UNIVERSIDADE FEDERAL DO TOCANTINS

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Danilo Maciel
Micael Ribeiro

Uma Rápida Análise de Algoritmos de Ordenação

Palmas, TO
2025

Resumo

Nosso objetivo aqui é comparar o desempenho dos seis principais algoritmos de ordenação — Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort — por meio de testes com listas de diferentes tamanhos (1.000 a 100.000 elementos) e distribuições (ordenadas, inversas e aleatórias). Foram analisados tempo de execução, número de comparações e trocas. Os resultados indicam melhor desempenho do Heap Sort e Merge Sort em listas grandes, enquanto os algoritmos simples tiveram baixo desempenho

Palavras-chave: Bubble Sort. Selection Sort. Insertion Sort. Merge Sort. Quick Sort. Heap Sort. Tempo de Execução. Ordenação.

Abstract

Our goal here is to compare the performance of the six main sorting algorithms — Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Heap Sort — through tests with lists of varying sizes (from 1,000 to 100,000 elements) and distributions (sorted, reverse, and random). Execution time, number of comparisons, and swaps were analyzed. The results indicate better performance by Heap Sort and Merge Sort on large lists, while the simpler algorithms showed lower performance.

Keywords: Sorting Algorithms. Empirical Analysis. Execution Time.

Capítulo 1

Introdução

A ordenação de dados é essencial para o funcionamento de sistemas no mundo moderno, impactando diretamente o desempenho de aplicações que lidam com grandes volumes de dados. A escolha do algoritmo adequado depende tanto do tamanho quanto da disposição dos dados, como listas ordenadas, invertidas ou aleatórias. Esta pesquisa analisa empiricamente seis algoritmos — Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort — considerando entradas de 1.000 a 100.000 elementos em diferentes distribuições. Os testes avaliam tempo de execução, número de comparações e trocas, permitindo comparar o desempenho e a eficiência de cada método em diferentes cenários.

Capítulo 2

Visão geral

2.1 Bubble Sort

Embora seja um dos algoritmos de ordenação mais fáceis de entender, o Bubble Sort apresenta um trade-off entre simplicidade e eficiência. Sua abordagem de comparar e permutar vizinhos resulta em uma complexidade de tempo de $O(n^2)$ para o caso médio e pior, limitando sua aplicabilidade em conjuntos de dados extensos. No entanto, sua performance melhora significativamente para O(n) em listas já ordenadas.

2.2 Selection Sort

Diferentemente do Bubble Sort, cuja performance varia conforme a entrada, o Selection Sort apresenta uma complexidade de tempo uniforme de $O(n^2)$. Seu funcionamento envolve a seleção contínua do menor ou maior item não ordenado e sua subsequente movimentação para a seção ordenada da lista.

2.3 Insertion Sort

O Insertion Sort ordena como organizar cartas, inserindo cada elemento na posição correta da sublista ordenada à esquerda. Simples de implementar e eficiente para listas pequenas ou quase ordenadas (O(n) no melhor caso). É um algoritmo in-place e estável. Contudo, para listas grandes e aleatórias, sua complexidade é $O(n^2)$, tornando-o menos eficiente que outros algoritmos avançados. Ideal para poucos dados ou inserção contínua.O Insertion Sort ordena como organizar cartas, inserindo cada elemento na posição correta da sublista ordenada à esquerda. Simples de implementar e eficiente para listas pequenas ou quase ordenadas (O(n) no melhor caso). É um algoritmo in-place e estável. Contudo, para listas grandes e aleatórias, sua complexidade é $O(n^2)$, tornando-o menos eficiente que outros algoritmos avançados. Ideal para poucos dados ou inserção contínua.

2.4 Merge Sort

- **Tipo:** Algoritmo de ordenação por divisão e conquista.
- **Funcionamento:** Divide recursivamente o array em duas metades até que cada subarray tenha um único elemento.

- **Combinação:** As sublistas são então fundidas (merge) em ordem crescente.
- **Complexidade:** $O(n \log n)$ em todos os casos (melhor, médio e pior).
- **Estável:** Sim – mantém a ordem relativa dos elementos iguais.
- **Espaço extra:** Requer espaço adicional proporcional a n para o array temporário.

2.5 Quick Sort

- **Tipo:** Algoritmo de ordenação por divisão e conquista.
- **Funcionamento:** Escolhe um pivô, particiona os elementos menores à esquerda e maiores à direita.
- **Recursão:** Aplica o mesmo processo recursivamente às sublistas.
- **Complexidade:** Melhor e médio caso: $O(n \log n)$; pior caso: $O(n^2)$.
- **Estável:** Não – pode trocar a ordem de elementos iguais.
- **Espaço extra:** $O(\log n)$ no caso médio (devido à pilha de recursão).

2.6 Heap Sort

- **Tipo:** Algoritmo de ordenação baseado em estrutura de heap binário.
- **Funcionamento:** Constrói um heap máximo, depois extrai repetidamente o maior elemento.
- **Etapas principais:** Construção do heap e ordenação por remoção da raiz.
- **Complexidade:** Sempre $O(n \log n)$, em todos os casos.
- **Estável:** Não – pode trocar a ordem de elementos iguais.
- **Espaço extra:** $O(1)$ – feito in-place, sem uso adicional de memória.

Capítulo 3

Metodologia

3.1 Hardware e software

- Notebook Dell Core i5 12500H
- 16 GB RAM DDR5
- Windows 11 Pro
- Ambiente de desenvolvimento: VS Code
- Linguagem: Python 3.12

3.2 Implementação dos Algoritmos

Os seis algoritimos de ordenação: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort foram implementados em linguagem python e de forma individual, para que o comportamento de cada um seja analisado de maneira independente.

Descrição dos Algoritmos:

1. **Bubble Sort:**
    - A cada iteração, o algoritmo compara dois elementos adjacentes e os troca de lugar se estiverem fora de ordem. O processo é repetido até que a lista esteja completamente ordenada.
    - **Complexidade:** O(n^2) no pior caso e liner (n) no melhor caso
2. **Selection Sort:**
    - O algoritmo encontra o menor elemento da lista e o move para a primeira posição. Em seguida, repete o processo para o restante da lista até que ela esteja ordenada.
    - **Complexidade:** O(n^2) em todos os casos
3. **Insertion Sort:**

- O algoritmo percorre a lista, inserindo cada elemento em sua posição correta, movendo os elementos maiores para a direita conforme necessário.
- **Complexidade:** O(n^2)no pior caso e linear (n) no melhor

4. **Merge Sort:**
   - Merge Sort é um algoritmo de ordenação do tipo divide e conquista, que divide recursivamente a lista em sublistas até que cada uma contenha um único elemento, e então as combina em ordem. Ele é estável e eficiente para grandes volumes de dados.
   - **Complexidade:** O($n \log n$) sempre
5. **Quick Sort:**
   - O algoritmo seleciona um pivô, particiona a lista em dois grupos (menores e maiores que o pivô), e ordena recursivamente cada parte.
   - **Complexidade:** O($n \log n$) em média, e O(n^2) no pior caso, dependendo da escolha do pivô.
6. **Heap Sort:**
   - Heap Sort é um algoritmo de ordenação que transforma a lista em uma estrutura de heap (geralmente um max-heap) e então extrai o maior elemento repetidamente para ordenar os dados. É eficiente e não utiliza espaço adicional significativo.
   - **Complexidade:** O($n \log n$) sempre

No python usamos a classe SortMetrics() para salvar o número de comparações e trocas realizadas por cada um dos algoritmos

3.3 Procedimento para Medir Desempenho

Os dados coletados durante a execução são:

- **Tempo de execução:** Em Segundos.
- **Número de comparações:** Quantidade de Comparações na execução.
- **Número de trocas:** Quantidade de trocas durante a execução.

Medição

Para avaliar o desempenho dos algoritmos de ordenação, implementamos um sistema automatizado que testa cada algoritmo de ordenação em diferentes condições de entrada. O procedimento seguido foi:

1. Implementamos os três algoritmos de ordenação (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort) em Python.

2. Criamos um algoritmo de testes que realiza a execução de todos os algoritmos de ordenação automaticamente. Para cada algoritmo, o sistema realiza os testes em listas com diferentes tamanhos (1.000, 10.000, 50.000 e 100.000 elementos) e em três tipos de listas: ordenada, inversa e aleatória.

3. Durante a execução de cada teste, o tempo de execução foi medido, juntamente com o número de comparações e o número de trocas realizadas pelo algoritmo. A cada execução, os resultados eram registrados e exibidos para análise.

Métricas

Durante os testes, as seguintes métricas foram coletadas para avaliar o desempenho dos algoritmos:

- **Tempo de execução:** O tempo total de execução do algoritmo, registrado em segundos ou minutos, dependendo da duração do teste.
- **Número de comparações:** O número de comparações realizadas entre elementos da lista durante a execução do algoritmo.
- **Número de trocas:** O número de vezes em que dois elementos foram trocados de posição durante a execução do algoritmo.

Essas métricas nos permitiram analisar a eficiência dos algoritmos em diferentes cenários de entrada, como listas já ordenadas, listas inversamente ordenadas e listas aleatórias.

Capítulo 4

Resultados

4.1 Listas Aleatórias

A análise dos resultados com listas aleatórias evidencia os pontos fortes e fracos dos algoritmos testados:

- **Insertion Sort** tem desempenho razoável apenas para listas pequenas, mas degrada rapidamente à medida que o tamanho aumenta, devido à sua complexidade quadrática. Ele realiza muitas comparações e trocas.
- **Bubble Sort** apresenta comportamento semelhante ao Insertion Sort, com desempenho fraco em listas grandes. Ele requer muitas passagens e trocas, tornando-o pouco eficiente nesse cenário.
- **Selection Sort** também sofre com sua complexidade $O(n^2)$, mas realiza menos trocas do que o Bubble Sort, mantendo um desempenho ligeiramente mais estável, embora ainda inadequado para listas grandes.
- **Merge Sort** mostra excelente desempenho em listas aleatórias, sustentado por sua complexidade $O(n \log n)$ e abordagem de divisão e conquista. Ele mantém tempos baixos mesmo em listas grandes, embora use espaço adicional.
- **Quick Sort** destaca-se em listas aleatórias, beneficiando-se de uma escolha adequada de pivô. Ele é frequentemente o mais rápido nesse contexto, combinando boa eficiência com um uso moderado de memória.
- **Heap Sort** também apresenta bom desempenho em listas aleatórias, com tempos consistentes. Embora faça mais trocas internas que o Merge Sort, ele tem a vantagem de ser um algoritmo in-place.

Algoritmo	Tipo de Lista	Tamanho da Lista	Tempo de Execução	Comparações	Trocas
Bubble Sort	Aleatória	1.000	0.065212 segundos	499.500	247.083
Bubble Sort	Aleatória	10.000	6.903174 segundos	49.995.000	25.156.840
Bubble Sort	Aleatória	50.000	3.095087 minutos	1.249.975.000	623.876.606
Bubble Sort	Aleatória	100.000	13.588555 minutos	4.999.950.000	2.492.083.301
Selection Sort	Aleatória	1.000	0.033453 segundos	499.500	994
Selection Sort	Aleatória	10.000	3.402797 segundos	49.995.000	9.992
Selection Sort	Aleatória	50.000	1.822204 minutos	1.249.975.000	49.985
Selection Sort	Aleatória	100.000	10.288943 minutos	4.999.950.000	99.989
Insertion Sort	Aleatória	1.000	0.040179 segundos	249.011	248.022
Insertion Sort	Aleatória	10.000	4.130067 segundos	25.046.973	25.036.983
Insertion Sort	Aleatória	50.000	1.687712 minutos	626.323.601	626.273.609
Insertion Sort	Aleatória	100.000	7.109576 minutos	2.503.004.459	2.502.904.470

Figura 4.1: Distribuição Aleatória Bubble Sort, Selection Sort e Insertion Sort

Algoritmo	Tipo de Lista	Tamanho da Lista	Tempo de Execução	Comparações	Trocas
Merge Sort	Aleatória	1,000	0.001898 ms	8,732	9,976
Merge Sort	Aleatória	10,000	0.019768 ms	120,484	133,616
Merge Sort	Aleatória	50,000	0.115451 ms	718,153	784,464
Merge Sort	Aleatória	100,000	0.247682 ms	1,536,395	1,668,928
Quick Sort	Aleatória	1,000	0.001691 ms	10,790	5,997
Quick Sort	Aleatória	10,000	0.012173 ms	151,882	81,502
Quick Sort	Aleatória	50,000	0.083304 ms	977,737	500,726
Quick Sort	Aleatória	100,000	0.174214 ms	2,004,505	1,091,522
Heap Sort	Aleatória	1,000	0.003029 ms	16,860	9,088
Heap Sort	Aleatória	10,000	0.028497 ms	235,232	124,135
Heap Sort	Aleatória	50,000	0.161106 ms	1,409,419	737,278
Heap Sort	Aleatória	100,000	0.378695 ms	3,019,867	1,575,184

Figura 4.2: Distribuição Aleatória Merge Sort, Quick Sort, Heap Sort

4.2 Listas Ordenadas

Insertion Sort é o algoritmo que apresenta o melhor desempenho nesse cenário. Como ele percorre a lista apenas verificando se os elementos estão em ordem e não realiza trocas desnecessárias, seu tempo de execução é extremamente baixo, mesmo em listas com grande número de elementos.

Merge Sort também apresenta desempenho muito bom em listas ordenadas. Apesar de sempre operar com complexidade $O(n \log n)$, o número de comparações e trocas permanece relativamente baixo, garantindo uma execução eficiente.

Quick Sort mostra um desempenho razoável, mas inferior ao Merge Sort e ao Insertion Sort. Ele depende fortemente da escolha do pivô e, em listas ordenadas, pode realizar mais comparações e trocas do que o necessário.

Heap Sort mantém um desempenho estável, mas realiza muitas comparações e trocas desnecessárias, pois sempre reconstrói a heap independentemente da ordem inicial da lista, o que gera uma sobrecarga de processamento.

Selection Sort e **Bubble Sort** não se beneficiam do fato de a lista já estar ordenada. Embora não realizem trocas nesse caso, continuam realizando o mesmo número de comparações que fariam em uma lista desordenada, o que resulta em tempos de execução elevados e pouca eficiência.

Algoritmo	Tipo de Lista	Tamanho da Lista	Tempo de Execução	Comparações	Trocas
Bubble Sort	Ordenada	1,000	0.039280 segundos	499,500	0
Bubble Sort	Ordenada	10,000	4.070777 segundos	49,995,000	0
Bubble Sort	Ordenada	50,000	1.702221 minutos	1,249,975,000	0
Bubble Sort	Ordenada	100,000	6.909751 minutos	4,999,950,000	0
Selection Sort	Ordenada	1,000	0.035362 segundos	499,500	0
Selection Sort	Ordenada	10,000	3.350349 segundos	49,995,000	0
Selection Sort	Ordenada	50,000	1.464752 minutos	1,249,975,000	0
Selection Sort	Ordenada	100,000	5.701959 minutos	4,999,950,000	0
Insertion Sort	Ordenada	1,000	0.000170 segundos	999	0
Insertion Sort	Ordenada	10,000	0.001574 segundos	9,999	0
Insertion Sort	Ordenada	50,000	0.007948 segundos	49,999	0
Insertion Sort	Ordenada	100,000	0.014887 segundos	99,999	0

Figura 4.3: Distribuição Ordenada (Insertion e Quick)

Algoritmo	Tipo de Lista	Tamanho da Lista	Tempo de Execução	Comparações	Trocas
Merge Sort	Ordenada	1,000	0.001787 ms	4,932	9,976
Merge Sort	Ordenada	10,000	0.015328 ms	64,608	133616
Merge Sort	Ordenada	50,000	0.088599 ms	382,512	784464
Merge Sort	Ordenada	100,000	0.182815 ms	815,024	1668928
Quick Sort	Ordenada	1,000	0.003820 ms	10,741	6137
Quick Sort	Ordenada	10,000	0.014559 ms	158,382	84073
Quick Sort	Ordenada	50,000	0.086653 ms	974,683	552994
Quick Sort	Ordenada	100,000	0.179099 ms	1,946,872	1098029
Heap Sort	Ordenada	1,000	0.002332 ms	17,583	9708
Heap Sort	Ordenada	10,000	0.029672 ms	244,460	131956
Heap Sort	Ordenada	50,000	0.165686 ms	1,455,438	773304
Heap Sort	Ordenada	100,000	0.351031 ms	3,112,517	1650854

Figura 4.4: Distribuição Ordenada (Bubble e Selection)

4.3 Listas Inversamente Ordenadas

A análise dos resultados com listas inversamente ordenadas revela diferenças marcantes entre os algoritmos testados:

- **Insertion Sort** sofre um impacto significativo nesse cenário, pois precisa deslocar cada elemento para a posição correta, resultando em um número muito alto de comparações e trocas. Esse é o pior caso para esse algoritmo.
- **Bubble Sort** também apresenta desempenho muito ruim em listas inversas, já que precisa fazer múltiplas passagens e trocas consecutivas até alcançar a ordenação correta. Ele é sensível à ordem inicial dos dados.
- **Selection Sort**, apesar de ter complexidade $O(n^2)$, é menos afetado pela ordem inicial, porque sempre procura o menor elemento e coloca na posição correta, independentemente da configuração inicial. Ainda assim, o desempenho geral permanece ruim para listas grandes.

- **Merge Sort** mantém uma performance robusta mesmo no pior caso, já que sua abordagem de dividir e conquistar não depende da ordem dos elementos. Ele garante um comportamento consistente, com baixa sensibilidade à ordem inicial.
- **Quick Sort** apresenta desempenho estável, mas pode degradar se o pivô não for bem escolhido. Em listas inversas, estratégias de pivô simples podem levar a mais comparações e trocas, mas o impacto ainda é moderado.
- **Heap Sort** também mantém boa performance e mostra-se estável nesse cenário, já que a estrutura de heap não depende da ordem inicial. No entanto, ele realiza muitas trocas e comparações internas, o que mantém sua execução um pouco mais pesada do que a do Merge Sort.

Algoritmo	Tipo de Lista	Tamanho da Lista	Tempo de Execução	Comparações	Trocas
Bubble Sort	Inversa	1,000	0.092097 segundos	499,500	499,500
Bubble Sort	Inversa	10,000	9.829806 segundos	49,995,000	49,995,000
Bubble Sort	Inversa	50,000	4.042058 minutos	1,249,975,000	1,249,975,000
Bubble Sort	Inversa	100,000	16.399115 minutos	4,999,950,000	4,999,950,000
Selection Sort	Inversa	1,000	0.035953 segundos	499,500	500
Selection Sort	Inversa	10,000	3.481336 segundos	49,995,000	5,000
Selection Sort	Inversa	50,000	1.501978 minutos	1,249,975,000	25,000
Selection Sort	Inversa	100,000	5.893926 minutos	4,999,950,000	50,000
Insertion Sort	Inversa	1,000	0.080337 segundos	499,500	499,500
Insertion Sort	Inversa	10,000	8.282029 segundos	49,995,000	49,995,000
Insertion Sort	Inversa	50,000	3.449592 minutos	1,249,975,000	1,249,975,000
Insertion Sort	Inversa	100,000	14.661454 minutos	4,999,950,000	4,999,950,000

Figura 4.5: Distribuição Inversa (Insertion e Quick)

Algoritmo	Tipo de Lista	Tamanho da Lista	Tempo de Execução	Comparações	Trocas
Merge Sort	Inversa	1,000	0.002039 ms	5,044	9,976
Merge Sort	Inversa	10,000	0.017615 ms	69,008	133,616
Merge Sort	Inversa	50,000	0.090151 ms	401,952	784,464
Merge Sort	Inversa	100,000	0.190597 ms	853,904	1,668,928
Quick Sort	Inversa	1,000	0.001282 ms	10,879	7,062
Quick Sort	Inversa	10,000	0.026097 ms	159,035	96,667
Quick Sort	Inversa	50,000	0.087720 ms	903,909	511,752
Quick Sort	Inversa	100,000	0.184251 ms	2,054,246	1,115,883
Heap Sort	Inversa	1,000	0.002975 ms	15,965	8,316
Heap Sort	Inversa	10,000	0.02848 ms	226,682	116,696
Heap Sort	Inversa	50,000	0.149034 ms	1,366,047	698,892
Heap Sort	Inversa	100,000	0.334033 ms	2,926,640	1,497,434

Figura 4.6: Distribuição Inversa (Bubble e Selection

Capítulo 5

Discussão

1. **Heap Sort e Merge Sort**

    - Heap Sort e Merge Sort apresentam desempenho consistente devido à sua complexidade $O(n \log n)$, destacando-se em listas de grande porte. Esses algoritmos mantêm boa eficiência independentemente da distribuição inicial dos dados, sendo apropriados para situações em que é necessário lidar com grandes volumes de informação.

2. **Quick Sort**

    - Quick Sort também é muito eficiente na prática, especialmente em listas aleatórias. No entanto, seu desempenho pode se degradar em listas já ordenadas ou quase ordenadas, devido à escolha do pivô, que influencia diretamente o número de comparações e a profundidade da recursão.

5.1 Facilidade de Implementação e Estabilidade

5.1.1 Facilidade de Implementação

- Bubble Sort, Selection Sort e Insertion Sort são fáceis de implementar, devido à simplicidade de sua lógica. Contudo, essa facilidade vem à custa de baixo desempenho.
- Heap Sort e Quick Sort requerem uma implementação mais sofisticada, envolvendo operações como particionamento (Quick Sort) e manutenção da estrutura de heap (Heap Sort).
- Merge Sort, apesar de simples de implementar recursivamente, exige cuidados adicionais no gerenciamento do espaço de memória extra.

5.1.2 Estabilidade

- Merge Sort é estável, preservando a ordem relativa de elementos iguais, característica importante em aplicações específicas.
- Quick Sort e Heap Sort não são estáveis, podendo alterar a ordem dos elementos iguais durante o processo de ordenação.

- Bubble Sort e Insertion Sort são estáveis, o que os torna úteis em cenários onde a estabilidade é fundamental, apesar de sua baixa eficiência em listas extensas.

5.2 Discussão sobre pontos a melhorar e alguns aspectos dos algoritmos

Os resultados obtidos nos experimentos permitem observar de forma clara o comportamento de cada algoritmo em diferentes cenários de entrada:

1. **Bubble Sort** e **Selection Sort** demonstram ser altamente ineficientes para listas de grande porte, independentemente da ordenação inicial dos dados. Ambos mantêm complexidade $O(n^2)$ em número de comparações mesmo em listas já ordenadas, o que evidencia a limitação estrutural desses algoritmos. No caso do Bubble Sort, embora o número de trocas caia a zero em listas ordenadas, o número de comparações permanece elevado, o que compromete a eficiência geral. Esses resultados empíricos estão em linha com a teoria, que sugere que esses algoritmos são ineficientes em listas grandes, especialmente em cenários desfavoráveis como listas inversamente ordenadas. Em contextos reais, algoritmos mais eficientes como Merge Sort, Quick Sort ou algoritmos híbridos (como o Timsort) são preferidos.

2. **Insertion Sort**, por outro lado, destaca-se positivamente em listas ordenadas ou quase ordenadas, apresentando complexidade próxima de $O(n)$. Isso se deve ao fato de que, nesses casos, o algoritmo praticamente não realiza trocas, e as comparações são minimizadas, o que explica os tempos extremamente baixos registrados nesses cenários. Esse comportamento o torna particularmente vantajoso em aplicações onde os dados tendem a estar quase organizados. A teoria confirma esse comportamento, com o algoritmo apresentando um desempenho significativamente melhor em listas parcialmente ordenadas. No entanto, sua performance diminui em listas aleatórias ou inversamente ordenadas, o que é esperado dado seu pior caso de $O(n^2)$.

3. A ordenação inicial dos dados exerce forte influência no desempenho dos algoritmos. Listas inversamente ordenadas representam o pior caso para todos os métodos testados, especialmente para o Insertion Sort e o Bubble Sort, que precisam realizar um número máximo de comparações e trocas para ordenar os elementos. Em contraste, listas já ordenadas representam o melhor caso para o Insertion Sort, onde seu tempo de execução se torna praticamente desprezível mesmo em listas com 100.000 elementos. Isso é confirmado pela teoria, que explica como a quantidade de trocas e comparações realizadas depende diretamente da ordenação inicial dos dados.

4. **Limitações e Recomendações**:

   - Embora os algoritmos Bubble Sort, Selection Sort e Insertion Sort sejam úteis do ponto de vista didático, sua aplicação em larga escala não é recomendada devido ao seu alto custo computacional em cenários desfavoráveis. A comparação entre os resultados empíricos e teóricos mostra que, embora simples de implementar, esses algoritmos se tornam ineficazes à medida que o tamanho da lista aumenta.

- Em cenários do mundo real, algoritmos como **Merge Sort** e **Quick Sort** são preferidos por sua melhor eficiência. Esses algoritmos possuem complexidade $O(n \log n)$, o que os torna muito mais adequados para aplicações de larga escala. A inclusão de outros algoritmos mais eficientes, como o **Timsort** (usado em Python), pode ser uma boa sugestão para melhorar o desempenho em listas grandes.

5. **Sugestões para Estudos Futuros**:

   - Analisar o consumo de memória de algoritmos como o **Quick Sort** e o **Merge Sort** para entender melhor o impacto de seu uso de memória em comparação com algoritmos mais simples.
   - Estudar alternativas como **Radix Sort**, que também têm complexidade $O(n \log n)$ e podem ser mais eficientes em certos contextos de dados.

Bubble Sort

Como funciona:

- Laço externo: passa n vezes pela lista.
- Laço interno: compara elementos adjacentes até o final da lista (diminuindo a cada passo).

for i in range(n) : for j in range$(0, n-i-1)$: if $arr[j] > arr[j+1]$: trocar

Quantas comparações?

- No $1^{\underline{o}}$ passo, compara $n-1$ pares: $arr[0]$ com $arr[1]$, até $arr[n-2]$ com $arr[n-1]$.
- No $2^{\underline{o}}$ passo, $n-2$ comparações (último já está no lugar).
- ...
- No último passo, 1 comparação.

A soma das comparações é:

$$(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} \quad \Rightarrow \quad O(n^2)$$

Trocas: No pior caso (lista invertida), cada comparação resulta em uma troca, logo também temos $O(n^2)$.

Selection Sort

Como funciona:

- Para cada posição i, você procura o menor elemento no resto da lista.
- Quando encontra, faz uma única troca no final da rodada.

for i in range(n) : min_index $= i$ for j in range$(i{+}1, n)$: if $arr[j] < arr[min\_index]$: min_inc

Quantas comparações?

- 1ª rodada: compara $n-1$ elementos.
- 2ª rodada: $n-2$ comparações.
- ...
- Última: 1 comparação.

A soma das comparações é:

$$(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} \quad \Rightarrow \quad O(n^2)$$

Trocas: Uma única troca por rodada (mesmo se não precisar).

$$\text{Máximo: } n-1 \text{ trocas} \quad \Rightarrow \quad O(n)$$

Insertion Sort

Como funciona:

- Vai inserindo cada elemento no lugar certo da parte já ordenada.
- Compara com os elementos anteriores até achar onde encaixar.

for i in range$(1, n)$: chave $= arr[i]$ $j = i{-}1$ while $j \geq 0$ and $arr[j] >$ chave : $arr[j{+}1] = arr[$

Quantas comparações?

- No melhor caso (lista já ordenada), $arr[j] >$ chave nunca é verdade, logo só uma comparação por i.
- Total: $n-1$ comparações, logo $O(n)$.
- No pior caso (lista invertida), cada chave é menor que todos os anteriores.
- 1ª inserção: 1 comparação.
- 2ª: 2 comparações.

- ...
- Última: $n - 1$ comparações.

A soma das comparações é:

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} \quad \Rightarrow \quad O(n^2)$$

Capítulo 6

Conclusão

A análise experimental realizada com os algoritmos de ordenação Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort revelou diferenças significativas de desempenho em termos de tempo de execução, número de comparações e número de trocas, especialmente à medida que o tamanho das listas aumentava.

Os algoritmos Bubble Sort, Selection Sort e Insertion Sort, todos de complexidade $O(n^2)$, apresentaram tempos de execução extremamente elevados em listas com 50.000 ou 100.000 elementos, chegando a consumir vários minutos. O Bubble Sort destacou-se negativamente pelo altíssimo número de trocas, mesmo em listas ordenadas e inversamente ordenadas, evidenciando sua baixa eficiência prática. O Selection Sort, embora também apresente alto custo em comparações, realizou pouquíssimas trocas, o que o torna previsível, mas não necessariamente eficiente. O Insertion Sort, por sua vez, apresentou bom desempenho em listas ordenadas, mas sofreu grande degradação com o aumento do tamanho e da desordem dos dados.

Os algoritmos de complexidade $O(n \log n)$ — Merge Sort, Quick Sort e Heap Sort — mostraram-se muito mais adequados para lidar com listas grandes. O Merge Sort foi consistente e estável, com baixíssimo tempo de execução mesmo em listas grandes, e sem realizar trocas no sentido clássico, o que é característico de seu funcionamento baseado em cópias e fusões. O Quick Sort apresentou excelente desempenho em quase todos os cenários, mas é importante destacar que seu desempenho depende fortemente da escolha do pivô, o que pode ser um ponto fraco em casos desfavoráveis. O Heap Sort mostrou-se robusto e competitivo, combinando um número controlado de comparações e trocas, independentemente da ordem inicial da lista.

Em termos de comparações, os algoritmos quadráticos foram, como esperado, muito menos eficientes do que os algoritmos $O(n \log n)$, chegando a ultrapassar bilhões de comparações nas listas maiores. Já em termos de trocas, os resultados variaram: enquanto Bubble Sort foi extremamente ineficiente, o Merge Sort apresentou virtualmente zero trocas, e Heap Sort manteve um equilíbrio notável.

De modo geral, os resultados confirmam que algoritmos $O(n \log n)$ são preferíveis para aplicações práticas envolvendo grandes volumes de dados, enquanto os algoritmos quadráticos devem ser utilizados apenas para fins didáticos ou em contextos específicos, como listas pequenas ou parcialmente ordenadas. Esta análise reforça a importância de compreender as características internas de cada algoritmo, permitindo uma escolha informada de acordo com as necessidades e limitações do problema.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press. Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-Wesley.