UEPB
Universidade Estadual da Paraíba
Departamento de Computação – DC
Laboratório de Estrutura de dados
Professor: Fábio Leite

KAYLLANE ARAÚJO DOS SANTOS

MARIA CLARA CUNHA PAULINO

TARCIO ELYAKIN AGRA DINIZ

ANÁLISE COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO

BOLSA DE VALORES BOVESPA 1994-2020

CAMPINA GRANDE – PB

27 de setembro de 2024

UEPB
Universidade Estadual da Paraíba
Departamento de Computação – DC
Laboratório de Estrutura de dados
Professor: Fábio Leite

KAYLLANE ARAÚJO DOS SANTOS

MARIA CLARA CUNHA PAULINO

TARCIO ELYAKIN AGRA DINIZ

ANÁLISE COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO

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: Fábio Leite

CAMPINA GRANDE – PB

27 de setembro de 2024

SUMÁRIO

1. INTRODUÇÃO

Entre o amplo universo da ciência da computação, a ordenação é uma atividade essencial na área, pois, desempenha um papel essencial na otimização de algoritmos e no aprimoramento do funcionamento de sistemas computacionais. 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, Heapsort 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 à 2020. Além disso, foi adicionado métodos para que seja fornecido o tempo de execução e uso de memória, e também, foi utilizado mecanismos para tratamento de possíveis erros ou falhas durante a execução.

A análise deste relatório aborda principalmente a implementação de todos os algoritmos mencionados anteriormente, explicando sobre as transformações, filtragens e ordenações de um conjuntos de dados.

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.

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

2. METODOLOGIA

Nesta seção, o leitor poderá visitar a máquina responsável por rodar o código e executar todos os testes. Além disso, será apresentada uma breve explicação de cada algoritmo utilizado durante o estudo. Por fim, será descrito como os testes foram conduzidos.

Abaixo, encontra-se uma tabela com informações relacionadas ao caderno do estudo:

Tabela 01: Características principais do notebook

Processador	13th Gen Intel(R) Core(TM) i5-13450HX
Memória RAM	16GB DDR5 4800mt/s
Armazenamento	NVIDIA GeForce RTX 3050 6GB Laptop
Sistema Operacional	Windows 11

Fonte: Autoria Própria

2.1 ALGORITMOS

O **Selection Sort** é um algoritmo de ordenação simples que funciona selecionando o menor (ou maior) elemento de uma lista e o colocando na posição correta, repetindo o processo para os n-1 elementos restantes até que a lista esteja ordenada. Ele possui uma complexidade de tempo $O(n^2)$ tanto no melhor quanto no pior caso, sendo um algoritmo in-place, ou seja, não requer memória extra significativa. No entanto, não é estável, o que significa que a ordem relativa dos elementos iguais pode ser alterada. Embora sua simplicidade o torne útil para ensino de algoritmos, ele é ineficiente para grandes volumes de dados.

O **Insertion Sort** também é um algoritmo de ordenação simples que funciona inserindo cada elemento na posição correta em relação aos elementos já ordenados. Ele é particularmente eficiente para conjuntos de dados pequenos ou quase ordenados, com uma complexidade de $O(n)$ no melhor caso e $O(n^2)$ no pior caso. O Insertion Sort é estável, mantendo a ordem relativa dos elementos iguais, e é um algoritmo in-place. Devido à sua simplicidade e bom desempenho em casos específicos, ele continua sendo uma escolha útil em determinadas situações, embora seja menos eficiente para grandes volumes de dados.

O **Merge Sort**, criado por John von Neumann em 1945, é um algoritmo que usa a estratégia de dividir para conquistar. Ele divide a lista em duas partes, ordena cada uma delas recursivamente e depois combina as duas partes ordenadas. Sua complexidade é O(n log n) em todos os casos, tornando-o mais eficiente que algoritmos com complexidade $O(n^2)$, como o Selection Sort e o Insertion Sort. O Merge Sort é estável e, embora exija espaço adicional para a mesclagem, é uma escolha eficiente para grandes conjuntos de dados.

O **Quick Sort**, desenvolvido por C.A.R. Hoare em 1960, também segue a estratégia de dividir para conquistar, mas de forma mais eficiente em termos de memória, já que geralmente é in-place. Ele escolhe um pivô e organiza a lista de modo que os elementos menores que o pivô fiquem à esquerda e os maiores à direita, repetindo o processo recursivamente. No pior caso, o Quick Sort tem complexidade $O(n^2)$, mas no melhor e no caso médio, sua complexidade é O(n log n). Apesar de ser rápido, ele não é estável e seu desempenho pode variar dependendo da escolha do pivô.

Por fim, o **Counting Sort**, criado por Harold Seward em 1954, é um algoritmo não comparativo que conta a frequência de cada elemento em um array e usa essas contagens para determinar as posições finais dos elementos no array ordenado. Sua complexidade é O(n+k), onde n é o número de elementos e k é o valor máximo dos elementos. É eficiente quando o intervalo dos valores é pequeno, mas pode se tornar ineficiente em termos de espaço quando o intervalo é muito grande. O Counting Sort é estável, mas geralmente é limitado a casos onde os dados têm um intervalo finito e conhecido.

O **Heap Sort** é um algoritmo de ordenação baseado na estrutura de dados Heap, geralmente utilizando um max-heap. Ele começa construindo um heap a partir do array de entrada e, em seguida, remove o maior elemento (a raiz do heap) e o coloca na última posição do array. Esse processo de "heapificação" é repetido para os elementos restantes, reorganizando o heap após cada remoção, até que todos os elementos estejam ordenados. O Heap Sort possui uma complexidade de O(n log n) em todos os casos, tornando-o eficiente e adequado para grandes conjuntos de dados. Além disso, é um algoritmo in-place, ou seja, não requer memória extra significativa para executar a ordenação. No entanto, ele não é estável, o que significa que a ordem relativa de elementos iguais pode ser alterada. Embora seja um algoritmo eficiente e consistente, o Heap Sort é menos utilizado que outros algoritmos como QuickSort e

MergeSort, devido à sua implementação mais complexa e à falta de estabilidade.

2.3 CASOS DE TESTES

Os testes foram feito utilizando o arquivo b3_stocks_1994_2020, que possui 1883204 linhas de registro.

Logo abaixo, estará listados os tópicos de classes e funções 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;
- CSVController: é um controlador para criar arquivos CSV a partir de dados de ações, onde os dados são primeiro ordenados pelo ticket antes de serem salvos. Ele utiliza um método de ordenação baseado no tipo de algoritmo fornecido e, ao final, cria um arquivo CSV formatado com um cabeçalho apropriado.
- StockDataTransformer: Tem como objetivo transformar datas em um formato específico (de yyyy-MM-dd para dd/MM/yyyy). Filtrar dados de ações para manter apenas os registros com o maior volume por data. e criar novos arquivos CSV com os dados transformados e filtrados.
- Main: Classe responsável por executar o código, bem como: exibir a memória utilizada, classificação, execução e algoritmo. Além do tempo em millisegundos , que foi calculado utilizando usando a funçaõ *currentTimeMillis.*

3. RESULTADOS E ANÁLISES

Nesta seção, apresentaremos as tabelas e gráficos comparativos dos sete algoritmos

utilizados como base para este estudo: Selection Sort, Insertion Sort, Counting Sort, Heap Sort, Quick Sort e Quick Sort com mediana de 3. As tabelas comparam o tempo de execução desses algoritmos em milissegundos (ms).

É importante destacar que os algoritmos foram testados em três cenários: melhor caso, pior caso e caso médio. No entanto, o Counting Sort não obteve dados, pois ele não é capaz de ordenar caracteres alfabéticos ou valores flutuantes, e por isso não aparece nas tabelas ou gráficos.

A seguir, apresentamos as tabelas e gráficos comparativos dos algoritmos de ordenação.

Tabela 02: **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	**1541 ms**	**1574 ms**	**1601 ms**
Quick Sort	**48668 ms**	**150974 ms**	**155217 ms**
Quick Sort com mediana de 3	**297401 ms**	**292433 ms**	**592642 ms**
Heap Sort	**1696 ms**	**1701 ms**	**1894 ms**
Couting Sort	**-**	**-**	**-**

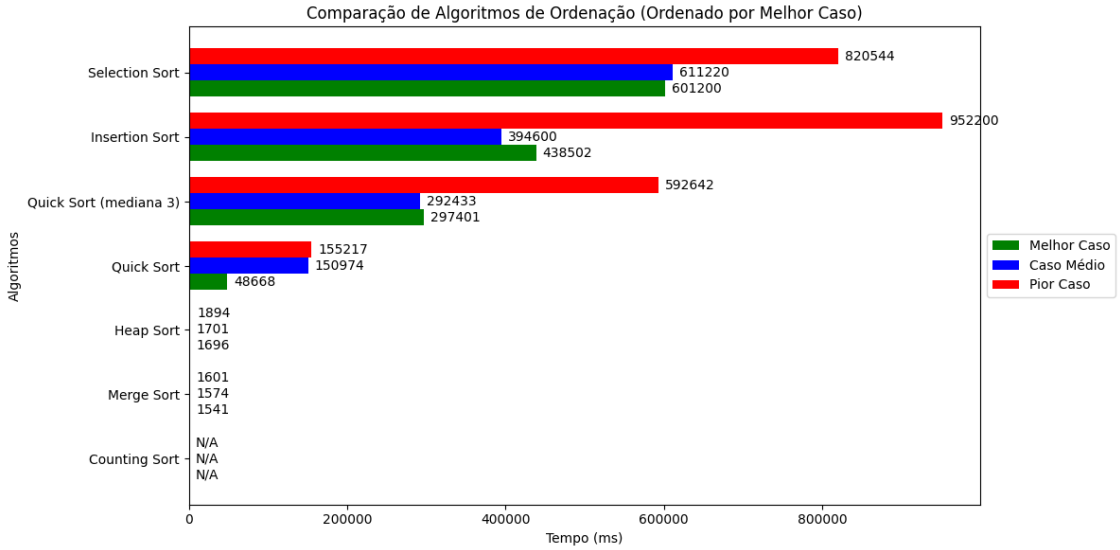
Tabela 03: **Ordenação dos Volumes em Ordem Crescente**

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	**3215 ms**	**3360 ms**	**4840 ms**
Quick Sort	**1800 ms**	**15700 ms**	**17355 ms**
Quick Sort com mediana de 3	**3215 ms**	**3360 ms**	**2500 ms**
Heap Sort	**5491**	**5535**	**5649**
Couting Sort	**-**	**-**	**-**

Tabela 04: **Ordenação das Variações Diárias em ordem Decrescente**

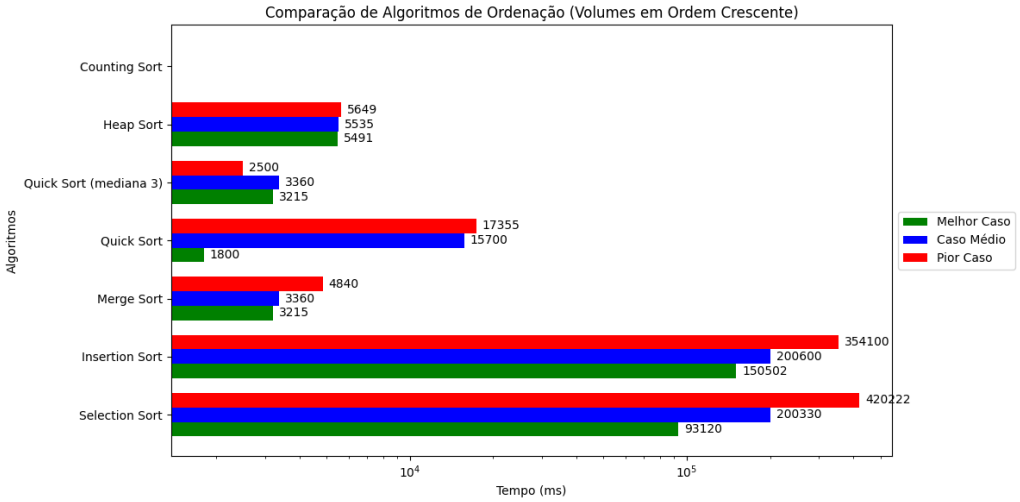
Algoritmos	Melhor Caso	Caso Médio	Pior Caso
Selection Sort	**103200 ms**	**218300 ms**	**77450 ms**
Insertion Sort	**23850 ms**	**192340 ms**	**100556 ms**
Merge Sort	**3811 ms**	**3824 ms**	**5809 m**
Quick Sort	**6400 ms**	**6900 ms**	**5400 ms**
Quick Sort com mediana de 3	**7800 ms**	**2000 ms**	**6000 ms**
Heap Sort	**8086 ms**	**8154 ms**	**8163 ms**
Couting Sort	**-**	**-**	**-**

Gráfico 01: **Ordenação dos tickets em Ordem Alfabética**



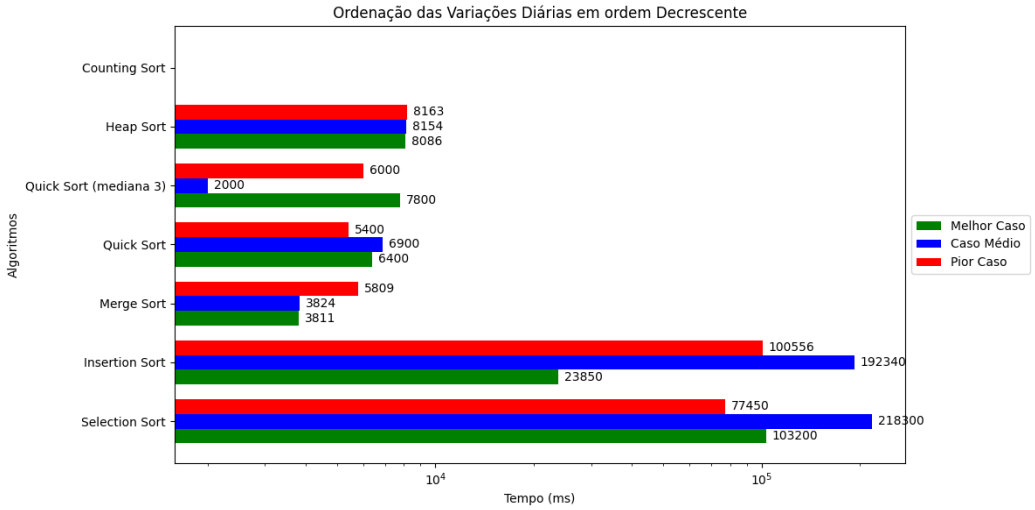
fonte: autoria Própria

Gráfico 02: **Ordenação dos Volumes em Ordem Crescente**



fonte: autoria Própria

Gráfico 03: **Ordenação das Variações Diárias em ordem Decrescente**



fonte: autoria Própria

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

Apesar de o **QuickSort** ter apresentado um excelente desempenho no **melhor caso** ao ordenar os volumes em ordem crescente (Tabela 03), ele sofre no **pior caso**, onde sua complexidade O(n^2) se manifesta, tornando-o menos confiável em algumas situações. A versão **QuickSort com mediana de 3** melhora a escolha do pivô, mas em alguns casos, como a **Tabela 02**, a sobrecarga adicional reduziu sua eficiência em comparação com o **QuickSort** padrão.

Insertion Sort mostrou boa performance no **melhor caso**, com complexidade O(n) quando a entrada já está quase ordenada, como visto na **Tabela 04** (23850 ms). No entanto, o **Selection Sort** e o **Insertion Sort** mostraram consistentemente tempos de execução altos em conjuntos de dados maiores devido à sua complexidade O(n^2) no **pior caso** e **caso médio**.

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.