Relatório - Análise Comparativa da Complexidade Algorítmica de Métodos de Ordenação

Alunos: Ana Luiza, Gabriel Henrique, Lucas Nass, Jhonatan de Jesus

Resumo

Este trabalho tem como objetivo principal analisar e comparar o desempenho de seis algoritmos de ordenação: **Bubble Sort, Insertion Sort, Heap Sort, Merge Sort, Quick Sort e Radix Sort**. A análise foca na complexidade algorítmica, medida pelo esforço computacional em termos de operações de comparação e troca de valores, sob a condição de "caso médio". Para isso, conjuntos de números inteiros de tamanhos variando de 1 a 1000 elementos foram gerados aleatoriamente, e os resultados de 30 execuções distintas foram promediados para garantir validade estatística. Os resultados serão apresentados em gráficos de linha, ilustrando as diferenças de complexidade entre os métodos.

Metodologia

A metodologia empregada para a análise comparativa seguiu os seguintes passos:

- Algoritmos Analisados

Foram implementados e testados os seguintes algoritmos de ordenação:

- **Bubble Sort**
- **Insertion Sort**
- **Heap Sort**
- **Merge Sort**
- **Quick Sort**
- **Radix Sort**

- Geração de Conjuntos de Dados

Conjuntos de números inteiros foram gerados aleatoriamente, com tamanhos variando de N=1 a N=1000. A função `rand()`, inicializada por `srand(time(NULL))`, foi utilizada para criar esses vetores, simulando o "caso médio" de distribuição de dados. Os números gerados estavam na faixa de 0 a 999.

- Métrica de Esforço Computacional

O esforço computacional de cada algoritmo foi quantificado pelo número total de operações realizadas. Uma operação foi definida como:

- Cada comparação entre elementos do vetor.
- Cada troca de valores e atribuições no vetor ordenado.

Essa contagem foi realizada por meio de uma variável **`cont`** incrementada dentro de cada função de ordenação e na função de troca **`swap`**, retornando o total de operações ao final da execução. Os algoritmos foram implementados com a contagem de operações integrada diretamente ao código-fonte de cada algoritmo.

- Validação Estatística

Cada algoritmo foi executado 30 vezes para cada tamanho de vetor N. Para cada tamanho N, foi feita a média dos 30 valores obtidos durante os testes e foi registrada para que a análise dos resultados fosse mais precisa e representasse melhor o esforço computacional do método de ordenação.

Resultados

Os resultados da análise de complexidade foram registrados em uma tabela e foi feito, no Power BI, um gráfico de linhas, onde o eixo x representa o tamanho do vetor, o eixo y representa a média de quantidade de operações realizadas e as linhas do gráfico representam os métodos de ordenação e o desempenho de cada um nos testes. As curvas foram colocadas em escala logarítmica para uma melhor diferenciação dos resultados.

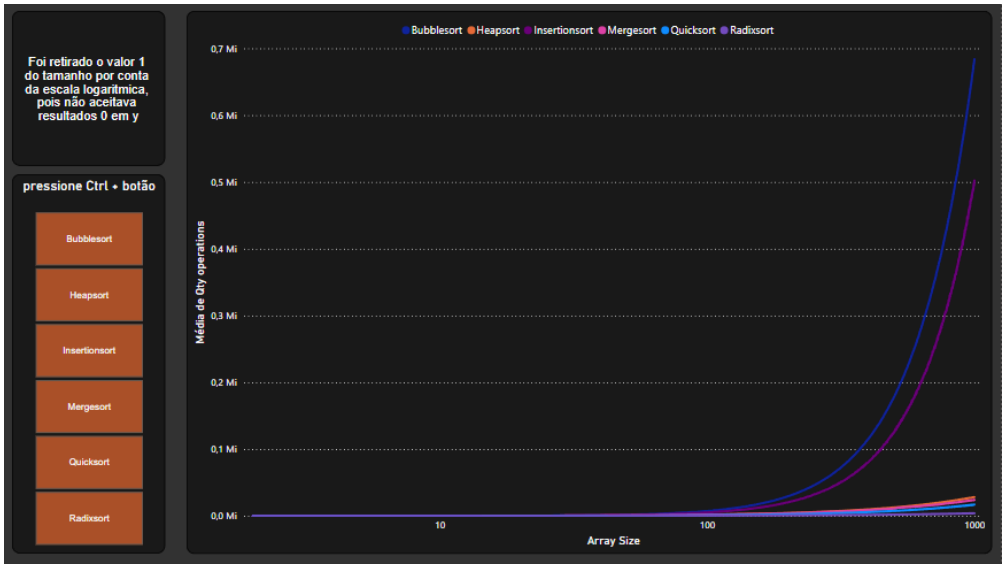

Figura 1: Gráfico com os resultados obtidos depois dos testes

Esforço Computacional Médio de Bubble Sort e Insertion Sort

Ambos Bubble Sort e Insertion Sort demonstraram um crescimento no número de operações que se alinha com a sua complexidade $O(n^2)$. As curvas são parabólicas, indicando que o esforço computacional aumenta drasticamente à medida que o tamanho do conjunto de dados (n) cresce. Para n=1000, o número de operações alcançou centenas de milhares, tornando-os impraticáveis para conjuntos de dados maiores.

Durante os testes, o **Insertion Sort** apresentou um desempenho levemente superior ao Bubble Sort. Isso ocorreu principalmente pelo comportamento do Insertion Sort com vetores parcialmente ordenados e, junto com isso, os vetores são gerados aleatoriamente, podendo facilitar a ordenação para o método. No entanto, a ordem de grandeza do crescimento é a mesma.

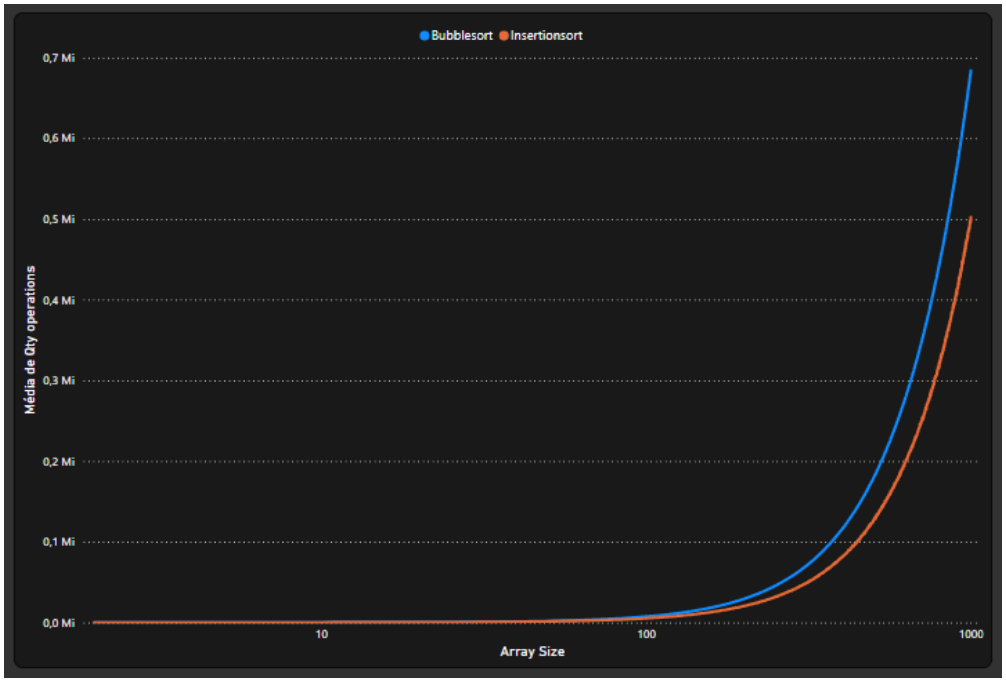

Figura 2 : Gráfico de comparação entre Bubble Sort e Insertion Sort

Esforço Computacional Médio de Heap Sort, Merge Sort, e Quick Sort

Merge Sort, Heap Sort e Quick Sort possuem uma complexidade O(n log n), sendo mais eficiente do que os métodos de complexidade O(n²), as curvas no gráfico deixam bem clara essa discrepância entre os métodos.

O **Heap Sort** apresentou uma quantidade de operações ligeiramente maiores que o Merge Sort e Quick Sort devido à sua estrutura de heap (no caso médio), apesar de ter a mesma complexidade assintótica teórica de O(n log n). A implementação da função **`heapify`** de maneira recursiva e o uso repetitivo da função **`swap`** acabam afetando no número de operações em grande escala.

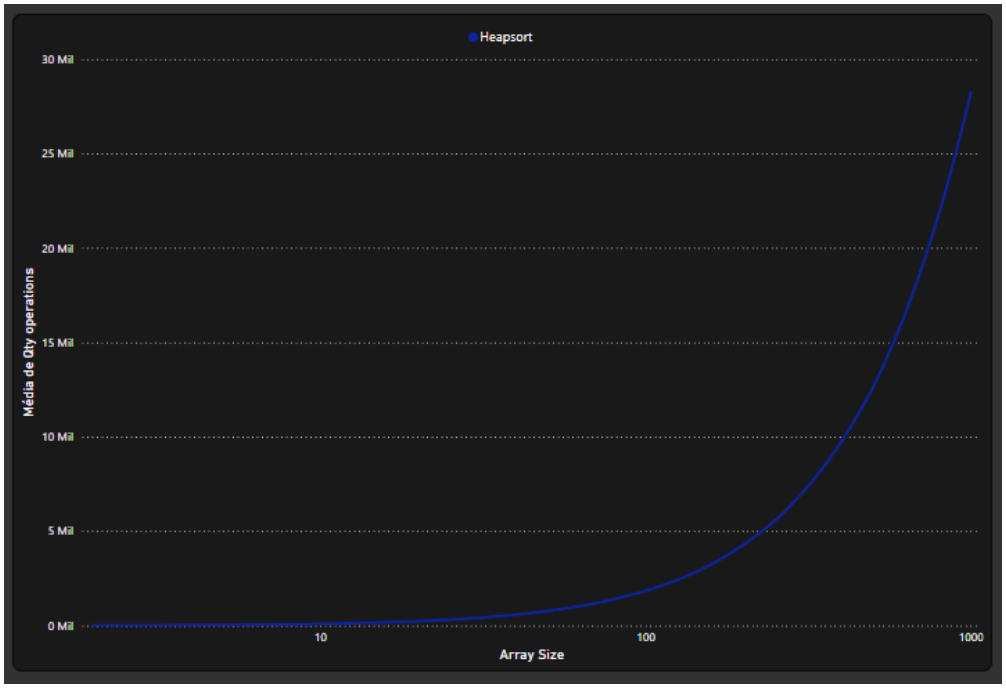

Figura 3: Gráfico mostrando a curva de eficiência do Heap Sort

O **Merge Sort** se saiu melhor do que o Heap Sort, mas pior do que o Quick Sort. Esse método, diferentemente do anterior, não faz uso da função **`swap`**, reduzindo grandemente o número de operações em relação ao Heap Sort. Porém, na hora de mesclar os vetores na função **`merge`**, são feitas diversas operações de comparação e de atribuição no vetor principal, contribuindo no aumento de operações realizadas.

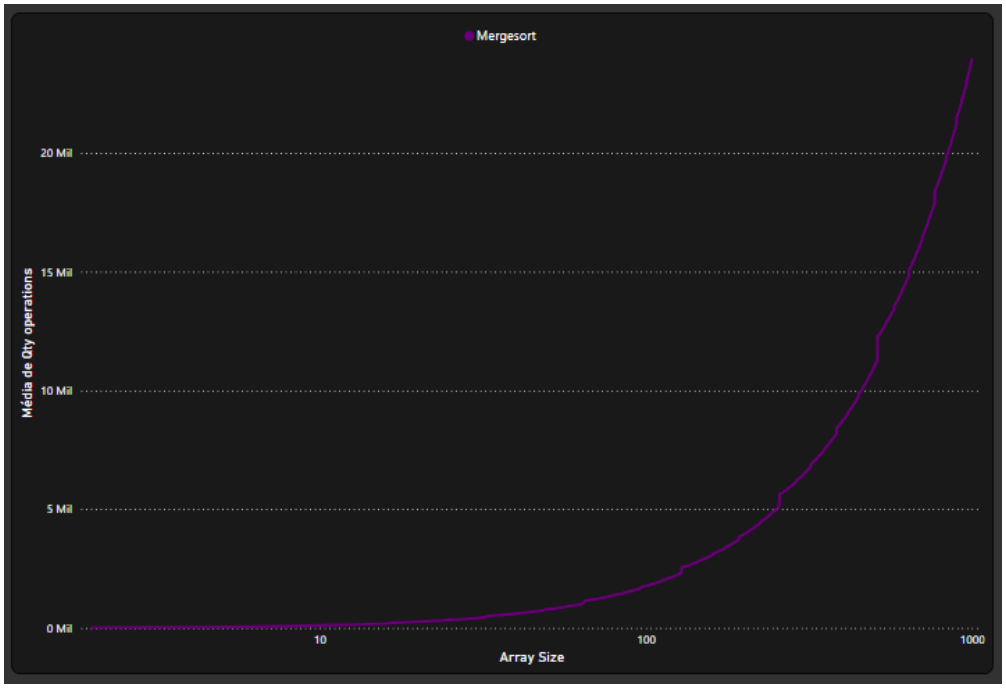

Figura 4: Gráfico mostrando a curva de eficiência do Merge Sort

O **Quick Sort** foi o método que menos realizou operações dentre os três, sendo o mais eficiente nos casos médios. Com sua função de particionar elementos a partir de um pivô, esse método realiza menos trocas durante a ordenação e também menos comparações, mesmo que seja recursiva como as outras apresentadas anteriormente. A maior diferença desse método para os outros no quesito de complexidade, é que em seu pior caso a complexidade é de $O(n^2)$, porém é muito difícil de acontecer (vetores quase ordenados) e não afetou os resultados obtidos no estudo.

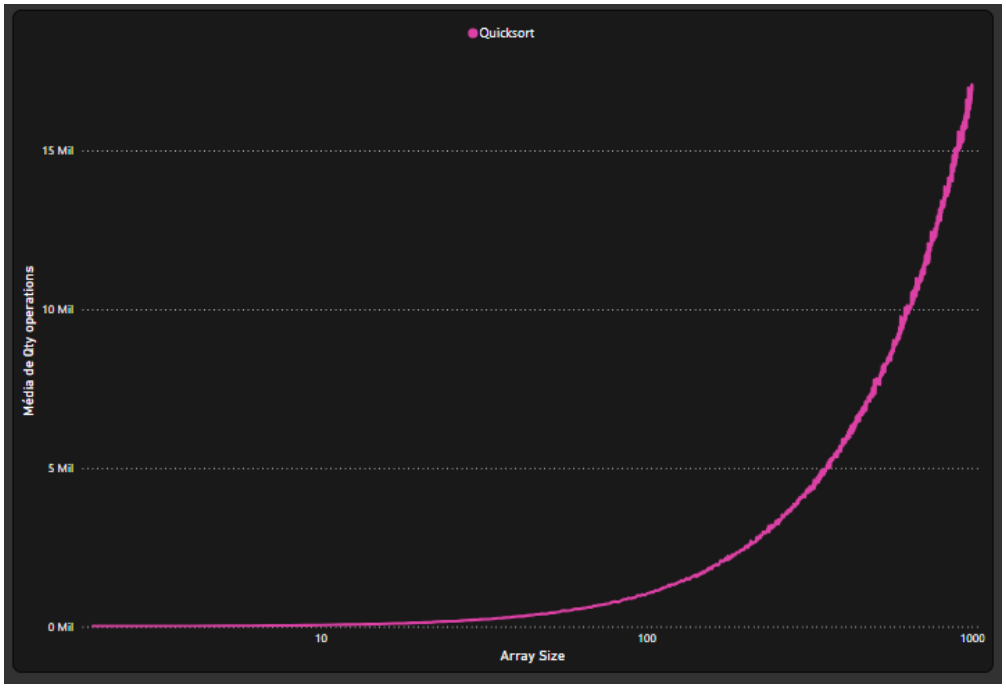

Figura 5: Gráfico mostrando a curva de eficiência do Quick Sort

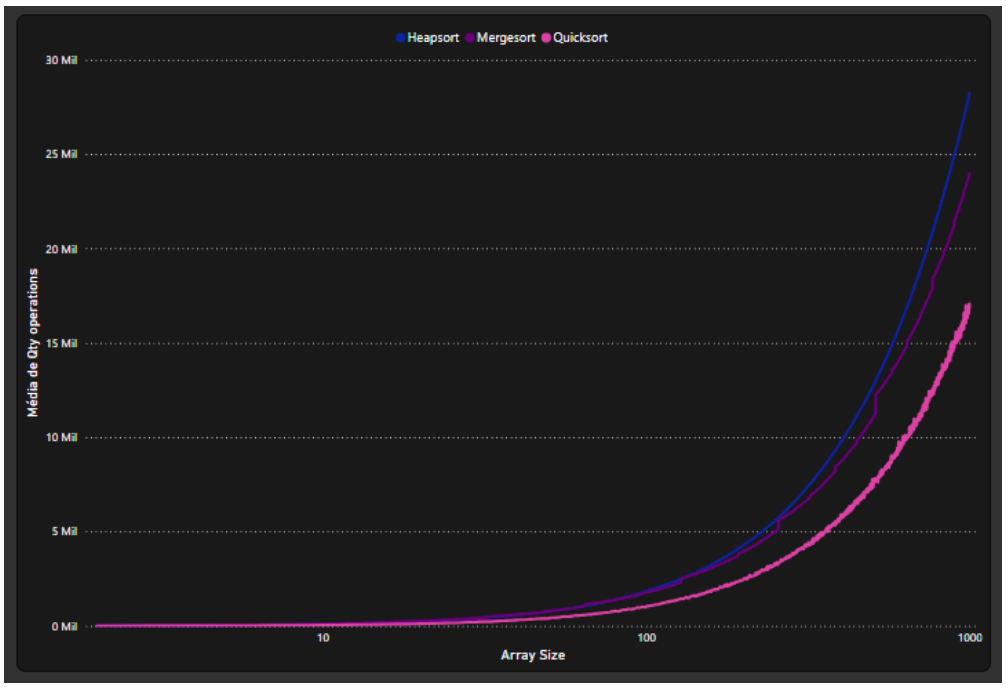

Figura 6: Gráfico de comparação entre Heap Sort, Merge Sort e Quick Sort

Esforço Computacional Médio de Radix Sort

O **Radix Sort** é um algoritmo de ordenação não comparativo. Ao contrário dos algoritmos como Quick Sort, Merge Sort e Heap Sort (que ordenam comparando elementos entre si), o Radix Sort ordena os elementos inspecionando e agrupando dígitos ou caracteres individuais (radix), começando do menos significativo para o mais significativo (LSD - Least Significant Digit) ou vice-versa (MSD - Most Significant Digit) e essa é a principal razão para sua superioridade na sua complexidade que é $O(n \cdot d)$, uma complexidade quase linear podendo ser melhor que O(n log n).

O Radix Sort não manteria sua superioridade se os elementos tivessem um número muito alto de dígitos (ou um valor máximo muito grande que se traduzisse em muitos dígitos/passes). Para que $O(n \cdot d)$ seja melhor que O(n log n), é necessário que d seja significativamente menor que logN. Se d for grande (muitos dígitos), a parte d da equação começa a dominar e o custo total do Radix Sort (que faz d passes sobre os n elementos) pode facilmente ultrapassar o custo de um algoritmo n log n como o Quick Sort ou Merge Sort. Um comportamento observado durante os testes foi que o Radix Sort, em vetores menores, teve um desempenho levemente pior que os métodos O(n log n), isso se dá por conta de que o Radix Sort inicialmente faz operações varrendo o vetor inteiro, e, por exemplo, o Quick Sort faz o particionamento, fazendo menos operações, sendo mais eficiente nesses casos de vetores menores. Neste estudo de caso, os testes foram realizados com números entre 0 a 999, ou seja, o número máximo de dígitos seria três, um número consideravelmente pequeno, portanto o Radix Sort foi o método mais eficiente dentre os outros.

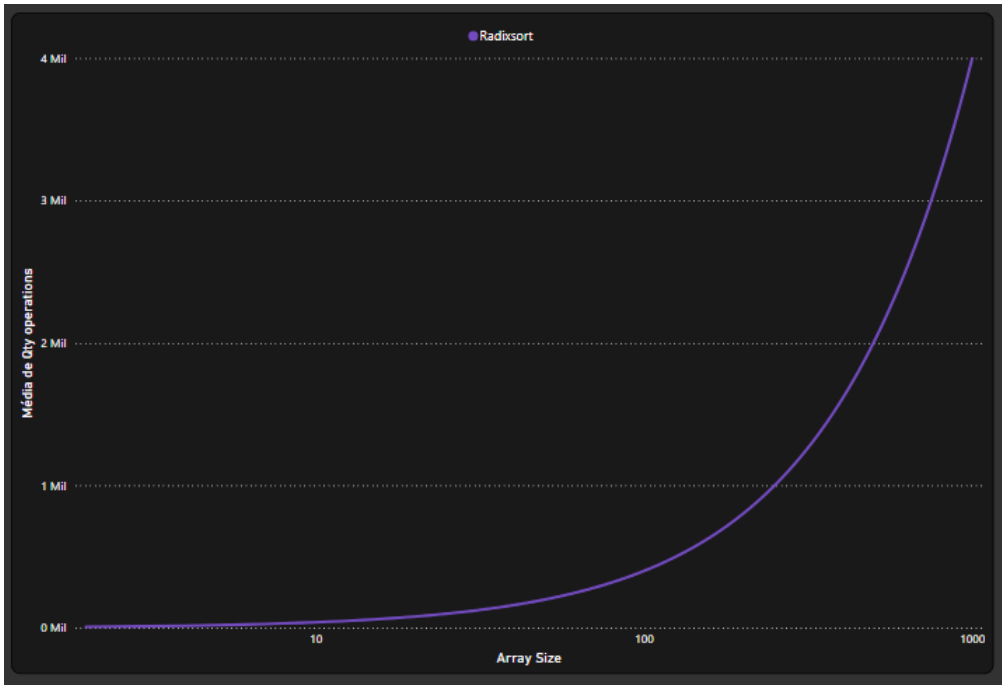

Figura 7: Gráfico mostrando a curva de eficiência do Radix Sort

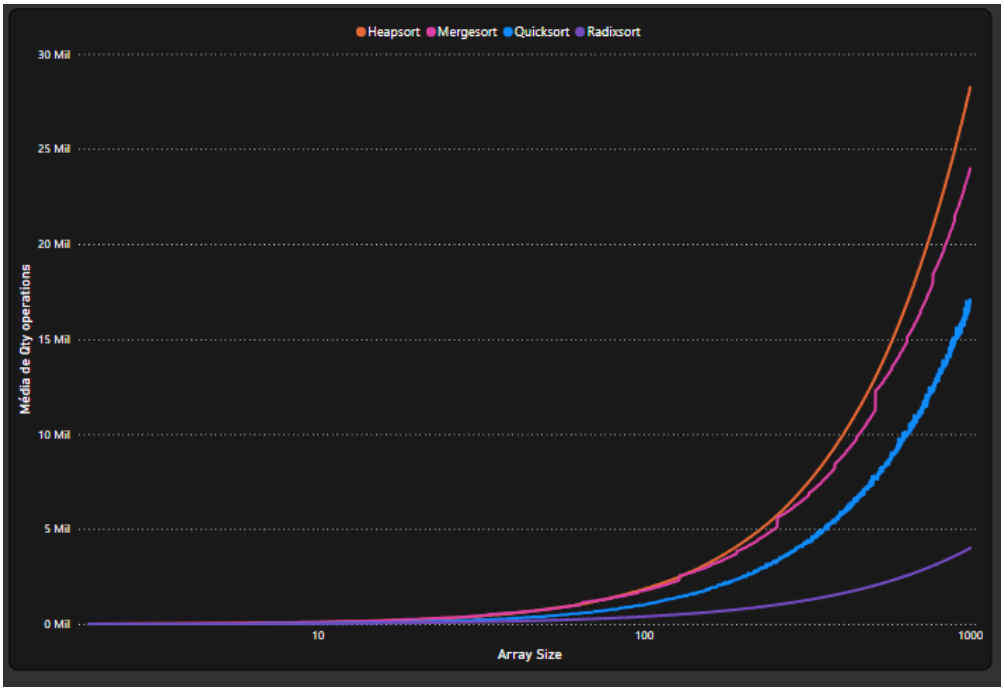

Figura 8: Gráfico de comparação entre o Radix Sort e os métodos O(n log n)

Conclusão

A análise do desempenho dos algoritmos de ordenação para vetores aleatórios de 1 a 1000 elementos revela que a complexidade assintótica dos métodos de ordenação é muito importante para analisar o desempenho de eficiência. Os métodos Bubble Sort e Insertion Sort, que possuem uma complexidade quadrática ($O(n^2)$), apresentaram uma quantidade de operações inviável para tamanhos maiores de vetores, no Bubble Sort, esse número chegou a quase 700 mil operações e no Insertion Sort ultrapassou as 500 mil operações. Os métodos de complexidade logarítmica ($O(n \log n)$) Heap Sort, Merge Sort e Quick Sort apresentaram resultados muito mais eficientes e viáveis, o Heap Sort apresentou um esforço máximo de quase 30 mil operações, o Merge Sort foi levemente melhor apresentando quase 25 mil operações e o Quick Sort, o qual foi o melhor, apresentou um pouco mais de 15 mil operações no esforço máximo. Por fim, o Radix Sort que possui uma complexidade parcialmente linear ($O(n \cdot d)$) foi o método de ordenação mais eficiente entre todos os outros, gerando um total de mais ou menos 5 mil operações no maior tamanho. Vale ressaltar que esse estudo de caso levou em consideração apenas o número de operações (comparações, trocas e atribuições), existem diversos outros fatores para analisar a eficiência do algoritmo, como acesso à

memória, localidade de referência, uso auxiliar de memória em determinados casos, etc.

Link do repositório no Github

https://github.com/AnaLuizaElert/sorting-methods.git