UNIVERSIDADE ESTADUAL DA PARAÍBA
CENTRO DE CIÊNCIAS E TECNOLOGIA – CAMPUS I
CURSO DE COMPUTAÇÃO (BACHARELADO)
COMPONENTE CURRICULAR: ESTRUTURA DE DADOS
PROF. HERON ARAGÃO MONTEIRO

RELATORIO SOBRE OS ALGORITMOS DE ORDENAÇÃO

JOÃO VITOR AGUIAR DE SOUSA
TÁSSIO SALES DOS SANTOS
YOSSEF ISAAC DE OLIVEIRA

CAMPINA GRANDE – PB
15 de setembro de 2023

INTRODUÇÃO

A ordenação é uma tarefa fundamental em ciência da computação e desempenha um papel crítico na otimização de algoritmos e na melhoria do desempenho de sistemas computacionais. Neste relatório, empreendemos uma análise detalhada e comparativa da performance de seis dos algoritmos de ordenação mais amplamente utilizados: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Counting Sort. A ordenação eficiente é essencial em uma variedade de contextos, desde a organização de grandes volumes de dados até a otimização de operações de pesquisa e recuperação. Portanto, compreender as características desses algoritmos é crucial para a tomada de decisões informadas na escolha do algoritmo mais apropriado para uma determinada tarefa.

Nossa análise será focada principalmente no tempo de execução de cada algoritmo para cada tipo de entrada que o mesmo recebe. Pretendemos identificar as situações em que cada algoritmo se destaca, bem como aquelas em que ele pode ser menos eficiente. Ao final deste estudo, esperamos fornecer informações valiosas que ajudarão a orientar a seleção e implementação de algoritmos de ordenação em uma ampla gama de aplicações computacionais, contribuindo assim para a melhoria da eficiência e do desempenho de sistemas e algoritmos.

METODOLOGIA

Para conduzir este estudo, implementamos todos os algoritmos utilizando a linguagem Java (versão JDK-20) e executamos as implementações por meio da IDE IntelliJ (versão 2023.2.1). Como conjunto de testes, criamos arrays de números inteiros com diferentes arranjos, incluindo ordem crescente, decrescente, aleatória, constante, parcialmente desordenada no final e parcialmente desordenada no início. Vale ressaltar que os arrays parcialmente ordenados foram configurados com 90% dos elementos em ordem crescente e 10% desordenados.

Para cada tipo de array mencionado, geramos conjuntos com tamanhos de 100,000, 500,000, 1.500.000, 2.000.000 e 3.000.000 de elementos. Cada algoritmo foi então executado em cada conjunto de dados correspondente, e os tempos de execução foram registrados utilizando o método nativo da biblioteca Java, ".nanoTime()".

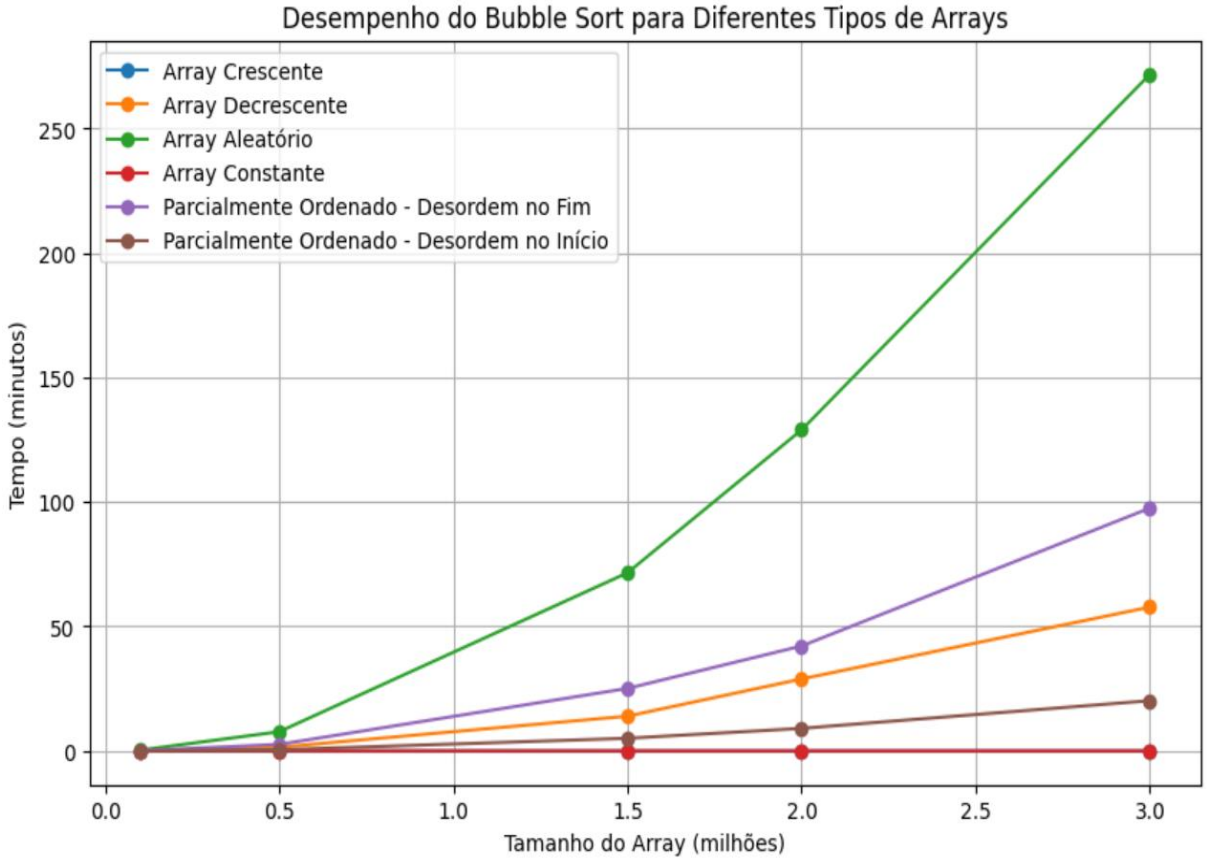
Além disso, é relevante informar as especificações da máquina utilizada para conduzir estas operações:

Placa Mãe	Biostar A320MH 2.0
Processador	AMD Ryzen 5 2400G with Radeon Vega Graphics 3.60 GHz
Memória RAM	12 GB 2400 MHz
Armazenamento	HDD de 500 GB
Placa de vídeo	NVIDIA GeForce RTX 2060 SUPER 8gb
Sistema operacional	Windows 10 Home

BUBBLE SORT

O Bubble Sort, também conhecido como "ordenamento da bolha," é um dos algoritmos de ordenação mais simples e intuitivos já criados. Apesar de sua simplicidade, ele desempenhou um papel fundamental na história da ciência da computação e na educação de programadores. A origem exata do Bubble Sort é um tanto difusa, pois se desenvolveu ao longo do tempo e não há um único criador claramente definido. Desse modo, é possível que tenha surgido nas primeiras décadas da computação, quando programadores começaram a desenvolver métodos para ordenar conjuntos de dados em computadores primitivos. Uma possível contribuição para o desenvolvimento do Bubble Sort pode ser atribuída a Edmund C. Berkeley, autor do livro "Giant Brains, or Machines That Think" publicado em 1949. Berkeley descreveu um algoritmo de ordenação chamado "sorting by exchange" (ordenando por trocas), que compartilha semelhanças com o Bubble Sort.

O algoritmo opera "inplace," o que significa que ele não requer memória adicional para armazenar elementos temporários durante a ordenação. Isso o torna eficiente em termos de espaço. Além disso, ele é um algoritmo estável, o que significa que ele preserva a ordem relativa de elementos com chaves iguais durante a ordenação. Porém, o Bubble Sort possui uma complexidade de tempo de pior caso de $O(n^2)$. Assim vejamos os resultados obtidos após os devidos testes com o mesmo:



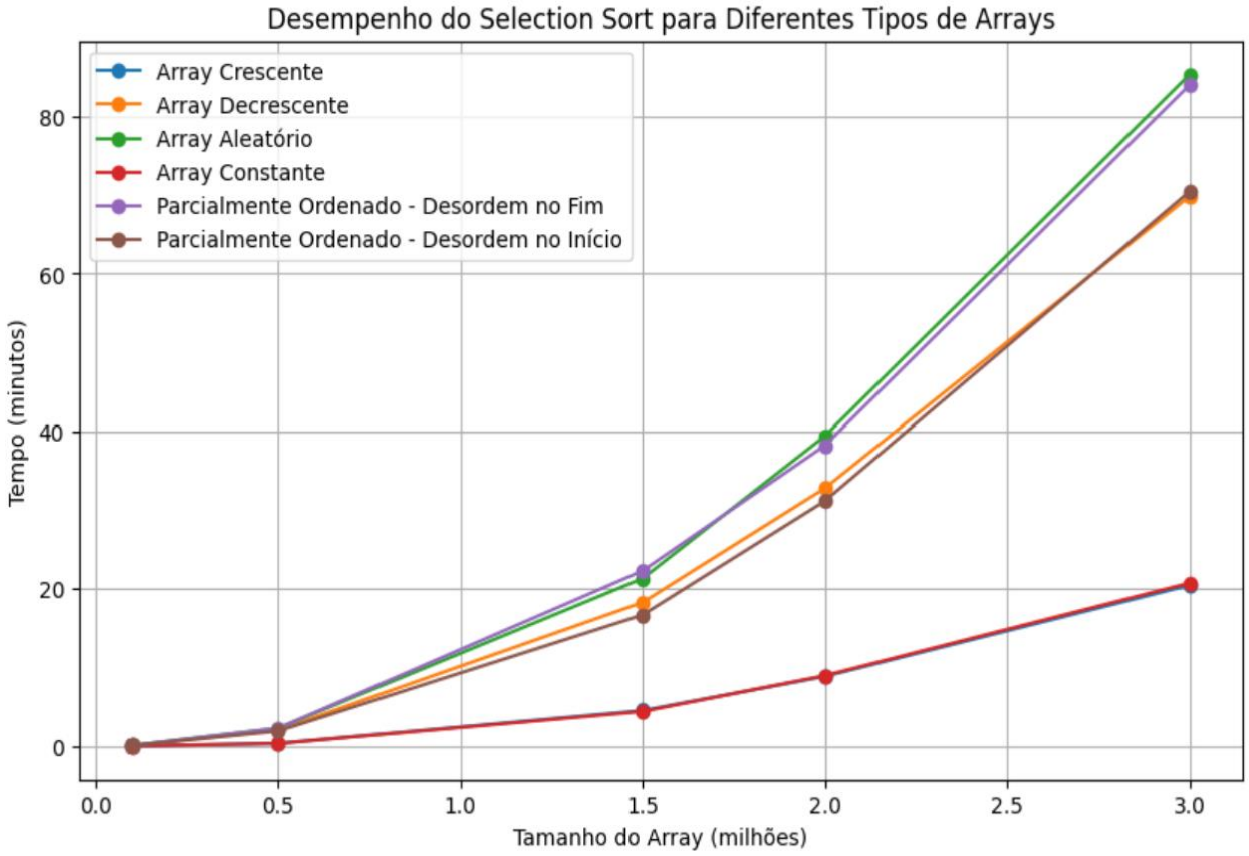
Tipo de Array / Tamanho	*100.000*	*500.000*	*1.500.000*	*2.000.000*	*3.000.000*
Crescente	0,000050 min	0,00018 min	0,0002028 min	0,0002023 min	0,00025 min
Decrescente	0,061 min	1,466 min	13,939 min	28,889 min	57,767 min
Aleatório	0,288 min	7,701 min	71,534 min	128,965 min	271.795 min
Constante	0,000055 min	0,00011 min	0,00014 min	0,00017 min	0,00024 min
Parcialmente ordenado – Dados no Fim	0,112 min	2,602 min	25,088 min	42,136 min	97,534 min
Parcialmente ordenado – Dados no início	0,021 min	0,466 min	5,056 min	9,040 min	20,163 min

Como pode ser visto no gráfico e na tabela acima, o Bubble Sort apresentou baixa eficiência principalmente ao lidar com o array aleatório e o parcialmente ordenado, mas com dados desordenados no fim. Já os seus melhores tempos de eficiência foi ao verificar um array crescente e um constante. Os demais arrays tiveram performance mediana.

SELECTION SORT

O algoritmo de ordenação conhecido como "Selection Sort" não tem um criador específico associado a ele, pois é uma técnica de ordenação bastante simples e intuitiva que tem sido usada por muitos programadores ao longo da história da computação. Ele é uma das primeiras abordagens de ordenação desenvolvidas. Embora não seja atribuído a uma pessoa em particular, o Selection Sort é amplamente utilizado como exemplo de algoritmo de ordenação em livros de programação, cursos de ciência da computação e recursos educacionais. Sua simplicidade o torna uma escolha popular para ensinar conceitos básicos de ordenação e algoritmos. Em resumo, o Selection Sort é uma técnica de ordenação que se originou da necessidade de classificar elementos em computação e é conhecida por sua simplicidade e facilidade de compreensão.

O algoritmo opera "inplace", o que significa que ele não requer memória adicional para armazenar elementos temporários durante a ordenação. Isso o torna eficiente em termos de espaço. Entretanto, o Selection Sort não é um algoritmo estável. Isso significa que, se houver chaves iguais nos dados de entrada, a ordem relativa dessas chaves pode não ser preservada após a ordenação. Além disso, o mesmo possui uma complexidade de tempo quadrática $O(n^2)$. Assim vejamos os resultados obtidos após os devidos testes com o mesmo:



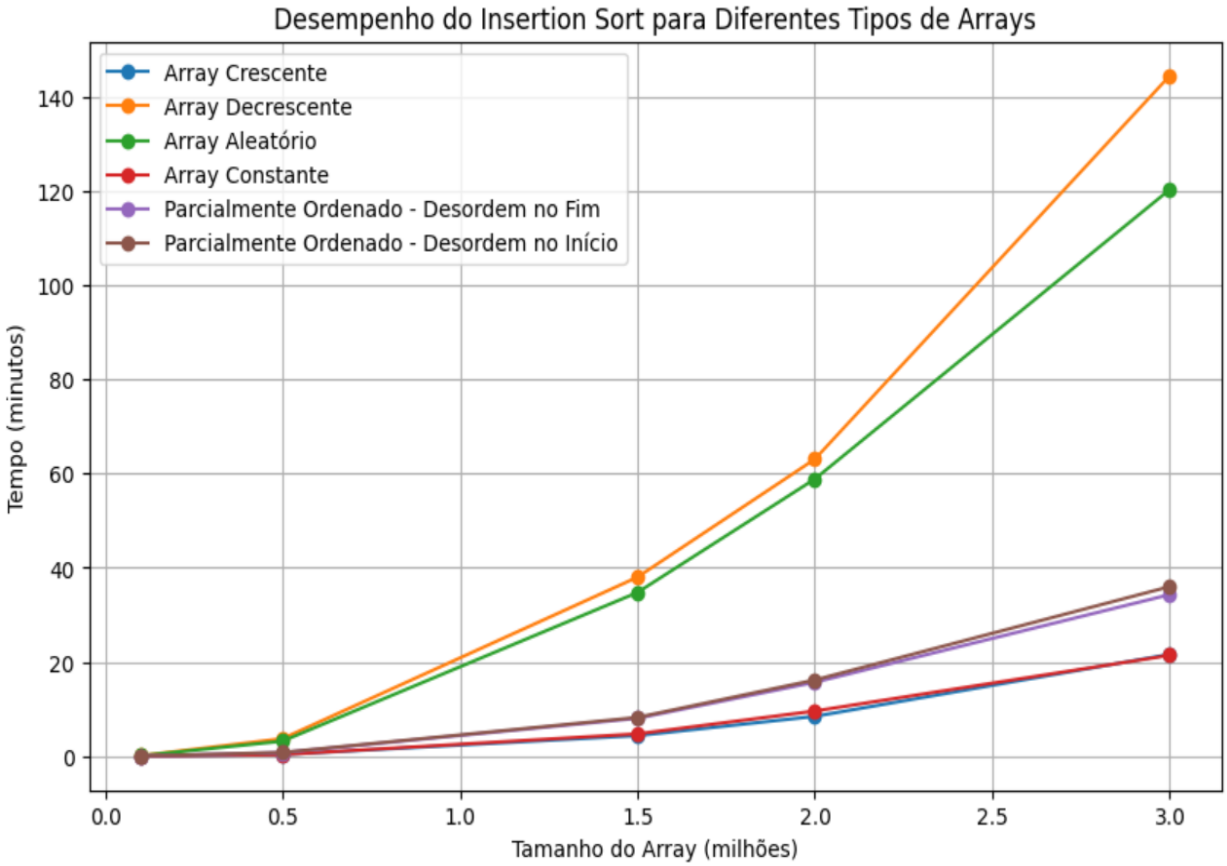
Tipo de Array / Tamanho	*100.000*	*500.000*	*1.500.000*	*2.000.000*	*3.000.000*
Crescente	0,0187 min	0,361 min	4,534 min	8,874 min	20,471 min
Decrescente	0,085 min	2,058 min	18,290 min	32,806 min	69,838 min
Aleatório	0,098 min	2,281 min	21,341 min	39,418 min	85,335 min
Constante	0,0180 min	0,341 min	4,416 min	8,977 min	20,754 min
Parcialmente ordenado – Dados no Fim	0,096 min	2,277 min	22,312 min	38,227 min	84,010 min
Parcialmente ordenado – Dados no início	0,075 min	1,925 min	16,702 min	31,179 min	70,456 min

Como pode ser visto no gráfico e na tabela acima, o Selection Sort apresentou baixa eficiência principalmente ao lidar com o array aleatório e o parcialmente ordenado, mas com dados desordenados no fim. Já os seus melhores tempos de eficiência foi ao verificar um array crescente e um constante. Os demais arrays ficaram no caso médio, mas com as suas eficiências mais próximas do pior caso.

INSERTION SORT

O Insertion Sort é um dos algoritmos de ordenação mais antigos conhecidos. Embora sua origem exata seja difícil de rastrear, ele tem sido utilizado por gerações de programadores e matemáticos. Sua ideia básica remonta à antiguidade e já foi utilizada por pensadores como Arquimedes para ordenar listas de números. Esse algoritmo é notável por sua simplicidade conceitual e de implementação. Pois, funciona da mesma maneira que muitas pessoas naturalmente ordenariam um baralho de cartas: inserindo uma carta na posição correta em uma mão ordenada.

O Insertion Sort opera "inplace", o que significa que não requer memória adicional para armazenar elementos temporários durante a ordenação. Isso torna o algoritmo eficiente em termos de espaço. Além disso, ele é um algoritmo estável, o que significa que ele preserva a ordem relativa de elementos com chaves iguais. Porém, sua complexidade de tempo média e pior caso é $O(n^2)$. Assim vejamos os resultados obtidos após os devidos testes com o mesmo:



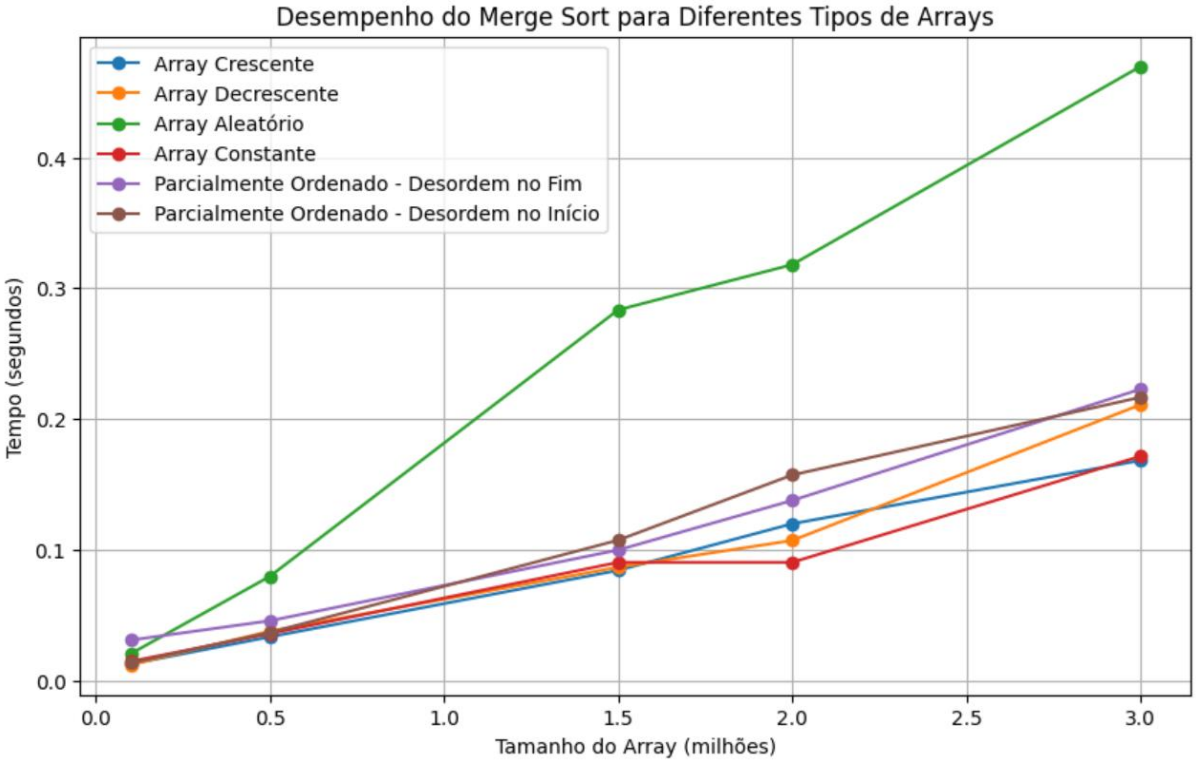
Tipo de Array / Tamanho	*100.000*	*500.000*	*1.500.000*	*2.000.000*	*3.000.000*
Crescente	0,020 min	0,359 min	4,367 min	8,457 min	21,629 min
Decrescente	0,116 min	3,736 min	38,014 min	63,077 min	144,495 min
Aleatório	0,105 min	3,198 min	34,762 min	58,863 min	120,318 min
Constante	0,015 min	0,351 min	4,746 min	9,597 min	21,430 min
Parcialmente ordenado – Dados no Fim	0,028 min	0,756 min	8,042 min	15,702 min	34,318 min
Parcialmente ordenado – Dados no início	0,035 min	0,806 min	8,238 min	16,192 min	36,014 min

Como pode ser visto no gráfico e na tabela acima, o Insertion Sort apresentou baixa eficiência principalmente ao lidar com o array decrescente e o array aleatório. Já os seus melhores tempos de eficiência foi ao verificar um array crescente e um constante. Os demais arrays ficaram no caso médio, mas com as suas eficiências mais próximas do melhor caso.

MERGE SORT

O Merge Sort é uma técnica de ordenação que tem suas raízes na década de 1940, no início do desenvolvimento da computação. Vários pesquisadores contribuíram para o desenvolvimento da ideia por trás do Merge Sort como John von Neumann, Hermann Grassmann, Arthur Burks e Herman Goldstine. Embora não haja um único criador do Merge Sort, a ideia por trás do algoritmo evoluiu ao longo de várias décadas e foi influenciada por várias mentes brilhantes da computação e da matemática.

O algoritmo baseia-se no paradigma "divisão e conquista", que divide o problema de ordenação em subproblemas menores, resolve esses subproblemas e, em seguida, combina suas soluções para obter o resultado final. Por isso, o Merge Sort é conhecido por sua eficiência e desempenho consistente. Ele é um algoritmo estável, o que significa que ele preserva a ordem relativa de elementos com chaves iguais durante a ordenação. Além disso, ele possui uma complexidade de tempo de O(n log n). Existem implementações inplace e não inplace do Merge Sort. A versão inplace consome menos memória, enquanto a versão não inplace é mais fácil de implementar. Nesse estudo, foi implementada a versão não inplace. Assim vejamos os resultados obtidos após os devidos testes com o mesmo:



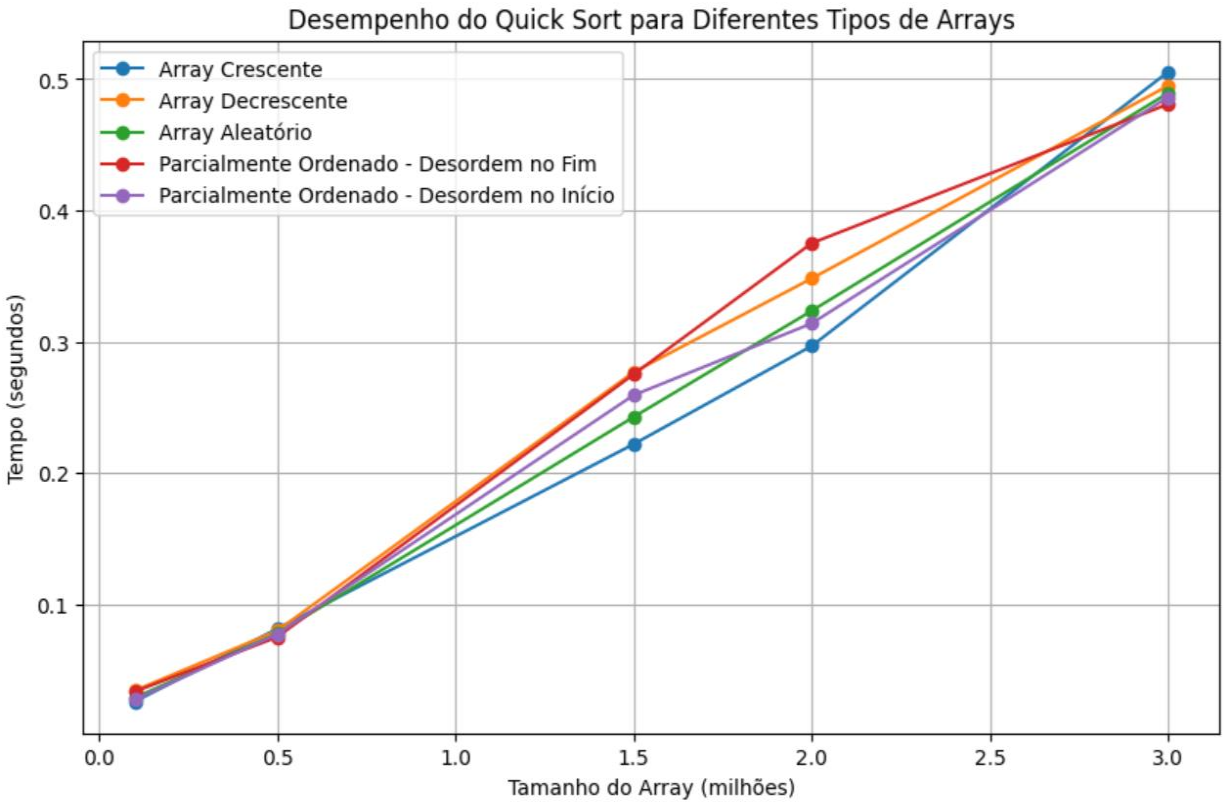
Tipo de Array / Tamanho	*100.000*	*500.000*	*1.500.000*	*2.000.000*	*3.000.000*
Crescente	0,0124 seg	0.033 seg	0,084 seg	0,119 seg	0,168 seg
Decrescente	0,011 seg	0,037 seg	0,086 seg	0,107 seg	0,210 seg
Aleatório	0,020 seg	0,079 seg	0,283 seg	0,318 seg	0,469 seg
Constante	0,014 seg	0,035 seg	0,090 seg	0,090 seg	0,171 seg
Parcialmente ordenado – Dados no Fim	0,030 seg	0,045 seg	0,099 seg	0,137 seg	0,222 seg
Parcialmente ordenado – Dados no início	0,013 seg	0,036 seg	0,107 seg	0,157 seg	0,216 seg

Como pode ser visto no gráfico e na tabela acima, o Merge Sort apresentou alta eficiência em todas as massas de testes. Mas, o maior destaque fica para o array crescente e para o array constante. A pior performance foi para o array aleatório e os demais arrays ficaram no caso médio, mas com as suas eficiências mais próximas do melhor caso.

QUICK SORT

O Quick Sort foi desenvolvido por Sir Charles Antony Richard Hoare, também conhecido como Tony Hoare, um cientista da computação britânico. Ele criou o algoritmo enquanto trabalhava na Universidade de Cambridge, no Reino Unido, em 1960. Inicialmente, Hoare desenvolveu o Quick Sort como uma solução mais eficiente do que os métodos de ordenação disponíveis na época, como o Bubble Sort e o Insertion Sort. Sua inovação e eficácia rapidamente o tornaram uma parte fundamental da ciência da computação e da programação.

O Quick Sort seleciona um elemento especial chamado de "pivô" a partir do conjunto de dados. Assim, os elementos são rearranjados de modo que os elementos menores que o pivô estejam à esquerda e os elementos maiores estejam à direita. Isso divide o conjunto em duas partes. Além disso, ele pode ser implementado como um algoritmo "in-place", o que significa que ele não requer espaço de memória adicional significativo para classificar a lista. Ele não é um algoritmo de ordenação estável, o que significa que a ordem relativa de elementos com chaves iguais pode não ser preservada após a classificação. Assim vejamos os resultados obtidos após os devidos testes com o mesmo:



Tipo de Array / Tamanho	*100.000*	*500.000*	*1.500.000*	*2.000.000*	*3.000.000*
Crescente	0,025 seg	0,081 seg	0,221 seg	0,296 seg	0,505 seg
Decrescente	0,034 seg	0,079 seg	0,276 seg	0,348 seg	0,494 seg
Aleatório	0,028 seg	0,077 seg	0,242 seg	0,323 seg	0,489 seg
Parcialmente ordenado – Dados no Fim	0,033 seg	0,075 seg	0,275 seg	0,375 seg	0,480 seg
Parcialmente ordenado – Dados no início	0,027 seg	0,077 seg	0,259 seg	0,313 seg	0,486 seg

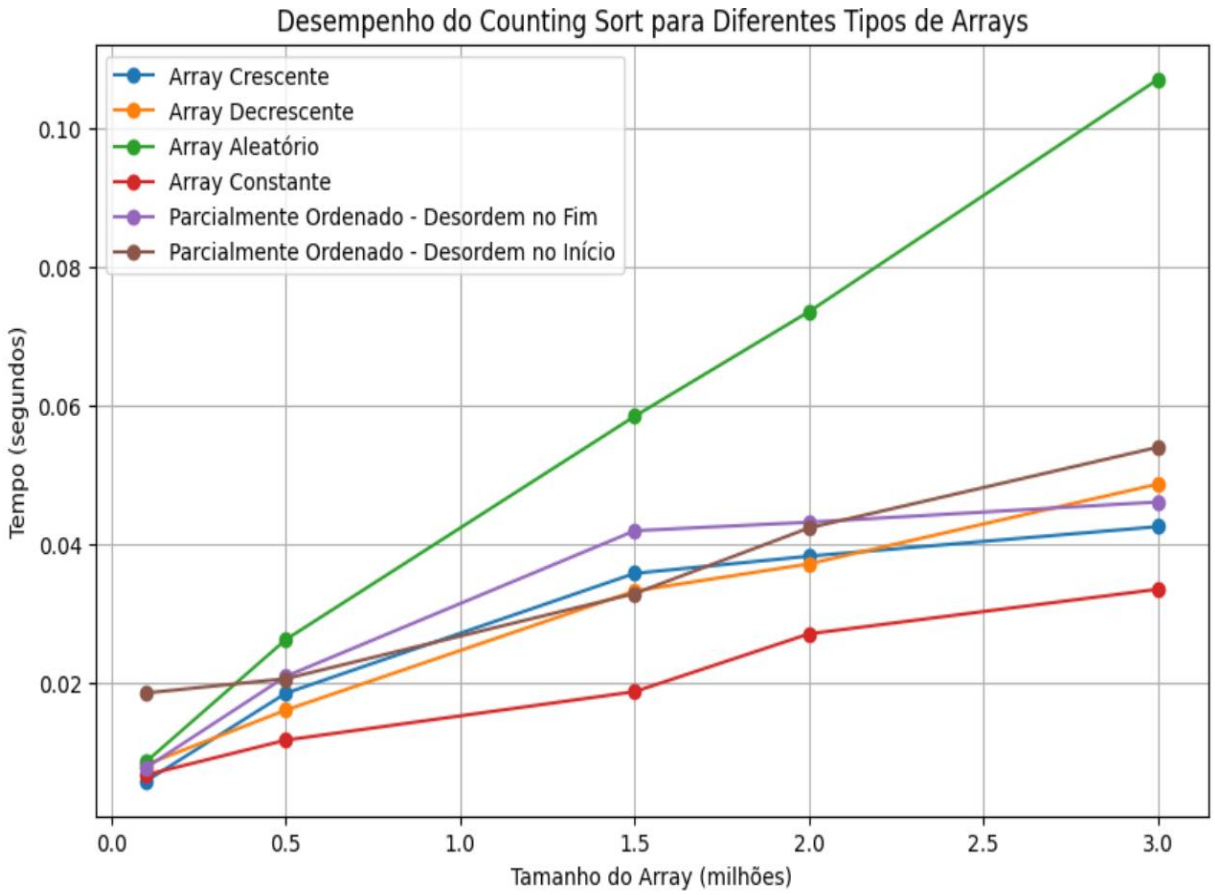
Como pode ser visto no gráfico e na tabela acima, o Quick Sort apresentou alta eficiência em cinco das seis massas de testes. Não tivemos um destaque de tempo considerável entre as massas de testes, pois todas apresentaram tempos muito próximos.

Durante os testes, o Quick Sort não conseguiu lidar com o array constante. O mesmo apresentou erro de stackoverfow a partir de arrays maiores que 12227 elementos. Valores abaixo disso ele conseguiu executar em tempo menor que 0,038 segundos.

COUTING SORT

O Counting Sort, também conhecido como ordenação de contagem, é um algoritmo de ordenação que se destaca por sua eficiência quando aplicado a uma ampla gama de situações específicas. Seu criador é Harold H. Seward e ele desenvolveu esse algoritmo como uma solução eficiente para ordenar grandes conjuntos de dados em uma época em que os computadores tinham recursos de memória limitados.

O Counting Sort tem uma complexidade de tempo linear O(n), onde "n" é o número de elementos no conjunto de dados. Isso torna o algoritmo muito eficiente para ordenar grandes quantidades de dados. Além disso, ele é um algoritmo de ordenação estável, o que significa que ele mantém a ordem relativa dos elementos com valores iguais. Assim vejamos os resultados obtidos após os devidos testes com o mesmo:

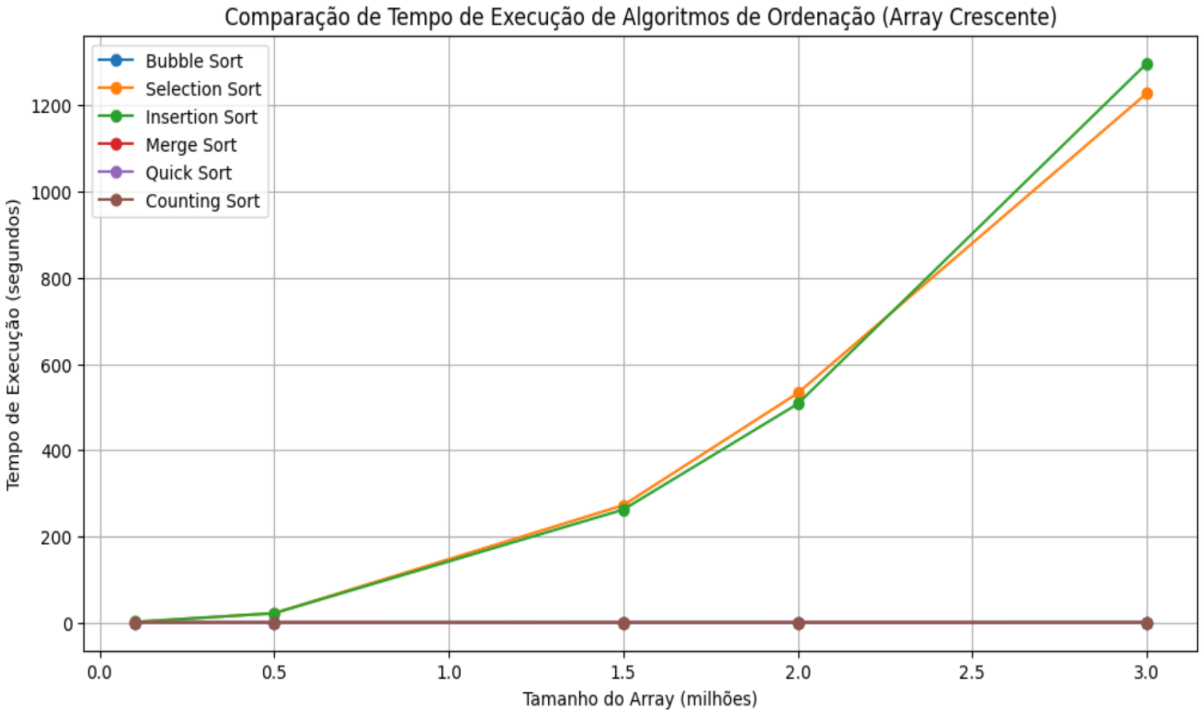


Tipo de Array / Tamanho	*100.000*	*500.000*	*1.500.000*	*2.000.000*	*3.000.000*
Crescente	0,005 seg	0,018 seg	0,035 seg	0,038 seg	0,042 seg
Decrescente	0,008 seg	0,016 seg	0,033 seg	0,037 seg	0,048 seg
Aleatório	0,008 seg	0,026 seg	0,058 seg	0,073 seg	0,107 seg
Constante	0,006 seg	0,011 seg	0,018 seg	0,027 seg	0,033 seg
Parcialmente ordenado – Dados no Fim	0,007 seg	0,020 seg	0,041 seg	0,043 seg	0,046 seg
Parcialmente ordenado – Dados no início	0,018 seg	0,020 seg	0,032 seg	0,042 seg	0,054 seg

Como pode ser visto no gráfico e na tabela acima, o Conting Sort também apresentou alta eficiência em todas as massas de testes. Assim, o maior destaque fica para o array constante. Já a pior performance foi para o array aleatório e os demais arrays ficaram com as suas eficiências bem próximas do melhor caso.

COMPARAÇÃO ENTRE OS ALGORITMOS

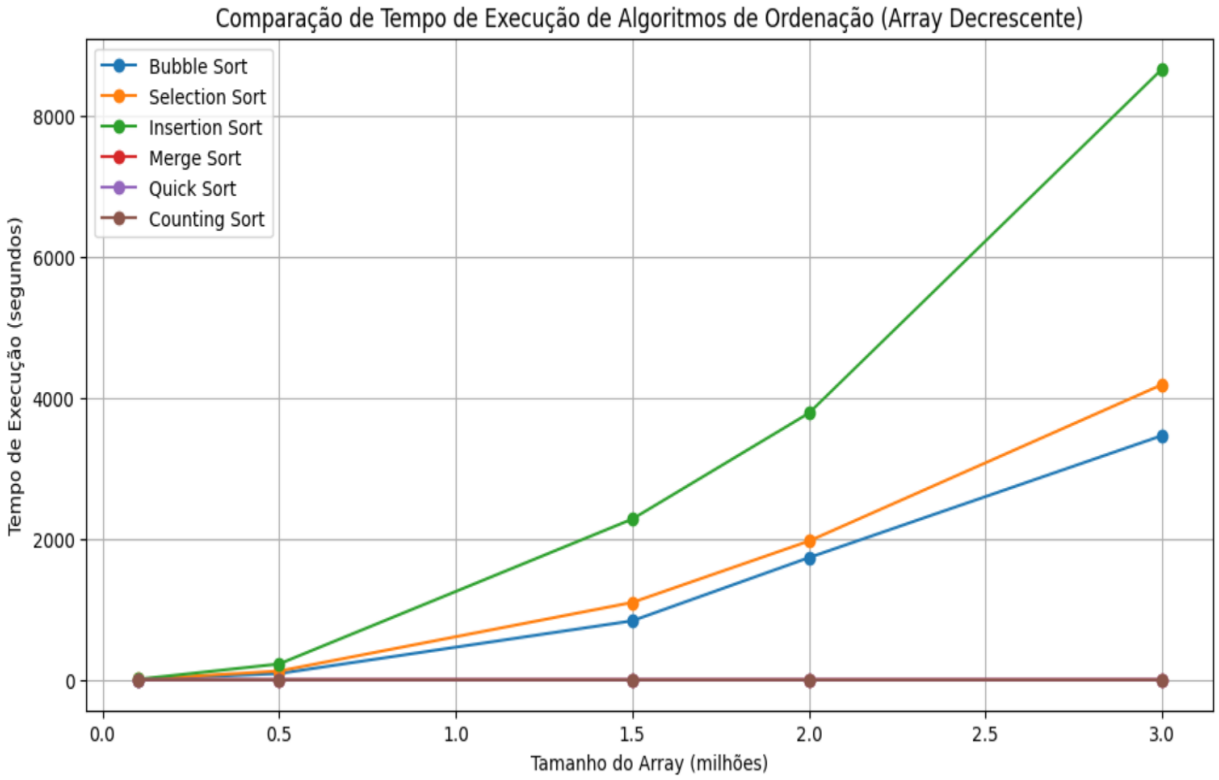
Após fazermos as comparações individualmente para cada algoritmo, agora será feita uma comparação entre eles para visualizarmos qual possui a melhor performance e qual possui a pior para cada tipo de array. Vejamos quais foram os resultados obtidos com o array crescente:



Tipo de Algoritmo / Tamanho	*100.000*	*500.000*	*1.500.000*	*2.000.000*	*3.000.000*
Bubble Sort	0,003 seg	0,011 seg	0,012 seg	0,012 seg	0,015 seg
Selection Sort	1,127 seg	21,676 seg	272,084 seg	532,474 seg	1228,271 seg
Insertion Sort	1,216 seg	21,567 seg	262,052 seg	507,461 seg	1297,766 seg
Merge Sort	0,012 seg	0.033 seg	0,084 seg	0,119 seg	0,168 seg
Quick Sort	0,025 seg	0,081 seg	0,221 seg	0,296 seg	0,505 seg
Counting Sort	0,005 seg	0,018 seg	0,035 seg	0,038 seg	0,042 seg

Como pode ser visto no gráfico e na tabela acima, o Bubble Sort apresentou a maior eficiência entre os algoritmos para o array crescente. Mas, tendo o Couting Sort, o Quick Sort e o Merge Sort também tiveram boas performances. Já o Insertion Sort e o Selection Sort foram os piores em eficiência.

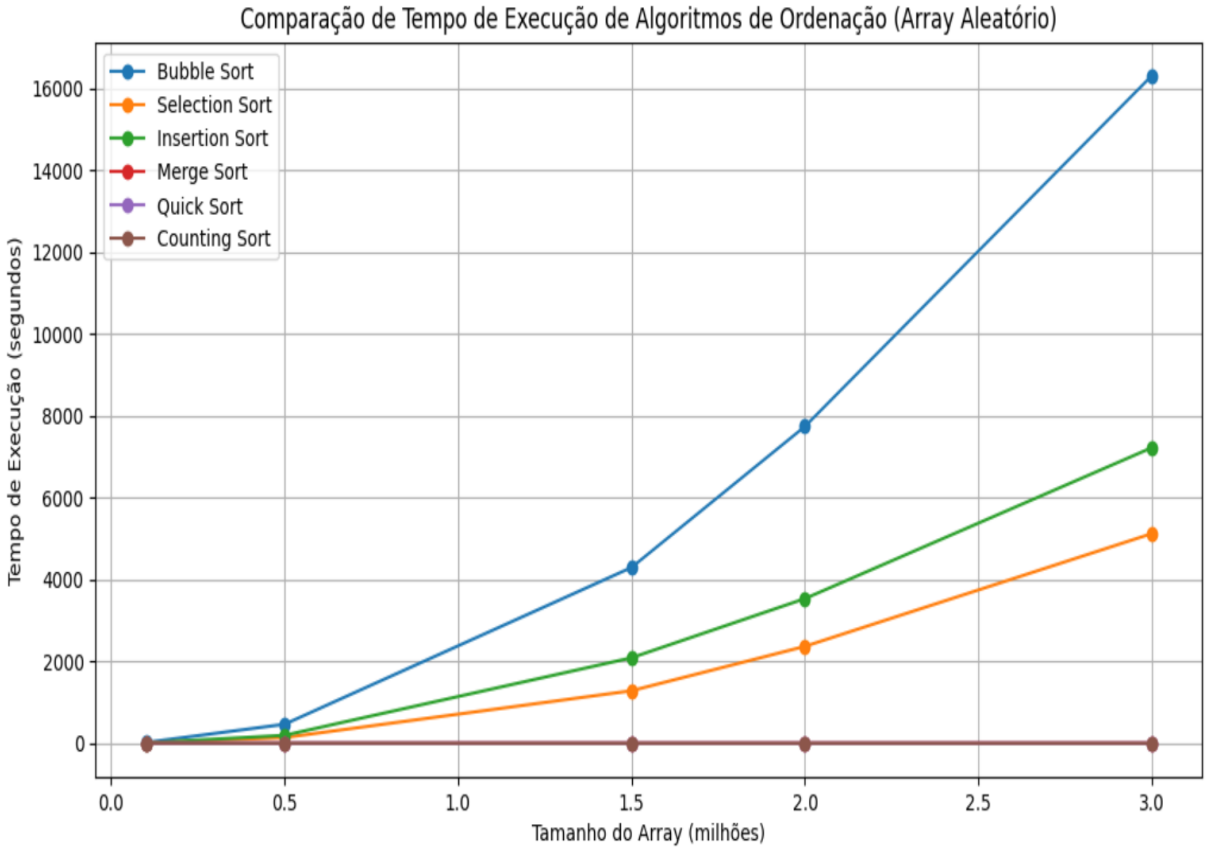
Vejamos quais foram os resultados obtidos com o array decrescente:



Tipo de Algoritmo / Tamanho	*100.000*	*500.000*	*1.500.000*	*2.000.000*	*3.000.000*
Bubble Sort	3,674 seg	87,989 seg	836,356 seg	1733,395 seg	3466.040 seg
Selection Sort	5,102 seg	123,527 seg	1097,444 seg	1968,419 seg	4190,281 seg
Insertion Sort	7,002 seg	224,215 seg	2280,841 seg	3784,644 seg	8669,715 seg
Merge Sort	0,011 seg	0,037 seg	0,086 seg	0,107 seg	0,210 seg
Quick Sort	0,034 seg	0,079 seg	0,276 seg	0,348 seg	0,494 seg
Counting Sort	0,008 seg	0,016 seg	0,033 seg	0,037 seg	0,048 seg

Como pode ser visto no gráfico e na tabela acima, o Counting Sort apresentou a maior eficiência entre os algoritmos para o array decrescente. Mas, tendo o Quick Sort e o Merge Sort também com boas performances. Já a pior performance novamente foi a do Insertion Sort, mas com o Selection Sort e Bubble Sort também entregando performances de baixa eficiência.

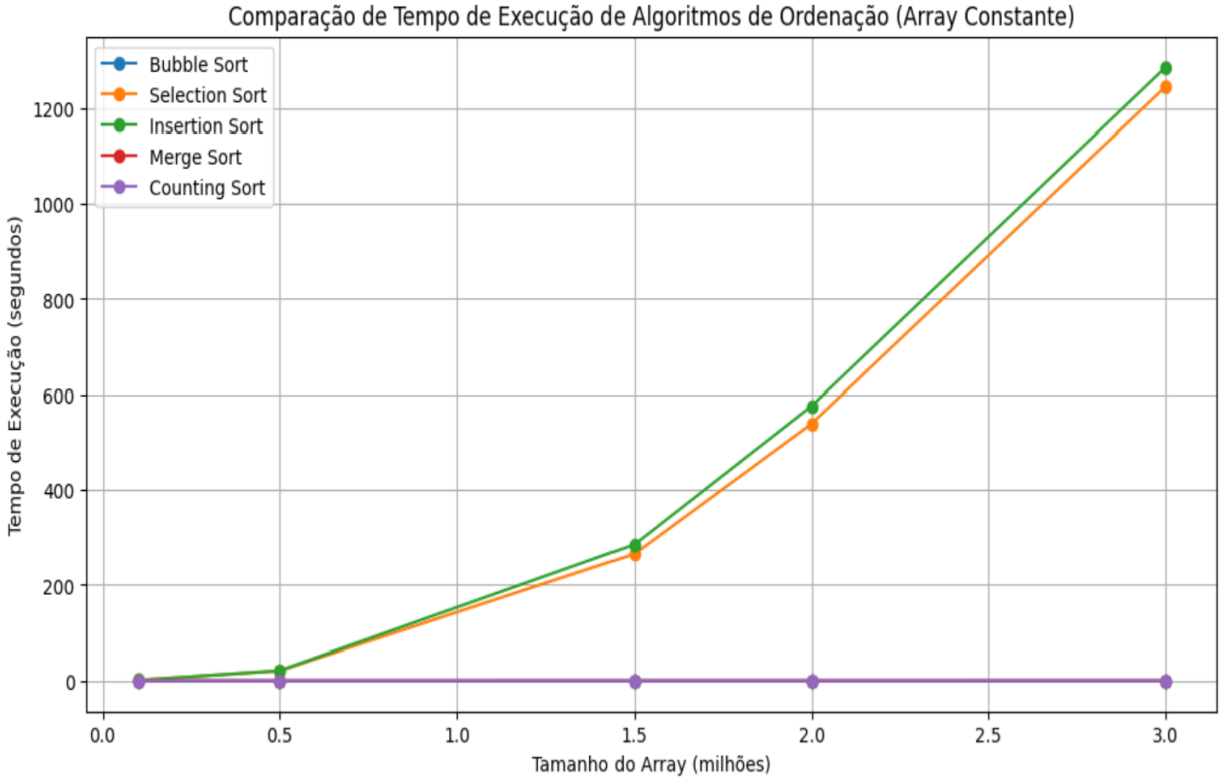
Vejamos quais foram os resultados obtidos com o array aleatório:



Tipo de Algoritmo / Tamanho	*100.000*	*500.000*	*1.500.000*	*2.000.000*	*3.000.000*
Bubble Sort	17,308 seg	462,077 seg	4292,093 seg	7737,914 seg	16307,748 seg
Selection Sort	5,886 seg	136,900 seg	1280,493 seg	2365,115 seg	5120,144 seg
Insertion Sort	6,345 seg	191,924 seg	2085,754 seg	3531,811 seg	7219,121 seg
Merge Sort	0,020 seg	0,079 seg	0,283 seg	0,318 seg	0,469 seg
Quick Sort	0,028 seg	0,077 seg	0,242 seg	0,323 seg	0,489 seg
Counting Sort	0,008 seg	0,026 seg	0,058 seg	0,073 seg	0,107 seg

Como pode ser visto no gráfico e na tabela acima, o Counting Sort apresentou a maior eficiência entre os algoritmos para o array aleatório. Mas, tendo o Quick Sort e o Merge Sort também com boas performances. Já a pior performance foi a do Bubble Sort, a qual foi a pior performance geral dentre todos os testes realizados. O Selection Sort e o Insertion Sort também tiveram performances de baixa eficiência.

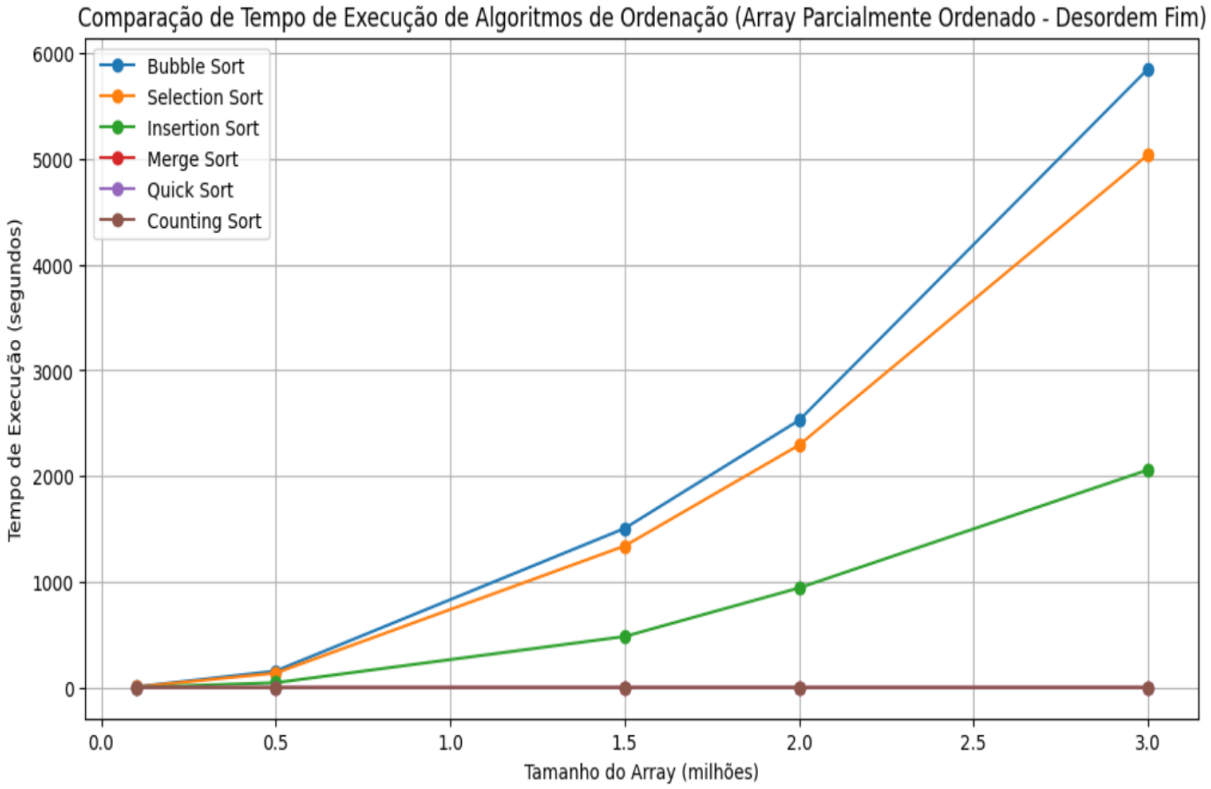
Vejamos quais foram os resultados obtidos com o array constante:



Tipo de Algoritmo / Tamanho	*100.000*	*500.000*	*1.500.000*	*2.000.000*	*3.000.000*
Bubble Sort	0,003 seg	0,006 seg	0,008 seg	0,010 seg	0,014 seg
Selection Sort	1,081 seg	20,519 seg	264,991 seg	538,671 seg	1245,269 seg
Insertion Sort	0,942 seg	21,092 seg	284,768 seg	575,838 seg	1285,811 seg
Merge Sort	0,014 seg	0,035 seg	0,090 seg	0,090 seg	0,171 seg
Counting Sort	0,018 seg	0,020 seg	0,032 seg	0,042 seg	0,054 seg

Como pode ser visto no gráfico e na tabela acima, o Bubble Sort apresentou a maior eficiência entre os algoritmos para o array constante. Mas, tendo o Counting Sort e o Merge Sort também com boas performances. Já a pior performance foi novamente a do Insertion Sort, mas com o Selection Sort tendo uma performance bem próxima. Como foi dito anteriormente, o Quick Sort não conseguiu verificar os arrays constantes devido ao erro de stackoverflow.

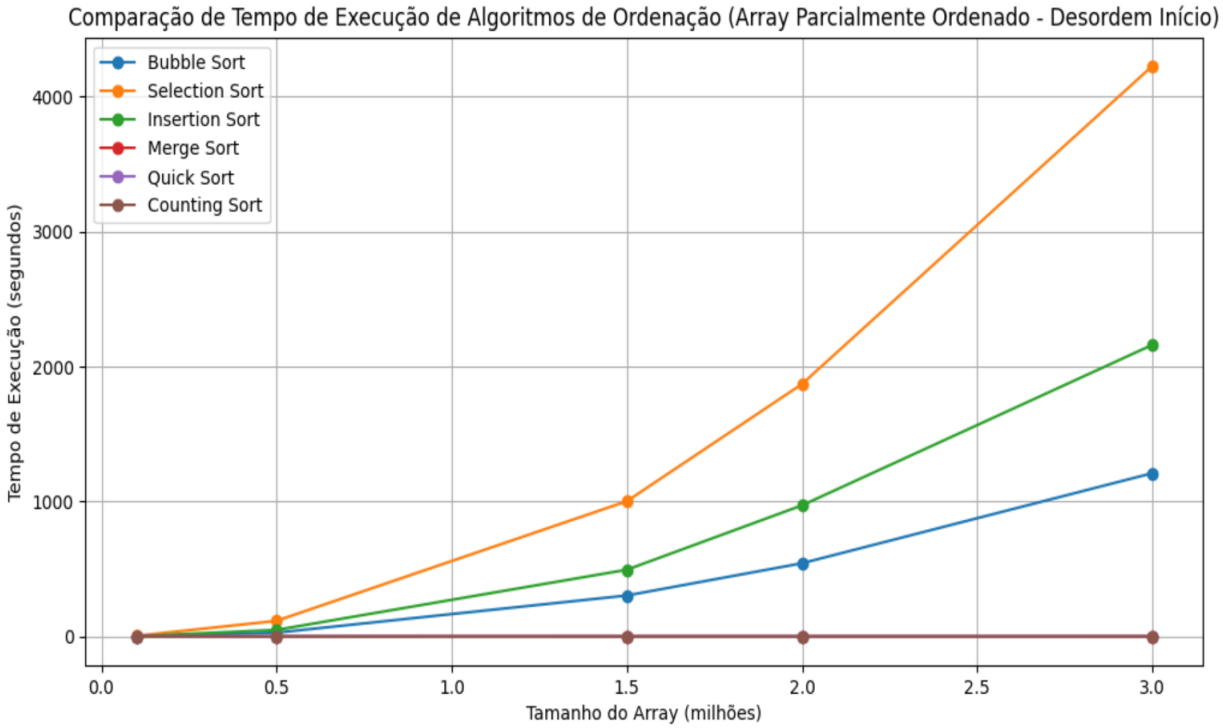
Vejamos quais foram os resultados obtidos com o array parcialmente ordenado e com dados desordenados no seu fim:



Tipo de Algoritmo / Tamanho	*100.000*	*500.000*	*1.500.000*	*2.000.000*	*3.000.000*
Bubble Sort	6,746 seg	156,164 seg	1505,322 seg	2528,161 seg	5852,046 seg
Selection Sort	5,804 seg	136,663 seg	1338,729 seg	2293,622 seg	5040,628 seg
Insertion Sort	1,700 seg	45,364 seg	482,566 seg	942,131 seg	2059,093 seg
Merge Sort	0,030 seg	0,045 seg	0,099 seg	0,137 seg	0,222 seg
Quick Sort	0,033 seg	0,075 seg	0,275 seg	0,375 seg	0,480 seg
Counting Sort	0,007 seg	0,020 seg	0,041 seg	0,043 seg	0,046 seg

Como pode ser visto no gráfico e na tabela acima, o Counting Sort apresentou a maior eficiência entre os algoritmos para o array parcialmente ordenado, mas com dados desordenados no seu fim. O Quick Sort e o Merge Sort também tiveram boas performances. Já a pior performance foi a do Bubble Sort. Mas, o Selection Sort e o Insertion Sort também tiveram performances de baixa eficiência.

Vejamos quais foram os resultados obtidos com o array parcialmente ordenado e com dados desordenados no seu início:



Tipo de Algoritmo / Tamanho	*100.000*	*500.000*	*1.500.000*	*2.000.000*	*3.000.000*
Bubble Sort	1,293 seg	27,982 seg	303,416 seg	542,424 seg	1209,821 seg
Selection Sort	4,502 seg	115,555 seg	1002,174 seg	1870,740 seg	4227,397 seg
Insertion Sort	2,145 seg	48,379 seg	494,322 seg	971,577 seg	2160,859 seg
Merge Sort	0,013 seg	0,036 seg	0,107 seg	0,157 seg	0,216 seg
Quick Sort	0,027 seg	0,077 seg	0,259 seg	0,313 seg	0,486 seg
Counting Sort	0,018 seg	0,020 seg	0,032 seg	0,042 seg	0,054 seg

Como pode ser visto no gráfico e na tabela acima, o Counting Sort apresentou a maior eficiência entre os algoritmos para o array parcialmente ordenado, mas com dados desordenados no seu início. O Quick Sort e o Merge Sort também tiveram boas performances. Já a pior performance foi a do Bubble Sort. Mas, o Selection Sort e o Insertion Sort também tiveram performances de baixa eficiência.

CONCLUSÃO

Após a análise dos resultados dos testes, identificamos que o algoritmo Counting Sort teve o melhor desempenho na maioria dos cenários de testes. Este algoritmo foi particularmente eficiente quando se lida com arrays decrescentes, aleatórios e parcialmente ordenados. Por outro lado, o algoritmo Insertion Sort demonstrou consistentemente o pior desempenho na maioria dos cenários de testes. Ele teve dificuldades para lidar principalmente com arrays crescentes, decrescentes e constantes.

Os algoritmos Merge Sort e Quick Sort também apresentaram um bom desempenho na maioria dos cenários de testes, mas o Quick Sort teve problemas ao lidar com array constantes devido à alocação de memória. Por isso, foi considerado o pior algoritmo para lidar com arrays constantes. O Bubble Sort teve o melhor desempenho com arrays crescentes e constantes, mas nos demais testes não apresentou bom desempenho. Já o Selection Sort não se destacou em positivamente em teste algum, o que torna o mesmo quase tão ineficiente quanto o Insertion Sort.

Em resumo, a escolha do algoritmo de ordenação deve ser feita com base nas características específicas dos dados e nos requisitos de desempenho do aplicativo. O Counting Sort se destacou como a escolha preferida na maioria dos cenários de teste, mas é importante considerar as necessidades individuais do projeto ao selecionar um algoritmo de ordenação adequado.