UNIVERSIDADE ESTADUAL DA PARAÍBA
CENTRO DE CIÊNCIA E TECNOLOGIA- CCT
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GUILHERME SILVA HENRIQUES DE SOUSA
LUIS GUSTAVO SOUZA LOURENÇO
WERLYS KAYAN SOARES DOS SANTOS

DOCENTE: FÁBIO LUIZ LEITE JÚNIOR

Relatório submetido no curso de Ciência da Computação da Universidade Estadual da Paraíba, na disciplina de LEDA, correspondente ao período de 2024.2.

CAMPINA GRANDE
2024

SUMÁRIO

1. RESUMO

Este relatório apresenta uma análise detalhada dos resultados obtidos na comparação de desempenho de diversos algoritmos de ordenação aplicados a um extenso Dataset de senhas contendo mais de 650 mil senhas, que são as mais utilizadas. A escolha de algoritmos variou desde os mais clássicos até os mais eficientes. Entre os algoritmos selecionados para essa análise estão o Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Counting Sort, Heap Sort, e Quick Sort com Mediana 3. Cada um desses algoritmos oferece características distintas de complexidade e desempenho.

O objetivo principal deste projeto é descrever os processos de classificação, transformação, filtragem e ordenação aplicados ao conjunto de dados (Dataset), mostrando como cada um dos algoritmos interagiu com as senhas presentes no Dataset. Será definido como as senhas foram preparadas para as análises que garantiram a consistência e a precisão dos testes. Além disso, iremos detalhar as técnicas e algoritmos utilizados, apresentaremos as análises desenvolvidas a partir dos resultados de ordenação. Faremos uma comparação criteriosa do desempenho dos algoritmos, levando em consideração o tempo de execução (em milissegundos).

Outro aspecto importante abordado neste relatório é o ambiente de execução (IDE), que inclui a descrição das configurações de hardware e software onde os testes foram realizados. Discutiremos como o ambiente influencia o desempenho dos algoritmos e as medidas adotadas para garantir a reprodutibilidade dos resultados. Por fim, o relatório incluirá uma visão detalhada de como o projeto foi estruturado, desde a organização dos dados até a implementação dos algoritmos e a análise final.

2. DESENVOLVIMENTO

2.1. ALGORITMOS DE ORDENAÇÃO

No projeto, foram aplicados sete algoritmos de ordenação para avaliar a eficiência, focando principalmente no tempo necessário para ordenar cada arquivo de teste. Os algoritmos selecionados para a análise foram:

- **Selection Sort:** É um algoritmo básico de ordenação que funciona dividindo a lista em duas seções: uma ordenada e outra não ordenada. A cada iteração, ele localiza o menor (ou maior, dependendo da ordem desejada) elemento na parte não ordenada e o troca com o primeiro elemento da seção não ordenada, expandindo assim a parte ordenada.
- **Insertion Sort:** É um algoritmo que organiza a lista dividindo-a em duas seções: uma parte ordenada à esquerda e uma parte não ordenada à direita. A cada passo, ele remove o próximo elemento da parte não ordenada e o insere na posição correta dentro da parte ordenada.
- **Quick Sort:** É um algoritmo de ordenação eficiente que funciona dividindo a lista em duas partes com base em um elemento pivô. Em seguida, o algoritmo ordena as duas partes de forma recursiva.
- **Merge Sort:** É um algoritmo de ordenação eficiente que opera dividindo a lista em duas metades. Cada uma dessas metades é ordenada de forma independente, utilizando a recursividade. Após a ordenação, as duas metades ordenadas são mescladas para formar uma única lista ordenada. Essa abordagem é especialmente eficaz para grandes volumes de dados.
- **Counting Sort:** O algoritmo funciona criando um array auxiliar (ou vetor de contagem) que registra o número de vezes que cada valor aparece no array de entrada. Depois de construir o array de contagem, o algoritmo utiliza essa informação para posicionar cada elemento no array ordenado de saída.
- **Heapsort:** O Heap Sort ordena elementos transformando inicialmente a lista de entrada em uma estrutura de dados chamada heap, que é uma árvore binária completa onde cada nó pai é maior (no caso de um max-heap) ou menor (no caso de um min-heap) que seus filhos.
- **Quick Sort com Mediana 3:** É uma variação do Quick Sort que seleciona o pivô como a mediana de três elementos da lista, geralmente o primeiro, o último e o elemento do meio. Essa abordagem ajuda a melhorar o desempenho do Quick Sort em listas

que já estão parcialmente ordenadas ou possuem padrões específicos.

Cada um desses algoritmos foi aplicado a um arquivo .csv, e o tempo de execução para ordenação foi registrado para análise posterior. A eficiência de cada algoritmo pode variar com base no tamanho e nas características dos dados. Neste relatório, discutiremos detalhadamente os resultados obtidos para cada algoritmo, destacando seu desempenho em diferentes cenários.

2.2. PROCESSAMENTO

Durante a fase de desenvolvimento, foram gerados casos de teste a partir do arquivo original "passwords.csv". Com isso, surgiram arquivos derivados, cada um com seu propósito. Inicialmente, foi atribuída uma classificação às senhas, categorizando-as como "muito ruim", "ruim", "fraca", "boa", "muito boa" ou "sem classificação". Esse processo gerou o arquivo "password_classifier.csv".

Com base no arquivo "password_classifier.csv", foi criado outro, mantendo as mesmas categorias, porém com uma modificação específica, onde o formato da coluna de data foi alterado para "dd/mm/aaaa". Assim, o arquivo resultante dessa etapa foi chamado de "passwords_formated_data.csv".

Arrays foram gerados para representar os cenários de melhor, pior e caso médio para cada tipo de ordenação, considerando diferentes critérios. Para as ordenações baseadas em mês e data, o melhor caso foi definido como os dados organizados em ordem crescente; o caso médio, como os dados em ordem aleatória; e o pior caso, como os dados em ordem decrescente. Já na análise baseada no comprimento das senhas, a lógica foi invertida: a ordenação em ordem decrescente foi considerada o cenário ideal, enquanto a crescente foi definida como o pior caso.

3. MÁQUINA DE EXECUÇÃO

Foi usado a linguagem JAVA com JDK 17.0.12 na IDE Eclipse em uma máquina Intel Core I5-10300H 2.50 GHz, 16GB DDR4 @2933MHz Dual Channel, NVIDIA GeForce GTX 1650 4GB usando o sistema operacional Windows 11 Pro.

4. ESTRUTURA DO CÓDIGO

Leitura e Manipulação de Dados: Para processar os dados contidos no arquivo "passwords.csv" e gerar novos arquivos ".csv" com as alterações solicitadas, foram implementadas três classes específicas, cada uma dedicada a uma etapa do processo. Essas classes foram projetadas para manipular e transformar os dados conforme os requisitos, garantindo que as mudanças necessárias fossem aplicadas corretamente antes de exportar os novos arquivos.

Formatação de Arrays: Três outras classes foram criadas para formatar os arrays nos cenários de melhor, pior e caso médio a partir do arquivo "passwords_formated_9k.csv" (que contém 9000 linhas). Nestas classes, diferentemente do restante do código, foram utilizados os métodos sort, reversed e o auxiliar comparator das bibliotecas do Java.

Algoritmos de Ordenação: Foi desenvolvida uma classe dedicada para cada algoritmo de ordenação, contendo métodos que executam a ordenação com base no tamanho das senhas, no mês e na data. Esses métodos são facilmente acessíveis em qualquer parte do código, permitindo gerar as saídas necessárias e criar arquivos ".csv" específicos para cada tipo de ordenação.

CriarArquivoCsv: Responsável por receber um array e gerar um arquivo ".csv" com base nos dados contidos nele.

GetInput: Utilizada para extrair informações específicas de cada índice do array, onde cada índice é uma string. A classe seleciona o valor correto necessário para os algoritmos de ordenação. Além disso, possui um método que converte um arquivo ".csv" em um array, sendo bastante útil nesse contexto.

ExecutandoTestes: Esta classe cria instâncias de cada uma das classes dos algoritmos de ordenação e executa os métodos conforme a categoria especificada, seja com base no tamanho da senha, mês ou data.

Main: A classe Main é responsável pela execução do projeto, integrando todos os métodos e classes desenvolvidas, o que permite uma execução centralizada e estruturada do programa.

5. RESULTADOS

Na seção a seguir, apresentamos as tabelas de resultados para cada um dos sete algoritmos de ordenação que analisamos. O objetivo deste projeto é realizar ordenações e análises utilizando o arquivo "passwords_formated_data.csv". Embora o código do projeto consiga processar o arquivo original, o tempo necessário para gerar resultados seria muito longo. Por isso, optamos por um arquivo reduzido, contendo 9.000 linhas, o que permite obter resultados mais rapidamente e facilita a reprodução em outras máquinas para fins de comparação. Cada tabela mostra o tempo de execução do algoritmo correspondente aos diferentes casos de teste. As tabelas permitem uma análise comparativa direta do desempenho dos algoritmos.

5.1. TEMPOS DE EXECUÇÃO P/ TAMANHO DA SENHA

Tempo de Execução (milissegundos)	MELHOR CASO	MÉDIO CASO	PIOR CASO
Selection Sort	**2877.00**	**3213.00**	**2939.00**
Insertion Sort	**1.00**	**1273.00**	**2462.00**
Quick Sort	**3261.00**	**405.00**	**1167.00**
Merge Sort	**7.00**	**18.00**	**6.00**
Counting Sort	**2.00**	**3.00**	**2.00**
Heap Sort	**19.00**	**17.00**	**17.00**
Quick Sort (mediana de três)	**18.00**	**14.00**	**11.00**

A partir da análise dessa tabela, notamos que o algoritmo de ordenação mais eficaz para esse caso de teste foi o CountingSort com um tempo de execução de 2-3 milissegundos. O algoritmo MergeSort demonstra um desempenho notável, com 6 milissegundos no pior caso, 18 milissegundos no caso médio e 7 milissegundos no melhor

caso. O algoritmo QuickSort com Mediana de 3 também apresentou um bom desempenho, com tempos de execução variando de 11 a 18 milissegundos em diferentes casos. O algoritmo HeapSort teve um desempenho variado de 17 a 19 milissegundos em diferentes cenários

5.2. TEMPOS DE EXECUÇÃO P/ MÊS

Tempo de Execução (milissegundos)	MELHOR CASO	MÉDIO CASO	PIOR CASO
Selection Sort	**301141.00**	**291280.00**	**290855.00**
Insertion Sort	**64.00**	**135022.00**	**269041.00**
Quick Sort	**291719.00**	**24268.00**	**96922.00**
Merge Sort	**441.00**	**771.00**	**513.00**
Counting Sort	**99.00**	**108.00**	**106.00**
Heap Sort	**1535.00**	**1540.00**	**1423.00**
Quick Sort (mediana de três)	**1303.00**	**1429.00**	**1406.00**

Ao analisar os resultados com base no mês, o algoritmo CountingSort se destaca, apresentando um tempo de execução de 99 milissegundos no melhor caso, 108 milissegundos no caso médio e 106 milissegundos no pior caso. O MergeSort também demonstra um desempenho notável, com tempos de execução consistentemente baixos em todos os casos. O algoritmo QuickSort, embora seja rápido, exibe uma variabilidade significativa, especialmente no melhor caso, com 291719 milissegundos.

5.3. TEMPOS DE EXECUÇÃO P/ DATA

Tempo de Execução (milissegundos)	MELHOR CASO	MÉDIO CASO	PIOR CASO
Selection Sort	**292148.00**	**288099.00**	**286684.00**
Insertion Sort	**62.00**	**145232.00**	**288342.00**
Quick Sort	**289135.00**	**1170.00**	**93415.00**
Merge Sort	**435.00**	**802.00**	**500.00**
Counting Sort	**134.00**	**141.00**	**110.00**
Heap Sort	**1662.00**	**1489.00**	**1441.00**
Quick Sort (mediana de três)	**1167.00**	**1423.00**	**1162.00**

Quando ordenado pela data completa, o algoritmo CountingSort se destaca mais uma vez, com tempos de execução baixos e consistentes em todos os casos. O MergeSort também demonstra um desempenho estável, com tempos de execução entre 435 e 802 milissegundos. O QuickSort, embora seja rápido, mostra uma variabilidade significativa, com o melhor caso atingindo 289135 milissegundos.

6. CONCLUSÃO

Após realizar uma avaliação minuciosa do comportamento de diversos algoritmos de ordenação, identificamos o Counting Sort como a opção com melhor desempenho em termos de tempo de processamento nas situações estudadas. Esse algoritmo, que não se baseia em comparações diretas entre os elementos, mostrou-se extremamente eficiente ao lidar com dados cujos valores estão restritos a um intervalo definido, proporcionando uma ordenação com complexidade linear, O(n + k), onde n é o número de itens e k representa o intervalo dos valores.

Em contrapartida, o MergeSort também apresentou resultados sólidos, sendo particularmente vantajoso quando aplicado a grandes volumes de dados desordenados. Contudo, apesar de sua eficiência em termos de tempo, uma de suas desvantagens é o uso elevado de memória, já que ele requer espaço extra para executar suas operações de fusão. Isso faz com que o MergeSort não seja ideal para cenários onde a otimização do uso de memória ou a economia de recursos computacionais seja uma preocupação fundamental.

Diante disso, a escolha do algoritmo de ordenação mais apropriado deve considerar não apenas as métricas de desempenho em termos de tempo de execução, mas também as características específicas do contexto da aplicação e dos dados a serem organizados. Fatores como a distribuição dos dados, a quantidade de elementos a serem ordenados e as restrições de memória disponíveis são cruciais para a tomada de decisão informada.

Em resumo, escolher o algoritmo de ordenação certo é essencial para otimizar o desempenho do sistema e usar os recursos computacionais de forma mais eficiente.