# Trabalho Prático I
# Estrutura de Dados II

**Gabriel Silva Xavier**[1], **Richard Rodrigues**[1], **Vitor Henrique**[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia do Sul de Minas Gerais (IFSULDEMINAS)
Contato (35) 3526-4856 – Passos – MG – Brazil

{gabriel.xavier, richard.rodrigues, vitor.almeida}@alunos.ifsuldeminas.edu.br

***Abstract.*** *Sorting algorithms are essential in various fields for organizing datasets. This study analyzes the performance of MergeSort, QuickSort, and HeapSort with different data types and sizes, also exploring the combination of MergeSort with InsertionSort. Additionally, a new algorithm, RecicleSort, is proposed. The algorithms were implemented to sort random values in different structures, and key metrics were recorded for analysis.*

***Resumo.*** *Os algoritmos de ordenação são fundamentais em diversas áreas para organizar conjuntos de dados. Este trabalho analisa o desempenho dos algoritmos MergeSort, QuickSort e HeapSort com diferentes tipos e tamanhos de dados, explorando também a integração do MergeSort com o InsertionSort. Além disso, é proposto o novo algoritmo RecicleSort. Os algoritmos foram implementados para ordenar valores aleatórios em diferentes estruturas, com métricas registradas para análise posterior.*

## 1. Introdução

Ordenar informações de forma eficiente é crucial em diversas áreas, como na estatística, especialmente para o cálculo de medidas de posição [MORETTING et al. 2010]. Nesse contexto, os algoritmos de ordenação são essenciais para organizar dados logicamente, especialmente em dispositivos computacionais.

Esses algoritmos se dividem entre básicos e sofisticados. Os básicos utilizam estruturas iterativas para ordenar os dados sequencialmente, enquanto os sofisticados empregam técnicas avançadas, como o paradigma de dividir para conquistar ou o uso de estruturas específicas, como o Heap Máximo. Embora os algoritmos básicos sejam mais fáceis de entender e implementar, seu custo computacional pode ser O($n^2$) no pior caso. Já os sofisticados, embora mais complexos, têm desempenho de O($n \log n$), sendo mais eficientes para grandes volumes de dados. Em alguns casos, a combinação de abordagens pode gerar melhores resultados, conforme as características do problema.

Assim sendo, este trabalho tem como objetivo analisar o desempenho dos algoritmos de ordenação QuickSort, MergeSort, HeapSort, uma abordagem híbrida com MergeSort e InsertionSort, e um novo algoritmo proposto, o RecicleSort. A análise será realizada a partir de três diferentes cenários, permitindo uma avaliação detalhada do comportamento de cada algoritmo com diferentes tamanhos e tipos de dados.

## 2. Fundamentação Teórica

### 2.1. InsertionSort

O InsertionSort organiza os dados de forma incremental, inserindo cada elemento ainda não ordenado na posição correta dentro da sequência, de forma que quando o último elemento for colocado na sua posição correta e todos os outros deslocados para suas posições, o conjunto estará ordenado. Sua complexidade é O($n^2$) no pior caso, o que o torna ineficiente para grandes volumes de dados.

### 2.2. MergeSort

O MergeSort utiliza a abordagem ”dividir para conquistar”. O conjunto de dados é dividido em sub-conjuntos cada vez menores, até que cada um contenha um único elemento (que já está ordenado). Em seguida, cada um desses sub-conjuntos vão sendo intercalados (”mesclados”) uns com os outros, momento em que é feita a ordenaçao. Com complexidade O($n \log n$), é consideravelmente eficiente para grandes volumes de dados e oferecendo um desempenho previsível. Contudo, requer espaço adicional O($n$) para armazenar as sublistas temporárias durante a mesclagem.

### 2.3. QuickSort

O QuickSort é um dos mais eficientes algoritmos de ordenação; ele seleciona um pivô e organiza os dados para que todos os elementos menores que o pivô fiquem à sua esquerda, e os maiores à sua direita. Esse processo é repetido recursivamente para as sublistas geradas. Com complexidade média de O($n \log n$), é o algoritmo considerado o mais rápido na literatura. No entanto, no pior caso, pode atingir O($n^2$), como quando o pivô é sempre o maior ou menor elemento. Neste trabalho, foi utilizado como pivo o primeiro valor de cada vetor ou lista, considerados à cada chamada recursiva.

### 2.4. HeapSort

O HeapSort é um algoritmo de ordenação baseado na estrutura Heap Máximo, em que cada nó é maior que os valores de seus filhos. No HeapSort, constrói-se inicialmente um heap máximo a partir dos dados. Em seguida, o maior elemento, que fica na raiz, é trocado com o último elemento da sequência, e o heap é sempre ajustado para manter sua propriedade. O espaço de ordenação é reduzido em um a cada vez. Com complexidade O($n \log n$), demonstra ser bastante eficiente. Contudo, como é utilizada uma árvore binária, o HeapSort não é indicado para ordenar conjuntos com muitos valores repetidos. Neste trabalho foi implementado o Heap com uma abordagem não recursiva.

### 2.5. RecicleSort

O RecicleSort é um algoritmo de ordenação inspirado na ideia da coleta seletiva. Antes de ordenar os elementos, ele percorre o vetor, segmentando-os conforme suas casas numéricas (unidade, dezena, centena, etc.), e alocando-os em vetores distintos. Durante esse processo de coleta e distribuição, cada vetor é progressivamente ordenado à medida que novos elementos são incluídos. Para a ordenação, o algoritmo adota uma versão aprimorada do QuickSort recursivo, na qual o pivô é selecionado com base na mediana dos valores, o que reduz o tempo de execução, onde em determinadas situações, torna o algoritmo mais eficiente do que outras abordagens tradicionais. [Ziviani et al. 2004]

## 3. Materiais e Métodos

Este estudo avaliou o desempenho de algoritmos de ordenação em três cenários distintos. Para cada cenário e seus sub-itens, para cada N valores foram realizados 5 testes independentes, e a média dos tempos de execução (em milissegundos) foi calculada, sendo esta considerada na análise. Para o cenário 1 também foi contabilizada a média do número de comparações de chaves.

Os testes foram realizados com diferentes estruturas de dados, sendo que os valores considerados eram sempre gerados aleatoriamente (na faixa de 0 à 999). Ademais, cada algoritmo foi executado em uma estrutura de testes padronizada, utilizando de maneira integral a linguagem C, para assegurar a comparabilidade dos resultados. Para a contabilização dos tempos, foi utilizada a função 'clock()'.

### 3.1. Cenários de Teste

- **Cenário 1:** Ordenação com o QuickSort de diferentes estruturas de dados, que foram executados em um hardware com as seguintes especificações: Processador: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz; RAM: 8GB DDR 4; Sistema Operacional: Windows 11 Home Single Language; Armazenamento: HD de 2 TB.
- **Cenário 2:** Analise do desempenho geral e a comparação dos tempos médios das variações do MergeSort foram analisados em dois diferentes conjuntos de hardware. Para os dois primeiros itens, os testes foram realizados em uma máquina com processador AMD Ryzen 5 Mobile 3500U CPU @ 2.1GHz, 6 GB de memória RAM DDR4, sistema operacional Windows 11 Home Single Language e armazenamento em HD de 1TB. Já para os dois últimos itens, os testes foram executados em um sistema com processador Intel(R) Core(TM) i3-7020U CPU @ 2.30GHz, 8 GB de memória RAM DDR4, sistema operacional Arch Linux (versão 6.11.1-arch1-1) e HD de 1TB.
- **Cenário 3:** Realizada a comparação do desempenho entre o tradicional HeapSort e o algoritmo proposto RecicleSort foi realizada em uma máquina com processador Intel(R) Core(TM) i3-7020U CPU @ 2.30GHz, 8 GB de memória RAM DDR4, sistema operacional Arch Linux (versão 6.11.1-arch1-1) e armazenamento em HD de 1TB.

## 4. Experimentos Computacionais

Os resultados dos experimentos realizados estão apresentados nas tabelas a seguir, com foco no tempo de execução dos algoritmos de ordenação em diferentes cenários e hardwares. A análise comparativa visa avaliar o desempenho de cada algoritmo de acordo com os diferentes tamanhos de vetores e as condições de teste. É importante ressaltar que devido ao espaço, são apresentados apenas os resultados obtidos à partir do cálculo da média de 5 valores obtidos para cada N.

### 4.1. Cenário (1)

Os resultados dos experimentos realizados, que envolvem a execução do algoritmo QuickSort para a ordenação de números em um vetor de inteiros, em uma lista duplamente encadeada e em um vetor de structs, estão apresentados na tabela a seguir. Nela estão as médias dos tempos e dos números de comparações de chaves.

| Cenário 1 – Quick Sort e Estruturas de Dados | | | | | |
|---|---|---|---|---|---|
| **Tamanho (N)** | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| Comp vetor inteiros | 11704,2 | 193315,4 | 6465014,4 | 514675973,2 | 2908231941 |
| Temp vetor inteiros | 0 | 1,2 | 19,8 | 1122,8 | 115804 |
| Comp lista | 11821 | 197631 | 6606091,4 | 515639263,6 | 2911073661 |
| Temp lista | 0 | 0 | 26,4 | 1549,2 | 197403,2 |
| Comp vetor structs | 11830 | 191263,6 | 6434994,4 | 514728100,8 | _ |
| Temp vetor structs | 0 | 6,4 | 153,2 | 3340,2 | _ |

**Tabela 1. Resultados do Cenário 1: QuickSort em diferentes estruturas**

### 4.2. Cenário (2)

Devido às limitações especificadas para o tamanho do documento, optou-se por apresentar a média dos tempos de execução das quatro implementações realizadas no Cenário 2. Essa escolha permitiu demonstrar os resultados de forma objetiva e representativa, assegurando a comparabilidade entre os testes realizados.

| Cenário 2 – Media e Comparação entre as implementações | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Media final para (N)** | 1000 | 5000 | 10000 | 50000 | 100000 | 500000 | 1000000 |
| Merge Recursivo | 0,6 | 2,6 | 7,6 | 35,4 | 69,6 | 253,8 | 488,8 |
| Merge Iterativo | 1 | 4,4 | 26,7 | 31,8 | 64,6 | 241 | 477 |
| M-Insertion (M = 10) | 0 | 1,2 | 3 | 13 | 25,2 | 120,6 | 243,6 |
| M-Insertion (M=100) | 0 | 2,6 | 2,4 | 11,2 | 38 | 115 | 236,4 |
| Multiway (k = 5) | 0,6 | 1,8 | 4,2 | 23,8 | 39,8 | 178,6 | 388 |
| Multiway (k=100) | 0,2 | 2 | 5,8 | 23,6 | 36 | 212,2 | 358 |

**Tabela 2. Resultados do Cenário 3: HeapSort para diferentes tamanhos.**

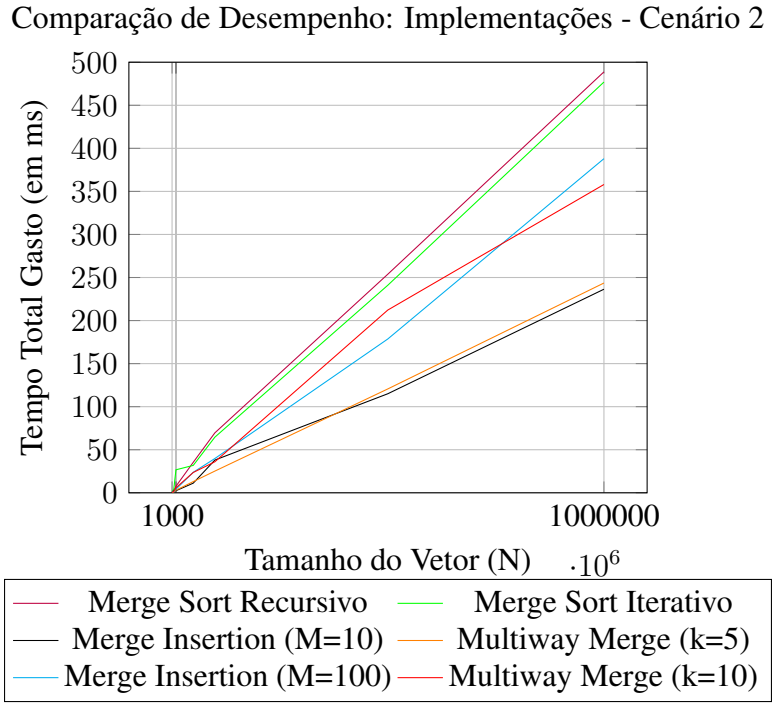


**Figura 1. Gráfico de Comparação de Comparação: Cenário 2**

### 4.3. Cenário (3)

Análise Comparativa de Desempenho entre os Algoritmos HeapSort e RecicleSort

| **Cenário 3 – Heap Sort: Vetor de Números Inteiros** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Tamanho do vetor (N)** | 1000 | 5000 | 10000 | 50000 | 100000 | 500000 | 1000000 |
| Temp 1 | 0 | 2 | 2 | 26 | 29 | 159 | 326 |
| Temp 2 | 0 | 1 | 2 | 13 | 30 | 159 | 326 |
| Temp 3 | 0 | 1 | 3 | 13 | 29 | 157 | 326 |
| Temp 4 | 0 | 2 | 7 | 15 | 29 | 161 | 326 |
| Temp 5 | 0 | 2 | 6 | 13 | 29 | 159 | 329 |
| **Tempo total gasto (em ms)** | | | | | | | |
| Média Temp | 0 | 1,6 | 4 | 16 | 29,2 | 159 | 326,6 |

**Tabela 3. Resultados do Cenário 3: HeapSort para diferentes tamanhos.**

| Cenário 3 – Aluno Sort: Vetor de Números Inteiros | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Tamanho do vetor (N)** | 1000 | 5000 | 10000 | 50000 | 100000 | 500000 | 1000000 |
| Temp 1 | 0 | 0 | 3 | 9 | 19 | 84 | 172 |
| Temp 2 | 0 | 1 | 5 | 23 | 16 | 86 | 170 |
| Temp 3 | 0 | 1 | 1 | 8 | 18 | 86 | 171 |
| Temp 4 | 0 | 2 | 5 | 8 | 17 | 84 | 171 |
| Temp 5 | 0 | 1 | 5 | 23 | 16 | 84 | 169 |
| **Tempo total gasto (em ms)** | | | | | | | |
| Média Temp | 0 | 1 | 3,8 | 14,2 | 17,2 | 84,8 | 170,6 |

**Tabela 4. Resultados do Cenário 3: RecicleSort para diferentes tamanhos.**

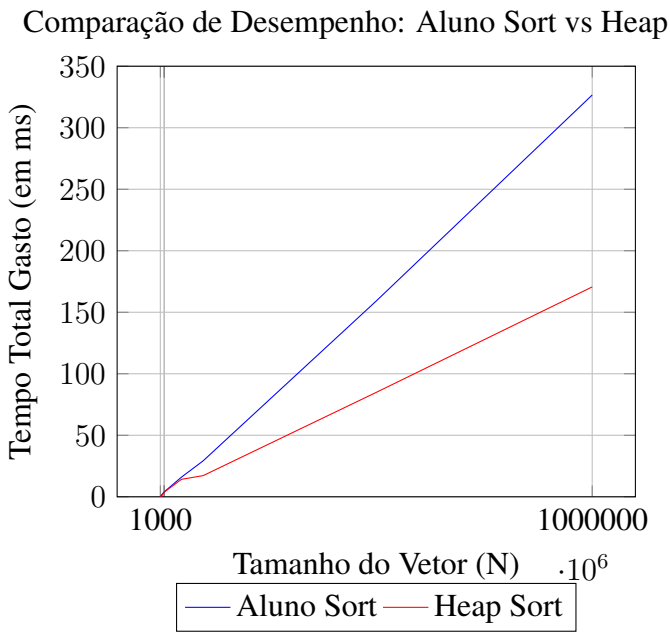


**Figura 2. Gráfico de Comparação de Desempenho: Aluno Sort vs Heap Sort**

## 5. Conclusão

Durante a execução deste trabalho, o conhecimento adquirido pelos integrantes do grupo ao longo do desenvolvimento foi de grande valia, uma vez que possibilitou um entendimento mais concreto sobre os algoritmos abordados. No entanto, alguns desafios foram enfrentados, tais como: a adaptação dos algoritmos para trabalhar com diferentes tipos de dados, a implementação de funcionalidades como o cálculo de tempo e o registro em arquivos, além da elaboração da documentação utilizando a ferramenta LaTeX, a qual era desconhecida por todos os membros do grupo, obrigando-nos a aprender seu uso do zero.

Em relação aos resultados obtidos, no cenário 1, foi evidente a superioridade do

algoritmo QuickSort na ordenação de dados em vetores, com seu pior desempenho registrado ao trabalhar com vetores de structs, não sendo possível sua execução para conjuntos de dados de 10.000.000 de elementos. No cenário 2, após a análise das métricas e o cálculo da média do tempo de execução de cada algoritmo, observou-se um desempenho ligeiramente superior do algoritmo Merge Insertion (M=10) em comparação com o Multiway Merge (k=5), o qual, apesar de um desempenho um pouco inferior, mostrou-se bastante eficaz em relação às outras implementações.

No cenário 3, observou-se uma diferença significativa entre o algoritmo RecicleSort e o HeapSort tradicional. Essa diferença pode ser atribuída a melhorias implementadas, especialmente no ajuste do QuickSort recursivo. Durante o processo, a separação inicial dos itens em diferentes casas numéricas não apenas organizou os elementos, mas também permitiu que cada vetor fosse automaticamente ordenado ao mesmo tempo. Assim, o QuickSort recursivo foi aplicado de forma mais eficiente, resultando em uma otimização no tempo de execução.

## Referências

MORETTING, P. et al. (2010). Estatística básica.

Ziviani, N. et al. (2004). *Projeto de algoritmos: com implementações em Pascal e C*, volume 2. Thomson Luton.