

UNIVERSIDADE FEDERAL DE SANTA MARIA  
CENTRO DE TECNOLOGIA  
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

João Vitor da Silva, João Pedro Azenha Righi e Arthur Bogacki  
Verissimo

**TRABALHO 1: ATIVIDADES DE IMPLEMENTAÇÃO COM O  
MERGESORT**

Santa Maria, RS  
2024

## **RESUMO**

### **TRABALHO 1: ATIVIDADES DE IMPLEMENTAÇÃO COM O MERGESORT**

**AUTORES:** João Vitor da Silva, João Pedro Azenha Righi e Arthur Bogacki Verissimo

**PROFESSORA:** Juliana Kaizer Vizzotto

Este relatório aborda a implementação e análise do algoritmo de ordenação Mergesort. O objetivo principal é aprofundar o conhecimento sobre a divisão de dados e a fusão de subarrays, além de comparar o desempenho de diferentes algoritmos de ordenação, como Quicksort, Bubblesort e Heapsort. Também foi desenvolvida uma visualização gráfica da árvore de recursão do Mergesort, com o intuito de ilustrar o processo de divisão e conquista. Ademais, explorou-se uma versão paralela do Mergesort para discutir os impactos da paralelização na eficiência do algoritmo. O relatório conclui com uma análise da complexidade teórica e prática do Mergesort, além de uma comparação entre as abordagens recursiva e iterativa, levando em consideração o desempenho e o consumo de memória.

**Palavras-chave:** Algoritmos de Ordenação. MergeSort. Complexidade. Desempenho de Algoritmos. Comparação de Algoritmos

## SUMÁRIO

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>INTRODUÇÃO .....</b>  | <b>4</b>  |
| <b>2</b> | <b>IMPLEMENTAÇÃO DO MERGESORT .....</b>                        | <b>5</b>  |
| 2.1      | DESCRIÇÃO DA IMPLEMENTAÇÃO .....                               | 5         |
| <b>3</b> | <b>COMPARAÇÃO DE ALGORITMOS DE ORDENAÇÃO .....</b>             | <b>6</b>  |
| 3.1      | IMPLEMENTAÇÃO E DADOS INICIAIS .....                           | 6         |
| 3.2      | BUBBLE SORT .....  | 6         |
| 3.3      | MERGE SORT .....   | 8         |
| 3.4      | ANÁLISE BUBBLE X MERGE .....                                   | 9         |
| 3.5      | TESTE DE ITERAÇÕES .....                                       | 9         |
| 3.6      | CONCLUSÃO .....  | 10        |
| <b>4</b> | <b>VISUALIZAÇÃO DA ÁRVORE DE RECURSÃO .....</b>                | <b>11</b> |
| <b>5</b> | <b>MERGESORT PARALELO .....</b>                                | <b>12</b> |
| 5.1      | PARALELIZAÇÃO DO MERGESORT .....                               | 12        |
| 5.2      | VANTAGENS E LIMITAÇÕES DA PARALELIZAÇÃO .....                  | 12        |
| 5.3      | IMPLEMENTAÇÃO PARALELA DO MERGESORT .....                      | 12        |
| <b>6</b> | <b>ANÁLISE DE COMPLEXIDADE .....</b>                           | <b>13</b> |
| 6.1      | COMPARAÇÃO TEÓRICA X PRÁTICA .....                             | 13        |
| 6.2      | CONCLUSÃO .....  | 13        |
| <b>7</b> | <b>COMPARAÇÃO DA IMPLEMENTAÇÃO RECURSIVA X ITERATIVA .....</b> | <b>15</b> |
| 7.1      | DESEMPENHO (TEMPO DE EXECUÇÃO) .....                           | 15        |
| 7.2      | CLAREZA E ELEGÂNCIA .....                                      | 15        |
| 7.3      | USO DE MEMÓRIA .....   | 15        |
| 7.4      | CONCLUSÃO .....  | 16        |

## 1 INTRODUÇÃO

Este relatório apresenta um estudo prático sobre a implementação do algoritmo de ordenação Mergesort e suas variações. O objetivo deste trabalho é consolidar o conhecimento teórico sobre algoritmos de ordenação, aplicando-os na prática para uma compreensão mais aprofundada dos desafios e soluções associadas ao processo de divisão e fusão de dados.

O Mergesort foi escolhido por sua relevância em diferentes cenários, sendo um algoritmo de ordenação eficiente com complexidade garantida em  $O(n \log n)$ . Além de sua implementação clássica, este trabalho explora também versões recursivas, iterativas e paralelas do Mergesort, oferecendo uma visão completa das diferentes abordagens e suas implicações em termos de desempenho.

Este relatório descreve em detalhes a implementação do algoritmo, a estrutura do código e os resultados obtidos. Também são apresentados gráficos e análises comparativas entre o Mergesort e outros algoritmos, como Quicksort, Bubblesort e Heapsort, para ilustrar suas vantagens e desvantagens em diferentes cenários de ordenação.

## 2 IMPLEMENTAÇÃO DO MERGESORT

- Descrição: implementar o algoritmo de mergesort do zero, em uma linguagem de programação à escolha.
- Objetivo: Compreender a divisão de dados e a fusão de subarrays.
- Adicionar comentários explicativos para cada parte do código, detalhando a lógica de divisão e fusão.

### 2.1 DESCRIÇÃO DA IMPLEMENTAÇÃO

O algoritmo Mergesort foi implementado em Python no diretório `trabPAA/ex1/mergesort.py`. Sua função recursiva recebe um array como entrada e o divide repetidamente em duas subpartes, até que as subdivisões contenham apenas um elemento, momento em que são consideradas ordenadas. Em seguida, os subarrays são recombinaados, comparando seus elementos e reordenando-os conforme são fundidos para formar um array ordenado. Esse processo de divisão e fusão ocorre de maneira recursiva, garantindo que, ao final, todos os elementos estejam devidamente ordenados no array original.

### 3 COMPARAÇÃO DE ALGORITMOS DE ORDENAÇÃO

- Descrição: Implementar outros algoritmos de ordenação (como quicksort, bubble-sort e heapsort) e comparar o desempenho deles com o mergesort em diferentes tamanhos de dados.
- Objetivo: Entender as vantagens e desvantagens de diferentes algoritmos de ordenação.
- Dicas: Crie um conjunto de dados variado para testar (melhor caso, pior caso e caso médio)

#### 3.1 IMPLEMENTAÇÃO E DADOS INICIAIS

Após a implementação de ambos algoritmos, Merge Sort e Bubble Sort, foi feita a análise do comportamento dos algoritmos em diferentes casos.

Dado o array:

```
[11, 12, 22, 25, 34, 64, 90]
```

Usamos como melhor caso:

```
array melhor = [11, 12, 22, 25, 34, 64, 90]
```

Usamos como caso médio:

```
array médio = [11, 12, 22, 90, 34, 25, 64]
```

Usamos como pior caso:

```
array pior = [90, 64, 34, 25, 22, 12, 11]
```

#### 3.2 BUBBLE SORT

No melhor caso do algoritmo bubble sort tivemos como resultado:

```

Melhor caso
Array inicial: [11, 12, 22, 25, 34, 64, 90]
Tempo de execução: 0.0 segundos

```

Figura 3.1 – Melhor caso bubble sort

No caso médio do algoritmo bubble sort tivemos como resultado:

```

● Caso medio
Array inicial: [11, 12, 22, 90, 34, 25, 64]
Iteração 1: [11, 12, 22, 34, 90, 25, 64]
Iteração 2: [11, 12, 22, 34, 25, 90, 64]
Iteração 3: [11, 12, 22, 34, 25, 64, 90]
Iteração 4: [11, 12, 22, 25, 34, 64, 90]
Tempo de execução: 0.0 segundos

```

Figura 3.2 – Caso médio bubble sort

No pior caso do algoritmo bubble sort tivemos como resultado:

```

Pior Caso
Array inicial: [90, 64, 34, 25, 22, 12, 11]
Iteração 1: [64, 90, 34, 25, 22, 12, 11]
Iteração 2: [64, 34, 90, 25, 22, 12, 11]
Iteração 3: [64, 34, 25, 90, 22, 12, 11]
Iteração 4: [64, 34, 25, 22, 90, 12, 11]
Iteração 5: [64, 34, 25, 22, 12, 90, 11]
Iteração 6: [64, 34, 25, 22, 12, 11, 90]
Iteração 7: [34, 64, 25, 22, 12, 11, 90]
Iteração 8: [34, 25, 64, 22, 12, 11, 90]
Iteração 9: [34, 25, 22, 64, 12, 11, 90]
Iteração 10: [34, 25, 22, 12, 64, 11, 90]
Iteração 11: [34, 25, 22, 12, 11, 64, 90]
Iteração 12: [25, 34, 22, 12, 11, 64, 90]
Iteração 13: [25, 22, 34, 12, 11, 64, 90]
Iteração 14: [25, 22, 12, 34, 11, 64, 90]
Iteração 15: [25, 22, 12, 11, 34, 64, 90]
Iteração 16: [22, 25, 12, 11, 34, 64, 90]
Iteração 17: [22, 12, 25, 11, 34, 64, 90]
Iteração 18: [22, 12, 11, 25, 34, 64, 90]
Iteração 19: [12, 22, 11, 25, 34, 64, 90]
Iteração 20: [12, 11, 22, 25, 34, 64, 90]
Iteração 21: [11, 12, 22, 25, 34, 64, 90]
Tempo de execução: 0.005764007568359375 segundos

```

Figura 3.3 – Pior caso bubble sort

### 3.3 MERGE SORT

No melhor caso do algoritmo merge sort tivemos como resultado:

```
Melhor caso
Array original: [11, 12, 22, 25, 34, 64, 90]
Iteração 1: [12, 22]
Iteração 2: [11, 12, 22]
Iteração 3: [25, 34]
Iteração 4: [64, 90]
Iteração 5: [25, 34, 64, 90]
Iteração 6: [11, 12, 22, 25, 34, 64, 90]
Total de iterações: 6
```

Figura 3.4 – Melhor caso merge sort

No caso médio do algoritmo merge sort tivemos como resultado:

```
Caso Medio
Array original: [11, 12, 22, 90, 34, 25, 64]
Iteração 1: [12, 22]
Iteração 2: [11, 12, 22]
Iteração 3: [34, 90]
Iteração 4: [25, 64]
Iteração 5: [25, 34, 64, 90]
Iteração 6: [11, 12, 22, 25, 34, 64, 90]
Tempo de execução: 0.008049249649047852 segundos
```

Figura 3.5 – Caso médio merge sort

No pior caso do algoritmo merge sort tivemos como resultado:

```
Pior Caso
Array original: [90, 64, 34, 25, 22, 12, 11]
Iteração 1: [34, 64]
Iteração 2: [34, 64, 90]
Iteração 3: [22, 25]
Iteração 4: [11, 12]
Iteração 5: [11, 12, 22, 25]
Iteração 6: [11, 12, 22, 25, 34, 64, 90]
Tempo de execução: 0.0006568431854248047 segundos
```

Figura 3.6 – Pior caso merge sort



### 3.4 ANÁLISE BUBBLE X MERGE

Ao analisar as tabelas geradas por cada algoritmo em cada caso específico, é notável perceber que o algoritmo merge sort se torna muito mais rápido e barato para realizar processos onde os dados se encontram mais embaralhados.

### 3.5 TESTE DE ITERAÇÕES

Array com mais elementos:

```
arrMedio = [23, 7, 45, 12, 88, 34, 67, 56, 91, 11, 49, 72, 6, 39, 80]
arrMelhor = [6, 7, 11, 12, 23, 34, 39, 45, 49, 56, 67, 72, 80, 88, 91]
arrPior = [91, 88, 80, 72, 67, 56, 49, 45, 39, 34, 23, 12, 11, 7, 6]
```

Figura 3.7 – Array aumentado

Para o algoritmo bubble sort foi obtido:

- **Melhor caso:** Apenas retorna o vetor inicial ordenado;
- **Caso Médio:** Temos 43 iterações;
- **Pior Caso:** Temos 105 iterações.

Tempo de execução: 0.005266904830932617 segundos

Figura 3.8 – Tempo no caso médio Bubble sort

Tempo de execução: 0.015620231628417969 segundos

Figura 3.9 – Tempo no pior caso Bubble sort

Para o algoritmo Merge sort foi obtido:

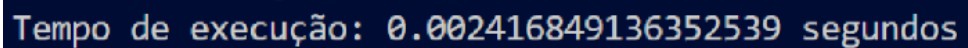
- **Melhor caso:** Temos 14 iterações;
- **Caso Médio:** Temos 14 iterações;
- **Pior Caso:** Temos 14 iterações.

Tempo de execução: 0.0 segundos

Figura 3.10 – Tempo no melhor caso Merge sort

Tempo de execução: 0.013520479202270508 segundos

Figura 3.11 – Tempo no caso médio Merge sort



Tempo de execução: 0.002416849136352539 segundos

Figura 3.12 – Tempo no pior caso Merge sort

### 3.6 CONCLUSÃO

Com esses dois conjuntos de dados foi possível perceber que quanto maior a entrada de dados o algoritmo mergesort se mostra cada vez mais eficiente em relação ao algoritmo bubble sort

Indo mais afundo, podemos realizar uma análise assintótica dos dois algoritmos e percebemos que:

- **Pior Caso Mergesort:**  $O(n \log n)$ ;
- **Pior Caso BubbleSort:**  $O(n^2)$ .

**Eficiência:** O Mergesort é significativamente mais eficiente do que o Bubble Sort, especialmente para grandes conjuntos de dados, devido à sua complexidade de tempo  $O(n \log n)$  em todos os casos.

**Uso de Espaço:** Bubble Sort é mais econômico em termos de espaço, pois não requer espaço adicional significativo, enquanto Merge Sort utiliza espaço extra proporcional ao tamanho do array.

**Aplicações:** Merge Sort é geralmente preferido em aplicações que requerem um desempenho consistente em grandes volumes de dados. Bubble Sort, por outro lado, é raramente usado na prática devido à sua ineficiência, embora possa ser útil para fins educacionais e em conjuntos de dados muito pequenos.

## 4 VISUALIZAÇÃO DA ÁRVORE DE RECURSÃO

- Descrição: criar uma visualização gráfica da árvore de recursão do mergesort durante a execução.
- Objetivo: Visualizar como o algoritmo divide e conquista os dados.
- Dicas: Usar bibliotecas de visualização (como Matplotlib em Python) pode ser interessante para este projeto.

Foi desenvolvida uma visualização gráfica da árvore de recursão do algoritmo Merge Sort usando as bibliotecas Matplotlib e NetworkX em Python. O grafo gerado representa cada chamada recursiva, ilustrando como o array é dividido e mesclado.

A função `merge_sort` cria um nó para cada subarray dividido, conectando-os ao nó pai. Já a função `merge` realiza a mesclagem final dos subarrays.

A função `draw_recursion_tree` desenha a árvore de recursão, organizando os nós hierarquicamente para visualizar o processo de divisão e mesclagem.

Esse gráfico oferece uma visão intuitiva do fluxo do Merge Sort, permitindo uma análise detalhada de sua execução.

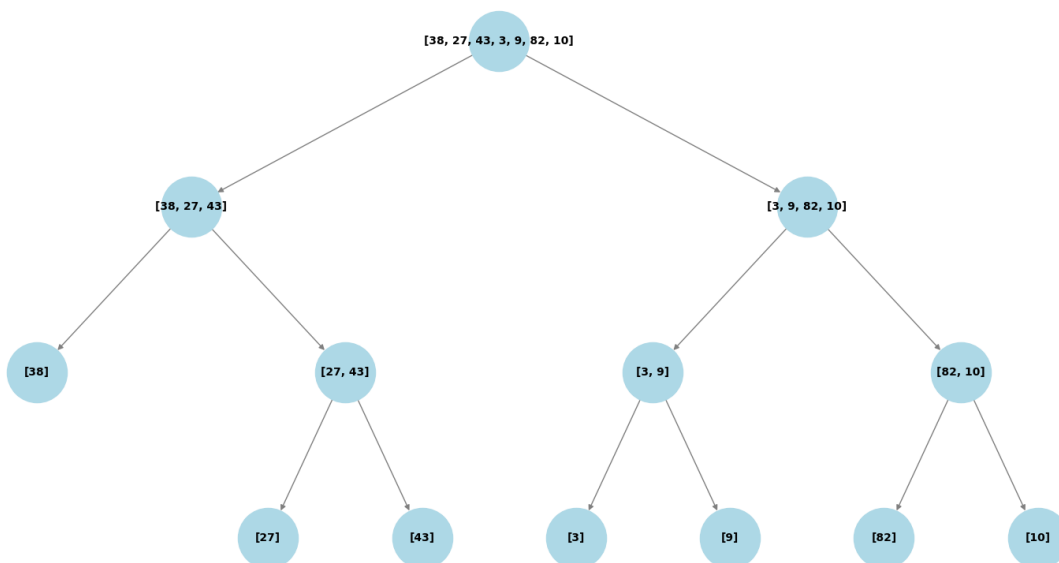


Figura 4.1 – Árvore de Recursão do Merge Sort gerada durante a execução do algoritmo.

O código completo que gera esta visualização está disponível no repositório GitHub do projeto. Esta visualização ajuda a consolidar o entendimento teórico do Merge Sort, destacando sua natureza recursiva e a estratégia de divisão e conquista usada para ordenar os dados.

## 5 MERGESORT PARALELO

- Descrição: Implementar uma versão paralela do mergesort, usando threads ou outras técnicas de paralelização.
- Objetivo: Explorar como a divisão e conquista pode ser aplicada em um contexto paralelo e discutir a complexidade envolvida.
- Dicas: Falar sobre o impacto da paralelização na eficiência do algoritmo e em situações práticas.

### 5.1 PARALELIZAÇÃO DO MERGESORT

A paralelização do MergeSort com threads explora o princípio de "divisão e conquista", dividindo o problema em subproblemas menores resolvidos de forma independente. Cada metade da lista pode ser ordenada simultaneamente por threads, aproveitando sistemas multi-core. No entanto, a eficiência depende da quantidade de núcleos e da sobrecarga na criação e gerenciamento das threads, o que pode ser contraproducente para entradas pequenas ou em sistemas com poucos recursos.

### 5.2 VANTAGENS E LIMITAÇÕES DA PARALELIZAÇÃO

Em dados maiores, a paralelização pode reduzir significativamente o tempo de execução. A complexidade  $O(n \log n)$  se mantém, mas a divisão do trabalho entre núcleos diminui o tempo percebido. Ainda assim, é crucial equilibrar o número de threads para evitar maior complexidade de gerenciamento e uso de memória, especialmente em sistemas com poucos recursos.

### 5.3 IMPLEMENTAÇÃO PARALELA DO MERGESORT

A implementação paralela do MergeSort, que está disponível no repositório GitHub do trabalho, segue essa abordagem de paralelização. O código utiliza a biblioteca `threading` para criar threads, permitindo que cada metade da lista seja ordenada em paralelo. Após a execução das threads, as sublistas ordenadas são mescladas com a função `merge`. Essa implementação demonstra uma solução simples e prática para aproveitar a paralelização, oferecendo ganhos de desempenho em cenários com grandes volumes de dados.

## 6 ANÁLISE DE COMPLEXIDADE

- Descrição: Após implementar o mergesort, os alunos devem analisar o tempo de execução em diferentes entradas e discutir a complexidade teórica versus a prática.
- Objetivo: Refletir sobre a eficiência do algoritmo na prática e entender o que pode influenciar a complexidade real.
- Dicas: Documentar suas observações com gráficos ou tabelas.

### 6.1 COMPARAÇÃO TEÓRICA X PRÁTICA

O algoritmo Merge Sort tem uma complexidade de tempo garantida de  $O(n \log n)$ , tanto no melhor, pior, quanto no caso médio. Isso significa que, teoricamente, o número de operações realizadas pelo algoritmo em todos esses cenários será aproximadamente o mesmo.

- **Melhor caso:** O array já está ordenado. O Merge Sort ainda faz comparações e divide o array, então a complexidade teórica continua sendo  $O(n \log n)$ ;
- **Caso Médio:** O array é aleatório. Esse caso reflete o comportamento típico do algoritmo. O número de divisões se mantém o mesmo;
- **Pior Caso:** O array está ordenado de forma inversa. O Merge Sort continua com a mesma complexidade  $O(n \log n)$ , com o mesmo número de divisões.

Portanto, na teoria, os tempos de execução para qualquer um dos casos deve ser o mesmo. Abaixo estão os resultados de testes feitos com 1000 entradas em cada um dos casos:

| Caso   | Nº de elementos | Tempo de execução |
|--------|-----------------|-------------------|
| Melhor | 1000            | 0,000993s         |
| Médio  | 1000            | 0,001000s         |
| Pior   | 1000            | 0,001053s         |

### 6.2 CONCLUSÃO

Embora a teoria sugira uma equivalência nos tempos de execução, os resultados práticos mostram que há uma variação mínima, que pode ser atribuída a fatores como gerenciamento de memória, uso de cache, eficiência da CPU e peculiaridades do sistema

operacional. Isso destaca a importância de considerar tanto a análise teórica quanto a prática ao avaliar o desempenho de algoritmos. Em suma, o Merge Sort se mantém uma escolha eficiente e robusta para a ordenação de dados em diversos cenários.

## 7 COMPARAÇÃO DA IMPLEMENTAÇÃO RECURSIVA X ITERATIVA

- **Descrição:** Implementar o mergesort de forma recursiva e iterativa e comparar os dois métodos em termos de desempenho e clareza do código.
- **Objetivo:** Entender as diferenças entre abordagens recursivas e iterativas.
- **Dicas:** Discutir a utilização de pilhas de chamada e consumo de memória.

### 7.1 DESEMPENHO (TEMPO DE EXECUÇÃO)

- **Versão recursiva:** Apresentou um tempo médio de 0,0351 segundos para 10 execuções;
- **Versão iterativa:** Apresentou um tempo médio de 0,0361 segundos para 10 execuções.

Apesar das diferenças serem mínimas, a versão recursiva teve uma leve vantagem em termos de desempenho para um array de 10.000 elementos. Isso sugere que, para conjuntos de dados moderados, a escolha entre recursivo e iterativo não trará um impacto significativo na performance.

### 7.2 CLAREZA E ELEGÂNCIA

- **Versão recursiva:** O código recursivo é mais intuitivo e segue uma abordagem de "dividir e conquistar", tornando-o mais fácil de entender;
- **Versão iterativa:** O código iterativo elimina o uso da pilha de chamadas, mas é um pouco mais complexo de implementar e pode ser menos claro.

### 7.3 USO DE MEMÓRIA

- **Versão recursiva:** Pode apresentar maior consumo de memória devido à profundidade da recursão. Cada chamada recursiva é armazenada na pilha de chamadas, o que aumenta o uso de memória;
- **Versão iterativa:** Mais eficiente em termos de uso de memória, pois evita a recursão, utilizando apenas loops.

## 7.4 CONCLUSÃO

A escolha entre recursiva e iterativa depende mais do contexto: se a prioridade é a legibilidade do código ou a eficiência de memória.