Interface gráfica do usuário, Texto, Aplicativo

O conteúdo gerado por IA pode estar incorreto.

JOÃO VICTOR MARINHO LOURENÇO

GUSTAVO AZEVEDO LÉLIS FÁRIAS

ANÁLISE COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO

**BOLSA DE VALORES BOVESPA 1994-2020**

CAMPINA GRANDE – PB

17 de maio de 2025

Interface gráfica do usuário, Texto, Aplicativo

O conteúdo gerado por IA pode estar incorreto.

JOÃO VICTOR MARINHO LOURENÇO

GUSTAVO AZEVEDO LÉLIS FÁRIAS

Relatório apresentado no curso de

Ciência da Computação da

Universidade Estadual da Paraíba e na

disciplina de Laboratório de Estrutura

de Dados referente ao período 2025.1

**Professor: Fábio Leite**

CAMPINA GRANDE – PB

17 de maio de 2025

**SUMÁRIO**

**1. INTRODUÇÃO.......................................................................................................................4**

**2. METODOLOGIA...................................................................................................................5**

**3. ALGORITMOS DE ORDENAÇÃO.....................................................................................5**

**4. CASOS DE TESTES...............................................................................................................6**

**5. RESULTADOS E ANÁLISES DOS ALGORITMOS..........................................................7**

**6.CONCLUSÃO..........................................................................................................................9**

**1. INTRODUÇÃO**

Este relatório tem como objetivo demonstrar os resultados obtidos na análise de diferentes algoritmos de ordenação, aplicados a um dataset com informações relacionadas às ações negociadas na BOVESPA de 1924 até 2020. Os algoritmos selecionados para este estudo incluem, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Counting Sort, Heap Sort e Quick Sort com Mediana 3.

O nosso projeto foi escrito em Java e possui implementações para que seja possível organizar o fluxo de execução, certificando-se de que os algoritmos de ordenação propostos anteriormente, estejam corretamente aplicados aos dados fornecidos contidos em um arquivo .csv (b3\_stocks\_1994\_2020.csv), contribuindo para a captura dos resultados finais correspondente às negociações da BOVESPA, entre 1994 e 2020.

As modificações que são realizadas no arquivo b3\_stocks\_1924\_2020.csv estão logo abaixo:

1. Criação do arquivo chamado b3stocks\_T1.csv, no qual é transformado a data para o formato DD/MM/AAAA (que anteriormente estava no formato YYYY-MM-DD)
2. Criação do arquivo b3stocks\_F1.csv, responsável por deixar apenas um registro por dia, considerando aquele que possui o maior volume negociado em bolsa.
3. Filtragem do arquivo b3stocks\_T1.csv, para que fique apenas os registros que possuírem volume acima da média diária.

Além disso, o código também realiza algumas ordenações no arquivo b3stocks\_T1.csv, segue as ordenações propostas:

1. Ordenar os tickets em ordem alfabética
2. Ordenar o arquivo de transações pelo volume, em ordem crescente.

Ordenar o arquivo de transações pelas maiores variações diárias em ordem decrescente.

**2. METODOLOGIA**

Para realizar este estudo, todos os algoritmos foram implementados utilizando a linguagem Java (versão JDK-20) e as implementações foram executadas por meio do IntelliJ IDEA (versão 2023.2.5). Para analisar a eficiência dos diversos algoritmos de ordenação, as tarefas acima foram executadas usando os seguintes algoritmos: Insertion Sort, Selection Sort, Quick Sort, Quick Sort com mediana de 3, Merge Sort, Heap Sort e Counting Sort. Cada um desses algoritmos foi executado 3 vezes, de modo a se obter o caso médio, o melhor caso e o pior caso.

Além disso, é importante ressaltar as especificações da máquina utilizada para realizar as operações:

|  |  |
| --- | --- |
| Placa Mãe | B660M Phantom Gaming 4 |
| Processador | Intel Core i5 12600K |
| Memória RAM | 16GB DDR4 3666MHz |
| Armazenamento | 500GB de SSD + 1TB de HD |
| Placa de vídeo | GeForce RTX 4060ti |
| Sistema Operacional | Windows 11 |

**3. ALGORITMOS DE ORDENAÇÃO**

O que podemos falar um pouco sobre cada algoritmo utilizado no código, é:

• **Insertion Sort**: esse método organiza a lista inserindo cada item no lugar certo, comparando com os que já foram colocados antes. Vai da esquerda para a direita e empurra os maiores para a frente quando encontra um menor. É bem simples e funciona bem em listas pequenas ou quase organizadas, mas fica lento com listas grandes (complexidade O(n²)).

• **Selection Sort**: percorre a lista várias vezes procurando sempre o menor valor e colocando ele na posição certa. É fácil de entender e implementar, mas não é rápido — também tem complexidade O(n²), então não é indicado pra listas muito grandes.

• **Quick Sort**: é um dos algoritmos mais eficientes. Ele escolhe um elemento como pivô, separa os menores e maiores em relação a ele, e repete isso nas partes menores. Normalmente roda em O (n log n), mas se o pivô for mal escolhido, pode piorar para O(n²).

• **Quick Sort com Mediana de 3**: essa variação do Quick Sort melhora a escolha do pivô pegando a mediana entre três elementos (início, meio e fim da lista). Isso ajuda a evitar divisões ruins e deixa o desempenho mais estável e rápido na prática.

• **Merge Sort**: divide a lista ao meio várias vezes até sobrar um item em cada parte, depois junta tudo de volta já em ordem. É rápido (O (n log n)) e muito confiável, mas usa mais memória, porque precisa de espaço extra para fazer as fusões.

• **Heap Sort**: transforma a lista em uma estrutura chamada heap (um tipo de árvore) e vai tirando os maiores valores um por um, colocando na posição certa. Tem boa performance (O (n log n)) e não precisa de memória extra, o que o torna eficiente e equilibrado.

• **Counting Sort**: diferente dos outros, esse não compara os elementos. Ele conta quantas vezes cada número aparece (dentro de um intervalo conhecido) e usa isso para montar a lista ordenada. É mais rápido (O (n + k)), mas só serve bem pra inteiros com pouca variação, caso contrário, pode acabar usando muita memória.

**4. CASOS DE TESTES**

Os testes foram feitos utilizando o arquivo b3\_stocks\_1994\_2020, que possui 1883204 linhas de registro.

Logo abaixo, estarão listados alguns tópicos de classes e funções principais que foram utilizados para escrever o programa, bem como, a função de cada uma:

- SelectionSort: Ordena o arquivo utilizando o Selection Sort;

- InsertionSort: Ordena o arquivo utilizando o Insertion Sort;

- QuickSort: Ordena o arquivo utilizando o Quick Sort;

- QuickSortMedianOfThree: Ordena o arquivo utilizando o Quick Sort com mediana de 3;

- MergeSort: Ordena o arquivo utilizando o Merge Sort;

- HeapSort: Ordena o arquivo utilizando o Heap Sort;

- CountingSort: Ordena o arquivo utilizando o Couting Sort;

- Main: Classe responsável por executar o código, bem como: exibir a memória

utilizada, classificação, execução e algoritmo. Além do tempo em milissegundos, que foi calculado utilizando usando a função *currentTimeMillis.*

**5. RESULTADOS E ANÁLISES DOS ALGORITMOS**

Nesta seção, apresentamos as tabelas comparativas de todos os algoritmos utilizados como base para este estudo, sendo eles: Selection Sort, Insertion Sort, Counting Sort, Heap Sort, Quick Sort e Quick Sort com mediana de 3. As tabelas comparam o tempo de execução dos algoritmos em milissegundos (ms).

Vale ressaltar que os algoritmos foram testados para três cenários diferentes, que são: melhor caso, pior caso e caso médio. Entretanto, o algoritmo do Counting Sort não obteve resultados, tendo em vista que ele não é capaz de ordenar caracteres alfabéticos e nem valores flutuantes, por esse motivo não irá aparecer nas tabelas a seguir.

Primeiramente teremos a tabela de ordenação dos tickets em ordem alfabética:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algoritmos |  | **Melhor Caso** | **Caso Médio** | **Pior Caso** |
| **Selection Sort** |  | **601200 ms** | **611220 ms** | **820544 ms** |
| **Insertion Sort** |  | **438502 ms** | **394600 ms** | **952200 ms** |
| **Merge Sort** |  | **1196 ms** | **1272 ms** | **1890 ms** |
| **Quick Sort** |  | **48668 ms** | **150974 ms** | **155217 ms** |
| **Quick Sort**  **mediana de 3** | **com** | **93643 ms** | **96529 ms** | **110214 ms** |
| **Heap Sort** |  | **2652 ms** | **3117 ms** | **3495 ms** |
| **Counting Sort** |  | **-** | **-** | **-** |

Após a ordenação dos tickets, temos agora a tabela de ordenação dos volumes em ordem crescente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algoritmos |  | **Melhor Caso** | **Caso Médio** | **Pior Caso** |
| **Selection Sort** |  | **93120 ms** | **200330 ms** | **420222 ms** |
| **Insertion Sort** |  | **150502 ms** | **200600 ms** | **354100 ms** |
| **Merge Sort** |  | **640 ms** | **683 ms** | **1096 ms** |
| **Quick Sort** |  | **1800 ms** | **15700 ms** | **17355 ms** |
| **Quick Sort**  **mediana de 3** | **com** | **2140 ms** | **2390 ms** | **3984 ms** |
| **Heap Sort** |  | **3114 ms** | **3409 ms** | **3474 ms** |
| **Counting Sort** |  | **-** | **-** | **-** |

Por fim, teremos a tabela de ordenação dos valores flutuantes em ordem decrescente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algoritmos |  | **Melhor Caso** | **Caso Médio** | **Pior Caso** |
| **Selection Sort** |  | **103200 ms** | **218300 ms** | **77450 ms** |
| **Insertion Sort** |  | **23850 ms** | **192340 ms** | **100556 ms** |
| **Merge Sort** |  | **867 ms** | **959 ms** | **1179 ms** |
| **Quick Sort** |  | **6400 ms** | **6900 ms** | **5400 ms** |
| **Quick Sort**  **mediana de 3** | **com** | **7800 ms** | **2000 ms** | **6000 ms** |
| **Heap Sort** |  | **2660 ms** | **2922 ms** | **2990 ms** |
| **Counting Sort** |  | **-** | **-** | **-** |

Após examinar as tabelas e gráficos fornecidos, observamos que Heap Sort e Merge Sort se destacam como os algoritmos mais eficazes em geral, tendo em vista que ambos possuem complexidade O(*n* log *n*), garantindo um bom desempenho mesmo em grandes conjuntos de dados. Merge Sort tem a vantagem adicional de ser estável, enquanto Heap Sort tende a ser mais eficiente em termos de uso de memória.

Apesar de o Quick Sort ter apresentado um excelente desempenho no melhor caso ao ordenar os volumes em ordem crescente como visto anteriormente, ele sofre no pior caso, onde sua complexidade O(*n²*) se manifesta, tornando-o menos confiável em algumas situações. A versão Quick Sort com mediana de 3 melhora a escolha do pivô, mas em alguns casos, a sobrecarga adicional reduziu sua eficiência em comparação com o Quick Sort padrão. Por fim, o Counting Sort não foi testado, pois ele não é adequado para ordenar dados alfabéticos ou valores flutuantes, sendo mais eficaz quando os dados são números inteiros com um intervalo conhecido.

**6. CONCLUSÃO**

Com base nas informações e tabelas analisadas, conclui-se que a escolha do algoritmo de ordenação mais adequado é fundamental para garantir um bom desempenho. Algoritmos como Merge Sort e Heap Sort se destacam como as melhores opções para grandes volumes de dados e aplicações gerais, devido à sua consistência e eficiência em diferentes cenários.

O Quick Sort também representa uma alternativa viável, especialmente quando o pior caso pode ser evitado ou minimizado. A variação com mediana de 3 contribui significativamente para isso, ao aprimorar a escolha do pivô e tornar as divisões mais equilibradas.

Em contrapartida, algoritmos mais simples como o Selection Sort e o Insertion Sort demonstram desempenho inferior em listas maiores, sendo recomendados apenas para situações específicas, como conjuntos de dados pequenos ou quase ordenados.

Portanto, o estudo reforça a importância de escolher o algoritmo de ordenação considerando o perfil dos dados e os requisitos de performance. Para cenários com grandes volumes de informação, algoritmos com complexidade O(n log n) são os mais indicados, por oferecerem um bom equilíbrio entre velocidade e confiabilidade.