Universidade Presbiteriana Mackenzie

ALGORITMOS DE ORDENAÇÃO EM VETORES DE TAMANHOS PRÉ DEFINIDOS

Projeto I - Algoritmos

DANIEL MANGABEIRA CORREIA
MESSIAS OLIVEIRA DA SILVA

SÃO PAULO
Setembro/2024

Sumário

INTRODUÇÃO

Este projeto tem como objetivo a análise comparativa de diferentes algoritmos de ordenação aplicados a vetores de dados com tamanhos variados, utilizando a linguagem de programação Python (versão 3.10.12). A comparação será realizada com vetores contendo 1.000, 10.000 e 100.000 elementos, gerados de três maneiras distintas: de forma aleatória, ordenados em ordem crescente e ordenados em ordem decrescente.

Os algoritmos de ordenação analisados incluem Bubble Sort, Bubble Sort Optimized, Selection Sort, Insertion Sort, Merge Sort e Quick Sort. A análise visa medir o desempenho de cada algoritmo em diferentes cenários, utilizando métricas de tempo de execução, comparando a eficiência prática dos algoritmos com suas respectivas complexidades assintóticas.

Os experimentos foram executados em um ambiente controlado para garantir a reprodutibilidade dos resultados. O hardware utilizado consistiu em um computador equipado com um processador Intel® Core™ i7-13650HX de 13ª geração, operando a 2.60 GHz, 16 GB de memória RAM (15,7 GB utilizável), e sistema operacional de 64 bits, processador baseado em x64. Embora o hardware tenha capacidades de alto desempenho, o programa Python foi executado em um ambiente virtual (venv) dentro do Windows Subsystem for Linux (WSL), o que simula um ambiente Linux dentro do Windows, proporcionando um cenário mais próximo ao uso de servidores Linux em ambientes de produção.

As bibliotecas `numpy` e `matplotlib` foram utilizadas para geração dos vetores e construção dos gráficos de desempenho, respectivamente. O relatório final incluirá gráficos comparativos dos tempos de execução dos algoritmos para cada tipo de vetor e uma análise crítica sobre a eficiência observada de cada algoritmo, levando em consideração a complexidade teórica (assintótica) e os resultados empíricos obtidos. Esta abordagem permite uma compreensão aprofundada do comportamento dos algoritmos de ordenação em diferentes contextos, destacando suas vantagens e limitações práticas.

OBJETIVO GERAL

O objetivo geral deste projeto é avaliar o desempenho de diferentes algoritmos de ordenação aplicados a vetores de tamanhos variados. A análise envolve a execução e comparação de seis algoritmos de ordenação (Bubble Sort, Bubble Sort Optimized, Selection Sort, Insertion Sort, Merge Sort e Quick Sort) em diferentes tipos de dados (aleatórios, ordenados crescentemente e ordenados decrescentemente) e a medição do tempo de execução de cada algoritmo, a fim de compreender suas características, comportamentos e eficiência em diferentes cenários.

OBJETIVOS ESPECÍFICOS

1. Implementar Algoritmos de Ordenação: Desenvolver e implementar os algoritmos de ordenação em Python, garantindo sua funcionalidade e correta aplicação em vetores de tamanhos variados.
2. Gerar Vetores de Dados: Criar três tipos diferentes de vetores (aleatórios, ordenados crescentemente e ordenados decrescentemente) para tamanhos de 1.000, 10.000 e 100.000 elementos, utilizando a biblioteca `numpy`.
3. Medir e Comparar Desempenho: Calcular o tempo de execução de cada algoritmo de ordenação aplicado a cada tipo de vetor, medindo o desempenho em milissegundos.
4. Salvar Resultados em Arquivos: Armazenar os tempos de execução em arquivos de texto separados para cada tamanho de vetor, permitindo a análise posterior dos resultados.
5. Gerar Gráficos de Desempenho: Criar gráficos comparativos dos tempos de execução para cada algoritmo e tamanho de vetor utilizando a biblioteca `matplotlib`.
6. Compactar Arquivos de Resultados: Organizar todos os arquivos gerados (vetores e resultados de execução) em um diretório e compactá-los em um arquivo ZIP para facilitar o armazenamento e compartilhamento dos resultados.
7. Registrar Informações do Sistema: Capturar e documentar as especificações de hardware e software do sistema utilizado para os testes, incluindo detalhes como o sistema operacional, processador e memória RAM disponível.
8. Documentar o Processo: Preparar um relatório que documenta os métodos, resultados, gráficos, e a análise dos algoritmos de ordenação, discutindo suas complexidades assintóticas e comparando o desempenho observado com as expectativas teóricas.

Esses objetivos garantem uma análise abrangente do desempenho dos algoritmos de ordenação, fornecendo dados empíricos e insights sobre suas eficiências e comportamentos em diferentes cenários.

MATERIAL E MÉTODOS

EQUIPAMENTO UTILIZADO

Os experimentos foram conduzidos em um computador com as seguintes especificações:

- Processador: 13ª Geração Intel® Core™ i7-13650HX, 2.60 GHz
- Memória RAM: 16,0 GB (utilizável: 15,7 GB)
- Sistema Operacional: Sistema operacional de 64 bits, processador baseado em x64

Apesar do hardware de alto desempenho, o código Python foi executado dentro de um ambiente virtual (‘env’) configurado no Windows Subsystem for Linux (WSL), proporcionando um ambiente de execução que simula o comportamento de servidores Linux.

MASSA DE DADOS

Foram utilizados vetores de três tamanhos distintos: 1.000, 10.000 e 100.000 elementos, gerados em três configurações diferentes:

1. Vetor Aleatório: Elementos gerados aleatoriamente.
2. Vetor Ordenado Crescente: Elementos ordenados de forma crescente.
3. Vetor Ordenado Decrescente: Elementos ordenados de forma decrescente.

Os vetores foram gerados utilizando a biblioteca `numpy`, e cada vetor foi salvo em arquivos de texto para facilitar a reprodução dos experimentos.

ALGORITMOS DE ORDENAÇÃO

Os seguintes algoritmos de ordenação foram implementados e testados:

1. Bubble Sort: Algoritmo simples de ordenação por comparação que percorre o vetor repetidamente, trocando elementos adjacentes fora de ordem.
2. Bubble Sort Optimized: Versão otimizada do Bubble Sort, que interrompe a

ordenação caso o vetor já esteja ordenado em uma das iterações.

3. Selection Sort: Algoritmo que repetidamente seleciona o menor elemento de uma parte não ordenada do vetor e o move para a posição correta.
4. Insertion Sort: Algoritmo que constrói uma lista ordenada inserindo elementos na posição correta a partir de uma sub lista já ordenada.
5. Merge Sort: Algoritmo de ordenação por divisão e conquista, que divide o vetor em sublistas menores, ordena essas sublistas e as combina para formar o vetor ordenado.
6. Quick Sort: Algoritmo de ordenação por divisão e conquista que escolhe um elemento como pivô e particiona o vetor em elementos menores e maiores que o pivô.

METODOLOGIA DE EXECUÇÃO

1. Geração dos Vetores: Os vetores de diferentes tamanhos e configurações foram gerados utilizando funções específicas e salvos em arquivos de texto para posterior análise.
2. Medição de Tempo: O tempo de execução de cada algoritmo foi medido utilizando a função `time()` da biblioteca `time`, que retorna o tempo em milissegundos. Para cada combinação de algoritmo e vetor, o tempo de execução foi registrado e salvo em arquivos de texto.
3. Análise dos Resultados: Os tempos de execução obtidos foram utilizados para gerar gráficos comparativos, utilizando a biblioteca `matplotlib`. Os gráficos representam o desempenho de cada algoritmo em diferentes cenários e permitem uma análise visual da eficiência de cada abordagem.
4. Relatório e Armazenamento de Dados: Os resultados e informações adicionais, como detalhes sobre o sistema, foram armazenados em um arquivo ZIP para facilitar a distribuição e o armazenamento.

FERRAMENTAS E BIBLIOTECAS UTILIZADAS

- Linguagem de Programação: Python (versão 3.10.12)
- Bibliotecas:

- ‘numpy’: Utilizada para geração dos vetores aleatórios.
- ‘matplotlib’: Utilizada para geração dos gráficos comparativos de tempo de execução.
- ‘zipfile’, ‘os’, ‘sys’, ‘psutil’, ‘platform’, ‘datetime’: Utilizadas para manipulação de arquivos, coleta de informações do sistema e geração de relatórios.

PROCEDIMENTO PARA EXECUÇÃO

1. Preparação do Ambiente: O ambiente virtual Python foi configurado no WSL, permitindo a execução do script em um ambiente Linux simulado.
2. Execução do Script: O script foi executado, gerando os vetores, aplicando os algoritmos de ordenação e medindo os tempos de execução.
3. Geração de Gráficos e Relatórios: Os tempos de execução foram utilizados para criar gráficos comparativos, e todos os dados relevantes foram salvos em arquivos de texto, que foram posteriormente compactados em um arquivo ZIP.
4. Armazenamento dos Resultados: As informações sobre o sistema e os resultados da execução dos algoritmos foram armazenados em um arquivo ZIP, garantindo a integridade e a facilidade de compartilhamento dos dados.

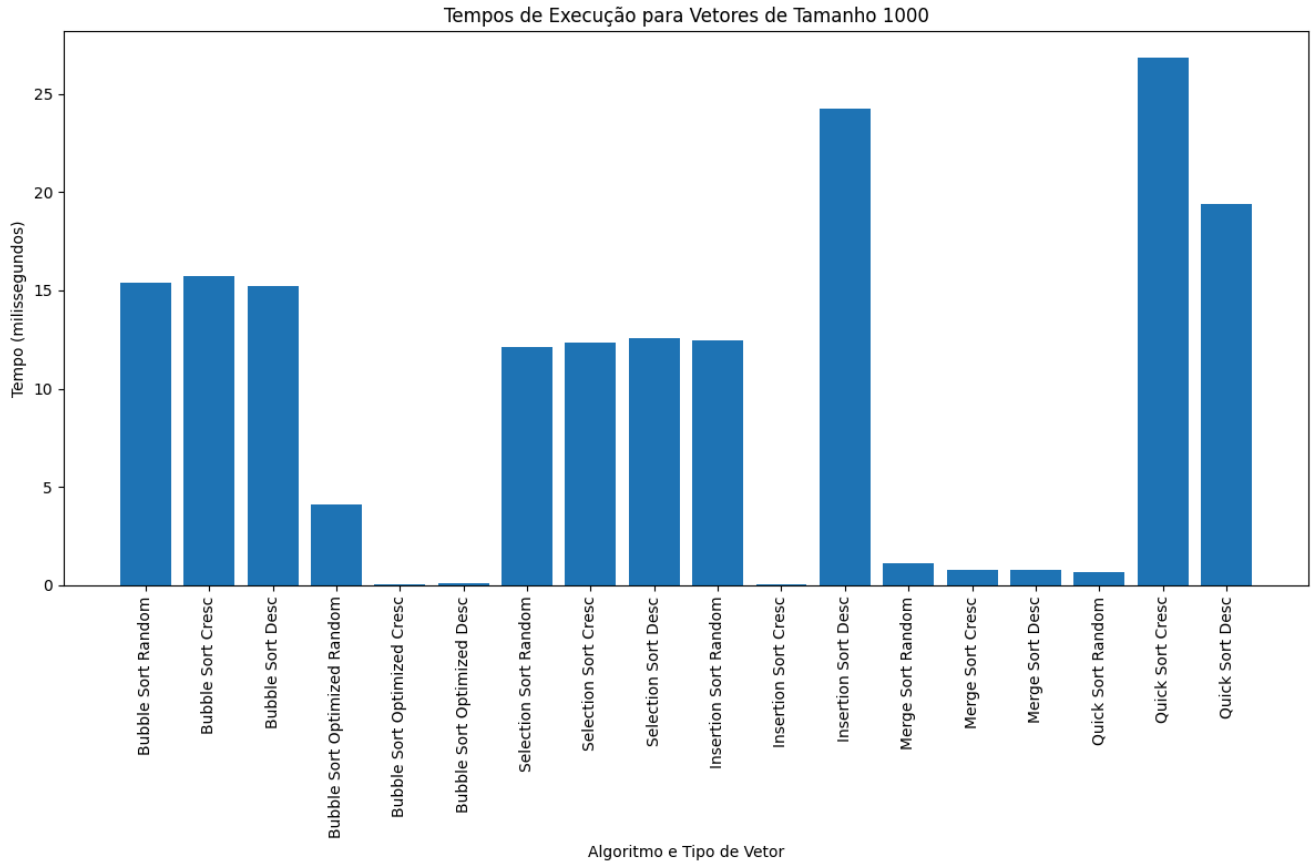
RESULTADOS E DISCUSSÃO

Nesta seção, são apresentados os resultados detalhados da execução dos algoritmos de ordenação em vetores de tamanhos variados (1.000, 10.000 e 100.000 elementos) e diferentes ordenações iniciais (aleatórios, ordenados crescentemente e ordenados decrescentemente).

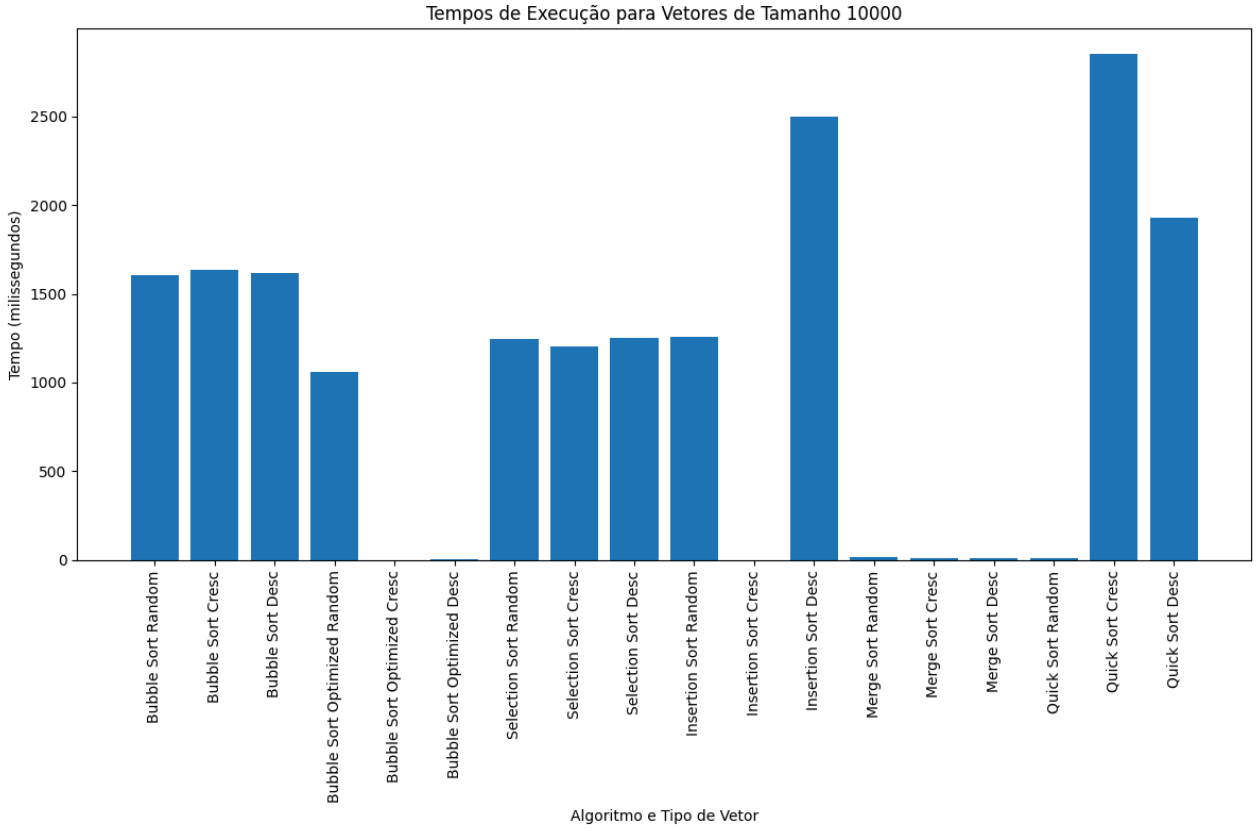
RESULTADOS OBTIDOS

Os tempos de execução (em milissegundos) de cada algoritmo para os três tamanhos de vetor e seus respectivos gráficos serão apresentados a seguir:

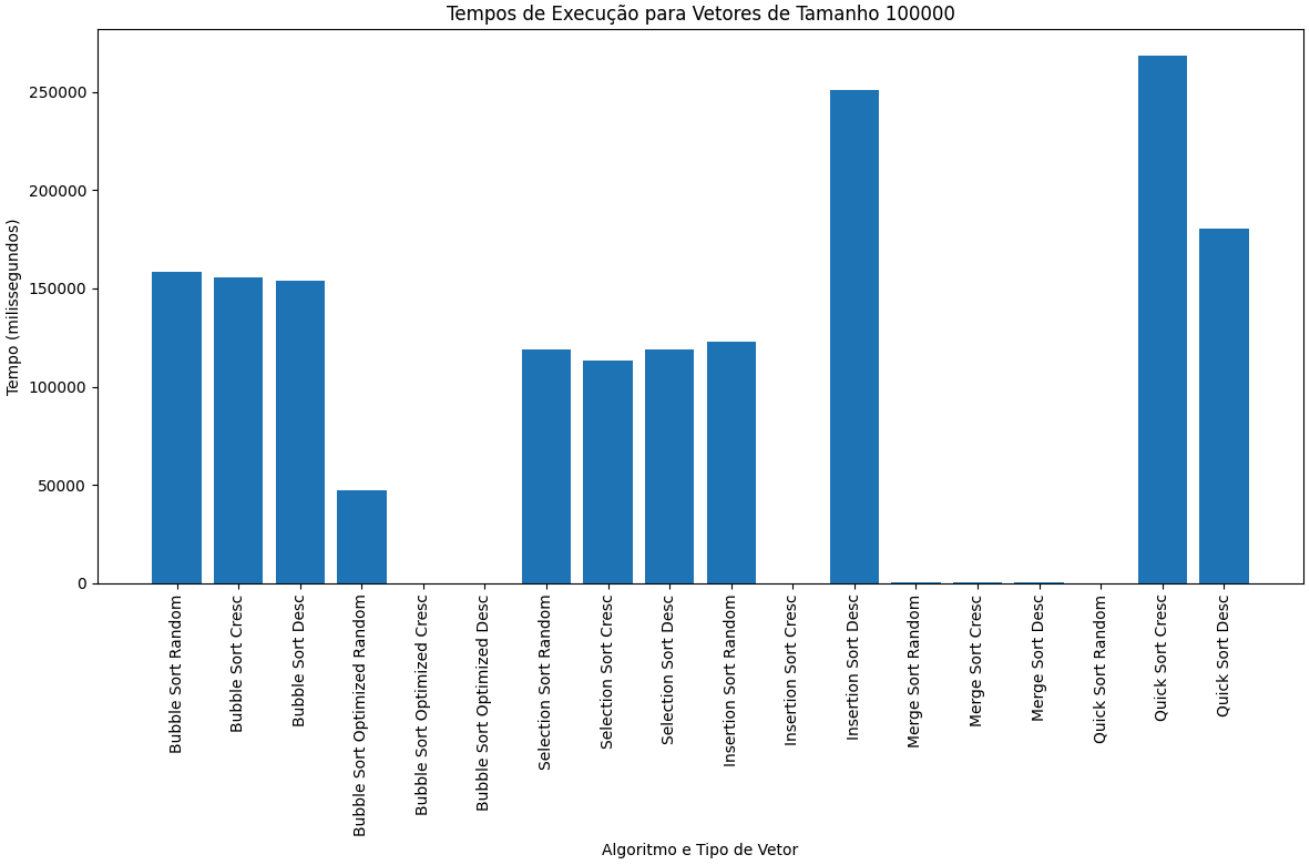
- **Vetores de 1.000 elementos**:
  - Bubble Sort e suas variantes apresentaram tempos similares em torno de 15 ms, exceto pelo Bubble Sort Otimizado que mostrou um desempenho muito melhor em vetores quase ordenados.
  - Insertion Sort teve um desempenho excelente em vetores crescentes (0.06 ms), mas sofreu uma grande queda no desempenho em vetores decrescentes (24.28 ms).
  - Quick Sort mostrou a melhor performance em vetores aleatórios (0.69 ms), mas apresentou um desempenho significativamente pior para vetores crescentes (26.85 ms).
  - Merge Sort teve um desempenho consistentemente bom em todos os casos, com tempos variando entre 0.8 ms e 1.09 ms.



- **Vetores de 10.000 elementos**:
  - Bubble Sort foi o algoritmo mais lento, com tempos em torno de 1600 ms.
  - Insertion Sort continuou a ter desempenho ruim em vetores decrescentes (2497.03 ms), enquanto Quick Sort teve um desempenho muito fraco em vetores crescentes (2851.48 ms), mas melhorou em vetores decrescentes (1927.54 ms).
  - Merge Sort e Quick Sort demonstraram tempos bem melhores em vetores aleatórios, com Merge Sort variando de 10.69 ms a 13.47 ms, e Quick Sort entre 9.11 ms e 1927.54 ms.



- **Vetores de 100.000 elementos**:
    - Quick Sort teve um desempenho ruim em vetores crescentes (268467.71 ms) e decrescentes (180634.71 ms), destacando a sua vulnerabilidade a casos de pior cenário.
    - Insertion Sort teve um desempenho ruim para vetores decrescentes (250823.92 ms).
    - Bubble Sort e suas variantes continuaram a ser os mais lentos, com tempos próximos de 150.000 ms para todos os casos.
    - Merge Sort apresentou o melhor desempenho geral, com tempos entre 126.53 ms e 157.28 ms, demonstrando eficiência e estabilidade, independentemente do tamanho e da ordenação do vetor.



ANÁLISE CRÍTICA SOBRE A EFICIÊNCIA DOS ALGORITMOS

BUBBLE SORT E BUBBLE SORT OTIMIZADO:

- Eficiência Prática: Ambos os algoritmos apresentam uma eficiência prática muito baixa para vetores grandes, especialmente em vetores aleatórios ou decrescentes. O Bubble Sort tradicional tem uma complexidade de tempo $O(n^2)$, o que se traduz em um desempenho ruim para conjuntos de dados grandes.
- Bubble Sort Otimizado: Apesar de melhorar o desempenho para vetores quase ordenados ao interromper o processamento quando nenhum swap é feito, ainda não é suficiente para vetores grandes devido ao seu tempo quadrático no pior caso.
- Vantagens e Limitações: A otimização é útil para vetores que já estão quase ordenados, mas a abordagem não compensa a ineficiência geral dos algoritmos quadráticos.

INSERTION SORT:

- Eficiência Prática: O Insertion Sort é bastante eficiente para vetores pequenos ou quase ordenados, mostrando excelente desempenho em vetores crescentes. No entanto, seu desempenho degrada-se drasticamente com vetores grandes e especialmente com vetores decrescentes, devido à sua complexidade $O(n^2)$ no pior caso.
- Vantagens e Limitações: Ideal para cenários onde a maioria dos dados já está ordenada ou para vetores pequenos. Não é recomendado para grandes conjuntos de dados devido à sua ineficiência em dados não ordenados.

SELECTION SORT:

- Eficiência Prática: Apresenta uma eficiência similar ao Bubble Sort, com uma complexidade de tempo $O(n^2)$. Seu desempenho é consistente independentemente da ordem inicial dos dados, mas continua sendo ineficaz para grandes volumes de dados.
- Vantagens e Limitações: Simples de implementar e entender, mas não é adequado para grandes conjuntos de dados devido à sua ineficiência.

MERGE SORT:

- Eficiência Prática: Merge Sort se destaca pela sua eficiência em grandes conjuntos de dados, mantendo um desempenho consistente independente da configuração inicial do vetor. Com uma complexidade de tempo $O(n \log n)$, é adequado para vetores grandes e variáveis.
- Vantagens e Limitações: É um algoritmo estável e eficiente para grandes conjuntos de dados, mas requer espaço adicional para a combinação dos vetores.

QUICK SORT:

- Eficiência Prática: Quick Sort é altamente eficiente em vetores aleatórios e menores, com um tempo médio $O(n \log n)$. No entanto, seu desempenho pode degradar para $O(n^2)$ em vetores ordenados, especialmente se o pivô não for escolhido de forma eficiente.
- Vantagens e Limitações: Ideal para a maioria dos casos, mas a escolha do pivô e a natureza dos dados podem impactar significativamente o desempenho.

ANÁLISE CRÍTICA SOBRE A ANÁLISE ASSINTÓTICA VS. TEMPOS OBTIDOS

COMPARAÇÃO DE RESULTADOS TEÓRICOS E PRÁTICOS:

- **Bubble Sort e Selection Sort:** Como esperado, os tempos de execução práticos confirmam a análise assintótica teórica. Ambos os algoritmos mostram um desempenho muito ruim para vetores grandes, o que está alinhado com a complexidade $O(n^2)$. A análise assintótica é uma boa previsão para a performance observada.
- **Insertion Sort:** A eficiência prática para vetores pequenos e quase ordenados corresponde bem à análise assintótica. No entanto, para vetores grandes e desordenados, os tempos de execução práticos são significativamente piores do que a análise teórica poderia sugerir, devido ao impacto real da complexidade $O(n^2)$ em grandes volumes de dados.
- **Merge Sort:** Os resultados práticos confirmam a análise assintótica. Merge Sort mantém um desempenho eficiente e consistente, refletindo bem sua complexidade $O(n \log n)$. O comportamento observado está de acordo com a teoria, demonstrando que o Merge Sort é efetivo para grandes volumes de dados.
- **Quick Sort:** Os resultados mostram que, enquanto Quick Sort é eficiente para vetores aleatórios e pequenos, o desempenho em cenários de pior caso (vetores ordenados) confirma a análise assintótica. A degradação de desempenho observada para vetores ordenados está alinhada com o comportamento esperado $O(n^2)$ no pior caso. A escolha do pivô e o impacto de dados ordenados são bem capturados pela análise teórica.

A análise assintótica fornece uma estimativa valiosa do desempenho dos algoritmos, mas os tempos de execução práticos podem ser afetados por fatores adicionais como a implementação específica, a escolha de pivôs em Quick Sort e as características do sistema. A análise empírica confirma muitas das previsões teóricas, mas também destaca a importância de considerar as variáveis do mundo real que podem impactar o desempenho real dos algoritmos.

DISCUSSÃO

Os resultados indicam que algoritmos com complexidade de tempo O(n2)O(n^2)O(n2), como Bubble Sort e Insertion Sort, não são adequados para conjuntos de dados grandes ou desordenados. Em particular:

1. **Bubble Sort e Bubble Sort Otimizado**: Como esperado, o Bubble Sort teve desempenho consistentemente ruim para todos os tamanhos e tipos de vetor devido à sua complexidade O(n2)O(n^2)O(n2). A versão otimizada demonstrou melhorias significativas para vetores quase ordenados, mas ainda apresentou tempos impraticáveis para vetores maiores.
2. **Insertion Sort**: Demonstrou excelente desempenho em vetores quase ordenados, mas teve desempenho muito ruim para vetores decrescentes, particularmente em tamanhos maiores (100.000 elementos), confirmando sua vulnerabilidade a dados desordenados.
3. **Quick Sort**: Enquanto Quick Sort foi muito eficiente em vetores aleatórios pequenos, seu desempenho caiu drasticamente para vetores ordenados (crescentes e decrescentes), especialmente para vetores maiores, confirmando sua ineficiência em cenários de pior caso.
4. **Merge Sort**: O algoritmo Merge Sort provou ser o mais consistente em todos os casos, confirmando sua adequação para grandes conjuntos de dados e demonstrando uma complexidade de tempo O(nlogn)O(n \log n)O(nlogn) que o torna preferível para aplicações que exigem alta eficiência.

Essas observações ressaltam a importância de escolher o algoritmo de ordenação apropriado com base nas características do conjunto de dados, como o tamanho e o grau de ordenação inicial.

CONCLUSÃO

Este estudo demonstrou as diferenças significativas de desempenho entre diversos algoritmos de ordenação, com base em experimentos realizados em vetores de tamanhos e ordenações variadas. Os resultados mostram que:

- Algoritmos como Bubble Sort e Insertion Sort são inadequados para grandes volumes de dados ou dados desordenados.
- Quick Sort, embora eficiente na maioria dos casos, apresenta problemas de desempenho em cenários de pior caso (dados ordenados crescentemente ou decrescentemente).
- Merge Sort é consistentemente eficiente, demonstrando ser a escolha mais adequada para grandes conjuntos de dados, independentemente do seu estado inicial.

Portanto, recomenda-se o uso de algoritmos de complexidade O(n log n), como Merge Sort, em situações onde a eficiência é crítica, especialmente em dados de grande escala.

REFERÊNCIAS BIBLIOGRÁFICAS

ChatGPT. Disponível em: https://chatgpt.com/c/99a6f74a-ecf2-453d-b82f-872ac9f1568d. Acesso em: 2 set. 2024.

CORMEN, T. H. et al. *Introduction to Algorithms*. 3. ed. [s.l.]: MIT Press, 2014.

HUNTER, J. D. (2007). "Matplotlib: A 2D Graphics Environment." *Computing in Science & Engineering*, 9, 90-95. Disponível em: https://www.scirp.org/reference/referencespapers?referenceid=1560579. Acesso em: 2 set. 2024.

NumPy documentation — NumPy v2.1 manual. Disponível em: https://numpy.org/doc/stable/index.html. Acesso em: 2 set. 2024.

SEDGEWICK, R.; WAYNE, K. *Algorithms*. [s.l.]: Addison-Wesley Professional, 2011.

VAN ROSSUM, G.; DRAKE, F. L. *Python 3 Reference Manual: (Python Documentation Manual part 2)*. [s.l.]: Createspace, 2009.