

Trabalho Prático I

Estrutura de Dados II

Gabriel Silva Xavier¹, Richard Rodrigues¹, Vitor Henrique¹

¹Instituto 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 fundamental for organizing data sets in different scenarios. This work aims to analyze the performance of the MergeSort, QuickSort and HeapSort algorithms with different types and sizes of data, also exploring the integration of MergeSort with InsertionSort. Furthermore, a new algorithm, the RecycleSort, is proposed. Algorithms were implemented to sort random values into different structures, with metrics recorded for later analysis.*

Resumo. *Os algoritmos de ordenação são fundamentais para organizar conjuntos de dados em diversos cenários. Assim, este trabalho tem como objetivo analisar 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 um novo algoritmo, o RecycleSort. 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 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 RecycleSort. 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

Nesta seção é feito o embasamento acerca dos algoritmos de ordenação abordados neste trabalho, que se mostram fundamentais para uma boa compreensão deste. Os algoritmos são o InsertionSort, MergeSort, QuickSort e HeapSort, já conhecidos pela literatura, e o algoritmo RecycleSort, desenvolvido neste trabalho.

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. Foi implementada a versão apresentada por [Backes 2017].

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 ordenação. 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. Foi implementada a versão apresentada por [Backes 2017].

2.3. QuickSort

O QuickSort é um dos mais eficientes algoritmos de ordenação, apresentado por [Hoare 1962]; 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. Foi implementada a versão apresentada por [Backes 2017].

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. Para este trabalho, foi implementada a versão apresentada por [Backes 2017]. 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. RecycleSort

O RecycleSort 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 QuickSort em diferentes estruturas de dados, executado em hardware com: Intel Core i7-8565U, 8GB RAM, Windows 11, HD de 2TB.
- **Cenário 2:** Análise de desempenho do MergeSort em dois hardwares distintos. Primeira máquina: AMD Ryzen 5 3500U, 6GB RAM, Windows 11, HD de 1TB. Segunda máquina: Intel Core i3-7020U, 8GB RAM, Arch Linux, HD de 1TB.
- **Cenário 3:** Comparação entre HeapSort e RecycleSort em uma máquina com: Intel Core i3-7020U, 8GB RAM, Arch Linux, HD de 1TB.

4. Experimentos Computacionais

Os resultados apresentados nas tabelas a seguir mostram o tempo de execução dos algoritmos em diferentes cenários e hardwares. A análise comparativa considera os tempos médios obtidos em 5 execuções para cada valor de 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	-

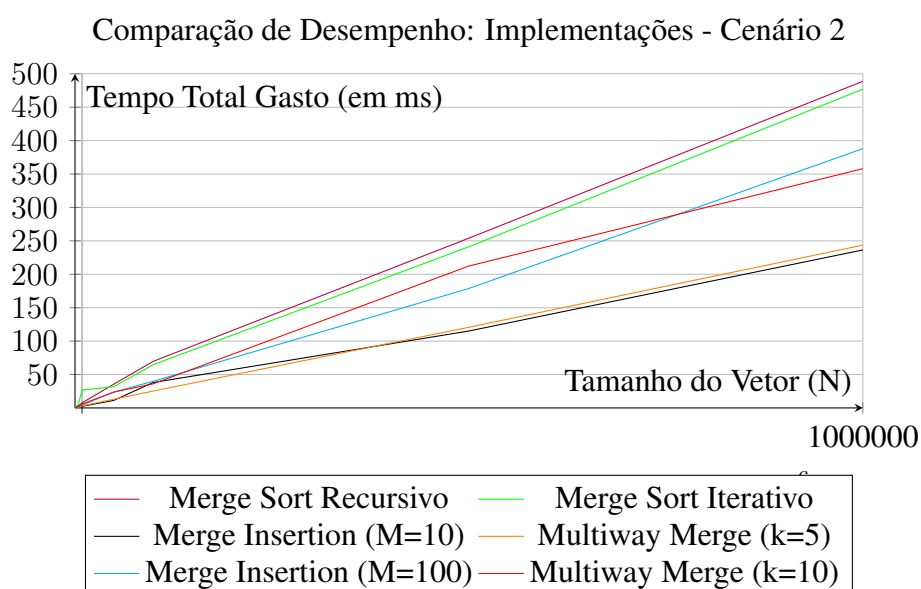
Tabela 1. Resultados do Cenário 1: QuickSort em diferentes estruturas

4.2. Cenário (2)

Análise da média dos tempos de execução das quatro implementações realizadas no Cenário 2.

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 2: Variações do Merge Sort.



4.3. Cenário (3)

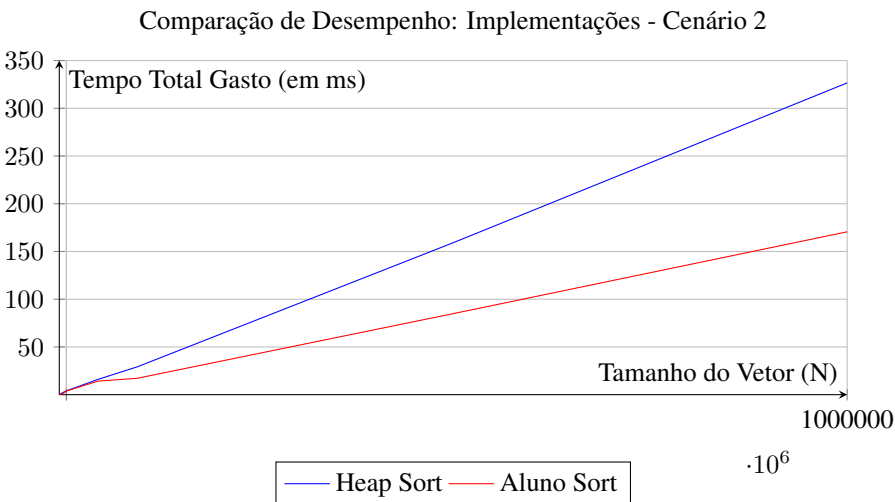
Análise Comparativa de Desempenho entre os Algoritmos HeapSort e RecycleSort

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: RecycleSort para diferentes tamanhos.



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 RecycleSort 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

- Backes, A. (2017). *Estrutura de Dados Descomplicada-em Linguagem C*. Elsevier Brasil.
- Hoare, C. A. (1962). Quicksort. *The computer journal*, 5(1):10–16.
- 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.