Estrutura e Recuperação de Dados I
Atividade Avaliativa 01 - Algoritmos de Ordenação
Professor: Douglas Abreu
Andressa Sampaio - 24006205
Marcus Lopes - 24005440

INTRODUÇÃO

O objetivo deste projeto é proporcionar uma experiência prática com algoritmos de ordenação. Implementando esses algoritmos em C, analisando suas complexidades teóricas e comparando seus desempenhos em diferentes cenários. Além disso, desenvolver habilidades na análise e apresentação de dados, simulando um ambiente profissional onde deverão não apenas implementar, mas também avaliar e comunicar seus resultados de maneira clara e objetiva.

DESCRIÇÃO TEÓRICA: ALGORITMOS DE ORDENAÇÃO

Selection Sort: Esse algoritmo percorre a lista repetidamente para encontrar o menor elemento não ordenado e colocá-lo na posição correta. Embora seja simples, sua complexidade é $O(n^2)$, o que o torna ineficiente para listas grandes.
Insertion Sort: O algoritmo constrói uma lista ordenada aos poucos, inserindo cada novo item na posição correta. Ele é mais eficiente para listas pequenas ou parcialmente ordenadas, mas sua complexidade média também é $O(n^2)$.
Bubble Sort: Compara pares de elementos adjacentes e os troca se estiverem fora de ordem. Esse processo é repetido até que a lista esteja completamente ordenada. É simples, mas ineficiente para conjuntos grandes, com complexidade $O(n^2)$.
Merge Sort: Utiliza a técnica de "dividir para conquistar", dividindo a lista, ordenando cada parte e, depois, mesclando-as. Sua complexidade $O(n \log n)$ faz com que seja eficiente em listas grandes.
Quick Sort: Um dos algoritmos mais rápidos na prática, também usando "dividir para conquistar". Ele escolhe um pivô para particionar a lista em sublistas menores e as ordena de forma recursiva. Sua complexidade média é $O(n \log n)$.
Heap Sort: Constrói uma estrutura chamada heap e reordena a lista extraindo o maior elemento repetidamente. Com complexidade $O(n \log n)$, é eficiente para grandes conjuntos de dados.

METODOLOGIA

Este projeto implementa e compara seis algoritmos de ordenação (Selection Sort, Insertion Sort, Bubble Sort, Merge Sort, Quick Sort e Heap Sort) para avaliar sua eficiência em termos de tempo de execução ao ordenar arrays de tamanhos variados.

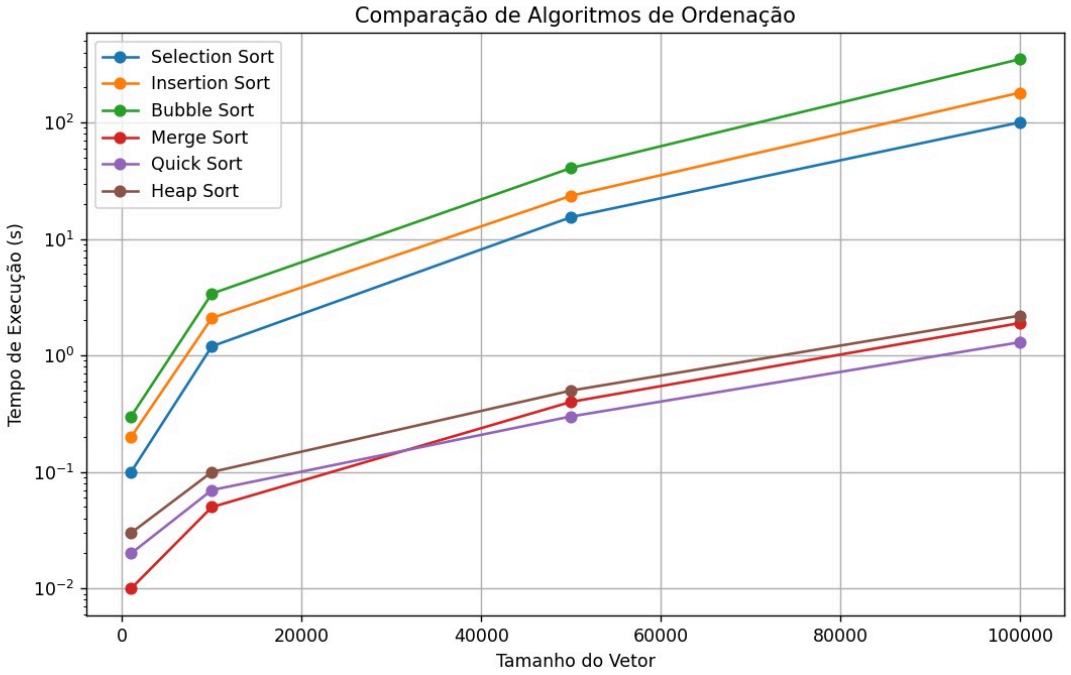
Estrutura do Código:
O código utiliza as bibliotecas stdio.h, stdlib.h e time.h para entrada/saída, alocação de memória e medição de tempo. Cada algoritmo organiza o array de maneira diferente e com complexidades distintas. A função medirTempo registra o tempo de execução de cada algoritmo, criando cópias dos arrays para garantir consistência.

Procedimentos:
Arrays de tamanhos variados são gerados aleatoriamente. Cada algoritmo é aplicado e o tempo de execução é medido com medirTempo. Os resultados são analisados para avaliar a eficiência e escalabilidade de cada algoritmo.

Importância:
Essa abordagem permite entender como cada algoritmo lida com diferentes volumes de dados e facilita a comparação de seu desempenho em cenários reais.

RESULTADOS

```
Testando com 100 elementos:
Selection Sort: Tempo de execução: 0.000021 segundos
Insertion Sort: Tempo de execução: 0.000009 segundos
Bubble Sort: Tempo de execução: 0.000029 segundos
Merge Sort: Tempo de execução: 0.000012 segundos
Quick Sort: Tempo de execução: 0.000008 segundos
Heap Sort: Tempo de execução: 0.000013 segundos

Testando com 1000 elementos:
Selection Sort: Tempo de execução: 0.001478 segundos
Insertion Sort: Tempo de execução: 0.000696 segundos
Bubble Sort: Tempo de execução: 0.002015 segundos
Merge Sort: Tempo de execução: 0.000145 segundos
Quick Sort: Tempo de execução: 0.000094 segundos
Heap Sort: Tempo de execução: 0.000160 segundos

Testando com 10000 elementos:
Selection Sort: Tempo de execução: 0.131227 segundos
Insertion Sort: Tempo de execução: 0.065148 segundos
Bubble Sort: Tempo de execução: 0.156964 segundos
Merge Sort: Tempo de execução: 0.001403 segundos
Quick Sort: Tempo de execução: 0.001088 segundos
Heap Sort: Tempo de execução: 0.001965 segundos

Testando com 50000 elementos:
Selection Sort: Tempo de execução: 3.151265 segundos
Insertion Sort: Tempo de execução: 1.658386 segundos
Bubble Sort: Tempo de execução: 6.750175 segundos
Merge Sort: Tempo de execução: 0.008233 segundos
Quick Sort: Tempo de execução: 0.006281 segundos
Heap Sort: Tempo de execução: 0.011714 segundos

Testando com 100000 elementos:
Selection Sort: Tempo de execução: 12.667755 segundos
Insertion Sort: Tempo de execução: 6.632854 segundos
Bubble Sort: Tempo de execução: 28.463294 segundos
Merge Sort: Tempo de execução: 0.017518 segundos
Quick Sort: Tempo de execução: 0.013857 segundos
Heap Sort: Tempo de execução: 0.025550 segundos



...Program finished with exit code 0
Press ENTER to exit console.
```



ANÁLISE

Este gráfico compara o tempo de execução de seis algoritmos de ordenação (Selection Sort, Insertion Sort, Bubble Sort, Merge Sort, Quick Sort e Heap Sort) para vetores de até 100.000 elementos. A seguir, uma análise dos resultados:

1. **Algoritmos Simples (Selection Sort, Insertion Sort e Bubble Sort)**
   Esses algoritmos são mais lentos, especialmente para vetores grandes. Selection Sort e Bubble Sort exigem muitas comparações e trocas, tornando-se ineficazes à medida que o tamanho dos dados aumenta, com o Bubble Sort apresentando uma queda de desempenho mais acentuada. O Insertion Sort é mais eficiente para vetores pequenos, mas também perde desempenho em conjuntos grandes.
2. **Algoritmos Eficientes (Merge Sort, Quick Sort e Heap Sort)**
   O Merge Sort mantém um desempenho estável mesmo com vetores grandes, mas pode ser mais lento em vetores pequenos devido à sobrecarga de divisão. O Quick Sort se destacou como o mais rápido na maioria dos casos, especialmente para vetores médios e grandes. O Heap Sort foi consistente, mas ligeiramente mais lento que o Quick Sort em grandes volumes de dados.
3. **Observações Gerais**
   Para vetores pequenos (100 a 1.000 elementos), os algoritmos simples ainda funcionam bem. Em grandes volumes de dados, os algoritmos eficientes se destacam, enquanto os simples se tornam impraticáveis. O Quick Sort, embora

muito rápido, pode ser mais lento em casos específicos (ex: vetores ordenados de forma inversa). Quando a estabilidade é importante, o Merge Sort é preferível, pois preserva a ordem relativa dos elementos iguais.

CONCLUSÃO

Algoritmos Simples (Bubble Sort, Selection Sort, Insertion Sort): São ineficazes para grandes volumes de dados devido à complexidade $O(n^2)$, sendo adequados apenas para conjuntos pequenos.
Algoritmos Eficientes (Merge Sort, Quick Sort e Heap Sort): Com complexidade $O(n \log n)$, são muito mais eficientes em grandes volumes de dados. O Quick Sort é rápido, mas pode ter um caso ruim raro. O Merge Sort é estável e previsível, e o Heap Sort é eficiente em termos de memória, mas um pouco mais lento.
Importância da Escolha do Algoritmo: A escolha do algoritmo impacta diretamente o desempenho em grandes volumes de dados. Para dados pequenos, a diferença é menor, mas para grandes volumes, a escolha errada pode levar a tempos de execução impraticáveis.
Considerações Práticas: Para grandes volumes de dados, algoritmos como Quick Sort ou Merge Sort são mais vantajosos. Quando a memória é limitada, o Heap Sort é uma boa opção devido ao seu baixo consumo de espaço.

CONCLUSÃO FINAL

A análise dos tempos de execução mostra como a eficiência dos algoritmos é importante para ordenar dados. Algoritmos mais simples, como Bubble Sort, Selection Sort e Insertion Sort, não são adequados para grandes quantidades de dados, pois são lentos. Já algoritmos como Merge Sort, Quick Sort e Heap Sort são mais rápidos e funcionam bem com grandes volumes de dados, sendo essenciais em áreas como ciência de dados e desenvolvimento de software. Esse estudo mostra como é importante entender esses algoritmos para tomar decisões em situações que exigem muito processamento e grandes quantidades de dados.

REFERENCIAS

GOOGLE. Google Acadêmico. Disponível em: https://scholar.google.com.br/

Academia.edu
Disponível em: https://www.academia.edu/

LINK REPOSITÓRIO

https://github.com/marcuslopes06/att_avaliativa