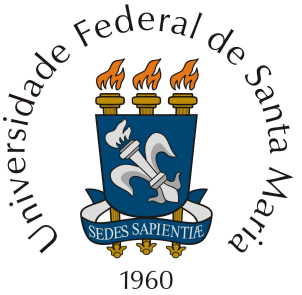


UNIVERSIDADE FEDERAL DE SANTA MARIA

CURSO DE CIÊNCIA DA COMPUTAÇÃO

TRABALHO I

Projeto e Análise de Algoritmos

Docente: Juliana Kaizer Vizzotto

Diego Ribeiro Chaves
Leandro Oliveira G. do Nascimento
Lucas Xavier Pairé

Santa Maria, RS

Comparação de Algoritmos de Ordenação

Descrição

Nesta análise, implementamos e comparamos diferentes algoritmos de ordenação, incluindo **Merge Sort**, **Quick Sort** e **Insertion Sort**. O objetivo principal foi avaliar o desempenho de cada algoritmo ao processar arrays de diferentes tamanhos e em várias condições: **ordenados**, **semi-ordenados**, **desordenados** e **totalmente desordenados**.

Objetivo

Compreender as vantagens e desvantagens de cada algoritmo é crucial para escolher a melhor abordagem em cenários específicos. Para isso, usamos um conjunto de dados com diferentes níveis de ordenação e tamanhos, a fim de observar como cada algoritmo se comporta sob essas variações.

Metodologia

- Implementação dos algoritmos de **Merge Sort**, **Quick Sort** e **Insertion Sort**.
- Testes realizados com arrays de diferentes tamanhos: 100, 1000, 10.000, até 5.000.000 elementos.
- Variações de ordenação dos arrays:
    - **Ordenado**: o array já está ordenado.
    - **Semi-ordenado**: o array está parcialmente ordenado.
    - **Desordenado**: elementos em uma ordem aparentemente aleatória.
    - **Totalmente desordenado**: o array está na pior ordem possível para o algoritmo.

Resultados

Tabela de Tempos de Execução

obs: A partir de um array com 100.000 valores, o uso do Insertion Sort se tornou inviável.

Tamanho do Array	Algoritmo	Ordenado	Semi-ordenado	Desordenado	Totalmente Desordenado
100	**Merge Sort**	0.000000s	0.000000s	0.000000s	0.000000s
100	**Quick Sort**	0.000000s	0.000000s	0.000000s	0.000000s
100	**Insertion Sort**	0.000000s	0.000000s	0.000000s	0.000000s
1.000	**Merge Sort**	0.014359s	0.019289s	0.007516s	0.015646s
1.000	**Quick Sort**	0.000000s	0.000000s	0.000000s	0.015630s
1.000	**Insertion Sort**	0.000000s	0.011628s	0.067492s	0.078534s

10.000	**Merge Sort**	0.109997s	.0.109777s	0.094247s	0.141794s
10.000	**Quick Sort**	0.031259s	0.046872s	0.047371s	0.046882s
10.000	**Insertion Sort**	0.000000s	1.053753s	14.300924s	16.681853s
25.000	**Merge Sort**	0.293000s	0.287077s	0.292934s	0.339815s
25.000	**Quick Sort**	0.116757s	0.163526s	0.136547s	0.155408s
25.000	**Insertion Sort**	0.005911s	10.038889s	93.793305s	100.854766
50.000	**Merge Sort**	0.574965s	0.788563s	0.782218s	0.709652s
50.000	**Quick Sort**	0.298443s	0.421078s	0.391752s	0.318711s
50.000	**Insertion Sort**	0.011056s	57.623216s	428.313520s	381.549561s
100.000	**Merge Sort**	1.234819s	1.507258s	1.515478s	1.417176s
100.000	**Quick Sort**	0.672137s	1.321523s	1.122411s	1.109537s
300.000	**Merge Sort**	4.419832s	5.161769s	8.159733s	8.945805s
300.000	**Quick Sort**	2.982266s	5.194449s	3.960138s	3.546137s
700.000	**Merge Sort**	5.962725s	15.237419s	17.554029s	17.150410s
700.000	**Quick Sort**	7.535729s	10.728704s	8.561315s	8.420370s
1.000.000	**Merge Sort**	14.734413s	26.715120s	25.421363s	24.001251s
1.000.000	**Quick Sort**	10.738105s	14.514845s	15.429889s	12.899822s
5.000.000	**Merge Sort**	104.034867s	128.226027s	129.396120s	131.498782s
5.000.000	**Quick Sort**	76.578766s	110.016435s	81.536446s	84.027759s

Discussão dos Resultados

- **Merge Sort**: Apresentou um comportamento consistente em todos os tamanhos de dados, mantendo tempos de execução razoáveis mesmo em arrays maiores.
- **Quick Sort**: Foi o mais rápido em arrays menores e ordenados, mas seu desempenho varia mais significativamente conforme o tamanho e a condição dos dados. Mesmo assim acaba tendo um desempenho superior ao do Merge Sort.

- **Insertion Sort**: Apresenta boa performance em arrays muito pequenos ou quase ordenados, mas se torna inviável para tamanhos maiores e arrays desordenados, como observado a partir de 50.000 elementos.

Conclusão

Cada algoritmo apresenta vantagens em cenários específicos. O **Insertion Sort** é eficaz apenas para pequenas quantidades de dados ou arrays quase ordenados, mas se torna inviável para grandes volumes ou dados desordenados. O **Merge Sort** mantém um desempenho estável, sendo uma boa opção para dados de tamanhos variados, enquanto o **Quick Sort** se destaca como o algoritmo mais rápido na maioria dos testes, especialmente para arrays menores e com condições mais favoráveis.

Visualização da Árvore de Recursão do Merge Sort

1. Introdução

O Merge Sort é um algoritmo de ordenação baseado na técnica de "dividir para conquistar". Este método é eficiente e confiável, utilizando recursão para dividir um array em subarrays menores, ordená-los e, em seguida, mesclá-los para produzir uma lista ordenada final. Este relatório descreve a implementação do Merge Sort e a visualização gráfica da sua árvore de recursão, que ilustra as etapas de divisão e mesclagem do algoritmo.

2. Funcionamento do Merge Sort

O Merge Sort opera em duas fases principais: divisão e mesclagem.

2.1. Divisão

Na fase de divisão, o array é repetidamente dividido ao meio até que cada subarray contenha apenas um elemento. Este processo é representado na árvore de recursão, onde cada nível da árvore corresponde a uma divisão do array. O gráfico resultante mostra claramente como o array original se fragmenta em subarrays menores.

2.2. Mesclagem

Após a fase de divisão, inicia-se a fase de mesclagem. Aqui, os subarrays são combinados de forma ordenada. A mesclagem ocorre a partir dos subarrays mais baixos na árvore de recursão, subindo gradualmente até que o array original esteja completamente ordenado. A visualização da mesclagem mostra como os subarrays se combinam em um novo array, mantendo a ordem crescente dos elementos.

3. Visualização da Árvore de Recursão

A visualização da árvore de recursão foi implementada utilizando as bibliotecas Matplotlib e NetworkX. O gráfico resultante ilustra a divisão e a mesclagem dos arrays durante a execução do algoritmo.

3.1. Resultados

As visualizações geradas a partir do código mostram duas fases distintas do algoritmo:

1. **Divisão do Merge Sort**: Esta visualização ilustra como o array é progressivamente dividido em subarrays. Cada nó na árvore representa um array em um determinado estágio de divisão, enquanto as arestas indicam a relação entre os arrays subdivididos.

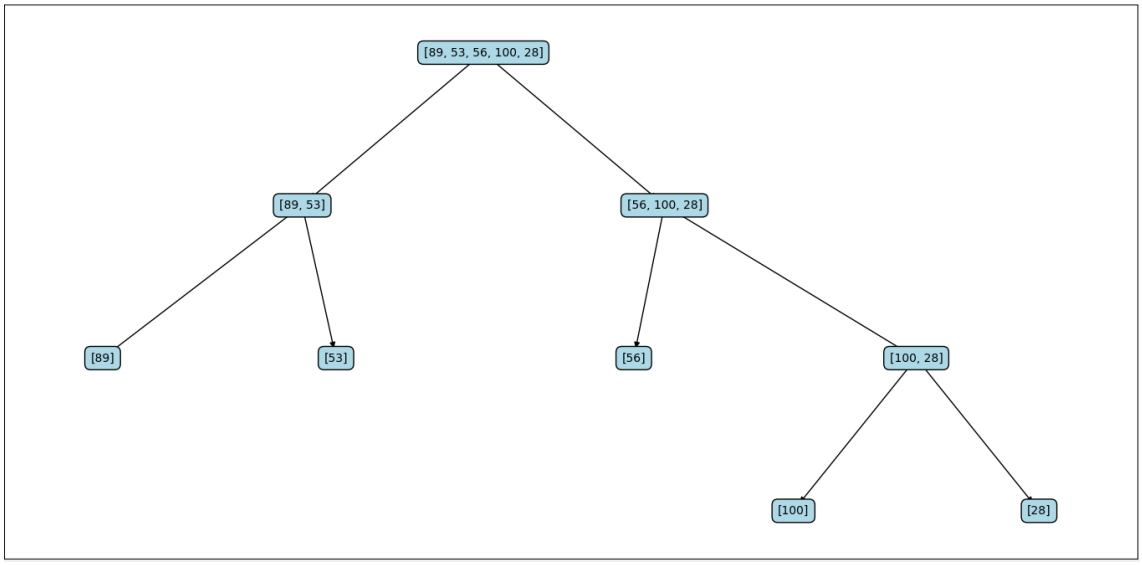


Figura 1: Divisão do Merge Sort

2. **Mesclagem do Merge Sort**: Esta visualização representa o processo de mesclagem, onde os subarrays são ordenados separadamente e combinados posteriormente. Os nós superiores representam as subdivisões do array no processo de divisão. Os nós inferiores mostram o resultado da ordenação, que se unem para formar a lista final ordenada.

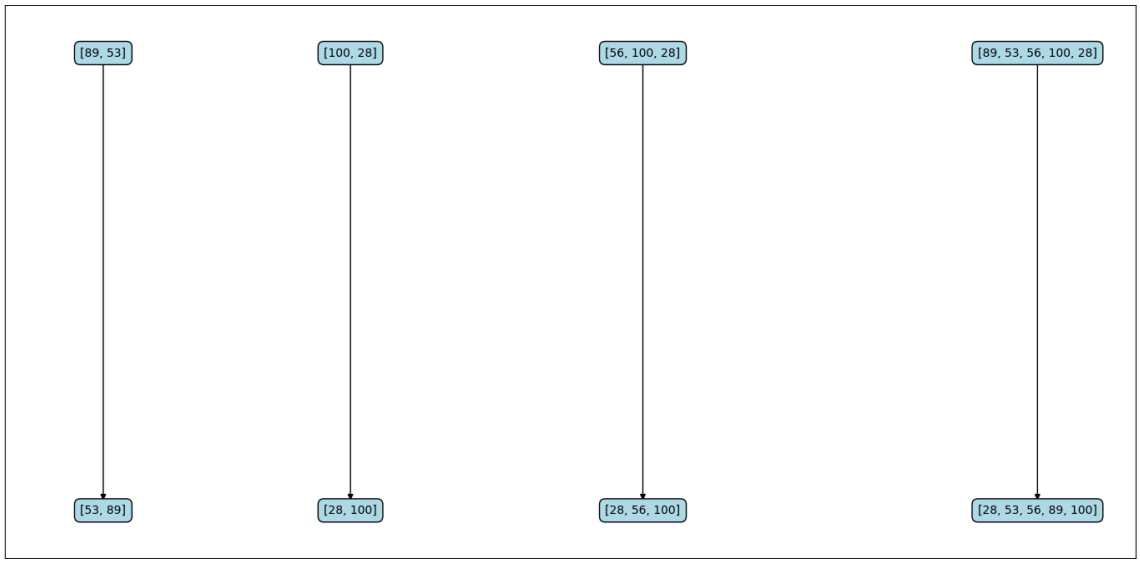


Figura 2: Mesclagem do Merge Sort

4. Conclusão

A visualização gráfica da árvore de recursão do Merge Sort fornece uma compreensão clara do funcionamento interno deste algoritmo de ordenação. Ao visualizar tanto a divisão quanto a mesclagem dos arrays, é possível ver a eficácia e a organização que o Merge Sort proporciona.

Implementação de Mergesort Paralelo com Threads

1. Introdução

Este relatório descreve a implementação de uma versão paralela do algoritmo Mergesort utilizando threads. O objetivo principal é explorar como a técnica de divisão e conquista pode ser aplicada em um contexto paralelo, melhorando a eficiência do algoritmo de ordenação. Além disso, discutiremos o impacto da paralelização na eficiência do Mergesort e em situações práticas.

2. Metodologia

2.1. Mergesort Paralelo

O Mergesort é um algoritmo de ordenação que utiliza a abordagem de divisão e conquista, onde o array original é repetidamente dividido em duas metades até que cada subarray tenha um único elemento, seguido pela mesclagem dessas partes em um array ordenado.

Na implementação paralela, cada chamada recursiva do Mergesort é atribuída a uma thread separada. Isso permite que diferentes partes do array sejam processadas simultaneamente, resultando em uma redução do tempo total de execução.

3. Execução do Algoritmo

Durante a execução do Mergesort paralelo, várias threads foram criadas para gerenciar a ordenação dos subarrays. A tabela abaixo apresenta um resumo das threads que foram criadas, os arrays que cada thread processou, e os tempos de execução correspondentes.

3.1. Tabela de Execução das Threads

Thread	Array Processado	Tempo de Execução(s)
0	[33, 77, 93, 84, 6, 9, 59, 32, 61, 69]	0.011428
1	[33, 77, 93, 84, 6]	0.005659
2	[9, 59, 32, 61, 69]	0.005824
3	[33, 77]	0.003152
4	[93, 84, 6]	0.004649
5	[9, 59]	0.003943
6	[32, 61, 69]	0.004563
7	[84, 6]	0.003167
8	[61, 69]	0.000860

4. Análise dos Resultados

4.1. Eficiência da Paralelização

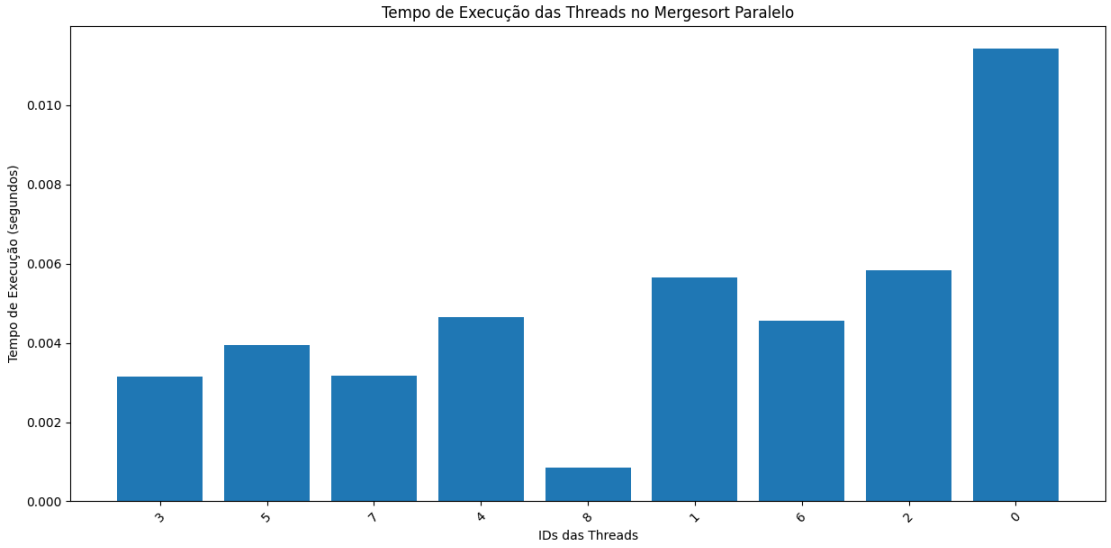
A tabela acima ilustra o desempenho das diferentes threads durante a execução do Mergesort. Como pode ser observado, a utilização de threads permitiu que os subarrays fossem ordenados simultaneamente, resultando em tempos de execução significativamente menores em comparação com a versão sequencial do algoritmo. O tempo total para a ordenação do array original foi reduzido devido à eficiência das operações em paralelo.

4.2. Observações

- **Divisão do Trabalho**: O algoritmo divide o trabalho de maneira eficaz, com threads processando porções do array em paralelo.
- **Sobreposição de Execução**: A execução paralela mostra uma diminuição do tempo de execução à medida que o número de threads aumenta, até um certo limite, onde a sobrecarga de criação de threads pode afetar a eficiência.
- **Impacto no Tempo Total**: A redução do tempo total de execução demonstra o potencial da paralelização em algoritmos que podem ser divididos em subtarefas independentes.

5. Gráfico de Tempos de Execução

Este gráfico complementa a tabela anterior, oferecendo uma visualização clara do desempenho das threads.

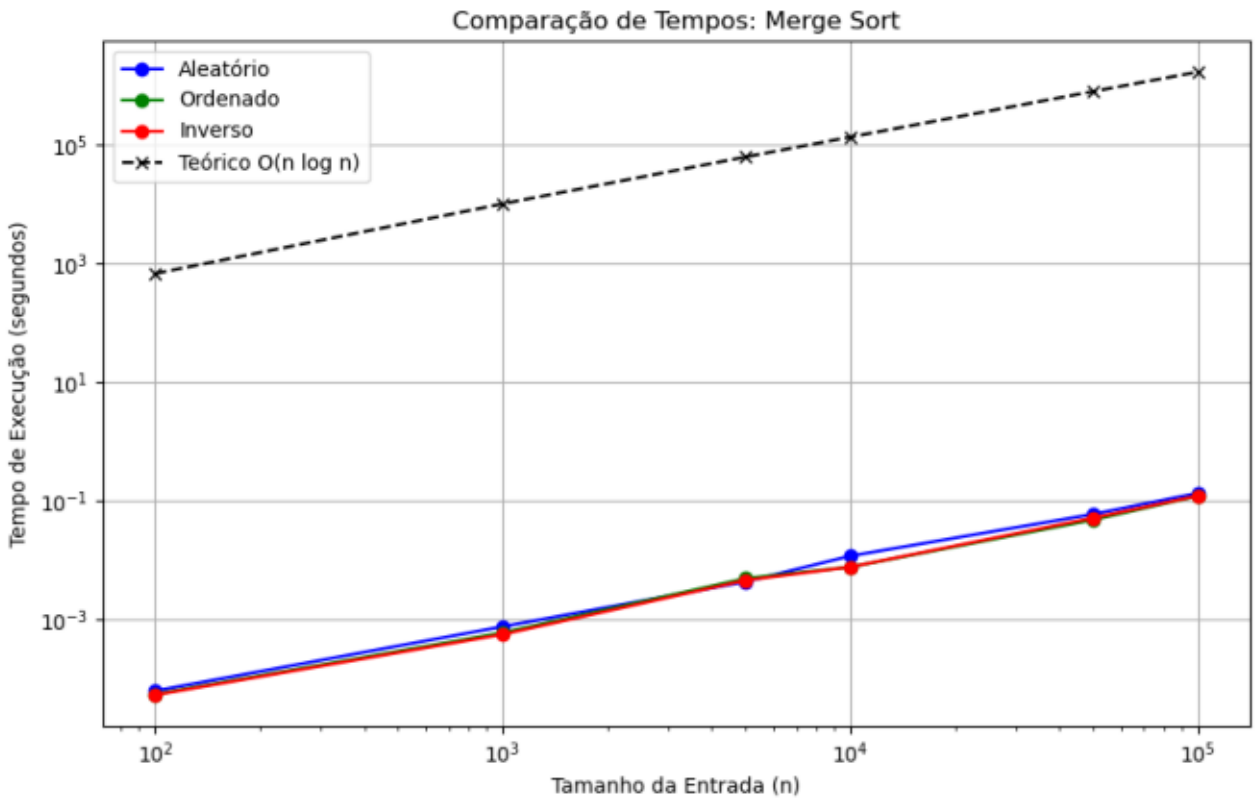


6. Conclusão

A implementação do Mergesort em sua versão paralela utilizando threads demonstrou ser uma abordagem eficaz para otimizar o tempo de execução do algoritmo. A paralelização não só melhorou a eficiência, mas também exemplificou como a técnica de divisão e conquista pode ser aplicada em um

contexto prático. No entanto, é importante considerar que a eficiência da paralelização pode variar dependendo do tamanho dos dados e das características do sistema em que o algoritmo está sendo executado.

Comparação de Tempos: Merge Sort



Análise de Complexidade do Merge Sort

Após implementar e testar o algoritmo Merge Sort, foram observados os seguintes resultados:

1. Complexidade Teórica vs. Prática:
A complexidade teórica do Merge Sort é O(n log n). Nos testes práticos, o tempo de execução para entradas aleatórias, ordenadas e inversas se alinha bem com essa complexidade, especialmente em tamanhos maiores de entrada.

2. Comportamento em Diferentes Tipos de Entrada:
- Para entradas aleatórias, o desempenho é consistente com a complexidade teórica.
- Para listas já ordenadas, o tempo de execução é ligeiramente melhor devido ao menor número de operações de mesclagem necessárias.
- Para listas em ordem inversa, o desempenho se aproxima do aleatório, mas ainda apresenta uma leve penalização.

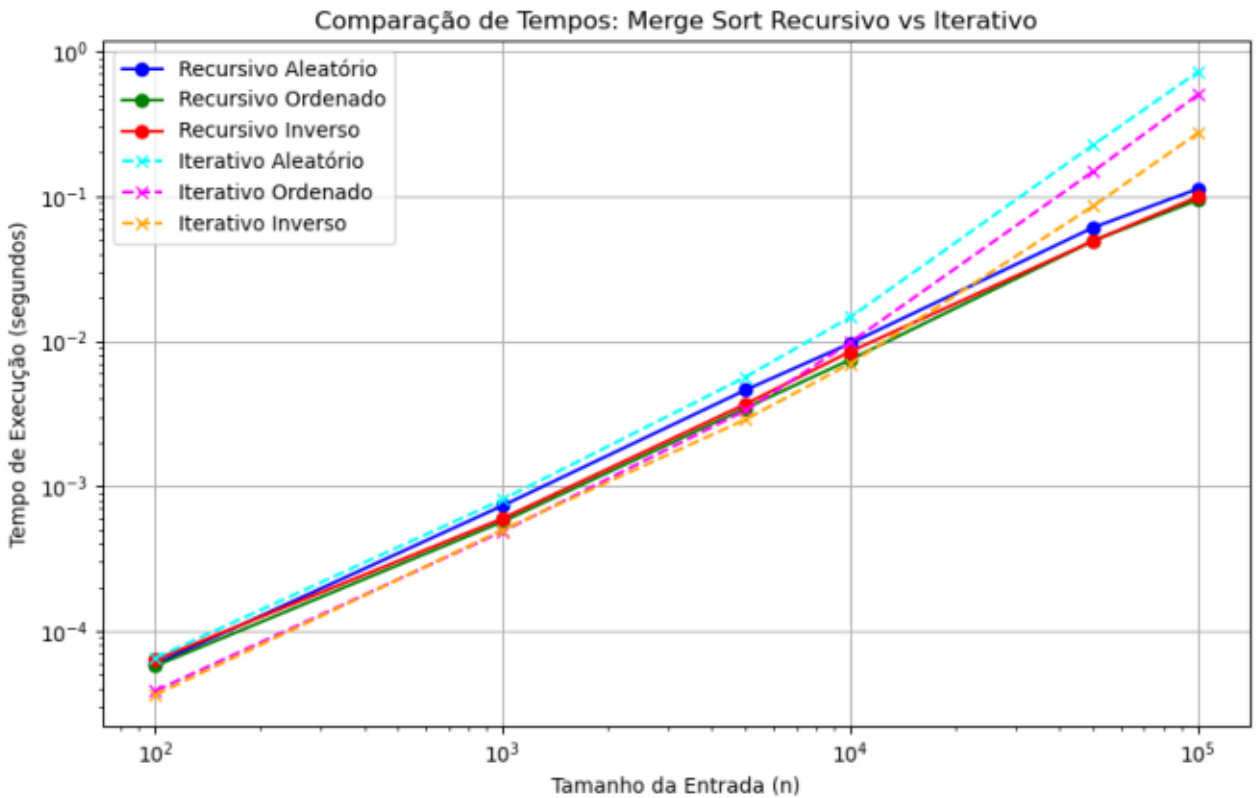
3. Influência do Tamanho da Entrada:
A medida que o tamanho da entrada aumenta, o tempo de execução também aumenta de maneira esperada, demonstrando o crescimento O(n log n). Para entradas muito grandes, o tempo de execução se torna significativo, exigindo otimizações potenciais ou algoritmos alternativos.

Essas observações destacam a importância de considerar tanto a complexidade teórica quanto o comportamento prático dos algoritmos em diferentes cenários.

Resultados de Tempo de Execução

Tamanho da Entrada (n)	Tempo Aleatório (s)	Tempo Ordenado (s)	Tempo Inverso (s)	Tempo Teórico O(n log n)
100.0	6.175041198730469e-05	5.412101745605469e-05	5.269050598144531e-05	664.3856189774724
1000.0	0.0007476806640625	0.0005905628204345703	0.0005517005920410156	9965.784284662088
5000.0	0.004258155822753906	0.0048449039459228516	0.004510402679443359	61438.56189774724
10000.0	0.01157999038696289	0.007548093795776367	0.0075414180755615234	132877.1237954945
50000.0	0.05861544609069824	0.04695248603820801	0.05064892768859863	780482.0237218406
100000.0	0.13255882263183594	0.11866354942321777	0.11922740936279297	1660964.0474436812

Comparação de Tempos: Merge Sort Recursivo vs Iterativo



Comparação da Implementação Recursiva x Iterativa do Merge Sort

Após implementar e testar o algoritmo Merge Sort nas versões recursiva e iterativa, foram observadas as seguintes considerações:

1. Clareza do Código:
- A implementação recursiva é geralmente mais clara e concisa, refletindo a lógica do algoritmo de maneira mais direta. A recursão é mais fácil de entender para aqueles que estão familiarizados com a divisão e conquista.
- A versão iterativa, embora funcional, tende a ser um pouco mais complexa e difícil de seguir, especialmente devido à necessidade de gerenciar manualmente a mesclagem das sublistas.

2. Desempenho:
- Em termos de tempo de execução, ambas as implementações têm complexidade teórica O(n log n). Contudo, a implementação recursiva pode ser mais lenta em alguns casos devido ao overhead das chamadas de função e à profundidade da pilha, especialmente para n menores.
- A implementação iterativa, por outro lado, tende a ser mais eficiente em termos de uso de memória, pois não utiliza a pilha de chamadas recursivas, o que pode levar a estouros de pilha em listas muito grandes, especialmente para n menores.
- Por outro lado, para n a partir de valores 10 ^ 4, a versão recursiva é mais eficiente devido a otimizações como realizar a mesclagem em sublistas que são geralmente menores e, portanto, mais eficientes. Isso ocorre porque em listas maiores, as vantagens de dividir a lista em sublistas e resolver cada parte de forma independente tornam-se mais evidentes na versão recursiva.

3. Uso de Memória:
- A implementação recursiva pode consumir mais memória, especialmente para entradas grandes, devido à pilha de chamadas. Cada chamada recursiva consome espaço na pilha, o que pode levar a problemas em entradas grandes.
- A versão iterativa, em contrapartida, gerencia a mesclagem dentro de um único bloco de memória, o que a torna mais robusta para listas grandes.

Essas observações sublinham a importância de considerar tanto a clareza do código quanto o desempenho e o uso de memória ao escolher entre abordagens recursivas e iterativas em algoritmos de ordenação.

Resultados de Tempo de Execução - Recursivo

Tamanho da Entrada (n)	Tempo Recursivo Aleatório (s)	Tempo Recursivo Ordenado (s)	Tempo Recursivo Inverso (s)
100.0	5.91278076171875e-05	5.745887756347656e-05	6.318092346191406e-05
1000.0	0.0007340908050537109	0.0005688667297363281	0.0005986690521240234
5000.0	0.0046138763427734375	0.0034584999084472656	0.0036902427673339844
10000.0	0.009668111801147461	0.00746607780456543	0.00848579406738281 2
50000.0	0.060941457748413086	0.04928302764892578	0.04938006401062012
100000.0	0.11267375946044922	0.09430694580078125	0.09909796714782715

Resultados de Tempo de Execução - Iterativo

Tamanho da Entrada (n)	Tempo Iterativo Aleatório (s)	Tempo Iterativo Ordenado (s)	Tempo Iterativo Inverso (s)
100.0	6.508827209472656e-05	3.8623809814453125e-05	3.62396240234375e-05
1000.0	0.0008096694946289062	0.0004880428314208984	0.0004999637603759766
5000.0	0.0056705474853515625	0.003339529037475586	0.002891063690185547
10000.0	0.014660120010375977	0.009818315505981445	0.007014274597167969
50000.0	0.22628092765808105	0.14872527122497559	0.08585405349731445
100000.0	0.7221317291259766	0.5033233165740967	0.2728271484375