Capítulo 1

Trabalho Prático I - EDII

Davi Henrique Garcia Araújo, Rafael de Padua Oliveira e Robson Ribeiro Filho

Abstract

This paper investigates and compares different sorting algorithms and data structures, analyzing their implementations and performances across various scenarios. The study is divided into three scenarios: the first addresses traditional sorting algorithms and their underlying data structures; the second explores variations of the MergeSort algorithm; and the third compares the performance of Heapsort and AlunoSort.

Resumo

Este trabalho investiga e compara diferentes algoritmos de ordenação e estruturas de dados, analisando suas implementações e desempenhos em diferentes cenários. O trabalho é dividido em três cenários: o primeiro trata de algoritmos de ordenação tradicionais e suas estruturas de dados subjacentes; o segundo explora variações do algoritmo MergeSort; e o terceiro compara o desempenho entre Heapsort e AlunoSort.

1.1. Informações Gerais

Os testes de desempenho foram realizados em um computador com as seguintes especificações: processador Intel Core i5 de 12ª geração, sistema operacional Ubuntu Linux 22.04 LTS, 16GB de memória RAM DDR4 e placa de vídeo Intel Iris Xe. As comparações entre os algoritmos são feitas com base no tempo de execução e no uso de recursos computacionais, utilizando a função getrusage do sistema operacional Linux para monitorar o consumo de recursos durante a execução.

1.2. Cenário 1: Algoritmos de ordenação e Estruturas de Dados

De acordo com Ziviani (2015), o QuickSort utiliza a estratégia de divisão e conquista, na qual um elemento é selecionado como pivô para particionar a lista em duas partes: uma

com os elementos menores que o pivô e outra com os maiores. Em seguida, a ordenação é realizada de forma recursiva sobre essas subdivisões.

O QuickSort é considerado eficiente para ordenar inteiros, apresentando uma complexidade média de $O(n \log n)$, devido à simplicidade das comparações entre os elementos. Embora o pior caso possa chegar a $O(n^2)$ quando a escolha do pivô não é ideal, essa situação pode ser mitigada com boas estratégias de seleção do pivô. Por isso, o QuickSort é geralmente preferido em relação a algoritmos mais simples, como o Bubble Sort (ZIVIANI, 2015).

N	**1.000**	**10.000**	**100.000**	**1.000.000**	**10.000.000**
Com1	11.908	193.604	5.595.866	423.694.013	41.535.134.898
Temp1	0,000166	0,001946	0,014772	0,427282	43,383890
Com2	10.445	183.102	5.539.616	426.029.573	41.835.418.084
Temp2	0,000202	0,002223	0,021388	0,424769	46,441712
Com3	11.971	183.102	5.533.207	426.220.969	41.737.502.257
Temp3	0,000166	0,001638	0,025105	0,422492	46,207378
Comp4	11.971	186.321	5.533.207	424.239.044	41.333.874.709
Temp4	0,000147	0,002098	0,020796	0,426339	45,448455
Comp5	11.088	186.321	5.683.387	426.599.552	41.743.601.648
Temp5	0,000162	0,002306	0,021126	0,425813	45,846310
CompM	11.908	186.321	5.539.616	426.029.573	41.737.502.257
TempM	0,000166	0,002098	0,021126	0,425813	45,846310

Tabela 1.1: QuickSort

Quando utilizado em listas duplamente encadeadas, o QuickSort torna-se menos eficiente. Ao invés de simplesmente trocar os elementos, o algoritmo precisa alterar os ponteiros de cada nó, o que aumenta a complexidade das operações. Esse aumento de complexidade torna o algoritmo menos adequado para listas encadeadas, sendo que o MergeSort, por exemplo, tende a ser mais eficiente nesse contexto (CORMEN et al., 2009).

N	**1.000**	**10.000**	**100.000**	**1.000.000**	**10.000.000**
Com1	10.571	230.559	11.311.868	-	-
Temp1	0,002952	0,125624	13,354818	-	-
Com2	11.546	211.276	11.139.915	-	-
Temp2	0,003669	0,126605	13,028776	-	-
Com3	10.285	217.809	11.214.047	-	-
Temp3	0,004041	0,125875	12,932265	-	-
Comp4	10.823	228.831	11.176.263	-	-
Temp4	0,002497	0,127018	13,703273	-	-
Comp5	10.823	240.570	11.426.269	-	-
Temp5	0,002385	0,126882	13,820160	-	-
CompM	10.823	228.831	11.214.047	-	-
TempM	0,002952	0,126605	13,354818	-	-

Tabela 1.2: QuickSort em Lista

Em estruturas de dados compostas, como as structs, a eficiência do QuickSort depende dos campos utilizados nas comparações. Se os campos forem simples, como inteiros ou floats, o desempenho do algoritmo se mantém bom. No entanto, quando as estruturas contêm campos mais complexos, como strings ou outros tipos de dados compostos,

as comparações podem se tornar um gargalo, impactando negativamente a performance do algoritmo (ZIVIANI, 2015).

N	1.000	10.000	100.000	1.000.000	10.000.000
Com1	10.678	181.912	5.589.277	430.267.367	-
Temp1	0,001060	0,014698	0,199350	2,639955	-
Com2	11.195	177.502	5.482.191	432.159.724	-
Temp2	0,001061	0,015484	0,204248	2,670156	-
Com3	9.930	176.434	5.617.156	429.713.798	-
Temp3	0,001499	0,015361	0,198122	2,645608	-
Comp4	11.031	181.153	5.600.214	426.781.568	-
Temp4	0,001029	0,014762	0,193450	2,662664	-
Comp5	10.374	181.153	5.657.298	426.325.326	-
Temp5	0,001475	0,014630	0,197718	2,707225	-
CompM	10.678	181.153	5.600.214	429.713.798	-
TempM	0,001061	0,014762	0,198122	2,662664	-

Tabela 1.3: QuickSort em Struct

1.3. Cenário 2: Variações do MergeSort

O MergeSort é um algoritmo de ordenação baseado em divisão e conquista, que divide o vetor em partes menores até obter subvetores com um único elemento e os combina de forma ordenada. Com complexidade$O(\log n)$, é eficiente para grandes conjuntos de dados, estável e preserva a ordem de elementos iguais, embora exija memória adicional para a mesclagem (Aqib; Nawaz; Butt, 2021).

1.3.1. MergeSort Recursivo

O MergeSort Recursivo ordena o vetor dividindo-o ao meio recursivamente até obter subvetores com um único elemento. Em seguida, esses subvetores são mesclados recursivamente, combinando-os em vetores maiores e ordenados até que o vetor completo esteja organizado. A recursividade guia todo o processo, gerenciando divisões e combinações de forma eficiente (Aqib; Nawaz; Butt, 2021).

1.3.2. MergeSort Interativo

O MergeSort Iterativo ordena o vetor sem chamadas recursivas, iniciando com subvetores de tamanho 1 e dobrando o tamanho a cada etapa (2, 4, 8, etc.). Em cada interação, os subvetores adjacentes são mesclados ordenadamente até que o vetor completo esteja organizado (Ishimwe; Nguyen; Nguyen, 2021).

1.3.3. MergeSort Inserção(N)

O MergeSort por Inserção combina a recursividade do MergeSort com a ordenação por inserção. O vetor é dividido até que os subvetores atinjam um tamanho "N", quando a divisão cessa e a ordenação por inserção é aplicada. Após ordenar os subvetores, eles são mesclados como no MergeSort tradicional. (Alqattan Et Al., 2023).

1.3.4. MergeSort Multiway(K)

O MergeSort Multiway (K) divide recursivamente o vetor em K partes, em vez de duas, até que os subvetores tenham um único elemento. Em seguida, os subvetores são mesclados em etapas, formando uma sequência ordenada (Salah Et Al., 2020).

1.3.5. Comparação dos resultados

A comparação das variações do MergeSort avalia o impacto de diferentes implementações na eficiência do algoritmo, destacando vantagens em cenários específicos. Vetores com 1.000 a 10.000.000 de valores aleatórios foram usados. Os resultados estão apresentados na Tabela 1.4. O desempenho das variações do MergeSort demonstra características distintas com base em seu funcionamento. O MergeSort Recursivo apresenta o maior tempo médio em vetores grandes devido à sobrecarga de chamadas recursivas. Já o MergeSort Inserção(10) é mais eficiente em vetores pequenos, pois utiliza o algoritmo de inserção para ordenar subvetores de 10 elementos antes da mesclagem. O MergeSort Inserção(100), apesar de seguir o mesmo princípio apresenta desempenho inferior ao Inserção(10), A variação Interativa se destaca em vetores pequenos, por ter a ausência de de chamadas recursivas, mas perde eficiência conforme o tamanho do vetor cresce. Por fim, os métodos Multiway, embora mais lentos para vetores pequenos, mostram melhorias significativas para vetores grandes, como o Multiway(5) apresentando uma leve vantagem sobre o Multiway(10).

N	Recursivo (s)	Inserção (10) (s)	Inserção (100) (s)	Iterativo (s)	Multiway (10) (s)	Multiway (5) (s)
1000	0.000258	0.000163	0.000129	0.000234	0.000196	0.000278
5000	0.001312	0.001005	0.000880	0.001219	0.001258	0.001320
10000	0.002539	0.001846	0.002034	0.002312	0.002422	0.002504
50000	0.013572	0.012436	0.013519	0.012507	0.014586	0.011970
100000	0.030725	0.021302	0.025876	0.022137	0.020402	0.024440
500000	0.104751	0.087559	0.080797	0.083842	0.105253	0.082108
1000000	0.195456	0.166443	0.153973	0.164828	0.159571	0.166206
5000000	1.023708	0.813007	0.848784	0.877705	0.968634	0.859729
10000000	2.111316	1.690932	1.743604	1.852401	1.678090	1.584498

Tabela 1.4: Tabela de tempos de execução dos algoritmos (em segundos).

1.4. Cenário 3: Heapsort X Aluno Sort

1.4.1. Introdução

Serão analisados dois algoritmos de ordenação: o Heapsort e o AlunoSort. A análise será baseada no tempo médio de execução ao ordenar vetores de inteiros positivos aleatórios com tamanhos variando de 1.000 a 1.000.000 elementos. Serão exploradas as características do AlunoSort, apresentando-se também os resultados obtidos e as diferenças de desempenho em relação ao Heapsort.

1.4.2. HeapSort

O heapsort é um algoritmo de ordenação eficiente que utiliza a estrutura de dados heap como base para organizar os elementos de uma lista. O heap é uma árvore binária que possui três propriedades fundamentais: a árvore é completamente preenchida, exceto talvez no último nível; cada nó contém uma chave. Portanto, o heapsort constroi inicialmente

um heap a partir do conjunto de dados e, em seguida, realiza sucessivas extrações da maior chave (no caso de um max-heap), reorganizando a estrutura a cada passo para manter a propriedade do heap.

1.4.3. AlunoSort

O algoritmo AlunoSort foi desenvolvido com base no modelo de divisão e conquista, inspirado no Quicksort, mas com uma abordagem diferente. Em vez de usar um pivô, o AlunoSort realiza um rearranjo iterativo entre as duas metades do vetor. Inicialmente, o vetor é dividido ao meio, e, na primeira metade, o algoritmo encontra o maior elemento, registrando seu índice em maxIndex. Na segunda metade, o processo é similar, buscando os menores elementos e armazenando seus índices em minIndex.

Após identificar os índices, o algoritmo compara os maiores de maxIndex com os menores de minIndex e os troca se necessário. Esse processo de trocas continua até que não haja mais mudanças, e então o AlunoSort é aplicado recursivamente nas duas metades. O algoritmo segue até que as metades tenham tamanho 1 ou menos, completando a ordenação do vetor.

1.4.4. Comparação dos resultados

O algoritmo AlunoSort tem um custo de $k \cdot n \cdot logn$, onde k é a quantidade de elementos a serem trocados. No melhor caso ($k = 1$), o custo é $n \cdot logn$, e no pior caso ($k = n$), o custo sobe para $n^2 \cdot logn$. O HeapSort, por outro lado, tem custo constante de $n \cdot logn$, devido à construção do heap e extração dos elementos.

No melhor caso, ambos os algoritmos têm desempenho equivalente ($n \cdot logn$), mas no pior caso, o custo do AlunoSort é muito maior, o que faz com que o HeapSort tenho uma grande vantagem. No cenário médio, o desempenho do AlunoSort é próximo ao do HeapSort, mas ainda cresce de forma exponencial, tornando-o ineficiente para grandes volumes de dados. Em resumo, o HeapSort é mais eficiente e estável que o AlunoSort.

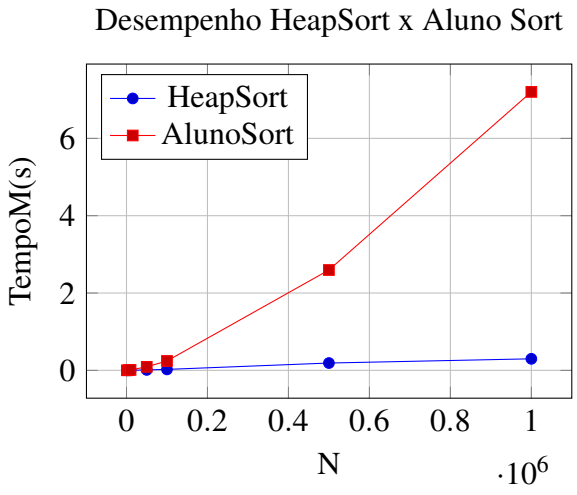

Figura 1.1: Gráfico de Desempenho

Referências

[1] ALQATTAN, Masuma M. et al. Comparison of Insertion, Merge, and Hybrid Sorting Algorithms Using C+. 2023.

[2] AQIB, Syed Muqeet; NAWAZ, Haque; BUTT, Shah Muhammad. Analysis of merge sort and bubble sort in python, php, javascript, and c language. *International Journal*, v. 10, n. 2, 2021.

[3] CORMEN, Thomas H. *Desmistificando algoritmos*. 1. ed. Rio de Janeiro: Elsevier, 2014.

[4] CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Introdução aos Algoritmos*. 3. ed. Rio de Janeiro: Elsevier, 2009.

[5] ISHIMWE, D.; NGUYEN, K.; NGUYEN, T. Dynaplex: analyzing program complexity using dynamically inferred recurrence relations. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA), 1-23, 2021.

[6] SALAH, Ahmad et al. A time-space efficient algorithm for parallel k-way in-place merging based on sequence partitioning and perfect shuffle. *ACM Transactions on Parallel Computing (TOPC)*, v. 7, n. 2, p. 1-23, 2020.

[7] SARKAR, Bishal et al. An Advanced Research on Enhancing Sorting and Searching Algorithms: A Comprehensive Study. Kolkata: Department of Electronics and Communication Engineering, Guru Nanak Institute of Technology.

[8] ZIVIANI, Nívio. *Projeto de Algoritmos com Implementação em Pascal e C*. 2. ed. Cengage Learning, 2015.