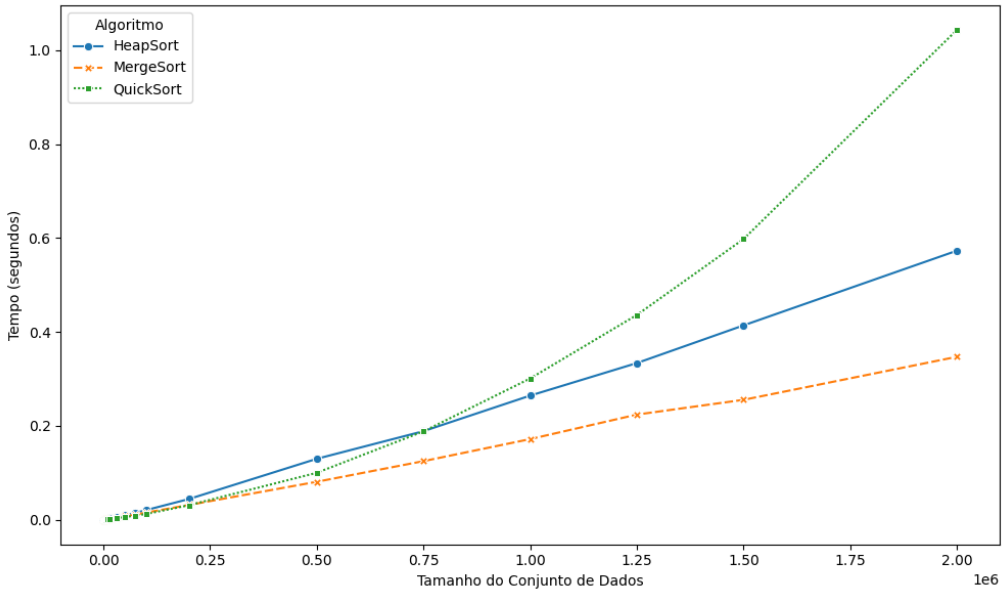

Figura 7. Tempo médio de execução dos algoritmos para dados aleatórios

é compatível com o seu pior caso teórico de complexidade $\mathcal{O}(n^2)$, que ocorre quando as partições geradas são altamente desbalanceadas, algo frequente quando o pivô escolhido é o primeiro, último ou central elemento em conjuntos já ordenados.

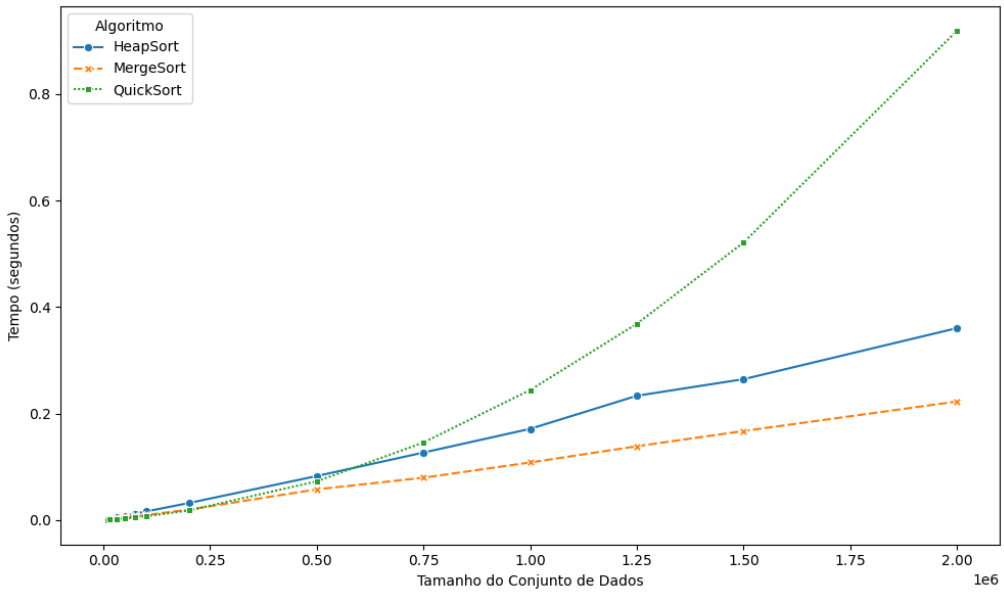

Figura 8. Tempo médio de execução dos algoritmos para dados ordenados

Enquanto isso, o `MergeSort` mantém sua curva de crescimento suave e previsível, condizente com sua complexidade garantida de $\mathcal{O}(n \log n)$, independentemente da ordenação dos dados. O `HeapSort`, embora também estável, apresenta tempo de execução ligeiramente superior ao `MergeSort`, o que pode ser atribuído ao custo adicional das operações de manutenção da estrutura de *heap*. Assim, os dados reforçam a vantagem dos algoritmos com complexidade estável em cenários de ordenação prévia.

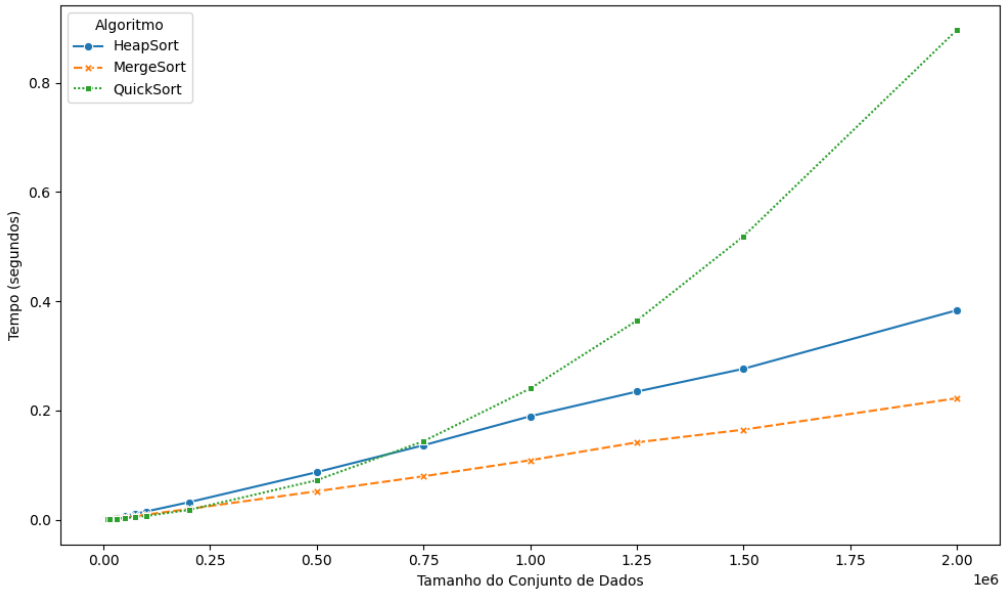

Figura 9. Tempo médio de execução dos algoritmos para dados decrescentes

A Figura 10, referente aos dados parcialmente ordenados, revela um comportamento intermediário entre os extremos totalmente ordenado e totalmente desordenado. O `QuickSort` apresenta desempenho melhor do que nos casos de pior cenário, mas ainda é superado pelo `HeapSort` e pelo `MergeSort` à medida que o tamanho da entrada cresce. Isso indica que a leve aleatoriedade presente nesses dados permite divisões um pouco mais equilibradas durante o particionamento, mitigando parcialmente o comportamento de pior caso ($\mathcal{O}(n^2)$), embora sem alcançar a eficiência esperada do caso médio ($\mathcal{O}(n \log n)$).

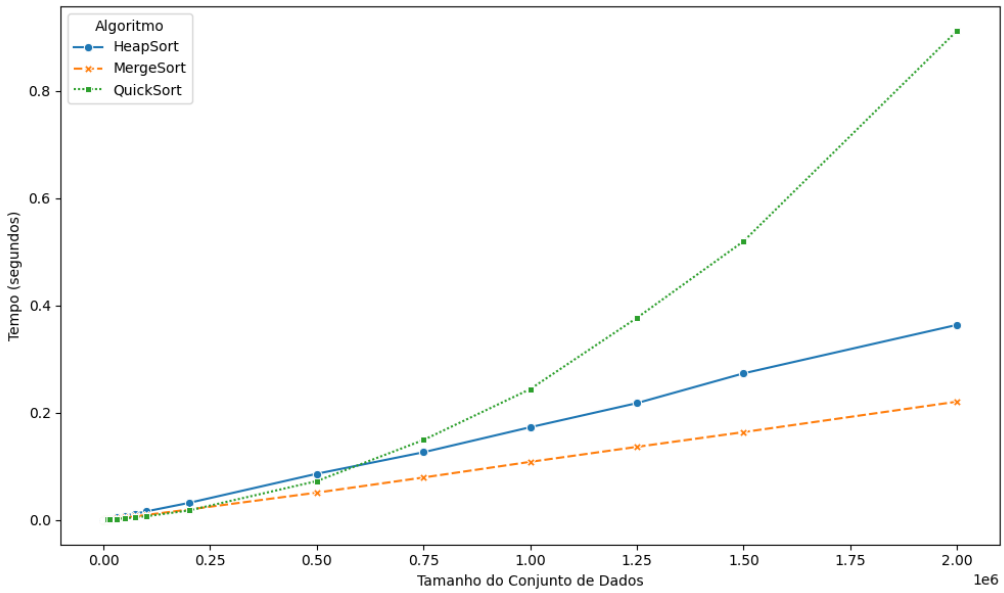

Figura 10. Tempo médio de execução dos algoritmos para dados parcialmente ordenados

O `MergeSort`, por sua vez, mantém sua regularidade e desempenho superior em todos os tamanhos testados, alinhado à sua complexidade garantida de $\mathcal{O}(n \log n)$, com partições sempre simétricas, independentemente da ordenação dos dados. O `HeapSort` também apresenta crescimento estável e previsível, porém ligeiramente acima do `MergeSort`, o que pode ser atribuído ao custo das operações de manutenção do heap.

Em síntese, o `MergeSort` se destaca como o mais eficiente nos diferentes cenários, enquanto o `QuickSort` demonstra alta variabilidade de desempenho, sensível à ordenação dos dados e à escolha do pivô. O `HeapSort`, embora um pouco mais lento, mostra-se estável frente à diversidade dos conjuntos analisados.

5. Conclusão

Este estudo apresentou uma análise teórica e experimental dos algoritmos de ordenação `QuickSort`, `MergeSort` e `HeapSort`, com foco em seu desempenho frente a diferentes padrões de entrada: aleatórios, ordenados, parcialmente ordenados e decrescentes. Os experimentos foram conduzidos em ambiente controlado, com vetores variando de 100 a 2 milhões de elementos, mensurando tempo de execução, número de comparações e uso de memória.

Os resultados empíricos evidenciaram diferenças relevantes no comportamento dos algoritmos. O `QuickSort` mostrou excelente desempenho em entradas aleatórias de pequeno e médio porte, mas revelou alta sensibilidade à ordenação dos dados. Mesmo com a adoção de pivô central, observou-se degradação de desempenho em entradas ordenadas ou decrescentes, refletida tanto no tempo de execução quanto na variabilidade entre execuções. Esse comportamento reforça a dependência do algoritmo à estratégia de particionamento.

O `MergeSort` destacou-se pela estabilidade e previsibilidade, mantendo tempo de execução uniforme independentemente do padrão da entrada. Sua estrutura de divisão fixa e uso de memória auxiliar asseguraram desempenho consistente, especialmente em conjuntos de grande volume ou estrutura adversa. Esse comportamento o consolidou como o mais eficiente nos experimentos, sendo indicado para aplicações em que previsibilidade de desempenho é requisito crítico.

O `HeapSort`, por sua vez, apresentou desempenho intermediário. Embora ligeiramente mais lento que o `MergeSort`, demonstrou comportamento estável e insensível à ordenação inicial, com baixos desvios padrão e uso restrito de memória auxiliar. Tais características o tornam adequado para sistemas com restrições de alocação dinâmica ou requisitos de previsibilidade sob diferentes padrões de entrada.

De forma geral, o estudo reforça que a escolha de um algoritmo de ordenação deve considerar sua complexidade assintótica e aspectos empíricos como a distribuição dos dados, o volume processado, a estabilidade entre execuções e as características do ambiente de execução. Não há algoritmo universalmente superior: a decisão ideal depende do contexto, da estrutura dos dados e das restrições da aplicação. Como trabalhos futuros, sugere-se a ampliação do experimento para incluir a análise do impacto da arquitetura de hardware (cache, paralelismo, otimizações de compilador) e de variações na implementação (iterativa *vs.* recursiva). Também seria relevante explorar métricas adicionais como consumo de memória e outras métricas para fins de comparação.

Referências

Backes, A. R. (2023). *Algoritmos e Estruturas de Dados em Linguagem C*. LTC, Rio de Janeiro. E-book. p.29. ISBN 9788521638315. Disponível em: `https://app.minhabiblioteca.com.br/reader/books/9788521638315/`. Acesso em: 03 mai. 2025.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3 edition.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2012). *Algoritmos*. Elsevier, Rio de Janeiro. Tradução de: Introduction to Algorithms, 3rd ed. Tradução de Arlete Simille Marques.

Mohammadagha, M. (2025). Hybridization and optimization modeling, analysis, and comparative study of sorting algorithms: Adaptive techniques, parallelization, for mergesort, heapsort, quicksort, insertion sort, selection sort, and bubble sort.

Wild, S., Nebel, M., and Neininger, R. (2013). Average case and distributional analysis of dual-pivot quicksort. *URL http://arxiv. org/abs/1304.0988*.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer.