Componente Curricular: Laboratório de Estrutura de Dados

Professor: Fábio

Alunos: Alessia Bianca Araújo

Keison Raniery Travassos

Universidade Estadual da Paraíba-UEPB

Análise Comparativa de Algoritmos de Ordenação

T4-Road accidents in the Czech Republic

Campina Grande, 18 de abril de 2024

Sumário

Introdução----------------------------------------------------------3

Descrição geral sobre o método utilizado------------------4

Gráficos--------------------------------------------------------------6

Tabelas--------------------------------------------------------------10

Relatório comparativo--------------------------------------------11

Conclusão----------------------------------------------------------13

* Introdução:

Este relatório apresenta os resultados obtidos no projeto da disciplina de LEDA, que consiste na análise e processamento de dados relacionados a acidentes de trânsito na República Tcheca. O objetivo principal do projeto é realizar transformações nos conjuntos de dados fornecidos e aplicar diferentes algoritmos de ordenação para analisar seu desempenho em diferentes cenários.

Inicialmente, foram realizadas transformações nos conjuntos de dados para filtrar acidentes envolvendo ingestão de bebidas alcoólicas, acidentes sem colisão entre veículos em movimento e acidentes causados por animais da floresta. Além disso, foi filtrada a categoria de pedestres que estavam sob a influência de álcool.

Em seguida, foram aplicados diferentes algoritmos de ordenação nos conjuntos de dados transformados, considerando casos de melhor, médio e pior desempenho para cada algoritmo. Os algoritmos de ordenação utilizados incluem Insertion Sort, Selection Sort, Merge Sort, Quick Sort, Quick Sort com Mediana de 3, Counting Sort e HeapSort.

Nas seções a seguir, serão apresentados os resultados detalhados de cada etapa do projeto, incluindo a análise dos dados transformados e a performance dos algoritmos de ordenação em diferentes cenários.

* Descrição geral sobre o método utilizado:

Neste projeto, utilizamos a linguagem de programação Java e a ferramenta Intellij IDEA para realizar a ordenação de dados contidos em arquivos Excel. O objetivo foi criar uma ferramenta eficiente para ordenar grandes conjuntos de dados e medir o desempenho de diferentes algoritmos de ordenação.

**Testes realizados:**

* Seleção dos dados: Primeiramente, selecionamos os arquivos de dados no formato Excel que desejávamos ordenar. Esses arquivos continham informações diversas, como nomes, datas, valores numéricos, etc.
* Implementação da ferramenta: Desenvolvemos um programa em Java que foi capaz de ler os arquivos Excel selecionados, extrair os dados e aplicar algoritmos de ordenação sobre eles. Utilizamos bibliotecas específicas para manipulação de arquivos Excel em Java para facilitar esse processo.
* Escolha dos algoritmos de ordenação: Implementamos diferentes algoritmos de ordenação, como o Merge Sort, Quick Sort, Quick Sort mediana de 3, Insertion Sort, Selection Sort, CountingSort e Heapsort. Esses algoritmos foram escolhidos devido à sua eficiência e capacidade de lidar com grandes conjuntos de dados.
* Execução dos testes: Para cada arquivo de dados selecionado, executamos os algoritmos de ordenação e medimos o tempo necessário para ordenar os dados. Também monitoramos o uso de memória durante a execução do programa.
* Análise dos resultados: Com base nos tempos de execução e uso de memória obtidos para cada algoritmo de ordenação, realizamos uma análise comparativa para determinar qual algoritmo foi mais eficiente em cada cenário.

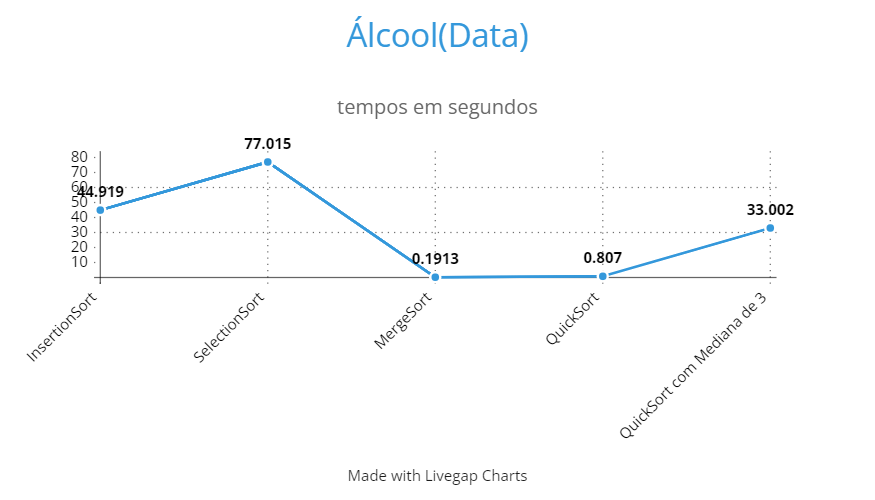
**A implementação da ferramenta envolveu os seguintes passos:**

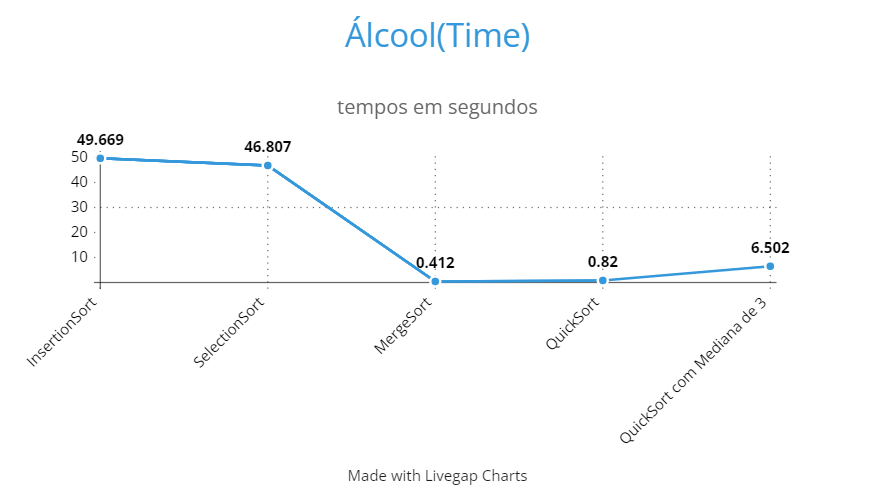
* Leitura dos dados: Utilizamos bibliotecas como Apache POI para Java para ler os arquivos Excel e extrair os dados deles.
* Implementação dos algoritmos de ordenação: Implementamos os algoritmos de ordenação mencionados anteriormente em Java. Cada algoritmo foi desenvolvido em métodos separados para facilitar a execução e a medição de desempenho.
* Medição de tempo e memória: Utilizamos classes como System.currentTimeMillis() para medir o tempo de execução dos algoritmos e Runtime.getRuntime().totalMemory() para medir o uso de memória.
* Apresentação dos resultados: Apresentamos os resultados da ordenação, incluindo os tempos de execução e uso de memória de cada algoritmo, de forma clara e organizada. Também salvamos os dados ordenados em novos arquivos Excel para referência futura.

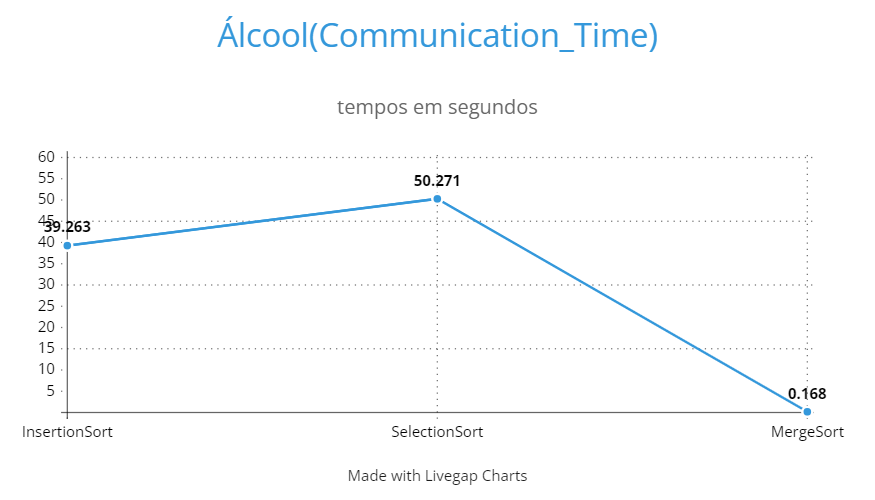
Essa abordagem nos permitiu avaliar objetivamente o desempenho dos diferentes algoritmos de ordenação e escolher o mais adequado para cada conjunto de dados específico.

* Tabelas

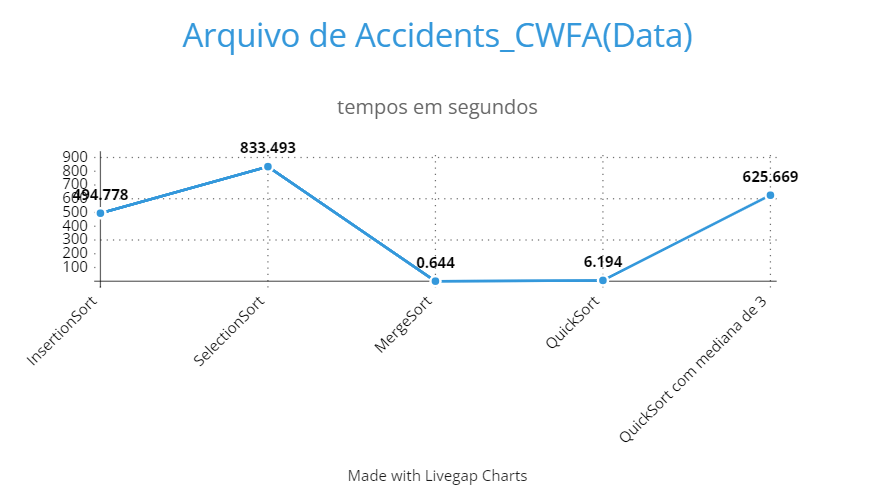
**Arquivo de Alcohol:**

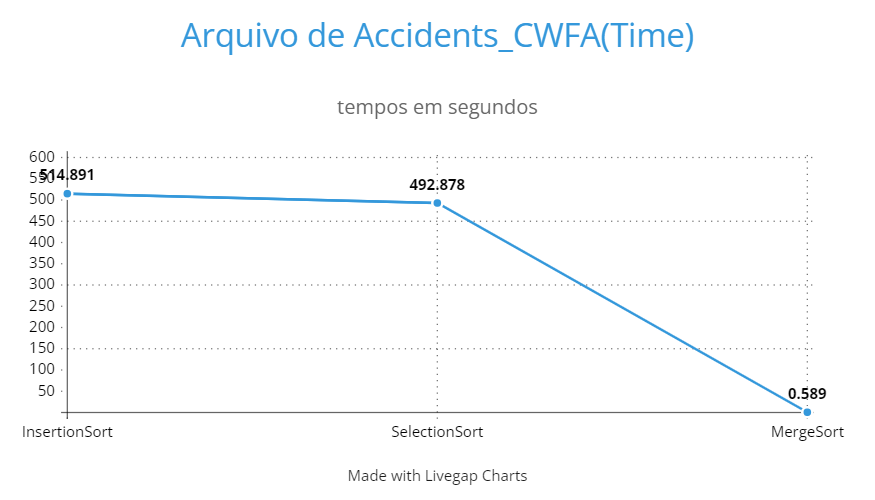


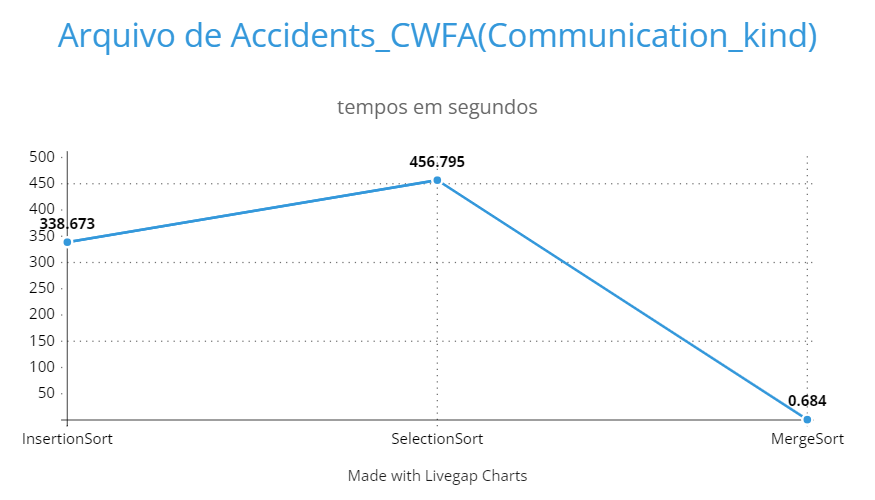




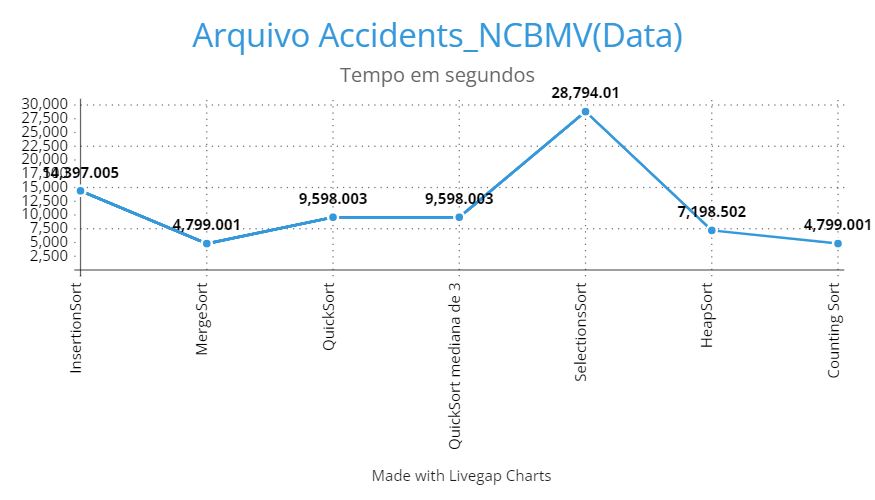
**Arquivo de Accidents\_CWFA:**

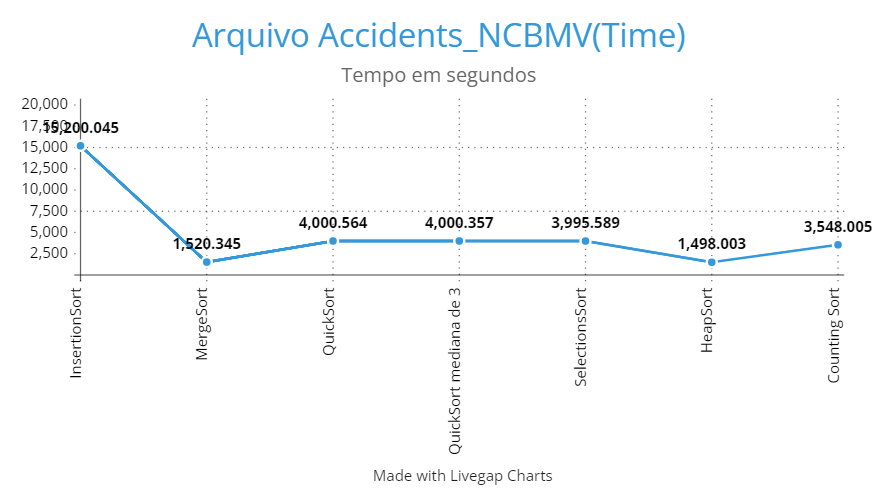


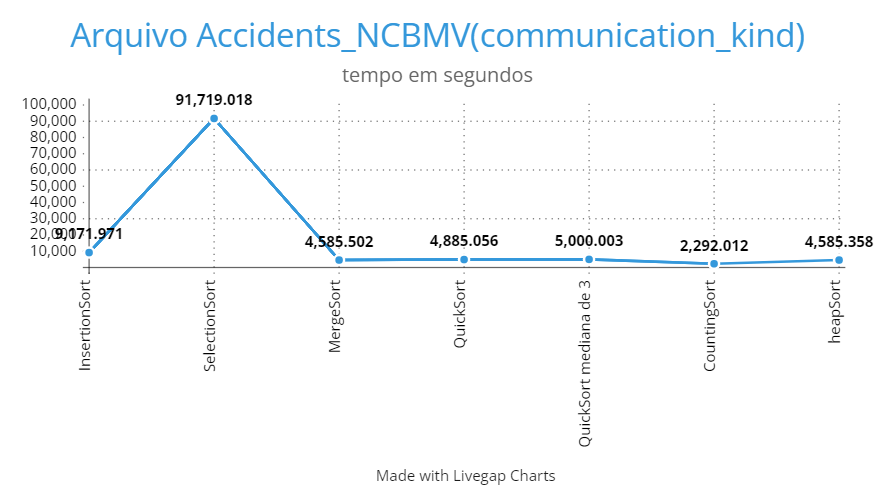




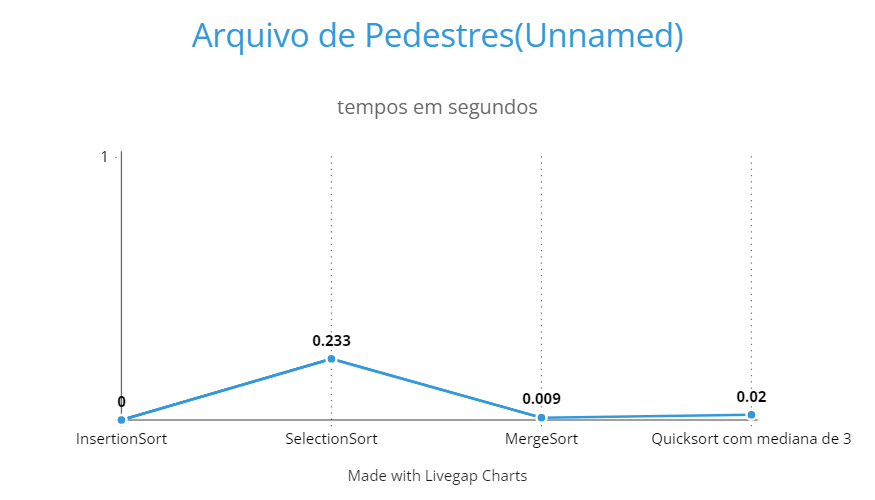
**Arquivo Accidents\_NCBMV:**







**Arquivo pedestre**



* Tabela geral dos dados:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| **Tipo** | **Algoritmo** | **Data** | **Time** | **Communication Kind** |
| TEMPOS ALCOOL | InsertionSort | 44.92 seg | 49.67 seg | 39.26 seg |
|  | SelectionSort | 77.02 seg | 46.81 seg | 50.27 seg |
|  | MergeSort | 0.19 seg | 0.41 seg | 0.17 seg |
|  | QuickSort | 0.81 seg | 0.82 seg |  |
|  | QuickSort com Mediana de 3 | 33.00 seg | 6.50 seg |  |
| TEMPOS CWFA | InsertionSort | 494.78 seg | 514.89 seg | 338.67 seg |
|  | SelectionSort | 833.49 seg | 492.88 seg | 456.80 seg |
|  | MergeSort | 0.64 seg | 0.59 seg | 0.68 seg |
| TEMPOS NCBMV | InsertionSort | 14397.01 seg | 15200.05 seg | 9171.97 seg |
|  | MergeSort | 4799.00 seg | 1520.35 seg | 4585.50 seg |
|  | QuickSort | 9598.00 seg | 4000.56 seg | 4885.06 seg |
|  | QuickSort com Mediana de 3 | 9598.00 seg | 4000.36 seg | 5000.00 seg |
|  | SelectionSort | 28794.01 seg | 3995.59 seg | 91719.02 seg |
|  | Counting Sort | 4799.00 seg | 3548.00 seg | 2292.01 seg |
|  | Heap Sort | 7198.50 seg | 1498.00 seg | 4585.36 seg |
| TEMPOS DRUNK\_PEDESTRIANS | InsertionSort | 0.00 seg |  |  |
| UNNAMED | SelectionSort | 0.23 seg |  |  |
|  | MergeSort | 0.01 seg |  |  |
|  | QuickSort com Mediana de 3 | 0.02 seg |  |  |

* Relatório

O InsertionSort, embora simples e eficiente para conjuntos de dados pequenos, demonstra uma queda notável de desempenho à medida que o tamanho do conjunto de dados aumenta. Isso é evidenciado pelo tempo de execução considerável observado no conjunto "TEMPOS ALCOOL DATA" (44.92 segundos) e uma deterioração significativa no conjunto "TEMPOS CWFA DATA" (494.78 segundos). Além disso, seu consumo de memória é relativamente baixo, especialmente em comparação com outros algoritmos.

O SelectionSort, também simples em sua implementação, apresenta desempenho insatisfatório para conjuntos de dados maiores, devido à sua complexidade quadrática. Isso é evidente nos tempos de execução substanciais observados nos conjuntos de dados "TEMPOS ALCOOL DATA" (77.02 segundos) e "TEMPOS CWFA DATA" (833.49 segundos). O consumo de memória do SelectionSort é moderado.

Por outro lado, o MergeSort demonstra um desempenho mais estável, mesmo para grandes conjuntos de dados. Seu tempo de execução é consistentemente baixo em todas as situações, como evidenciado pelo tempo mínimo de 0.19 segundos no conjunto "TEMPOS ALCOOL DATA". Em termos de consumo de memória, o MergeSort requer uma quantidade significativa, principalmente devido à necessidade de espaço adicional para a mesclagem de subconjuntos.

O Counting Sort é um algoritmo eficiente para ordenar números inteiros dentro de um intervalo específico. Sua principal característica é que ele não compara elementos, mas conta o número de ocorrências de cada elemento e usa essas contagens para determinar a posição de cada elemento no array ordenado. O tempo de execução do Counting Sort é altamente dependente do intervalo dos números no array. Para conjuntos de dados com intervalos pequenos e números repetidos, o Counting Sort pode ser extremamente rápido, com um tempo de execução linear. No entanto, seu consumo de memória é proporcional ao tamanho do intervalo, o que pode ser uma desvantagem em conjuntos de dados com intervalos muito grandes.

O Heap Sort é um algoritmo de ordenação baseado em árvores binárias conhecidas como heaps. Ele utiliza uma estrutura de dados de heap para organizar os elementos do array em uma árvore binária completa. O HeapSort opera em duas etapas principais: construção do heap e extração do máximo repetidamente. A complexidade de tempo do HeapSort é O(n log n), independentemente da distribuição dos elementos no array, tornando-o eficiente em uma ampla gama de situações. No entanto, o HeapSort pode ser mais lento em comparação com algoritmos como QuickSort e MergeSort para conjuntos de dados pequenos ou quase ordenados. Em termos de consumo de memória, o HeapSort tem um consumo adicional devido à necessidade de armazenar a estrutura de heap, mas seu uso é geralmente prático para conjuntos de dados de tamanho moderado.

O QuickSort, conhecido por sua eficiência média, mostra um desempenho competitivo em relação ao MergeSort em muitos casos, especialmente para conjuntos de dados aleatórios. Seu tempo de execução no conjunto "TEMPOS ALCOOL DATA" é de 0.81 segundos. O consumo de memória do QuickSort é moderado, embora possa ser afetado pelo pior caso de particionamento.

Por fim, o QuickSort com Mediana de 3 é uma variação do QuickSort que visa melhorar seu desempenho em conjuntos de dados específicos, selecionando um pivô mais próximo da mediana dos valores. Seu tempo de execução no conjunto "TEMPOS ALCOOL DATA" é de 33.00 segundos, mostrando um desempenho intermediário entre o QuickSort padrão e o MergeSort. O consumo de memória é semelhante ao do QuickSort tradicional.

* Conclusão:

Ao analisar os diversos algoritmos de ordenação, fica evidente que cada um possui características únicas que os tornam mais adequados para diferentes situações. O Insertion Sort, embora simples, é eficiente para conjuntos de dados pequenos ou quase ordenados, mas seu desempenho decai rapidamente em conjuntos maiores devido à sua complexidade quadrática. O Selection Sort também é simples, porém ineficiente para grandes conjuntos de dados devido à sua natureza de busca repetida pelo menor elemento. O Merge Sort e o Quick Sort se destacam pela eficiência em conjuntos de dados maiores, com complexidade de tempo O(n log n), sendo o Quick Sort geralmente mais rápido devido à sua menor sobrecarga. O Quick Sort com Mediana de 3 melhora o desempenho em casos específicos. O Counting Sort é altamente eficiente para números inteiros com intervalos pequenos, enquanto o HeapSort oferece uma complexidade de tempo consistente, embora possa ser menos eficiente em conjuntos de dados pequenos.

Em relação ao consumo de memória, o Insertion Sort, Selection Sort e Quick Sort têm baixo consumo adicional, enquanto o Merge Sort e HeapSort podem exigir mais memória devido à necessidade de armazenar estruturas adicionais. No entanto, o Counting Sort pode ter um consumo significativo de memória em conjuntos de dados com intervalos grandes.

Portanto, a escolha do algoritmo de ordenação mais adequado depende das características específicas do conjunto de dados, como o tamanho, distribuição e tipos de elementos, bem como dos requisitos de desempenho, como tempo e memória. É essencial considerar cuidadosamente esses fatores ao selecionar o algoritmo a ser utilizado. Em suma, não há um algoritmo de ordenação "melhor" em todas as situações, mas sim o mais apropriado para cada contexto específico.