
**UNIVERSIDADE ESTADUAL DA PARAÍBA**

**CENTRO DE CIÊNCIA E TECNOLOGIA- CCT**

**CIÊNCIA DA COMPUTAÇÃO**

LUCAS ANACLETO BATISTA ALMEIDA

JOÃO GABRIEL DOS SANTOS PEREIRA

# PROJETO 1: TEMA 2 - PASSWORDS

Análise comparativa de algoritmos de ordenação

**CAMPINA GRANDE**

**2024**

**SUMÁRIO**

# INTRODUÇÃO

Este relatório tem como objetivo demonstrar os resultados obtidos na análise comparativa de diferentes algoritmos de ordenação aplicados a um dataset com mais de 600 mil senhas, consideradas as mais comuns utilizadas. Os algoritmos selecionados para este estudo incluem Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Counting Sort, Heapsort e Quick Sort com Mediana 3. Todos os algoritmos escolhidos foram vistos em sala de aula, sendo falado pelo professor sobre sua eficiência e usabilidade.

Nosso objetivo para o que vem a seguir trata-se de explicar como foi feito as classificações, transformações, filtragens e ordenações de um conjunto de dados contendo milhares de senhas. Além disso, falaremos um pouco sobre os algoritmos utilizados e mostraremos análises feitas e deduzidas a partir da ordenação desses dados. Também citaremos sobre o ambiente de execução e como o projeto foi organizado.

# DESENVOLVIMENTO

## 2.1 ALGORITMOS

No projeto, foram utilizados sete algoritmos de ordenação para comparar a eficiência, levando em consideração o tempo de ordenação de cada arquivo de teste. Os algoritmos utilizados foram:

- Selection Sort: Este é um algoritmo simples que divide a lista em duas partes: a parte ordenada e a parte não ordenada. O algoritmo funciona encontrando o menor (ou maior, dependendo da ordem de classificação) elemento na parte não ordenada e trocando-o com o elemento mais à esquerda não ordenado. Este algoritmo é mais eficaz para listas pequenas e para listas onde a troca de elementos é cara.
- Insertion Sort: Este algoritmo divide a lista em duas partes: a parte ordenada à esquerda e a parte não ordenada à direita. O algoritmo pega cada elemento da parte não ordenada e insere na posição correta na parte ordenada. Este algoritmo é eficiente para listas quase ordenadas e listas pequenas.
- Quick Sort: Este 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. Este algoritmo é eficaz para listas grandes e tem uma boa média de tempo de execução.
- Merge Sort: Este é um algoritmo de ordenação eficiente que funciona dividindo a lista em duas metades, ordenando as duas metades de forma independente e, em seguida, mesclando as duas metades ordenadas. Este algoritmo é eficaz para listas grandes e garante um tempo de execução estável.
- Counting Sort: Este é um algoritmo de ordenação não comparativo que funciona contando o número de objetos que possuem valores distintos de chave. Este algoritmo é eficaz para listas onde o intervalo de valores possíveis é limitado.
- Heapsort: Este é um algoritmo de ordenação comparativo que funciona transformando a lista em uma estrutura de dados chamada heap. O processo é repetido até que o heap esteja vazio. Este algoritmo é eficaz para listas grandes e garante um tempo de execução estável.
- Quick Sort com Mediana 3: Esta é uma variação do Quick Sort que funciona escolhendo o pivô como a mediana de três elementos da lista. Isso pode melhorar o

desempenho do Quick Sort para certos tipos de listas, especialmente aquelas que já estão parcialmente ordenadas.

Cada um desses algoritmos foi aplicado a um arquivo .csv, e o tempo de ordenação foi registrado para análise posterior. A eficiência de cada algoritmo pode variar dependendo do tamanho e da natureza dos dados. Ainda neste relatório, discutiremos os resultados obtidos de cada algoritmo.

## 2.2 CASOS DE TESTE

A etapa de desenvolvimento incluiu a geração de casos de testes a partir do arquivo original "passwords.csv". Dessa forma, foram produzidos arquivos derivados com diferentes propósitos. Primeiramente, foi adicionada uma classificação para todas as senhas, segmentando-as em categorias como "muito ruim", "ruim", "fraca", "boa", "muito boa" e "sem classificação". Isso resultou no arquivo "password_classifier.csv".

A partir do arquivo "password_classifier.csv", foram gerados outros com a mesma categoria, mas com uma alteração específica: a formatação da coluna de data para o formato "dd/mm/aaaa". O arquivo resultante desse processo foi denominado "passwords_formated_data.csv".

Posteriormente, com o objetivo de agilizar o processo de análise, foi construído um novo arquivo, o "passwords_test.csv", a partir do "passwords_formated_data.csv". Esse novo arquivo é significativamente menor, permitindo a obtenção de resultados em um tempo hábil e será o utilizado nas ordenações.

Com o arquivo "passwords_test.csv", foram criados arrays representando o melhor, o pior e o caso médio para cada tipo de ordenação. Essa segmentação foi orientada pelo professor, considerando diferentes critérios. Para as ordenações baseadas no mês e na data, o melhor caso foi definido como os dados em ordem crescente, o caso médio como os dados sem formatação e o pior caso como os dados em ordem decrescente. Para a análise baseada no tamanho da senha, seguiu-se o mesmo critério, no entanto, a ordenação seria em ordem decrescente, ou seja, para pior e melhor caso foi considerado o inverso dos demais.

## 2.3 AMBIENTE DE EXECUÇÃO

O projeto foi desenvolvido utilizando a IDE IntelliJ IDEA, uma das IDEs mais populares para desenvolvimento em Java, que oferece uma variedade de recursos para facilitar o desenvolvimento.

O projeto foi estruturado em várias classes, cada uma com funções específicas, a fim de garantir uma organização eficiente e evitar repetição de código.

**Estrutura do Código**

**Leitura e Manipulação de Dados:** Para consumir os dados do arquivo "passwords.csv" e retornar novos arquivos ".csv" com as mudanças exigidas, foram utilizadas três classes dedicadas.

**Formatação de Arrays:** Outras três classes foram desenvolvidas para retornar os arrays formatados para os casos de melhor, pior e médio, a partir do arquivo especificado. Para o projeto, o arquivo "passwords_test.csv" foi utilizado na construção. Nessas classes, em contraste com o restante do código, foram usados os métodos sort e reversed, bem como o auxiliar Comparator, que fazem parte das bibliotecas do Java.

**Algoritmos de Ordenação:** Foi criada uma classe para cada algoritmo de ordenação, cada uma com métodos que realizam as ordenações baseadas no tamanho da senha, mês e data. Esses métodos podem ser facilmente chamados em qualquer parte do código, gerando as saídas e criando os arquivos ".csv" para cada ordenação.

**Classes Auxiliares:**

CreateFileCsv: Responsável por consumir um array e criar um arquivo ".csv" a partir dele.

GetVariables: Utilizada para extrair os dados necessários de cada índice do array. Cada índice representa uma string e é feita uma seleção para pegar o valor correto, sendo de grande utilizadas para os algoritmos de ordenação. Esta classe também contém um método que

transforma um arquivo ".csv" em um array, o que será também bastante utilizado.

RunTests: Esta classe cria instâncias para cada classe dos algoritmos de ordenação e os chama de acordo com sua categoria, seja para o tamanho da senha, mês ou data.

**Classe Principal:** A classe Main é a executada e engloba todos os métodos e classes criadas, proporcionando uma execução centralizada e organizada do projeto.

## 2.4 ANÁLISE COMPARATIVA DOS ALGORITMOS

Na seção a seguir, apresentamos as tabelas de resultados para cada um dos sete algoritmos de ordenação que analisamos: Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Counting Sort, Heap Sort e Quick Sort com Mediana 3. 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. Ao examinar os resultados, podemos observar como o tempo de execução de cada algoritmo varia com os casos de testes propostos pelo orientador.

**TEMPOS DE EXECUÇÃO A PARTIR DO TAMANHO DA SENHA**

| ( ~em milissegundos ) | PIOR CASO | MÉDIO CASO | MELHOR CASO |
|---|---|---|---|
| SELECTION SORT | 316,00 | 327,00 | 312,00 |
| INSERTION SORT | 255,00 | 129,00 | 0,00 |
| QUICK SORT | 123,00 | 45,00 | 293,00 |
| MERGE SORT | 1,00 | 6,00 | 3,00 |
| COUNTING SORT | 0,00 | 0,00 | 0,00 |
| HEAPSORT | 3,00 | 3,00 | 4,00 |
| QUICK SORT COM MEDIANA 3 | 3,00 | 3,00 | 3,00 |

Os resultados obtidos a partir da ordenação pelo tamanho da senha mostram que o algoritmo CountingSort é o mais eficiente, com um tempo de execução de 0 milissegundos em todos os casos. Em seguida, o algoritmo HeapSort demonstra um desempenho notável, com 4 milissegundos no melhor caso, 3 milissegundos no caso médio e 3 milissegundos no pior caso. O algoritmo MergeSort também apresentou um bom desempenho, com tempos de execução variando de 1 a 6 milissegundos em diferentes casos. O algoritmo QuickSort com Mediana de 3 teve um desempenho constante em todos os casos, com 3 milissegundos em

cada cenário. O QuickSort tradicional, embora apresente tempos de execução baixos, mostra uma variabilidade significativa, principalmente no pior caso, onde atinge 123 milissegundos.

**TEMPOS DE EXECUÇÃO A PARTIR DO MÊS**

| ( ~em milissegundos ) | PIOR CASO | MÉDIO CASO | MELHOR CASO |
|---|---|---|---|
| SELECTION SORT | 22981,00 | 22973,00 | 23809,00 |
| INSERTION SORT | 21365,00 | 11680,00 | 15,00 |
| QUICK SORT | 10593,00 | 2018,00 | 23092,00 |
| MERGE SORT | 112,00 | 160,00 | 95,00 |
| COUNTING SORT | 29,00 | 23,00 | 23,00 |
| HEAPSORT | 283,00 | 304,00 | 285,00 |
| QUICK SORT COM MEDIANA 3 | 258,00 | 267,00 | 290,00 |

Ao analisar os resultados com base no mês, o algoritmo MergeSort se destaca, apresentando um tempo de execução de 95 milissegundos no melhor caso, 160 milissegundos no caso médio e 112 milissegundos no pior caso. O CountingSort 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 pior caso, com 10593 milissegundos.

**TEMPOS DE EXECUÇÃO A PARTIR DA DATA**

| ( ~em milissegundos ) | PIOR CASO | MÉDIO CASO | MELHOR CASO |
|---|---|---|---|
| SELECTION SORT | 23381,00 | 23961,00 | 23455,00 |
| INSERTION SORT | 23346,00 | 11684,00 | 14,00 |
| QUICK SORT | 11173,00 | 210,00 | 24896,00 |
| MERGE SORT | 98,00 | 167,00 | 94,00 |
| COUNTING SORT | 35,00 | 47,00 | 52,00 |
| HEAPSORT | 314,00 | 326,00 | 332,00 |
| QUICK SORT COM MEDIANA 3 | 261,00 | 273,00 | 264,00 |

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 94 e 167 milissegundos. O QuickSort, embora seja rápido, mostra uma variabilidade significativa, com o pior caso atingindo 11173 milissegundos.

## CONCLUSÃO

Concluímos que, ao analisar o desempenho dos algoritmos, observamos que, em termos de tempo de execução, o CountingSort se destacou como a melhor opção em todos os cenários. O MergeSort também apresentou resultados consistentemente bons em todos os casos. No entanto, deve-se observar que o MergeSort pode exigir mais espaço de memória do que outros algoritmos. Portanto, a escolha do algoritmo de ordenação mais apropriado deve ser feita considerando o contexto específico de aplicação e a natureza dos dados a serem ordenados.

A estrutura do código, desenvolvida na IDE IntelliJ IDEA, proporcionou uma organização eficiente e facilitou o desenvolvimento, com classes dedicadas para leitura e manipulação de dados, formatação de arrays, implementação dos algoritmos de ordenação e classes auxiliares. O ambiente de execução foi controlado e permitiu uma análise precisa dos algoritmos, com os casos de teste construídos.

Em suma, a análise demonstra que a escolha do algoritmo de ordenação apropriado é crucial para otimizar o desempenho, reduzir o tempo de execução e melhorar a eficiência do sistema.