Relatório de Análise de Algoritmos de Ordenação

Isabelle Vizzu, Maria Fernanda e Julia Helena.

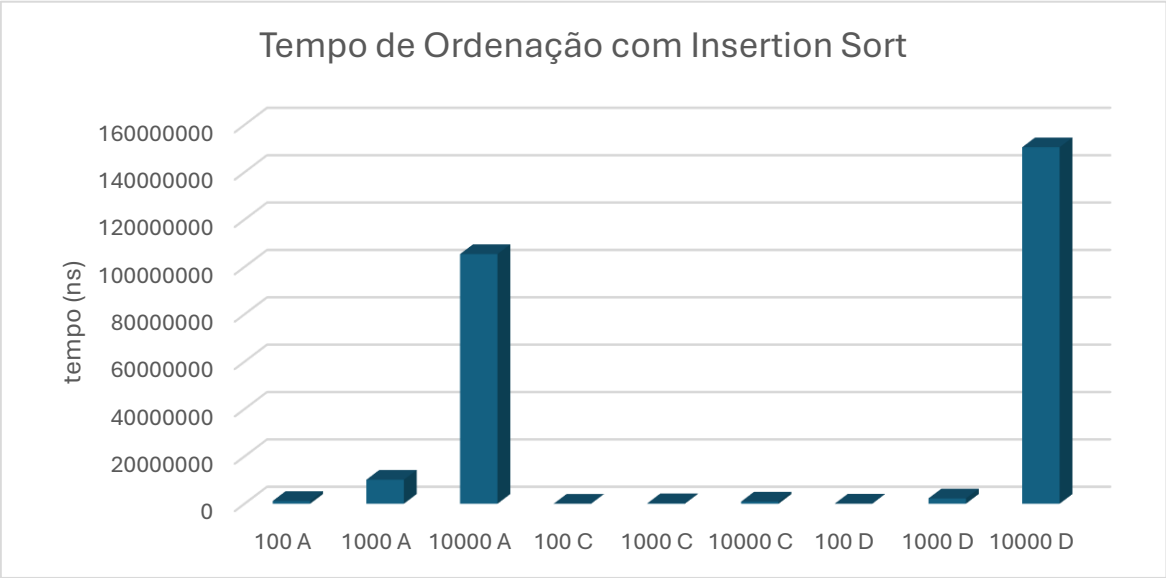
1. Insertion Sort:

1.1 Tabela de resultados do tempo de execução para cada vetor:

Insertion Sort								
Aleatório			Crescente			Decrescente		
100 A	1000 A	10000 A	100 C	1000 C	10000 C	100 D	1000 D	10000 D
1158300 ns	10142600 ns	105414300 ns	23200 ns	215900 ns	911300 ns	35600 ns	2235300 ns	150537000 ns

Legenda: ns = nanossegundo

1.2 Gráfico:



Legenda:

- **100 A**: Vetor com 100 dados em ordem aleatória
- **1000 A**: Vetor com 1.000 dados em ordem aleatória
- **10000 A**: Vetor com 10.000 dados em ordem aleatória
- **100 C**: Vetor com 100 dados em ordem crescente
- **1000 C**: Vetor com 1.000 dados em ordem crescente
- **10000 C**: Vetor com 10.000 dados em ordem crescente
- **100 D**: Vetor com 100 dados em ordem decrescente
- **1000 D**: Vetor com 1.000 dados em ordem decrescente
- **10000 D**: Vetor com 10.000 dados em ordem decrescente

1.3 Análise:

Os resultados apresentados nos levam a algumas conclusões sobre a eficiência do Insertion Sort. Primeiramente, esse algoritmo demonstra alta eficiência ao ser aplicado a um vetor já ordenado de forma crescente. Isso ocorre porque ele verifica, para cada elemento, se o atual é menor que o anterior. Como essa condição sempre será verdadeira em um vetor ordenado, o algoritmo reconhece que o elemento já

está na posição correta e evita a troca. Em nosso código, quando o atual é maior que o anterior, um break é acionado, permitindo que o algoritmo vá imediatamente para o próximo elemento, aumentando ainda mais a eficiência, como podemos ver no resultado, o qual foi o menor (911300 nanossegundos para 10000 dados).

Em segundo lugar, o pior cenário para o Insertion Sort acontece quando o vetor está ordenado em ordem decrescente. Nesse caso, sempre que o elemento atual for menor que seu antecessor, ocorrerá uma troca. Como o antecessor é sempre maior (já que o vetor está em ordem decrescente), todas as trocas que poderiam ocorrer, ocorrerão, resultando na menor eficiência do algoritmo, justificando assim, o tempo apresentado, o qual foi o maior (150537000 nanossegundos para 10000 dados).

Por último, quando o Insertion Sort é aplicado a um vetor de ordem aleatória, ele apresenta uma “eficiência média”. Essa eficiência varia conforme o número de trocas que serão realizadas, podendo ser muitas ou poucas, dependendo do nível de desordem do vetor. Por isso, o resultado foi 'alto' em termos de tempo, mas ainda não foi o maior (105414300 nanossegundos para 10000 dados) se comparado com o vetor em ordem decrescente.

Em suma, podemos dizer que o Insertion Sort possui alta eficiência em vetores crescentes, eficiência média em vetores aleatórios e baixa eficiência em vetores decrescentes.

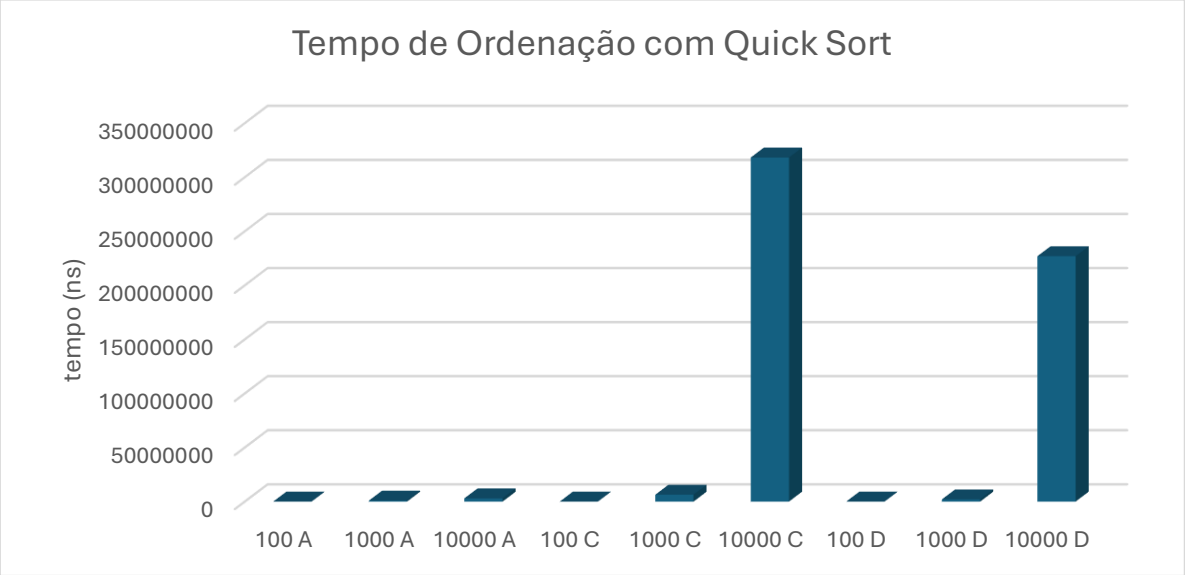
2. Quick Sort:

2.1 Tabela de resultados do tempo de execução para cada vetor:

Quick Sort								
Aleatório			Crescente			Decrescente		
100 A	1000 A	10000 A	100 C	1000 C	10000 C	100 D	1000 D	10000 D
91000 ns	924800 ns	3008000 ns	67300 ns	6370700 ns	318225900 ns	65700 ns	2051200 ns	226941600 ns

Legenda: ns = nanossegundo

2.2 Gráfico:



Legenda:

- **100 A**: Vetor com 100 dados em ordem aleatória
- **1000 A**: Vetor com 1.000 dados em ordem aleatória
- **10000 A**: Vetor com 10.000 dados em ordem aleatória
- **100 C**: Vetor com 100 dados em ordem crescente
- **1000 C**: Vetor com 1.000 dados em ordem crescente
- **10000 C**: Vetor com 10.000 dados em ordem crescente
- **100 D**: Vetor com 100 dados em ordem decrescente
- **1000 D**: Vetor com 1.000 dados em ordem decrescente
- **10000 D**: Vetor com 10.000 dados em ordem decrescente

2.3 Análise:

Com os resultados obtidos, podemos realizar uma análise sobre o algoritmo de ordenação Quick Sort desenvolvido neste projeto. Esse algoritmo é geralmente mais eficiente para arrays de inteiros em ordem aleatória, como mostrado no gráfico, pois seu desempenho tende a ser melhor quando os elementos estão distribuídos de forma imprevisível. Esse é o cenário ideal para o Quick Sort, pois a chance de dividir o array em partes aproximadamente iguais é maior, levando a uma complexidade média de O(n log n) e evitando divisões desbalanceadas.

Para os arrays em ordem crescente e decrescente, o desempenho foi semelhante, devido à escolha fixa do pivô (o último elemento). Essa escolha causou uma divisão desbalanceada, uma vez que o pivô foi o menor valor nos arrays decrescentes e o maior nos crescentes. Como resultado, o algoritmo não conseguiu dividir o array de maneira eficiente, realizando apenas uma partição por chamada recursiva. Esse comportamento gerou uma complexidade próxima a O(n^2), o que caracteriza o pior caso para o Quick Sort. Esse impacto

foi especialmente notável em arrays maiores, como os com 10.000 elementos, onde o tempo de execução aumentou significativamente.

Em resumo, a análise do algoritmo Quick Sort revelou que, em cenários com arrays aleatórios, onde há maior chance de criar partições balanceadas, sua eficiência é maior. Contudo, a escolha do pivô pode impactar significativamente o desempenho. No caso deste projeto, a utilização do último elemento como pivô resultou em divisões desbalanceadas, principalmente em arrays já ordenados, o que levou a uma complexidade de tempo próxima a O(n^2).

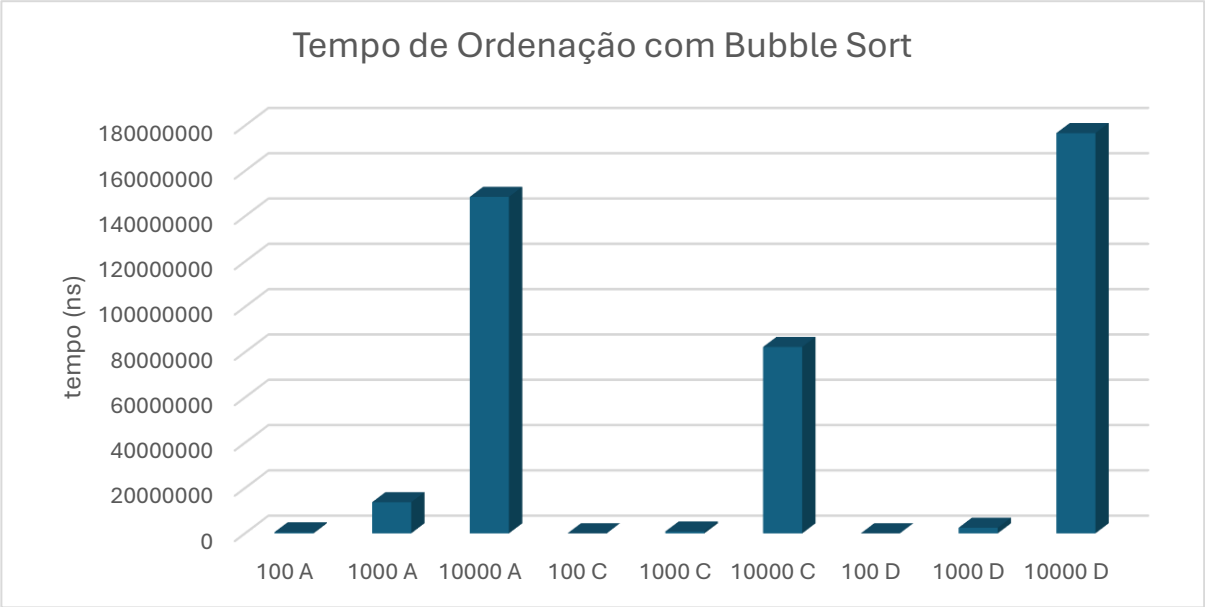
3. Bubble Sort:

3.1 Tabela de resultados do tempo de execução para cada vetor:

Bubble Sort								
Aleatório			Crescente			Decrescente		
100 A	1000 A	10000 A	100 C	1000 C	10000 C	100 D	1000 D	10000 D
464700 ns	13744500 ns	148536400 ns	8900 ns	812100 ns	82274100 ns	18000 ns	2457700 ns	176651200 ns

Legenda: ns = nanossegundo

3.2 Gráfico:



Legenda:

- **100 A**: Vetor com 100 dados em ordem aleatória
- **1000 A**: Vetor com 1.000 dados em ordem aleatória
- **10000 A**: Vetor com 10.000 dados em ordem aleatória
- **100 C**: Vetor com 100 dados em ordem crescente
- **1000 C**: Vetor com 1.000 dados em ordem crescente
- **10000 C**: Vetor com 10.000 dados em ordem crescente
- **100 D**: Vetor com 100 dados em ordem decrescente
- **1000 D**: Vetor com 1.000 dados em ordem decrescente
- **10000 D**: Vetor com 10.000 dados em ordem decrescente

3.3 Análise:

Os resultados nos mostraram algumas conclusões sobre a eficiência do Bubble Sort. Primeiro, esse algoritmo é bem mais rápido em um vetor já ordenado de forma crescente. Isso ocorre porque ele só precisa verificar que os elementos estão na ordem certa, sem fazer trocas desnecessárias. Por isso, os tempos foram bem menores para dados crescentes: 8900 nanosegundos para 100 dados, 812100 nanosegundos para 1000 dados e 82274100 nanosegundos para 10000 dados.

Por outro lado, o pior caso para o Bubble Sort é quando o vetor está em ordem decrescente. Nesse cenário, ele precisa fazer o maior número de trocas, o que aumenta bastante o tempo de execução. O caso mais lento foi com 10000 dados decrescentes, que levou 176651200 nanosegundos.

Já com dados em ordem aleatória, o Bubble Sort apresenta uma "eficiência média". O tempo depende da quantidade de trocas necessárias, que varia de acordo com a desordem dos dados. Ainda assim, o tempo é alto, como vemos em 10000 dados aleatórios, com 148536400 nanosegundos.

Em resumo, o Bubble Sort funciona bem em dados crescentes, tem eficiência média em dados aleatórios e é mais lento em dados decrescentes.

4. Análise Final:

Em resumo, podemos dizer de em nossas implementações dos três algoritmos Bubble Sort, Quick Sort e Insertion Sort, a ordem do melhor resultado (menor tempo) para o pior resultado da ordenação (maior tempo) dos arrays decrescentes (de 10000 dados) para cada tipo de ordenação é:

1. Mais eficiente: Insertion Sort com 150537000 ns.
2. Bubble Sort com 176651200 ns.
3. Menos eficiente: Quick Sort com 226941600 ns.

Já a ordem do melhor resultado (menor tempo) para o pior resultado da ordenação (maior tempo) dos arrays crescentes (de 10000 dados) para cada tipo de ordenação é:

1. Mais eficiente: Insertion Sort com 911300 ns.
2. Bubble Sort com 82274100 ns.
3. Menos eficiente: Quick Sort com 318225900ns.

Já a ordem do melhor resultado (menor tempo) para o pior resultado da ordenação (maior tempo) dos arrays aleatórios (de 10000 dados) para cada tipo de ordenação é:

1. Mais eficiente: Quick Sort com 3008000 ns.
2. Insertion Sort com 105414300 ns.
3. Menos eficiente: Bubble Sort com 148536400 ns.