Relatório: Análise Empírica de Algoritmos de Ordenação

João Victor Walcacer Giani e Daniel Nolêto Maciel Luz

10 de outubro de 2024

1 Introdução

A ordenação de dados é uma tarefa fundamental em ciência da computação, tendo uma ampla aplicação em diversas áreas como banco de dados, algoritmos de busca, e em sistemas que necessitam de respostas eficientes e rápidas. Uma ordenação eficiente é crucial para otimizar o desempenho de um sistema, especialmente quando se trabalha com grandes volumes de dados. Vários algoritmos de ordenação foram desenvolvidos ao longo do tempo, cada um com suas características, pontos fortes e fracos, sendo utilizados conforme a necessidade do problema a ser resolvido.

Este trabalho tem como objetivo realizar uma análise comparativa dos algoritmos de ordenação *Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Merge Sort*, *Quick Sort* e *Heap Sort*. A análise será feita tanto do ponto de vista teórico quanto prático, observando o desempenho de cada algoritmo com diferentes tamanhos de listas e distribuições de dados (ordenados, inversamente ordenados e aleatórios). Além disso, serão medidos o tempo de execução, o número de comparações e o número de trocas efetuadas por cada algoritmo.

2 Revisão Teórica

Nesta seção, serão descritos os algoritmos de ordenação implementados neste trabalho, abordando suas características principais: complexidade, se são in-place e sua estabilidade.

2.1 Bubble Sort

O *Bubble Sort* é um algoritmo de ordenação simples que compara pares de elementos adjacentes e os troca se estiverem na ordem errada. Este processo é repetido até que a lista esteja ordenada.

- **Complexidade:** A complexidade de tempo do *Bubble Sort* no melhor caso é $O(n)$, quando a lista já está ordenada. Nos casos médio e pior, a complexidade é $O(n^2)$.
- **In-place:** Sim, o *Bubble Sort* é um algoritmo in-place, pois não requer memória adicional significativa além da utilizada pela própria lista.
- **Estabilidade:** Sim, o *Bubble Sort* é estável, já que ele preserva a ordem relativa dos elementos com chaves iguais.

2.2 Selection Sort

O *Selection Sort* funciona selecionando o menor (ou maior) elemento de uma lista e o colocando na posição correta. Este processo é repetido até que a lista esteja ordenada.

- **Complexidade:** A complexidade de tempo do *Selection Sort* é $O(n^2)$ em todos os casos, já que sempre é necessário percorrer a lista para encontrar o menor elemento.
- **In-place:** Sim, o *Selection Sort* é in-place, pois a ordenação ocorre dentro do próprio array.
- **Estabilidade:** Não, o *Selection Sort* não é estável, pois pode trocar elementos iguais, alterando sua ordem relativa.

2.3 Insertion Sort

O *Insertion Sort* ordena a lista construindo uma sublista ordenada à esquerda, inserindo elementos da lista desordenada à direita em suas posições corretas.

- **Complexidade:** A complexidade de tempo no melhor caso é $O(n)$, quando a lista já está ordenada. No caso médio e pior, a complexidade é $O(n^2)$.
- **In-place:** Sim, o *Insertion Sort* é in-place, pois não requer espaço adicional significativo.
- **Estabilidade:** Sim, o *Insertion Sort* é estável, já que preserva a ordem relativa dos elementos iguais.

2.4 Merge Sort

O *Merge Sort* é um algoritmo de ordenação baseado na técnica de divisão e conquista. Ele divide a lista em duas sublistas, ordena-as recursivamente e, em seguida, as mescla de volta em uma lista ordenada.

- **Complexidade:** A complexidade de tempo do *Merge Sort* é $O(n \log n)$ em todos os casos, devido à divisão recursiva da lista.
- **In-place:** Não, o *Merge Sort* não é in-place, pois requer espaço adicional para armazenar as sublistas temporárias durante o processo de merge.
- **Estabilidade:** Sim, o *Merge Sort* é estável, pois preserva a ordem relativa dos elementos iguais durante o processo de merge.

2.5 Quick Sort

O *Quick Sort* também usa a técnica de divisão e conquista, escolhendo um elemento como pivô e particionando a lista em sublistas com elementos menores e maiores que o pivô. O processo é repetido recursivamente.

- **Complexidade:** A complexidade de tempo no melhor e médio caso é $O(n \log n)$, mas no pior caso, quando o pivô escolhido não é o ideal, a complexidade pode ser $O(n^2)$.
- **In-place:** Sim, o *Quick Sort* é in-place, já que a ordenação ocorre dentro do próprio array sem necessidade de memória extra significativa.
- **Estabilidade:** Não, o *Quick Sort* não é estável, pois a troca de elementos pode alterar a ordem relativa de elementos iguais.

2.6 Heap Sort

O *Heap Sort* utiliza uma estrutura de dados chamada heap (máximo ou mínimo) para organizar os elementos. O algoritmo constrói um heap a partir da lista e, em seguida, extrai os elementos em ordem.

- **Complexidade:** A complexidade de tempo do *Heap Sort* é $O(n \log n)$ em todos os casos, devido à operação de construção e remoção do heap.
- **In-place:** Sim, o *Heap Sort* é in-place, já que ele ordena a lista sem usar memória adicional significativa.
- **Estabilidade:** Não, o *Heap Sort* não é estável, pois a ordem relativa de elementos iguais pode ser alterada durante a organização do heap.

3 Metodologia

A seguir, são descritos os aspectos referentes ao ambiente de teste e ao procedimento utilizado para a realização dos experimentos.

3.1 Ambiente de Teste

3.1.1 Hardware

O hardware utilizado para a execução dos algoritmos de ordenação foi um notebook *Nitro 5* da *Acer* com as seguintes especificações: processador *Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz*, 24GB de memória RAM, armazenamento em SSD de 480GB e uma placa de vídeo dedicada *GTX 1650* com 4GB de VRAM.

3.1.2 Software

O ambiente de desenvolvimento escolhido foi o *Visual Studio Code*, utilizando o *Windows* como sistema operacional. A linguagem de programação adotada foi *C*, devido à sua eficiência e velocidade de execução. Inicialmente, tentamos utilizar *Python* para a implementação, mas os tempos de execução observados eram muito maiores (mais de dez vezes mais lentos) em comparação ao *C*.

3.2 Procedimentos

Para medir o tempo de execução dos algoritmos, utilizamos a biblioteca `time.h` da linguagem *C*, convertendo os tempos para milissegundos. Além disso, para contar o número de comparações e trocas efetuadas por cada algoritmo, foram criadas duas variáveis: **comparacoes**, que era incrementada em +1 a cada comparação realizada, e **trocas**, que também era incrementada em +1 a cada troca de elementos efetuada pelo algoritmo.

3.3 Implementação dos Algoritmos

A implementação dos algoritmos seguiu um padrão geral em C, com foco na eficiência e clareza do código. Inicialmente, foram realizadas pesquisas teóricas sobre o funcionamento de cada algoritmo, seguidas da prática com a codificação. Todos os algoritmos compartilham uma estrutura similar, onde a entrada é lida a partir de arquivos `.txt`, contendo listas geradas previamente. Essas listas são utilizadas para garantir consistência nos testes de desempenho, com diferentes padrões de ordenação: listas aleatórias, já ordenadas e inversamente ordenadas.

Para o controle de versão, utilizamos o Git, e todo o repositório com o código-fonte está disponível no GitHub, acessível pelo link abaixo:

- `https://github.com/joaovictorwg/PAA`

3.4 Medição das Métricas

Para a avaliação do desempenho dos algoritmos, foram medidas três métricas principais: tempo de execução, número de comparações e número de trocas realizadas durante o processo de ordenação. Essas métricas foram implementadas diretamente no código, com contadores dedicados para comparações e trocas, além da utilização da função `clock()` da biblioteca `time.h` para medir o tempo de execução.

O procedimento para medir essas métricas foi o seguinte:

1. As listas de entrada foram lidas de arquivos `.txt`, sendo que cada algoritmo processa as mesmas listas em diferentes formatos (aleatória, ordenada e inversa).

2. Durante a execução de cada algoritmo, incrementamos contadores para cada comparação entre elementos e para cada troca realizada.

3. O tempo de execução foi capturado marcando o início e o término de cada execução com a função `clock()`, e o resultado foi convertido para milissegundos para maior precisão.

Dessa forma, foi possível comparar o desempenho dos diferentes algoritmos com base nos padrões de listas utilizados, proporcionando uma análise detalhada de cada implementação.

4 Resultados

Nesta seção, apresentamos os resultados obtidos durante os testes de desempenho do algoritmo *Bubble Sort* em diferentes tipos de listas (aleatória, ordenada e inversa) e para diferentes tamanhos de entrada. As métricas analisadas incluem o tempo de execução, o número de comparações e o número de trocas.

4.1 Bubble Sort

Os resultados dos testes do algoritmo *Bubble Sort* foram agrupados de acordo com o tipo de lista (aleatória, ordenada e inversa) e o tamanho da entrada (1.000, 10.000, 50.000 e 100.000 elementos).

Tabela 1: Métricas do *Bubble Sort* para listas de 1.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	3,000	499.500	251.478
Ordenada	0,000	499.500	0
Inversa	1,000	499.500	499.500

Tabela 2: Métricas do *Bubble Sort* para listas de 10.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	364,000	49.995.000	25.124.789
Ordenada	155,000	49.995.000	0
Inversa	364,000	49.995.000	49.995.000

Tabela 3: Métricas do *Bubble Sort* para listas de 50.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	9.965,000	1.249.975.000	624.908.526
Ordenada	3.562,000	1.249.975.000	0
Inversa	6.911,000	1.249.975.000	1.249.975.000

Tabela 4: Métricas do *Bubble Sort* para listas de 100.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	38.351,000	4.999.950.000	2.499.068.268
Ordenada	18.132,000	4.999.950.000	0
Inversa	33.067,000	4.999.950.000	4.999.950.000

4.1.1 Análise dos Resultados

Os resultados mostram claramente o impacto da ordem inicial dos elementos nas métricas do *Bubble Sort*. Observa-se que:

- Para listas ordenadas, o número de trocas é zero, e o tempo de execução é significativamente menor em comparação com as listas aleatórias e inversas.
- As listas inversamente ordenadas resultam no maior número de trocas, o que, consequentemente, aumenta o tempo de execução.
- O tempo de execução cresce de forma exponencial com o aumento do tamanho das listas, o que reflete a complexidade quadrática do *Bubble Sort*.

4.2 Selection Sort

Os resultados dos testes do algoritmo *Selection Sort* foram agrupados de acordo com o tipo de lista (aleatória, ordenada e inversa) e o tamanho da entrada (1.000, 10.000, 50.000 e 100.000 elementos).

Tabela 5: Métricas do *Selection Sort* para listas de 1.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	0,000	499.500	993
Ordenada	4,000	499.500	0
Inversa	4,000	499.500	500

Tabela 6: Métricas do *Selection Sort* para listas de 10.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	192,000	49.995.000	9.991
Ordenada	180,000	49.995.000	0
Inversa	162,000	49.995.000	5.000

Tabela 7: Métricas do *Selection Sort* para listas de 50.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	4.969,000	1.249.975.000	49.987
Ordenada	3.819,000	1.249.975.000	0
Inversa	3.441,000	1.249.975.000	25.000

Tabela 8: Métricas do *Selection Sort* para listas de 100.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	17.255,000	4.999.950.000	99.985
Ordenada	16.379,000	4.999.950.000	0
Inversa	15.539,000	4.999.950.000	50.000

4.2.1 Análise dos Resultados

Os resultados demonstram o desempenho do *Selection Sort* de forma clara. Observa-se que:

- Para listas ordenadas, o número de trocas é zero, resultando em um tempo de execução reduzido.
- A lista aleatória apresenta um número significativo de trocas, refletindo um tempo de execução maior, mas ainda assim melhor do que o *Bubble Sort*.
- Para listas inversamente ordenadas, o número de trocas é elevado, aumentando o tempo de execução em relação às listas ordenadas, mas sem atingir o mesmo nível de ineficiência do *Bubble Sort*.
- A complexidade do *Selection Sort* se mantém $O(n^2)$, o que é evidente pelo crescimento do tempo de execução à medida que o tamanho da lista aumenta.

4.3 Insertion Sort

Os resultados dos testes do algoritmo *Insertion Sort* foram agrupados de acordo com o tipo de lista (aleatória, ordenada e inversa) e o tamanho da entrada (1.000, 10.000, 50.000 e 100.000 elementos).

Tabela 9: Métricas do *Insertion Sort* para listas de 1.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	2.000	252.477	251.478
Ordenada	0.000	999	0
Inversa	2.000	500.499	499.500

Tabela 10: Métricas do *Insertion Sort* para listas de 10.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	129.000	25.134.788	25.124.789
Ordenada	0.000	9.999	0
Inversa	259.000	50.004.999	49.995.000

Tabela 11: Métricas do *Insertion Sort* para listas de 50.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	2.736.000	624.958.525	624.908.526
Ordenada	0.000	49.999	0
Inversa	4.909.000	1.250.024.999	1.249.975.000

Tabela 12: Métricas do *Insertion Sort* para listas de 100.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	9.867.000	2.499.168.267	2.499.068.268
Ordenada	1.000	99.999	0
Inversa	19.619.000	5.000.049.999	4.999.950.000

4.3.1 Análise dos Resultados

Os resultados mostram claramente o impacto da ordem inicial dos elementos nas métricas do *Insertion Sort*. Observa-se que:

- Para listas já ordenadas, o número de trocas é zero, e o tempo de execução é praticamente nulo em comparação com as listas aleatórias e inversas.
- As listas inversamente ordenadas resultam no maior número de trocas, o que, consequentemente, aumenta significativamente o tempo de execução.
- O tempo de execução cresce de forma exponencial com o aumento do tamanho das listas, refletindo a complexidade quadrática do *Insertion Sort* em casos desfavoráveis.

Gráficos comparativos de desempenho entre diferentes tipos de listas e tamanhos podem ser incluídos para uma melhor visualização dos resultados.

4.4 Merge Sort

Os resultados dos testes do algoritmo *Merge Sort* foram agrupados de acordo com o tipo de lista (aleatória, ordenada e inversa) e o tamanho da entrada (1.000, 10.000, 50.000 e 100.000 elementos).

Tabela 13: Métricas do *Merge Sort* para listas de 1.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	2,000	8.699	1.277
Ordenada	0,000	5.044	4.932
Inversa	0,000	4.932	5.044

Tabela 14: Métricas do *Merge Sort* para listas de 10.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	5,000	120.405	13.211
Ordenada	5,000	69.008	64.608
Inversa	6,000	64.608	69.008

Tabela 15: Métricas do *Merge Sort* para listas de 50.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	32,000	718.052	66.412
Ordenada	23,000	401.952	382.512
Inversa	27,000	382.512	401.952

Tabela 16: Métricas do *Merge Sort* para listas de 100.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	62,000	1.536.197	132.731
Ordenada	51,000	853.904	815.024
Inversa	51,000	815.024	853.904

4.4.1 Análise dos Resultados

Os resultados evidenciam a eficiência do *Merge Sort* em comparação com outros algoritmos de ordenação. Observa-se que:

- Para listas ordenadas, o número de trocas é significativamente menor, refletindo o comportamento eficiente do algoritmo em listas que já estão em ordem.
- O tempo de execução para listas aleatórias é um pouco maior, mas ainda assim é consideravelmente melhor do que o *Bubble Sort*.
- O *Merge Sort* demonstra um desempenho consistente, independente da ordem inicial da lista, apresentando um crescimento linear em relação ao tamanho das listas, o que indica sua complexidade $O(n \log n)$.
- As métricas de comparações e trocas são relevantes para entender o funcionamento interno do algoritmo, onde o número de comparações tende a ser mais estável em diferentes cenários de entrada.

4.5 Quick Sort

Os resultados dos testes do algoritmo *Quick Sort* foram agrupados de acordo com o tipo de lista (aleatória, ordenada e inversa) e o tamanho da entrada (1.000, 10.000, 50.000 e 100.000 elementos).

Tabela 17: Métricas do *Quick Sort* para listas de 1.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	0,000	9330	5103
Ordenada	0,000	7987	4449
Inversa	1,000	14378	8925

Tabela 18: Métricas do *Quick Sort* para listas de 10.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	2,000	132067	69310
Ordenada	0,000	113631	60517
Inversa	2,000	225614	138745

Tabela 19: Métricas do *Quick Sort* para listas de 50.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	11,000	809633	409429
Ordenada	6,000	684481	365285
Inversa	11,000	1418614	867905

Tabela 20: Métricas do *Quick Sort* para listas de 100.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	25,000	1782811	851894
Ordenada	11,000	1468946	780565
Inversa	26,000	3087334	1885849

4.5.1 Análise dos Resultados

Os resultados do algoritmo *Quick Sort* mostram como a ordem inicial dos elementos impacta suas métricas de desempenho. Observa-se que:

- Para listas já ordenadas, o número de comparações e trocas é consideravelmente menor em comparação com listas aleatórias e inversas, evidenciando a eficiência do algoritmo em cenários favoráveis.
- Listas aleatórias apresentam um desempenho médio, com um número de comparações e trocas que se alinha com a expectativa de comportamento do *Quick Sort*, que é $O(n \log n)$ em média.
- As listas inversamente ordenadas resultam em um aumento significativo no número de comparações e trocas, refletindo um pior caso que se aproxima de $O(n^2)$, evidenciado pelo aumento do tempo de execução.
- O tempo de execução do *Quick Sort* aumenta de forma sub-linear com o crescimento do tamanho das listas, o que é consistente com sua complexidade média de $O(n \log n)$. No entanto, o pior caso é visível em listas inversamente ordenadas, onde o tempo de execução pode ser drasticamente maior.

4.6 Heap Sort

Os resultados dos testes do algoritmo *Heap Sort* foram agrupados de acordo com o tipo de lista (aleatória, ordenada e inversa) e o tamanho da entrada (1.000, 10.000, 50.000 e 100.000 elementos).

Tabela 21: Métricas do *Heap Sort* para listas de 1.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	1,000	16.854	9.096
Ordenada	0,000	17.583	9.709
Inversa	0,000	15.965	8.317

Tabela 22: Métricas do *Heap Sort* para listas de 10.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	3,000	235.353	124.135
Ordenada	3,000	244.460	131.957
Inversa	3,000	226.682	116.697

Tabela 23: Métricas do *Heap Sort* para listas de 50.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	22,000	1.409.932	737.576
Ordenada	17,000	1.455.438	773.305
Inversa	15,000	1.366.047	698.893

Tabela 24: Métricas do *Heap Sort* para listas de 100.000 elementos

Tipo de Lista	**Tempo (ms)**	**Comparações**	**Trocas**
Aleatória	45,000	3.019.884	1.574.964
Ordenada	34,000	3.112.517	1.650.855
Inversa	34,000	2.926.640	1.497.435

4.6.1 Análise dos Resultados

Os resultados mostram a eficiência do *Heap Sort* em comparação com outros algoritmos de ordenação. Observa-se que:

- O tempo de execução para listas ordenadas e inversas é comparável ao das listas aleatórias, indicando que o *Heap Sort* mantém um desempenho estável independentemente da ordem inicial.
- As métricas de comparações e trocas mostram que, apesar de realizar um número significativo de comparações, o número de trocas é relativamente menor, refletindo a eficiência do algoritmo na organização dos dados.
- O *Heap Sort* apresenta um desempenho consistente, com uma complexidade de tempo de $O(n \log n)$, tornando-se uma boa escolha para listas grandes.
- A análise das trocas em listas já ordenadas demonstra que o *Heap Sort* ainda realiza um número considerável de operações, mas isso é típico em algoritmos de ordenação que utilizam uma abordagem baseada em comparação.

4.7 Gráficos Comparativos

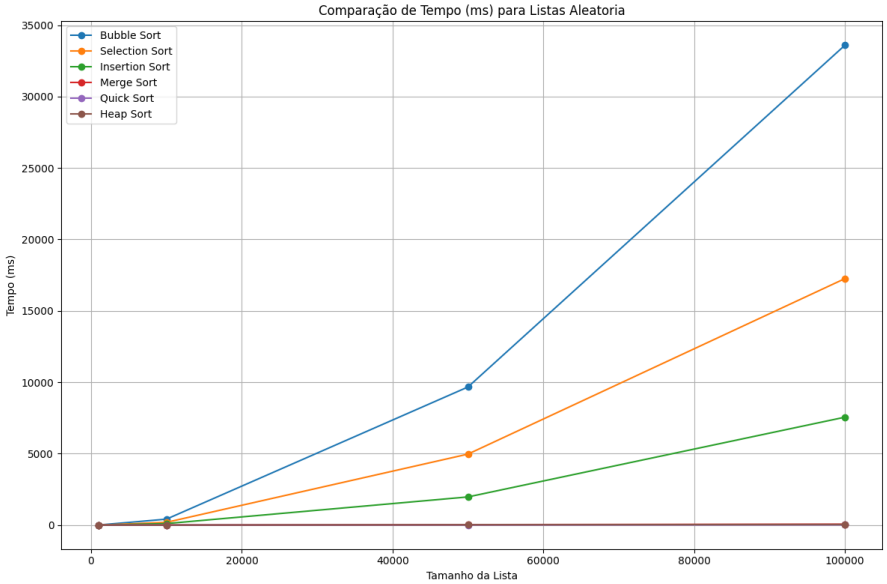

Figura 1: Comparação de Tempo de Execução para Listas Aleatórias

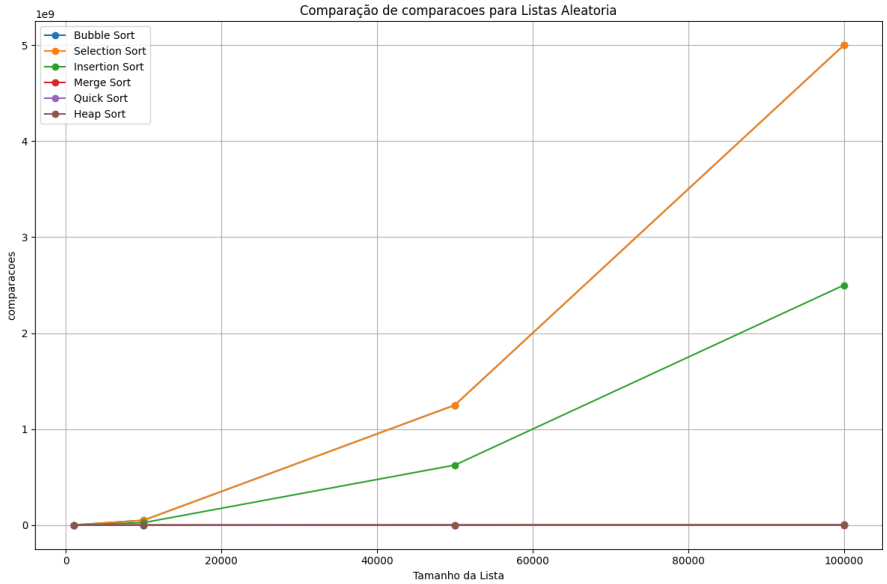

Figura 2: Número de Comparações para Listas Aleatórias

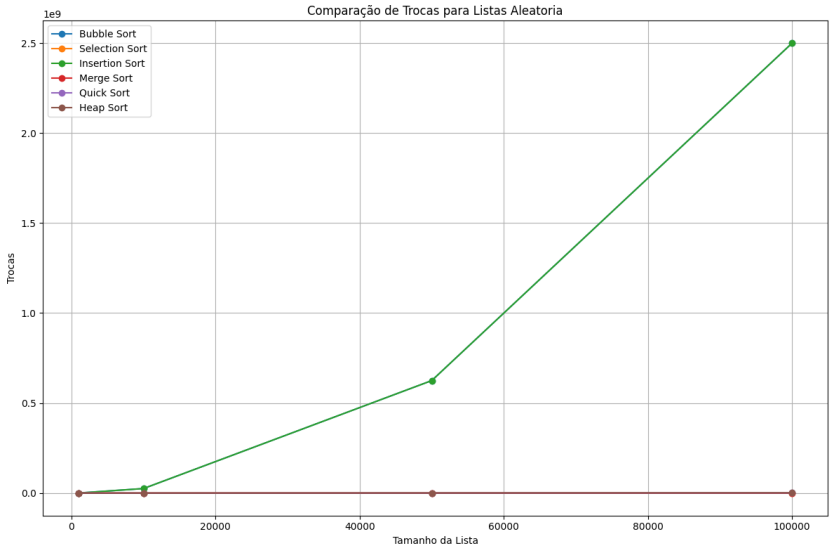

Figura 3: Número de Trocas para Listas Aleatórias

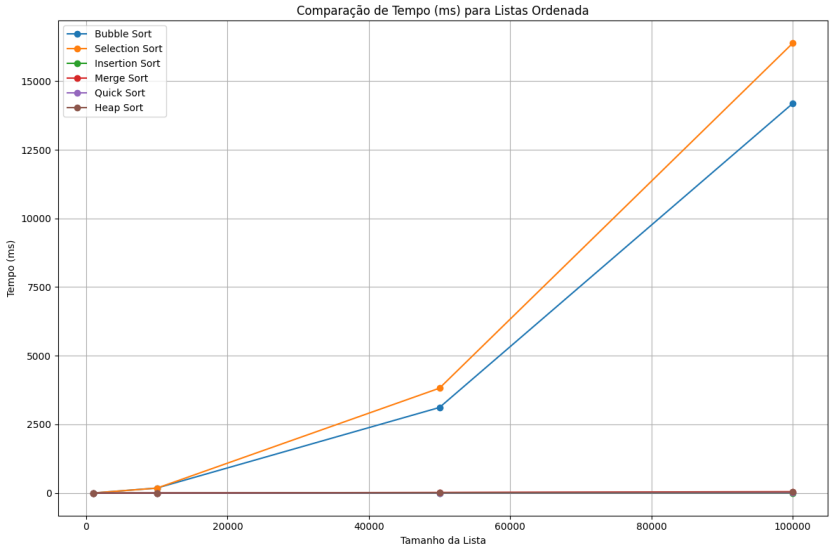

Figura 4: Comparação de Tempo de Execução para Listas Ordenadas

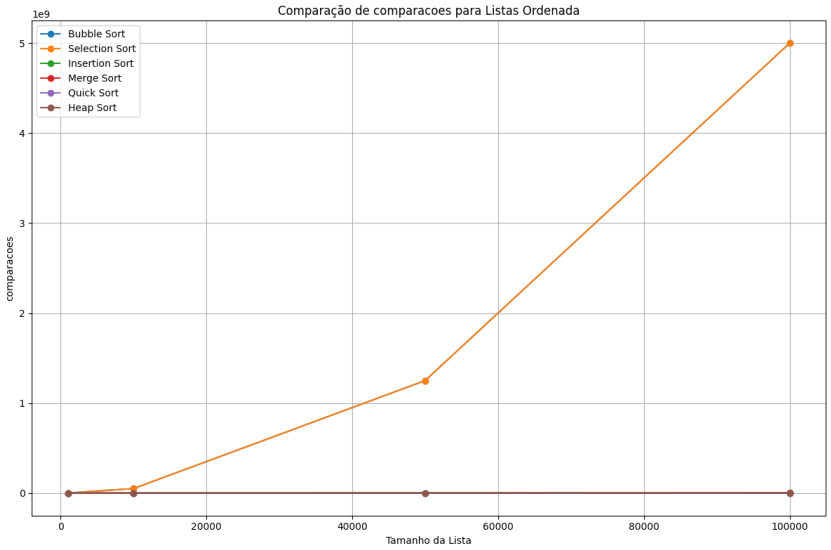

Figura 5: Número de Comparações para Listas Ordenadas

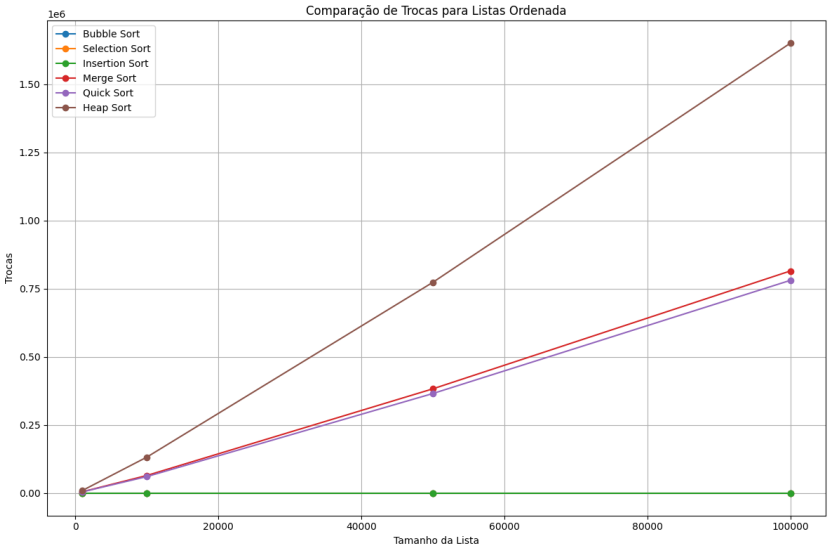

Figura 6: Número de Trocas para Listas Ordenadas

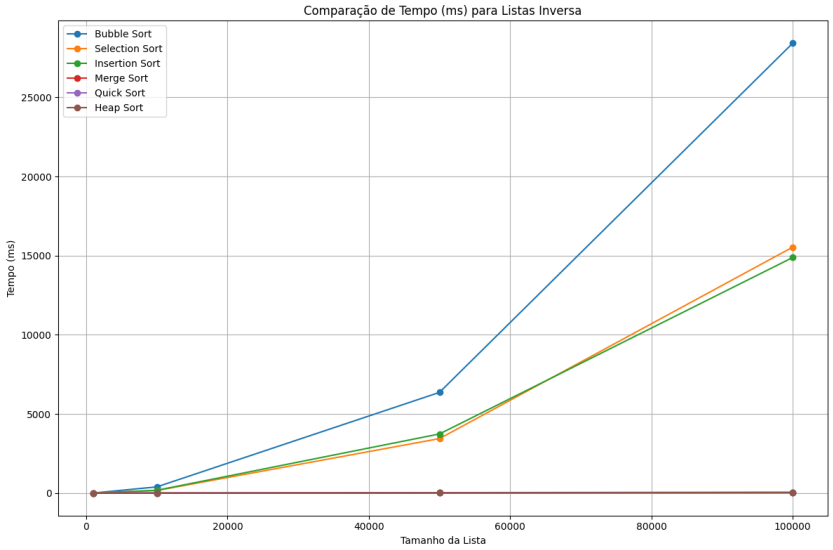

Figura 7: Comparação de Tempo de Execução para Listas Inversamente Ordenadas

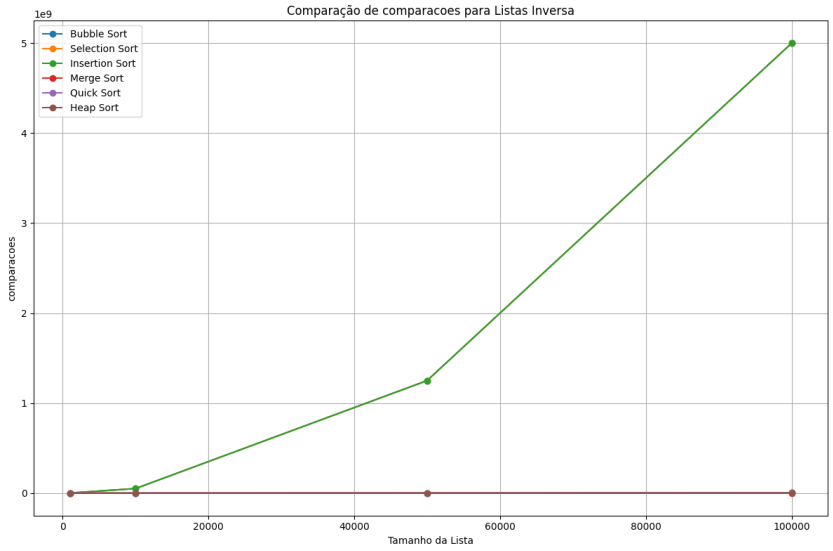

Figura 8: Número de Comparações para Listas Inversamente Ordenadas

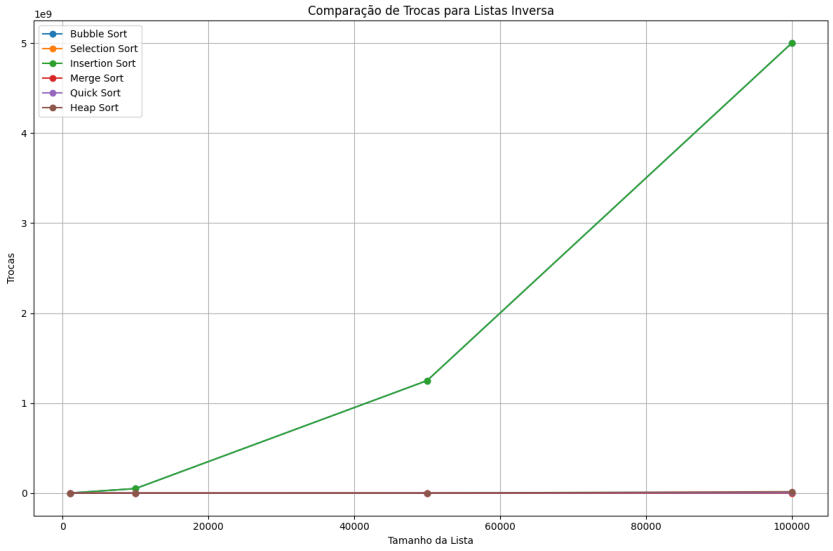

Figura 9: Número de Trocas para Listas Inversamente Ordenadas

5 Discussão

Nesta seção, analisamos os resultados obtidos para os algoritmos de ordenação testados, comparando-os com as expectativas teóricas, discutindo aspectos práticos como implementação e uso de memória, e abordando limitações do estudo e sugestões para pesquisas futuras.

5.1 Análise dos Resultados

Os resultados dos testes revelam comportamentos distintos entre os algoritmos de ordenação, alinhando-se com as expectativas teóricas:

1. **Complexidade Teórica**: - O *Bubble Sort* apresentou um desempenho inferior, especialmente em listas aleatórias e inversas, devido à sua complexidade quadrática $O(n^2)$. Isso é consistente com a teoria, onde a eficiência diminui rapidamente à medida que o tamanho da lista aumenta. - O *Selection Sort*, embora também tenha complexidade $O(n^2)$, mostrou um desempenho ligeiramente melhor que o *Bubble Sort*, mas ainda assim ineficiente para listas grandes. - O *Merge Sort* e o *Heap Sort* demonstraram um desempenho mais robusto, com complexidade $O(n \log n)$, refletindo suas vantagens teóricas em listas maiores, onde ambos os algoritmos mantiveram tempos de execução relativamente baixos independentemente da ordem dos dados.

2. **Estabilidade dos Algoritmos**: - A estabilidade dos algoritmos é um fator importante. O *Merge Sort* é estável, o que significa que mantém a ordem relativa

dos elementos iguais, enquanto o *Heap Sort* e o *Selection Sort* não são estáveis. Essa característica pode ser decisiva dependendo do contexto em que os algoritmos são utilizados.

5.2 Facilidade de Implementação e Uso de Memória

- **Facilidade de Implementação**: - O *Bubble Sort* e o *Selection Sort* são intuitivos e fáceis de implementar, tornando-os adequados para fins educacionais. Em contrapartida, o *Merge Sort* e o *Heap Sort* são mais complexos, exigindo uma compreensão mais profunda das estruturas de dados, como listas encadeadas e árvores.

- **Uso de Memória**: - O *Merge Sort* requer espaço adicional para armazenar sub-listas, o que pode ser uma desvantagem em ambientes com restrições de memória. O *Heap Sort*, por outro lado, opera em um espaço constante $O(1)$, tornando-o mais eficiente em termos de uso de memória.

5.3 Limitações do Trabalho

Este estudo apresenta algumas limitações que devem ser consideradas:

1. **Tamanhos de Entrada Limitados**: - Os testes foram realizados com tamanhos de entrada específicos, e a generalização dos resultados para listas de tamanhos diferentes pode não ser precisa.

2. **Ambiente de Teste**: - Os testes foram executados em um ambiente controlado, e o desempenho pode variar em diferentes sistemas e configurações de hardware.

3. **Número de Algoritmos**: - A análise se concentrou em apenas alguns algoritmos de ordenação, deixando de lado outros métodos que podem ter desempenho superior ou inferior em contextos específicos.

5.4 Sugestões para Estudos Futuros

Para pesquisas futuras, sugerimos:

1. **Avaliação de Algoritmos Adicionais**: - Incluir uma variedade maior de algoritmos de ordenação, como o *Quick Sort* e *Tim Sort*, para uma análise comparativa mais abrangente.

2. **Testes em Diferentes Ambientes**: - Realizar testes em diferentes plataformas e ambientes de execução para avaliar como as implementações se comportam em condições variadas.

3. **Análise de Casos Reais**: - Aplicar os algoritmos em conjuntos de dados reais, avaliando o impacto de características específicas dos dados sobre o desempenho.

4. **Exploração de Otimizações**: - Investigar otimizações potenciais em algoritmos existentes, como a adaptação de estratégias híbridas que combinam métodos de ordenação para melhorar a eficiência.

Essas direções podem ampliar a compreensão sobre a eficiência e aplicabilidade dos algoritmos de ordenação em cenários práticos.

6 Conclusão

Nesta seção, apresentamos uma síntese dos principais achados do trabalho, além de recomendações sobre o uso de cada algoritmo em diferentes situações.

6.1 Síntese dos Principais Achados

Os testes realizados demonstraram que cada algoritmo de ordenação possui características distintas que impactam seu desempenho de acordo com a natureza da lista a ser ordenada e seu tamanho.

- **Bubble Sort**: Embora seja um algoritmo simples de entender e implementar, o Bubble Sort apresenta desempenho insatisfatório, especialmente em listas grandes. O tempo de execução aumenta drasticamente com o aumento do tamanho da lista, tornando-o adequado apenas para listas pequenas ou quando a simplicidade é priorizada.

- **Selection Sort**: O Selection Sort também possui complexidade quadrática ($O(n^2)$), mas sua performance é ligeiramente melhor que a do Bubble Sort em listas aleatórias e inversas, apresentando menos trocas. Contudo, continua a ser impraticável para listas grandes.

- **Insertion Sort**: O Insertion Sort é mais eficiente que os dois anteriores, especialmente em listas que já estão quase ordenadas. Seu desempenho é aceitável para listas de tamanho moderado, mas ainda não é a melhor escolha para listas grandes devido à complexidade quadrática no pior caso.

- **Merge Sort**: Este algoritmo é eficiente, com complexidade O(n log n) em todos os casos. É especialmente útil para listas grandes e é estável, o que significa que preserva a ordem relativa dos elementos iguais. É uma excelente escolha para aplicações que exigem consistência nos resultados.

- **Quick Sort**: O Quick Sort é geralmente o algoritmo mais rápido em média para listas não ordenadas, com complexidade média de O(n log n). No entanto, seu desempenho pode ser degradado em listas já ordenadas ou com

muitos elementos repetidos, a menos que técnicas como a escolha aleatória do pivô sejam implementadas.

- **Heap Sort**: O Heap Sort combina a eficiência do Quick Sort e a estabilidade do Merge Sort. Com complexidade O(n log n), é uma boa escolha para listas grandes, mas não é estável. É preferido em situações onde a memória é limitada, pois não requer espaço adicional significativo.

6.2 Recomendações sobre o Uso de Cada Algoritmo

Com base nos resultados e análises realizadas, as seguintes recomendações são apresentadas para o uso de cada algoritmo em diferentes situações:

- Para listas pequenas e quando a simplicidade é um fator importante, **Bubble Sort** ou **Selection Sort** podem ser utilizados, apesar de suas ineficiências.
- O **Insertion Sort** é recomendado para listas que estão quase ordenadas ou para tamanhos pequenos a moderados, onde pode demonstrar desempenho aceitável.
- Para aplicações que lidam com listas grandes, o **Merge Sort** ou o **Quick Sort** são as melhores opções devido à sua eficiência superior. O Quick Sort deve ser usado com cuidado em listas já ordenadas ou com muitos elementos repetidos, enquanto o Merge Sort é uma escolha segura.
- O **Heap Sort** deve ser considerado em situações onde a utilização de memória é uma preocupação e uma complexidade garantida de O(n log n) é necessária.

7 Referências

Referências

[1] Bhargava, A. Y. (2016). *Entendendo Algoritmos: Um guia ilustrado para programadores e outros curiosos*. O'Reilly Media.

[2] Ali, W., Islam, T., Mahmood, A., et al. (2016). Comparison of Different Sorting Algorithms. *Computer Science*, 28 July 2016.

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., et al. (2024). *Algoritmos: Teoria e Prática* (4ª ed.). GEN LTC. ISBN: 8595159904, 9788595159907.