Comparativo dos Métodos $O(n \log n)$: MergeSort, QuickSort e HeapSort

Bruno Stafuzza Maion[1], **Lucca Abbado Neres**[1], **Rafael Hoffman**[1]

[1]Colegiado de Ciência da Computação – Universidade Estadual do Oeste do Paraná (Unioeste)
Cascavel – PR – Brasil

bruno.maion@unioeste.br, Luccaneres@hotmail.com, rafaeldinh@hotmail.com

Resumo. *Este estudo analisa a complexidade e o desempenho de três algoritmos de ordenação: MergeSort, QuickSort e HeapSort. São examinadas as operações de cada método e as diferenças de desempenho em diferentes conjuntos de dados, como aleatórios, crescentes, parcialmente ordenados e decrescentes. O MergeSort apresenta a complexidade de $O(n \log n)$ em todos os casos. O QuickSort tem um desempenho variado entre $O(n \log n)$ no melhor caso e $O(n^2)$ no pior. Já o HeapSort mantém uma complexidade de $O(n \log n)$ em todos os casos. Os resultados experimentais revelam que o QuickSort tem um desempenho superior*

1. Introdução

Neste trabalho, realizamos um estudo comparativo do desempenho de três algoritmos de ordenação amplamente utilizados: MergeSort, QuickSort e HeapSort. Através de experimentos com diferentes conjuntos de dados, avaliamos a eficiência de cada algoritmo em termos de tempo de execução. Os resultados obtidos permitem identificar o algoritmo mais adequado para diferentes cenários e contribuem para uma melhor compreensão das características de cada um.

O MergeSort é um algoritmo de ordenação baseado na estratégia de divisão e conquista. Ele divide o conjunto de dados em partes menores, ordena cada uma delas de forma recursiva e, em seguida, mescla essas partes em uma única sequência ordenada. Sua complexidade é $O(n \log n)$ no pior, médio e melhor caso, tornando-o eficiente mesmo para grandes volumes de dados. No entanto, devido à necessidade de espaço adicional para as sublistas durante o processo de mesclagem, o MergeSort não é in-place, o que pode ser uma desvantagem em sistemas com memória limitada [ORMEN et al. 2002]..

O QuickSort é um algoritmo de ordenação eficiente que também utiliza a estratégia de divisão e conquista. Ele escolhe um elemento como pivô, particiona o conjunto de dados ao redor desse pivô e, então, ordena as sublistas resultantes de forma recursiva. Sua complexidade no melhor e médio caso é $O(n \log n)$, mas pode atingir $O(n^2)$ no pior caso, quando o pivô escolhido não é ideal. Apesar disso, sua implementação in-place e a ausência de uso significativo de memória adicional tornam o QuickSort uma escolha popular para muitos cenários práticos [ORMEN et al. 2002].

O HeapSort é um algoritmo de ordenação que utiliza a estrutura de dados heap, especificamente um heap máximo ou mínimo. Ele constrói o heap a partir do conjunto de dados, extrai o maior (ou menor) elemento repetidamente e reorganiza o heap até que todos os elementos estejam ordenados. O HeapSort tem complexidade $O(n \log n)$ em

todos os casos e é in-place, não exigindo memória adicional significativa. Embora sua eficiência seja comparável à do MergeSort e QuickSort, ele é frequentemente mais lento em aplicações práticas devido ao maior número de operações de troca e reordenação do heap [ORMEN et al. 2002]..

2. Objetivos

Realizar um estudo comparativo de desempenho de três algoritmos de ordenação: MergeSort, QuickSort e HeapSort, com foco em sua eficiência em termos de tempo de execução.

3. Material

O experimento foi realizado em um notebook equipado com um processador Intel Core i5 de 4ª geração, 8GB de memória RAM DDR3 com frequência de 1600MHz e um SSD de 240GB para armazenamento. O sistema operacional utilizado foi o Ubuntu 22.04.3 LTS. Os códigos estão hospedados no repositório do GitHub[1].

4. Metodologia

Os algoritmos MergeSort, QuickSort e HeapSort foram implementados em C++, utilizando a biblioteca padrão para medir o tempo de execução. A análise dos resultados foi realizada em Python. O algortimo HeapSort utiliza a função *MaxHeap*, ou seja, cada nó pai é maior ou igual aos seus filhos. Para cada algoritmo, foram utilizados 4 tipos de ordenação inicial como entrada, sendo o conjunto de dados aleatórios; conjunto de dados ordenados de forma crescente; conjunto de dados ordenados parcialmente; e, conjunto de dados ordenados de forma decrescente - com cada contendo 20 conjuntos de tamanhos variando entre 100 e 2.000.000 elementos em formato de arquivo de texto.

Cada combinação de tipo e tamanho de conjunto foi submetida a 20 execuções para cada algoritmo, sendo calculada a média para avaliar o desempenho em diferentes cenários de dados."

4.1. MergeSort

A função **merge**, que mescla dois subvetores ordenados, tem complexidade $O(n)$, pois percorre todos os elementos do vetor para realizar a fusão. No caso do algoritmo, as comparações e inserções dos elementos entre os dois subvetores resultam em um total de n operações, onde n é o número total de elementos na parte do vetor que está sendo processada.

O algoritmo recursivo **MergeSort** divide o vetor em duas metades até que as sublistas tenham tamanho 1. A cada divisão, o vetor é cortado ao meio, o que leva a $O(\log n)$ divisões, onde n é o tamanho do vetor original. Após as divisões, o vetor é mesclado de volta utilizando a função **merge**.

Como a função **merge** é executada $O(n)$ vezes e a recursão ocorre $O(\log n)$ vezes, a complexidade total do **MergeSort** é dada por:

$$O(n \log n)$$

[1] GitHub.

Essa complexidade é a mesma para os melhores, piores e casos médios, pois a divisão e a fusão sempre ocorrem da mesma forma, independentemente da ordem dos elementos.

4.2. QuickSort

A complexidade do algoritmo **QuickSort** é analisada com base no número de operações realizadas durante o particionamento e nas chamadas recursivas [ORMEN et al. 2002].

- n comparações nos laços **while**.
- Até 10 operações de trocas e incrementos/decrementos.

Assim, o custo do particionamento é proporcional a c, onde c é uma constante.

O vetor é dividido em duas partes:

- Melhor caso: partições são balanceadas ($n/2$ cada).
- Pior caso: uma partição tem $n - 1$ elementos e a outra apenas 1 elemento.

A relação de recorrência para o tempo total é:

$$T(n) = T(k) + T(n - k - 1) + c$$

- Melhor caso e caso médio: O particionamento é balanceado, resultando em $T(n) = O(n \log n)$.
- Pior caso: O particionamento é altamente desequilibrado, resultando em $T(n) = O(n^2)$.

4.3. HeapSort

A função *heapify* ajusta a posição dos elementos para manter essa propriedade. Especificamente, ela verifica se os filhos esquerdo e direito são maiores que o nó pai e, se for o caso, troca o pai com o maior dos filhos e chama recursivamente *heapify* para garantir que a sub-árvore também satisfaça a propriedade de *MaxHeap* [ORMEN et al. 2002]..

A função **heapify** tem complexidade $O(\log n)$, pois ajusta um nó na árvore binária, o que envolve até duas comparações e trocas, com a altura da árvore sendo $O(\log n)$. A construção do **heap**, realizada através de chamadas sucessivas de **heapify** de $n/2$ até 1, tem complexidade $O(n)$. Após construir o **heap**, o processo de extração e reinicialização do heap envolve $O(\log n)$ para cada extração, com um total de n extrações, resultando em uma complexidade de $O(n \log n)$. Portanto, a complexidade total do algoritmo **HeapSort** é $O(n \log n)$ no pior caso, melhor caso e caso médio.

5. Resultados

Nesta seção, os resultados são apresentados de duas maneiras: agrupados por métodos de ordenação, analisando em quais conjuntos de dados cada método obteve melhor desempenho; e organizados por conjuntos de dados, comparando qual método de ordenação teve o melhor desempenho em cada tipo de conjunto (aleatório, ordenado crescentemente, parcialmente ordenado e ordenado decrescente). Todas as tabelas apresentadas neste relatório possuem o destaque para os piores resultados em vermelho, enquanto os melhores estão em azul para cada tamanho de conjunto.

5.1. Por métodos de ordenação

A tabela 1 apresenta as médias de desempenho do algoritmo de ordenação MergeSort para diferentes tamanhos de conjuntos de dados. O pior cenário é observado no conjunto Aleatório. Diferentemente, o melhor cenário é alternado entre os conjuntos Crescente e Parcialmente, mas sem diferença significativa para o conjunto Decrescente.

(N)	Aleatório	Crescente	Parcialmente	Decrescente
100	0,023	0,024	0,022	0,020
200	0,046	0,034	0,044	0,038
500	0,119	0,075	0,088	0,084
1.000	0,224	0,165	0,168	0,171
2.000	0,446	0,275	0,287	0,298
5.000	1,242	0,842	0,834	0,890
7.500	1,910	1,088	1,077	1,220
10.000	2,347	1,466	1,565	1,729
15.000	3,612	2,199	2,247	2,801
30.000	8,460	5,066	5,058	5,424
50.000	14,754	8,441	8,417	9,785
75.000	21,738	12,981	12,973	13,883
100.000	28,394	18,496	18,992	20,367
200.000	58,555	35,945	35,934	38,893
500.000	151,952	90,842	91,510	99,033
750.000	234,040	144,587	142,383	150,269
1.000.000	297,744	193,381	189,219	211,662
1.250.000	368,519	257,399	243,484	261,402
1.500.000	444,377	287,643	288,623	326,286
2.000.000	601,846	383,894	386,258	412,486

Tabela 1. Tempo (milissegundos) do método de ordenação MergeSort nos diferentes conjuntos de dados aleatório.

Por sua vez, a figura 1 ilustra graficamente esses dados, mostrando que a curva do conjunto aleatório (em azul) se destaca negativamente das demais à medida que o número de elementos no conjunto aumenta.

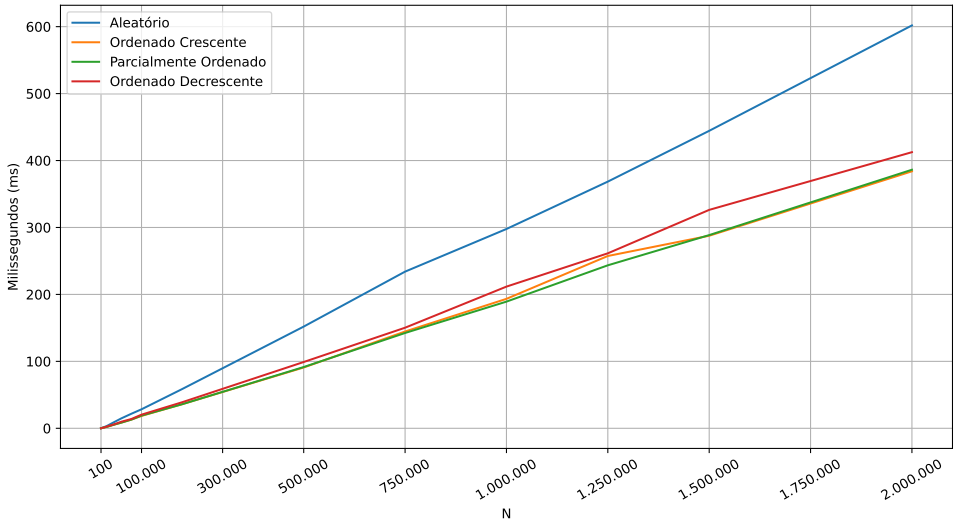

Figura 1. Comparativo de tempo do método MergeSort para diferentes tamanhos de conjunto de dados (N).

A tabela 2 apresenta os tempos médios de execução do algoritmo de ordenação QuickSort em diferentes tamanhos de conjuntos de dados. Os piores desempenhos, ocorrem principalmente para os conjuntos aleatórios. Por vez, os melhores tempos são observados para o conjunto Crescente, devido à menor necessidade de reorganização dos elementos. Esses padrões de desempenho são ilustrados na figura 2, evidenciando como o QuickSort é influenciado pela distribuição dos dados.

(N)	Aleatório	Crescente	Parcialmente	Decrescente
100	0,012	0,004	0,008	0,005
200	0,023	0,008	0,013	0,009
500	0,064	0,022	0,029	0,021
1.000	0,128	0,048	0,055	0,049
2.000	0,273	0,102	0,107	0,105
5.000	0,677	0,294	0,292	0,306
7.500	1,084	0,436	0,450	0,454
10.000	1,365	0,606	0,622	0,633
15.000	2,188	0,988	0,988	1,022
30.000	4,471	1,964	1,950	2,000
50.000	8,135	3,159	3,195	3,361
75.000	10,895	4,725	4,686	4,854
100.000	15,068	6,254	6,273	6,645
200.000	30,044	13,779	13,968	14,119
500.000	76,673	34,341	34,548	35,590
750.000	113,022	55,022	55,245	54,810
1.000.000	148,191	71,985	72,189	75,635
1.250.000	183,743	92,074	91,988	95,266
1.500.000	221,196	110,446	111,200	116,337
2.000.000	295,490	146,427	145,394	150,954

Tabela 2. Tempo (milissegundos) do método de ordenação QuickSort para diferentes tamanhos de conjunto de dados (N).

Nota-se na figura 2 que o QuickSort apresenta resultados parecidos para os conjuntos Crescente (em laranja), Parcialmente (em verde) e Decrescente (em vermelho).

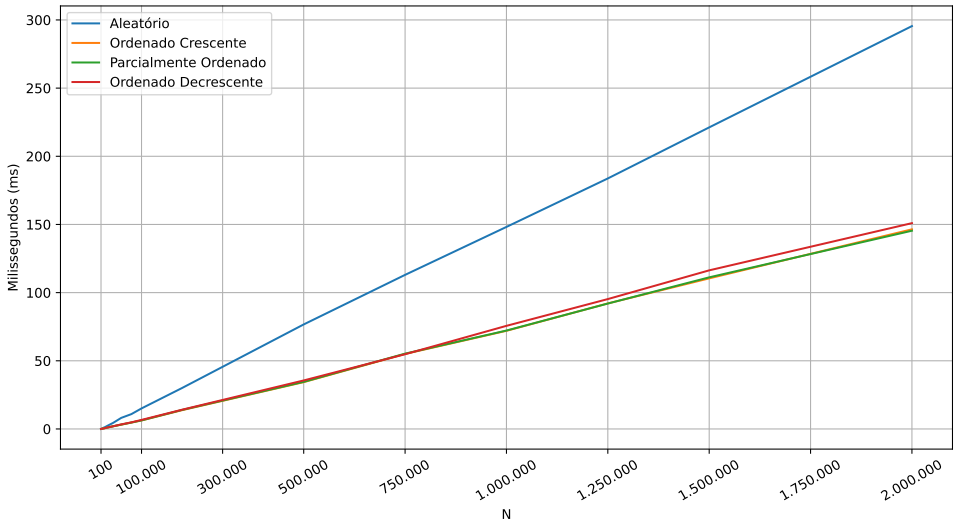

Figura 2. Comparativo de tempo do método QuickSort para diferentes tamanhos de conjunto de dados (N)

Os resultados do HeapSort estão apresentados na tabela 3. Destaca-se que nos

conjuntos Crescente, Parcialmente e Decrescente o HeapSort obteve resultados parecidos enquanto seu pior resultado foi evidenciado no conjunto aleatório.

(N)	Aleatório	Crescente	Parcialmente	Decrescente
100	0,025	0,025	0,025	0,022
200	0,054	0,054	0,058	0,051
500	0,157	0,146	0,145	0,133
1.000	0,327	0,291	0,307	0,289
2.000	0,688	0,621	0,630	0,581
5.000	1,990	1,742	1,704	1,653
7.500	3,128	2,936	2,888	2,695
10.000	4,096	3,867	3,889	3,923
15.000	6,450	5,595	5,662	5,424
30.000	14,732	11,698	11,941	11,764
50.000	26,326	20,555	20,562	20,214
75.000	40,737	32,297	32,441	31,111
100.000	53,854	43,866	44,062	45,350
200.000	120,091	85,701	86,271	90,777
500.000	342,959	220,126	224,067	245,129
750.000	524,079	343,114	337,584	368,207
1.000.000	671,038	466,964	461,563	527,957
1.250.000	874,476	583,802	581,495	651,647
1.500.000	1054,728	697,411	706,807	806,139
2.000.000	1498,042	943,106	946,240	1054,553

Tabela 3. Tempo (milissegundos) do método de ordenação HeapSort para diferentes tamanhos de conjunto de dados (N).

A figura 3 apresenta um comparativo do desempenho do algoritmo HeapSort nos diferentes cenários de entrada. A curva azul representa o desempenho em dados aleatórios, sendo o pior, a laranja em dados ordenados de forma crescente, a verde em dados parcialmente ordenados e a vermelha em dados ordenados de forma decrescente com resultados consideravelmente semelhantes.

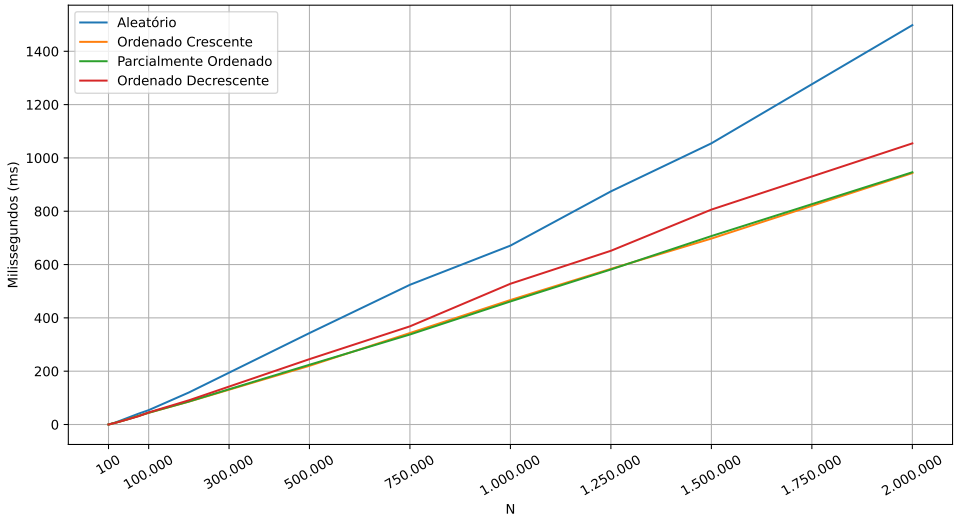

Figura 3. Comparativo de tempo do método HeapSort para diferentes tamanhos de conjunto de dados (N)

5.2. Por conjunto de dados

O resultados do conjunto de dados Aleatório estão apresentados na tabela 4. O QuickSort teve o melhor o melhor desempenho entre os algoritmos, enquanto o HeapSort obteve o pior desempenho.

(N)	MergeSort	QuickSort	HeapSort
100	0,023	0,012	0,025
200	0,046	0,023	0,054
500	0,119	0,064	0,157
1.000	0,224	0,128	0,327
2.000	0,446	0,273	0,688
5.000	1,242	0,677	1,990
7.500	1,910	1,084	3,128
10.000	2,347	1,365	4,096
15.000	3,612	2,188	6,450
30.000	8,460	4,471	14,732
50.000	14,754	8,135	26,326
75.000	21,738	10,895	40,737
100.000	28,394	15,068	53,854
200.000	58,555	30,044	120,091
500.000	151,952	76,673	342,959
750.000	234,040	113,022	524,079
1.000000	297,744	148,191	671,038
1.250000	368,519	183,743	874,476
1.500000	444,377	221,196	1054,728
2.000000	601,846	295,490	1498,042

Tabela 4. Tempo (milissegundos) dos métodos de ordenação no conjunto de dados aleatório.

A figura 4 apresenta o desempenho dos algoritmos MergeSort (em azul), o QuickSort (em laranja) sendo o melhor com 295 ms e o HeapSort (em verde) sendo o pior com 1.498 ms para o conjunto de tamanho 2 milhões.

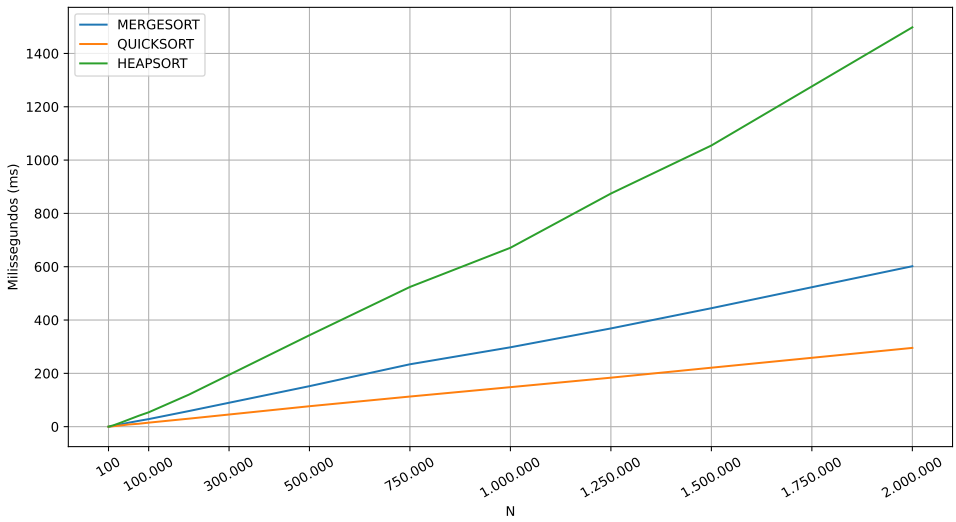

Figura 4. Comparativo de tempo entre os métodos de ordenação no conjunto de dados (N) aleatório.

Os resultados apresentados na tabela 5 expõem o desempenho dos três algoritmos de ordenação no conjuntos de dados ordenados de forma crescente. O QuickSort se destaca como o algoritmo mais eficiente, apresentando os menores tempos de execução em todos os tamanhos de conjuntos de dados. Por outro lado, o HeapSort tempos de execução maiores.

(N)	MergeSort	QuickSort	HeapSort
100	0,020	0,004	0,025
200	0,038	0,008	0,054
500	0,084	0,022	0,146
1.000	0,171	0,048	0,291
2.000	0,298	0,102	0,621
5.000	0,890	0,294	1,742
7.500	1,220	0,436	2,936
10.000	1,729	0,606	3,867
15.000	2,801	0,988	5,595
30.000	5,424	1,964	11,698
50.000	9,785	3,159	20,555
75.000	13,883	4,725	32,297
100.000	20,367	6,254	43,866
200.000	38,893	13,779	85,701
500.000	99,033	34,341	220,126
750.000	150,269	55,022	343,114
1.000000	211,662	71,985	466,964
1.250000	261,402	92,074	583,802
1.500000	326,286	110,446	697,411
2.000000	412,486	146,427	943,106

Tabela 5. Tempo (milissegundos) dos métodos de ordenação no conjunto de dados (N) ordenados de forma crescente.

A figura 5 ilustra as diferenças de desempenho, com o QuickSort (em laranja) mostrando uma curva significativamente mais baixa, indicando tempos de execução muito menores que os dos outros dois algoritmos. Enquanto o desempenho do HeapSort (em verde) aumenta consideravelmente.

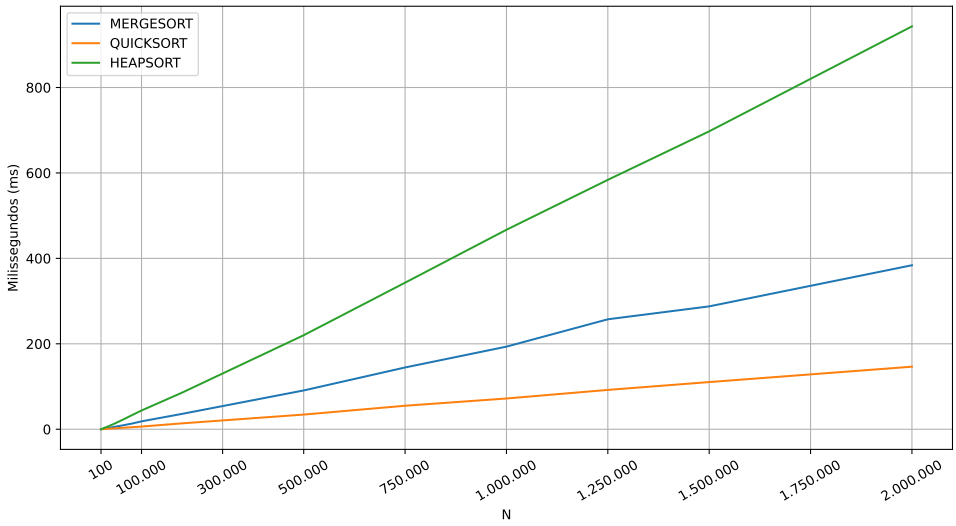

Figura 5. Comparativo de tempo entre os métodos de ordenação no conjunto de dados (N) ordenados de forma crescente.

Como apresentado na tabela 6, o algoritmo QuickSort apresentou o melhor desempenho em todos os tamanhos de conjuntos de dados, enquanto o HeapSort obteve os piores resultados.

(N)	MergeSort	QuickSort	HeapSort
100	0,022	0,008	0,025
200	0,044	0,013	0,058
500	0,088	0,029	0,145
1.000	0,168	0,055	0,307
2.000	0,287	0,107	0,630
5.000	0,834	0,292	1,704
7.500	1,077	0,450	2,888
10.000	1,565	0,622	3,889
15.000	2,247	0,988	5,662
30.000	5,058	1,950	11,941
50.000	8,417	3,195	20,562
75.000	12,973	4,686	32,441
100.000	18,992	6,273	44,062
200.000	35,934	13,968	86,271
500.000	91,510	34,548	224,067
750.000	142,383	55,245	337,584
1.000.000	189,219	72,189	461,563
1.250.000	243,484	91,988	581,495
1.500.000	288,623	111,200	706,807
2.000.000	386,258	145,394	946,240

Tabela 6. Tempo (milissegundos) dos métodos de ordenação no conjunto de dados (N) parcialmente ordenados.

A figura 6 apresenta graficamente os desempenhos dos algoritmos. A curva em laranja representa o QuickSort, com umas diferença considerável para os outros algoritmos.

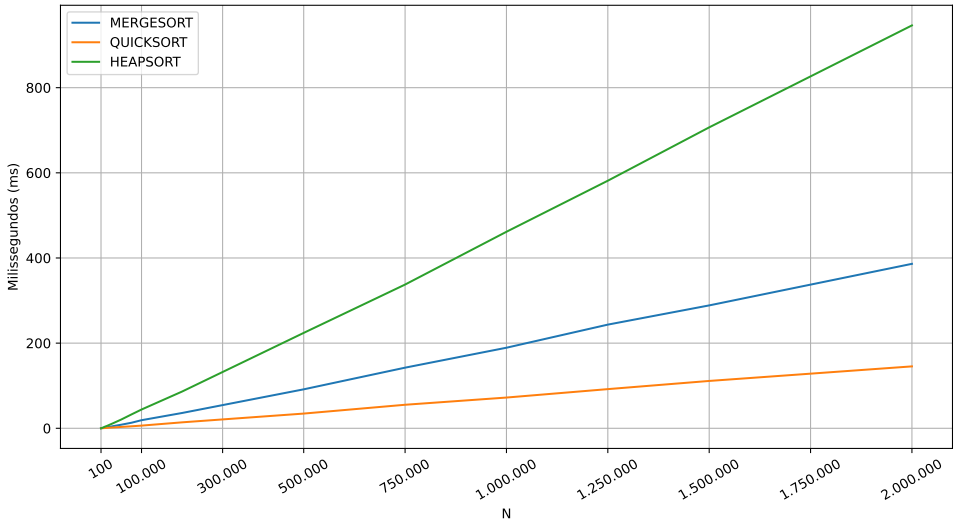

Figura 6. Comparativo de tempo entre os métodos de ordenação no conjunto de dados (N) parcialmente ordenados.

A tabela 7 apresenta os resultados para o conjunto de dados decrescente. Neta configuração dos dados, o QuickSort obteve o melhor desempenho enquanto o HeapSort obteve o pior. Conforme o gráfico apresentado na figura 7, nota-se no conjunto de tamanho 2 milhões de elementos que o desempenho do HeapSort foi aproxidamente 7 vezes maior que o QuickSort.

(N)	MergeSort	QuickSort	HeapSort
100	0,020	0,005	0,022
200	0,038	0,009	0,051
500	0,084	0,021	0,133
1.000	0,171	0,049	0,289
2.000	0,298	0,105	0,581
5.000	0,890	0,306	1,653
7.500	1,220	0,454	2,695
10.000	1,729	0,633	3,923
15.000	2,801	1,022	5,424
30.000	5,424	2,000	11,764
50.000	9,785	3,361	20,214
75.000	13,883	4,854	31,111
100.000	20,367	6,645	45,350
200.000	38,893	14,119	90,777
500.000	99,033	35,590	245,129
750.000	150,269	54,810	368,207
1.000.000	211,662	75,635	527,957
1.250.000	261,402	95,266	651,647
1.500.000	326,286	116,337	806,139
2.000.000	412,486	150,954	1054,553

Tabela 7. Tempo (milissegundos) dos métodos de ordenação no conjunto de dados (N) ordenados de forma decrescente

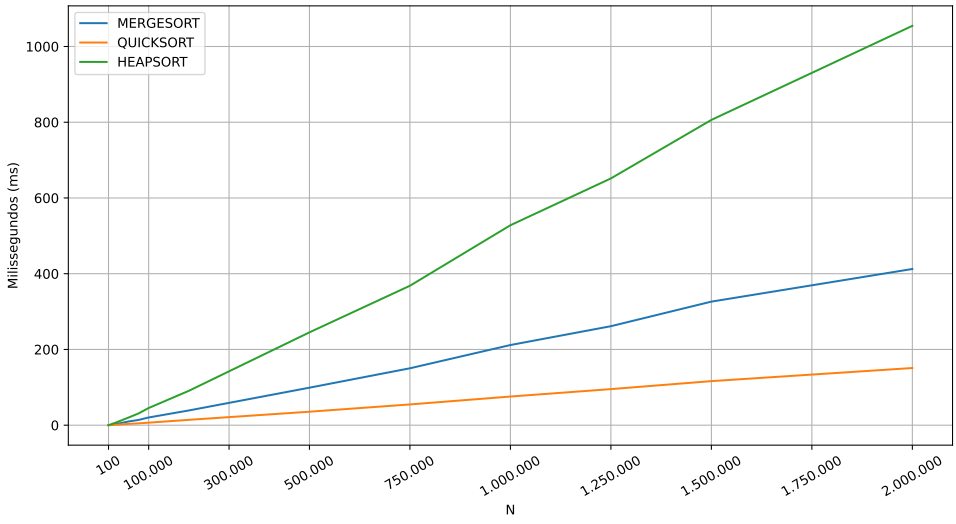

Figura 7. Comparativo de tempo entre os métodos de ordenação no conjunto de dados (N) ordenados de forma decrescente.

6. Conclusão

Os resultados obtidos neste estudo evidenciam que o desempenho dos algoritmos de ordenação MergeSort, QuickSort e HeapSort é significativamente influenciado pela

distribuição dos dados. O QuickSort se mostrou o algoritmo mais eficiente em todos os cenários. O HeapSort, por sua vez, apresentou o pior desempenho em todos os casos, especialmente para dados ordenados ou parcialmente ordenados. Esses resultados corroboram com os estudos prévios da literatura, que indicam o QuickSort como um dos algoritmos de ordenação mais eficientes na prática.

Referências

ORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., and STEIN, C. (2002). *Algoritmos – Teoria e Prática. Rio de Janeiro*. Elsevier.