UEPB
Universidade Estadual da Paraíba
Departamento de Computação – DC
Laboratório de estrutura de dados
Professor: Fabio Luiz Leite Junior

ARTHUR MARQUES ARAÚJO

ASCENDINO MARTINS DE AZEVEDO NETO

KEVIN SANTOS MARQUES

ANÁLISE COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO COM A BOLSA DE VALORES BOVESPA 1994-2020

CAMPINA GRANDE – PB

27 de setembro de 2024

ARTHUR MARQUES ARAÚJO

ASCENDINO MARTINS DE AZEVEDO NETO

KEVIN SANTOS MARQUES

ANÁLISE COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO COM A BOLSA DE VALORES BOVESPA 1994-2020

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 2024.2

Professor: Fabio Luiz Leite Junior

CAMPINA GRANDE – PB

27 de setembro de 2024

SUMÁRIO

INTRODUÇÃO

O presente relatório tem como objetivo apresentar um estudo detalhado sobre a manipulação e análise de dados de negociações realizadas na BOVESPA (Bolsa de Valores Brasileira), no período compreendido entre 1994 e 2020. O desenvolvimento desse projeto teve como principal intuito manipular e analisar as aplicações dos algoritmos de ordenação em contextos do mundo real.

Com o objetivo de avaliar a eficiência e o desempenho de diferentes algoritmos de ordenação sobre os dados transformados, foram aplicadas várias técnicas de ordenação: Insertion Sort, Selection Sort, Quicksort, Quicksort com mediana de 3, Merge Sort, Heap Sort e Counting Sort. Esses algoritmos foram executados em diferentes cenários, analisando o desempenho em termos de tempo de execução e complexidade computacional nos casos médios, melhores e piores. Para cada uma das execuções, novos arquivos CSV foram gerados, apresentando os dados em diferentes formas de ordenação: os tickers foram ordenados alfabeticamente, os volumes negociados de forma crescente, e as variações de preço, de maneira decrescente.

Este estudo incluiu não apenas a transformação e filtragem eficiente de um grande volume de dados históricos, mas também a avaliação comparativa dos principais algoritmos de ordenação em cenários práticos. Os resultados obtidos demonstram a relevância da escolha do algoritmo adequado para diferentes tipos de dados e critério de desempenho, contribuindo assim para a análise do comportamento dos algoritmos e para uma compreensão mais profunda das operações de mercado na BOVESPA ao longo do período investigado.

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), visando manipular e modificar as informações contidas em um arquivo .csv (b3_stocks_1994_2020.csv). Esse arquivo corresponde às informações das negociações da BOVESPA (Bolsa de Valores Brasileira) entre os anos de 1994 à 2020.

As modificações realizadas no arquivo “b3_stocks_1994_2020.csv”, incluíram:

- A criação de um arquivo “b3stocks_T1.csv”, no qual o formato da data é DD/MM/AAAA (que antes estava no formato AAAA/MM/DD);

- A criação de um arquivo “b3stocks_F1.csv” para fazer uma filtragem de modo que fique apenas um registro por dia. Esse registro deve ser apenas aquele que possuir o maior volume negociado em bolsa.
- A realização de uma filtragem no arquivo “b3stocks_T1.csv”, para que fique apenas os registros que possuírem volume negociado acima da média diária.

O código também realiza algumas ordenações no arquivo “b3stocks_T1.csv”, incluindo a ordenação dos tickers por ordem alfabética, a ordenação do volume por ordem crescente e a ordenação das variações em ordem decrescente.

Para analisar a eficiência dos diversos algoritmos de ordenação, as tarefas acima foram executadas usando os seguintes algoritmos: Insertion Sort, Selection Sort, Quicksort, Quicksort 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. Para cada uma dessas execuções, o código gera um novo arquivo .csv, para mostrar, assim, as informações de forma ordenada.

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

Tabela 01: Características principais do computador utilizado.

Placa Mãe	BRX H110
Processador	Intel Core i7 8700
Memória RAM	16GB DDR4 3200MHz
Armazenamento	500GB de SSD + 500GB de HD
Placa de vídeo	rx 580 2048sp
Sistema Operacional	Windows 10

Fonte: Autoria Própria.

ALGORITMOS DE ORDENAÇÃO

A eficiência de cada algoritmo pode variar dependendo do tamanho e da natureza dos dados. Entre os sete algoritmos utilizados no código, destacam-se as principais características:

- **Insertion Sort:** ordena uma lista inserindo cada elemento na posição correta em relação aos elementos já ordenados. Ele percorre a lista da esquerda para a direita, deslocando os maiores elementos à medida que encontra os menores. Embora seja simples, sua complexidade O (n^2) o torna ineficiente para grandes listas, mas é útil para pequenas ou quase ordenadas.

- **Selection Sort**: percorre a lista várias vezes, selecionando o menor elemento a cada passagem e colocando-o na posição correta. Apesar de ser fácil de entender, seu desempenho O (n^2) o torna impraticável para grandes volumes de dados, sendo menos eficiente que outros algoritmos mais avançados.

- **Quicksort**; é um algoritmo eficiente que escolhe um pivô, organiza os elementos em torno dele e aplica o processo recursivamente às sublistas. Sua complexidade é O (n log n) na maioria dos casos, mas pode chegar a O (n^2) em cenários desfavoráveis, como quando o pivô é mal escolhido.

- **Quicksort com Mediana de 3:** essa variação do Quicksort melhora a escolha do pivô usando a mediana de três elementos (o primeiro, o último e o central), reduzindo as chances de divisões desequilibradas e melhorando o desempenho prático em relação ao Quicksort tradicional.

- **Merge Sort**: divide recursivamente a lista em duas partes até que cada sublista tenha um único elemento, e depois as mescla (merge) em ordem crescente. Ele garante uma complexidade de O (n log n), sendo eficiente para grandes listas, mas consome mais memória do que o Quicksort.

- **Heap Sort**: transforma a lista em um heap máximo e, repetidamente, extrai o maior elemento, colocando-o na posição correta. Ele possui complexidade O (n log n) e é eficiente tanto em tempo quanto em espaço, pois não exige memória adicional significativa.

- **Counting Sort:** é não comparativo e funciona contando as ocorrências de cada valor em uma lista de inteiros dentro de um intervalo limitado. Ele é muito rápido (O(n+k)), mas requer espaço proporcional ao intervalo de valores, sendo inadequado para listas com grande variação de números.

ESTRUTURA DO CÓDIGO

Ao total, o código foi construído em 13 classes, nas quais suas funções incluem:

- **FiltrarRegistro:** cria o arquivo b3stocks_F1.csv;
- **TransformarData:** cria o arquivo b3stocks_T1.csv;
- **FiltrarMediaDiaria:** faz a filtragem no arquivo b3stocks_T1.csv;
- **Registro:** define o registro e seus campos (data, ticker, open, close, high, low, volume e linha);
- **Funcoes:** define as principais funções que serão utilizadas pelas demais classes herdeiras;
- **InsertionSort:** ordena o arquivo utilizando insertion sort;
- **SelectionSort:** ordena o arquivo utilizando selection sort;
- **QuickSort:** ordena o arquivo utilizando quick sort;
- **QuickSortMedianaDe3:** ordena o arquivo utilizando quicksort com mediana de 3;
- **MergeSort:** ordena o arquivo utilizando merge sort;
- **HeapSort:** ordena o arquivo utilizando heap sort;
- **CountingSort:** ordena o arquivo utilizando counting sort;
- **Main:** executa o código.

Além disso, para capturar o tempo de execução de cada algoritmo, foi utilizada a função do java currentTimeMillis.

CASOS DE TESTES

A etapa de desenvolvimento incluiu a geração de casos de testes a partir do arquivo original "b3_stocks_1994_2020.csv", que possui 1883204 linhas de registro e é formada pela data, o papel negociado (ticker), valor de abertura e fechamento, valor de alta e baixa do dia e o volume negociado. Assim, foram produzidos arquivos derivados com diferentes propósitos.

Primeiramente, foi realizada uma alteração no datetime, que passou para o formato DD/MM/AAAA. Isso resultou no arquivo "b3stocks_T1.csv". A partir do arquivo "b3stocks_T1.csv", foi gerada uma filtragem específica: deve ficar constando apenas o registro diário com maior volume negociado em bolsa. O arquivo resultante desse processo foi denominado "b3stocks_F1.csv". E, ainda considerando o arquivo "b3stocks_T1.csv", foi

realizada uma filtragem de modo que ficassem apenas os registros que possuíssem volume negociado acima da média diária.

Posteriormente, para as ordenações, foi considerado como entrada dos dados o arquivo resultante da transformação da data “b3stocks_T1.csv” para a criação de arrays representando o melhor, o pior e o médio caso para cada tipo de ordenação. Essa segmentação foi orientada pelo professor, considerando diferentes critérios, em que foi gerado um arquivo para cada algoritmo de ordenação e o tipo de caso.

Para a ordenação do arquivo completo de transações pelos nomes dos papéis negociados (campo ticker) em ordem alfabética, foram criados os arquivos: “b3stocks_ticker_insertionSort_medioCaso.csv”, “b3stocks_ticker_insertionSort_piorCaso .csv” e “b3stocks_ticker_insertionSort_melhorCaso.csv”.

Para a ordenação do arquivo de transações pelo volume (campo volume) do menor para o maior, foram criados os arquivos: “b3stocks_volume_insertionSort_medioCaso.csv”, “b3stocks_volume_insertionSort_piorCaso.csv” e “b3stocks_volume_insertionSort_melhor Caso.csv”.

Já para a ordenação do arquivo de transações pelas maiores variações diárias (campo High - Low) do maior para o menor, foram criados os arquivos: “b3stocks_fluctuations_insertionSort_medioCaso.csv”, “b3stocks_fluctuations_insertionSort_ piorCaso.csv” e “b3stocks_fluctuations_insertionSort_melhorCaso.csv”.

ANÁLISE COMPARATIVA DOS ALGORITMOS

Foram criadas tabelas comparativas do tempo de execução em milissegundos (ms) de cada algoritmo de ordenação, com base nos resultados de três ordenações distintas: a ordenação alfabética dos registros com base em seus tickets, a ordenação crescente dos registros com base em seus volumes, e a ordenação decrescente dos registros com base em suas variações diárias.

Os algoritmos foram testados em três cenários diferentes: o melhor caso, o pior caso e o caso médio. É importante ressaltar que o algoritmo counting sort não é capaz de ordenar caracteres alfabéticos (o que o torna inválido para a primeira ordenação) nem valores flutuantes (o que o torna inválido para a segunda e terceira ordenações), logo, ele não aparece nas tabelas.

Tabela 02: Ordenação de tickers em ordem alfabética.

	Caso Melhor	Caso Médio	Caso Pior
Insertion Sort	443623 ms	401912 ms	923765 ms
Selection Sort	642521 ms	667311 ms	883504 ms
Quick Sort	51429 ms	163942 ms	171134 ms
Quick Sort com mediana de 3	301873 ms	298091 ms	602421 ms
Merge Sort	1802 ms	1581 ms	1998 ms
Heap Sort	1712 ms	1433 ms	1656 ms
Counting Sort	-	-	-

Fonte: Autoria Própria.

Nesta tabela, é possível observar que os algoritmos Insertion Sort e Selection Sort têm os piores desempenhos, especialmente no pior caso, devido à sua complexidade quadrática O (n^2). Isso os torna inadequados para listas grandes, com tempos de execução que chegam a 923765 ms e 883504 ms ms, respectivamente. Por outro lado, algoritmos mais eficientes, como o Quicksort e sua variação com Mediana de 3, apresentam resultados razoavelmente bons, com tempos de 51429 ms a 602421 ms. Já o Merge Sort e o Heap Sort demonstram o melhor desempenho, com tempos baixos e consistentes, evidenciando que são escolhas mais apropriadas para tarefas de ordenação de strings.

Tabela 03: Ordenação dos volumes em ordem crescente.

	Melhor Caso	Médio Caso	Pior Caso
Insertion Sort	161504 ms	223607 ms	378133 ms
Selection Sort	98134 ms	219599 ms	431044 ms
Quick Sort	16135 ms	2017 ms	19220 ms
Quick Sort com mediana de 3	13349 ms	3180 ms	11789 ms
Merge Sort	2701 ms	2897 ms	4968 ms
Heap Sort	2239 ms	4986 ms	2596 ms
Counting Sort	-	-	-

Fonte: Autoria Própria.

Na Tabela 03, os algoritmos Quicksort e Quicksort com Mediana de 3 continuam tendo bons tempos, especialmente no caso médio, com tempos de 2017 ms e 3180 ms. E o Merge Sort e o Heap Sort continuam sendo os mais eficientes. Em contrapartida, o Insertion Sort e o Selection Sort, novamente, mostram-se inadequados para esse tipo de tarefa, com tempos de execução mais elevados, especialmente no pior caso, onde atingem 378133 ms e 431044 ms.

Tabela 04: Ordenação de variações diárias em ordem decrescente.

	Melhor Caso	Médio Caso	Pior Caso
Insertion Sort	24091 ms	217310 ms	252789 ms
Selection Sort	113582 ms	267102 ms	234039 ms
Quick Sort	6550 ms	6870 ms	5509 ms
Quick Sort com mediana de 3	7798 ms	2609 ms	6123 ms
Merge Sort	1998 ms	2103 ms	2798 ms
Heap Sort	1409 ms	2157 ms	2732 ms
Counting Sort	-	-	-

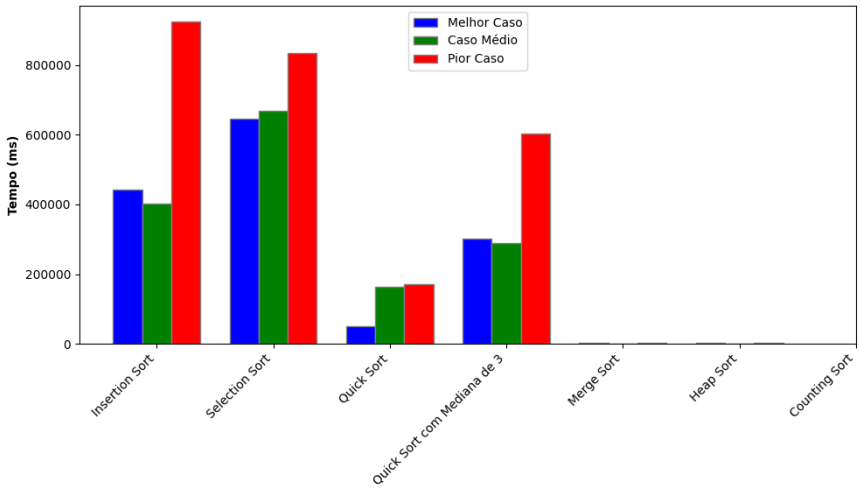
Fonte: Autoria Própria.

Já na Tabela 04, o Insertion Sort trouxe um tempo consideravelmente bom no melhor caso ao comparar com o Selection Sort. No entanto, para o médio caso e pior caso, ambos continuaram com os piores tempos. Os algoritmos Quicksort e Quicksort com Mediana de 3 continuam a se destacar, com tempos baixos mesmo no pior caso, sendo 5509 ms e 6123 ms, respectivamente. E o Merge Sort e o Heap Sort mantêm os seus bons desempenhos, com tempos consistentes em todos os cenários.

RESULTADOS E DISCUSSÕES

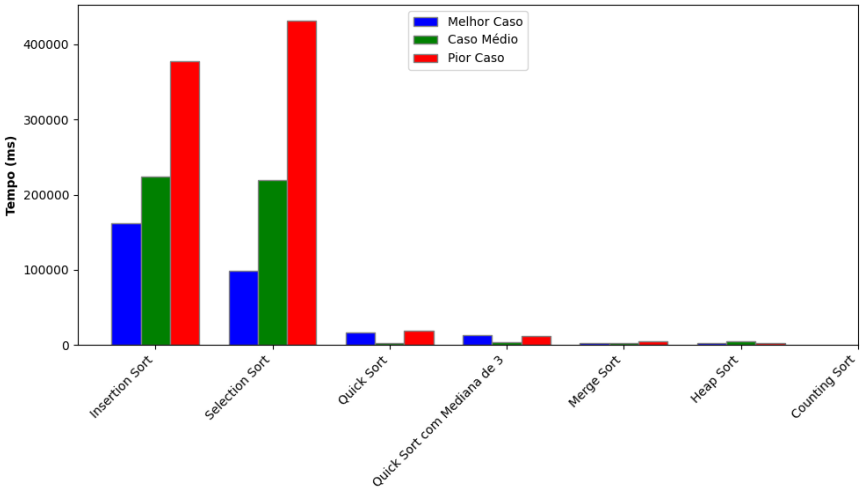
Em geral, os resultados demonstram que, embora algoritmos como Insertion Sort e Selection Sort sejam simples e intuitivos, eles não são recomendados para grandes conjuntos de dados. Algoritmos como Quicksort, Heap Sort e Merge Sort mostram-se mais adequados, apresentando desempenho significativamente melhor e tempos de execução baixos mesmo em cenários de pior caso. Nesse sentido, além da geração de tabelas, também foram plotados gráficos que visam comparar o tempo de execução dos algoritmos em cada uma das ordenações.

Figura 01: Gráfico em relação à ordenação dos tickers em ordem alfabética.



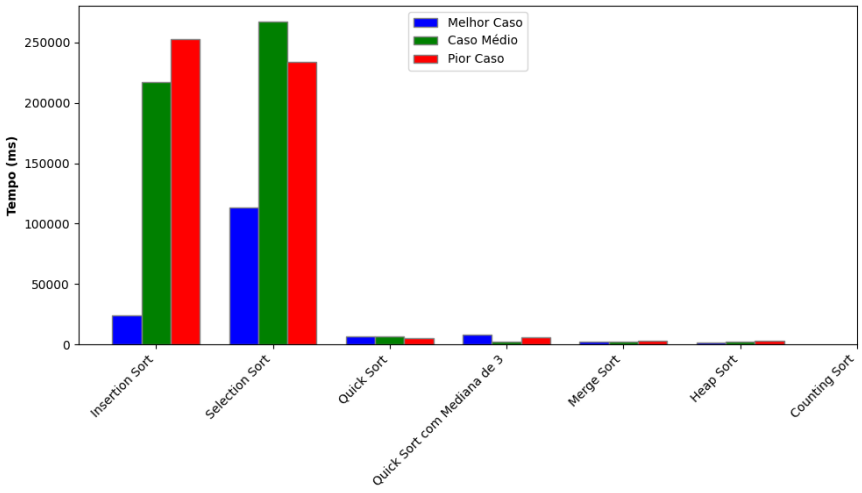
Fonte: Autoria Própria.

Figura 02: Gráfico em relação à ordenação dos volumes em ordem crescente.



Fonte: Autoria Própria.

Figura 03: Gráfico em relação à ordenação de variações diárias em ordem decrescente.



Fonte: Autoria Própria.

Ao examinar os gráficos fornecidos, fica claro que o Heapsort e o Merge Sort se destacaram como os algoritmos mais eficazes em geral, pois possuem complexidade O(n log(n)) em todos os três casos, enquanto os outros algoritmos têm complexidade $O(n^2)$. Embora o Quicksort também tenha complexidade O(n log(n)) no melhor e caso médio, sua eficiência foi inferior à do Heapsort e do Merge Sort devido ao grande tamanho do array de entrada (quase 2 milhões de registros), o que diminuiu a eficiência do Quicksort.

Em contrapartida, o Insertion Sort, que possui complexidade O(n) no melhor caso, foi, juntamente com o Selection Sort, os algoritmos menos eficazes em todos os casos de testes

Dessa forma, para obter melhor desempenho na ordenação de grandes conjuntos de dados, recomenda-se a utilização de algoritmos com complexidade O(n log(n)), que conseguem aliar rapidez e estabilidade, oferecendo uma solução mais robusta e eficiente para tarefas de ordenação complexas.

CONCLUSÃO

O presente relatório permitiu uma análise aprofundada da aplicação de algoritmos de ordenação sobre um grande volume de dados históricos da BOVESPA, evidenciando a importância da escolha adequada do algoritmo conforme as características dos dados.

Os experimentos realizados demonstraram que algoritmos com complexidade linear ou quase-linear, como o Merge Sort e o Heap Sort, apresentaram melhor desempenho na maioria dos casos, especialmente em grandes volumes de dados. Em contrapartida, algoritmos mais simples, como o Insertion Sort e o Selection Sort, apresentaram desempenho inferior. Além disso, a transformação e filtragem dos dados originais, resultando em novas representações e estruturas de arquivo, mostraram-se essenciais para garantir uma análise precisa e otimizada, possibilitando a avaliação eficiente dos algoritmos. A implementação em Java, utilizando as versões mais recentes das ferramentas de desenvolvimento, assegurou um ambiente robusto e eficaz para a execução das tarefas propostas.

Por fim, esse estudo destaca a relevância de um entendimento profundo das características dos dados a serem processado, contribuindo para a compreensão dos comportamentos dos algoritmos em contextos reais de mercado.

REFERÊNCIAS

BRAGA, Henrique. *Algoritmos de Ordenação: Ordenação por Inserção. Disponível em: <https://henriquebraga92.medium.com/algoritmos-de-ordena%C3%A7%C3%A3o-iii-insertion-sort-bfade66c6bf1>* Acesso em: 20 de setembro de 2024.

DEVMEDIA. *Algoritmos de ordenação: análise e comparação. Disponível em: <https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261>* Acesso em: 20 de setembro de 2024.

MUNDO BIT A BIT. *Método de Ordenação – Insertion Sort. Disponível em: < https://mundobitabitblog.wordpress.com/2017/07/03/101/>* Acesso em: 20 de setembro de 2024.

PENNA, Lucas. *Counting Sort. Disponível em: <http://desenvolvendosoftware.com.br/algoritmos/ordenacao/counting-sort.html>* Acesso em: 20 de setembro de 2024.

VIANA, Daniel. Conheça os principais algoritmos de ordenação. Disponível em: <https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao> Acesso em: 20 de setembro de 2024.