UNIVERSIDADE FEDERAL DO PIAUÍ - UFPI
CENTRO DE CIÊNCIAS DA NATUREZA - CCN
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
DISCIPLINA: Estruturas de Dados
PROFESSOR: Raimundo Santos Moura: 2025.2

GRUPO 2:

Antonio Gemesson Sousa de Oliveira

Gleidson Luan Sena Alves

Ivan Vitor Dias de Oliveira

1.INTRODUÇÃO

Um algoritmo de ordenação é um procedimento usado para organizar um conjunto de dados (como números, palavras ou objetos) em uma ordem específica, geralmente crescente ou decrescente. Ele define uma sequência de passos lógicos para comparar e rearranjar os elementos, facilitando operações posteriores, como busca e análise.

2.OBJETIVO

Este relatório visa documentar a implementação e análise comparativa de diferentes algoritmos de ordenação, aplicados a conjuntos de dados textuais de diferentes tamanhos.

3.ESCOLHAS DE PROJETO

Durante o desenvolvimento deste trabalho algumas escolhas foram feitas na estruturação do projeto, sendo elas: quais as versões e linguagem dos algoritmos a serem utilizadas, as ferramentas e técnica necessárias no processo e a interface responsável pela exibição dos resultados de forma intuitiva ao usuário.

3.1 – Algoritmos

Todos os algoritmos analisados foram retirados do acervo de códigos do site Geeks for Geeks, implementados em C++ já nas versões descritas. Foram alterados apenas em relação aos parâmetros de comparação, afim de ordenar strings ao invés de valores numéricos.

3.1.1- Bubble Sort (https://www.geeksforgeeks.org/dsa/bubble-sort-algorithm/)

Implementamos uma versão clássica do Bubble Sort, apenas utilizando uma flag para otimizá-la em relação as iterações desnecessárias para um array já ordenado. A complexidade no pior caso continua sendo O(N²), mas no melhor caso (quando os dados já estão ordenados) cai para O(N) devido à verificação de trocas.

3.1.2- Selection Sort (https://www.geeksforgeeks.org/dsa/selection-sort-algorithm-2/)

Implementamos o Selection Sort clássico, o qual realiza sempre n–1 trocas no total, mas efetua O(N²) comparações independentemente da entrada, sem otimizações para casos parcialmente ordenados.

3.1.3 - Insertion Sort (https://www.geeksforgeeks.org/dsa/insertion-sort-algorithm/)

Implementamos o Insertion Sort clássico, com complexidade O(N²) no pior caso, mas O(N) no melhor caso (quando os dados já estão ordenados). É um algoritmo estável, preservando a ordem relativa de elementos iguais.

3.1.4- Shell Sort (https://www.geeksforgeeks.org/dsa/shell-sort/)

Implementamos o Shell Sort clássico, utilizando um gap inicial de N/2, que vai sendo dividido por 2 a cada iteração até chegar a 1. Mantemos o padrão de Insertion Sort adaptado para gaps, que garante que o array fique parcialmente ordenado a cada iteração.

3.1.5 - Merge Sort (https://www.geeksforgeeks.org/cpp/cpp-program-for-merge-sort/)

Implementamos a versão clássica recursiva top-down do Merge Sort, que opera dividindo recursivamente o array até subarrays unitários e combinando dois subarrays em um único segmento ordenado. Assim, a implementação tem complexidade de tempo O(N log N) e custo adicional de espaço O(N) devido às cópias em vetores auxiliares.

3.1.6 - Quick Sort (https://www.geeksforgeeks.org/dsa/quick-sort-algorithm/)

Implementamos o Quick Sort clássico na versão de Lomuto, na qual a função de partição escolhe sempre o último elemento como pivô e reorganiza o array com elementos menores à esquerda e maiores ou iguais à direita. Possue complexidade O(N log N) quando o pivô divide o array de forma equilibrada, e O(n²) no pior caso, que ocorre quando o pivô é sempre o maior ou menor elemento (como em vetores já ordenados).

3.1.7 - Heap Sort (https://www.geeksforgeeks.org/dsa/heap-sort/)

Implementamos o Heap Sort clássico, que utiliza a estrutura de heap máximo, afim de sempre extrair o maior elemento (na raiz) e o colocar no final do vetor. Esse algoritmo possui complexidade O(N log N) tanto no melhor quanto no pior caso.

3.2 – Ferramentas e Técnicas

Git/Github: Utilizado para desenvolvimento colaborativo e controle de versão.

Python, C++: Linguagem utilizadas no projeto.

Pybind11, SetupTools: Bibliotecas utilizadas para criação de pacotes em C++ para Python.

Pyside6, Matplotlib: Bibliotecas Python utilizados para geração gráfica.

3.3 – Interface

Para a interface do programa foram utilizados os recursos de criação de interface gráfica presentes no *PySide6.* A interface é definida pela classe *MyWindow* (presente no arquivo *MyApp.py*), e o programa roda a partir do arquivo *main.py.* A aplicação possui as seguintes opções para o usuário:

Visualizar grafícos de desempenho: Exibe outra tela com botões para retorno a tela anterior e para geração do gráfico referente ao algoritmo definido no campo superior.

Realizar teste de algoritmos: Exibe outra tela contendo um campo superior para definição do algortimo e três botões: Um para seleção de um arquivo .txt ao qual o algoritmo será aplicado; um para iniciar o teste; e um para retornar a tela anterior. Ao enviar um arquivo para teste, haverá a exibição de outra janela com os resultados da execução.

Comparar algoritmos: Exibe outra tela com um esquema de caixinhas para marcar os algoritmos a serem comparados, além de botões para exibição do gráfico e retorno a tela anterior.

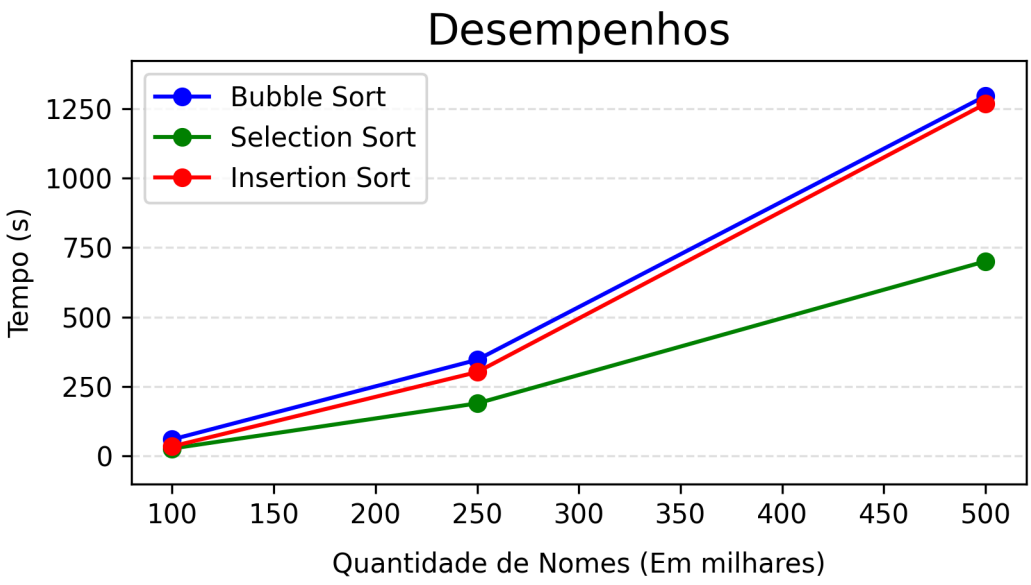
Sair: Sai do programa.

OBS: Por questões de inviabilidade técnica em vários computadores, instituímos uma limitação de 25K para testes realizados nos algoritmos BubbleSort, SelectionSort e Insertion Sort via interface gráfica. Execuções de testes com arquivos que ultrapassem esse limite irão considerar apenas a quantidade suportada.

4.CONCLUSÃO

Segue abaixo os resultados finais do projeto, em formato de gráficos comparativos do desempenho dos algoritmos. Eles foram gerados utilizando a biblioteca Matplotlib do Python, a partir dos dados registrados com a função time.process_time(), que mede o tempo efetivo gasto pela CPU na execução do código. Todos os testes foram executados um a um em um notebook Lenovo ideapad s145 com a seguinte configuração: 20GBs de memória RAM; Processador AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx; e Clock de 3.5 Ghz.

ALGORITMOS QUADRÁTICOS



ALGORITMOS LINEARÍTMICOS