RELATÓRIO DE ALGORITMOS DE ORDENAÇÃO

Discussões sobre os resultados obtidos

Kauan Cardoso

Marco Antonio Menegati

14.09.2024
IMPLEMENTAÇÃO ALGORÍTMICA-T01-2024-2

INTRODUÇÃO

Este relatório propõe inferir sobre a eficiência e execução dos algoritmos de ordenação requeridos: BubbleSort, InsertionSort, MergeSort, HeapSort, QuickSort e CountingSort. Acompanhado dos gráficos e tabelas de cada vetor de teste para mais fácil compreensão.

Este relatório observa testes dos algoritmos com 4 tipos de vetores: Aleatório(RANDOM), Reverso(REVERSE), Ordenado(SORTED) e Quase Ordenado(NEARLY SORTED).

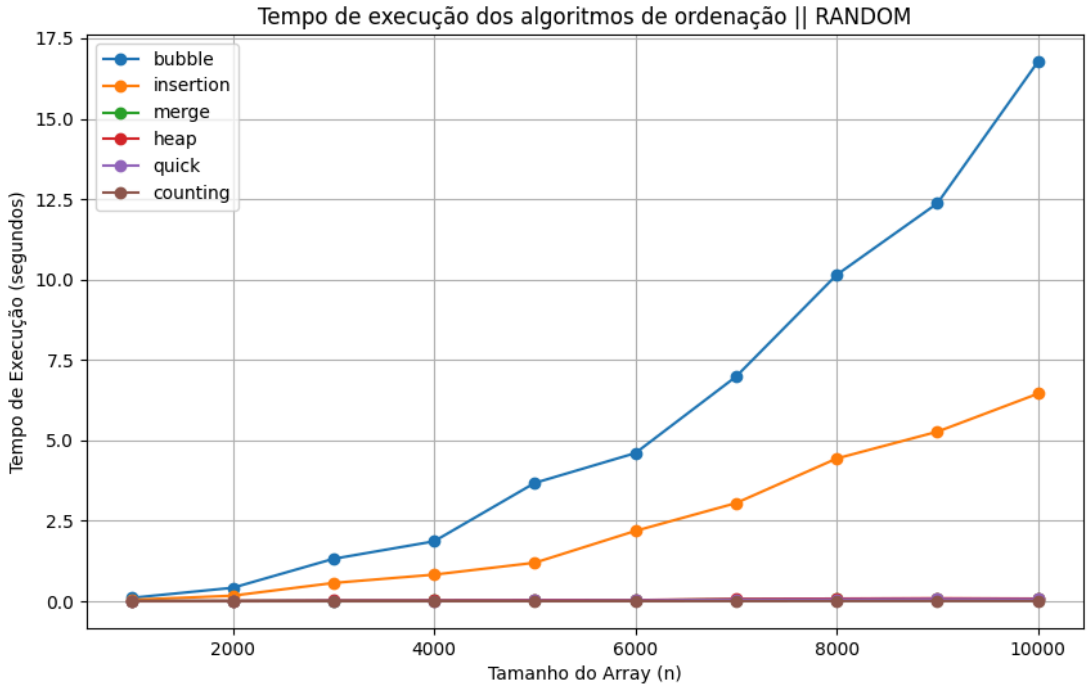
Para os testes, os valores de parâmetro utilizados foram:

- inc, para Valor Inicial, = 1000;
- fim, para Valor Final, = 10.000;
- stp, para Intervalo entre dois valores, = 1000 e
- rpt, para Valor de repetições a serem realizadas, , = 10.

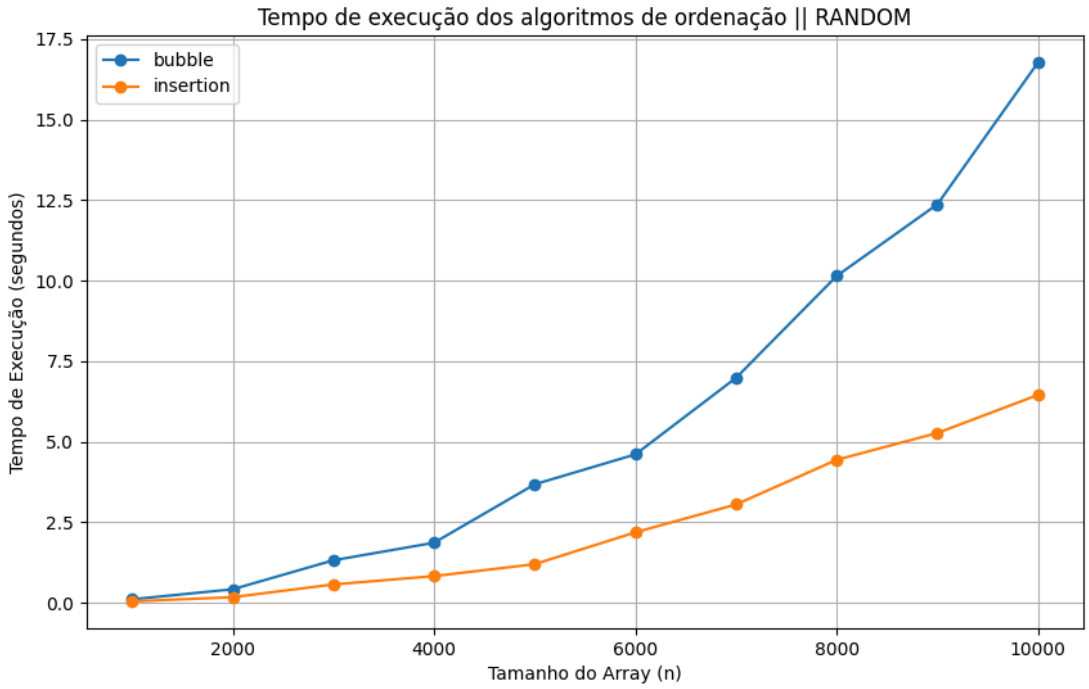
OBS: Como instruído posteriormente utilizamos 2n para o intervalo de geração de valores dos vetores.

VETOR RANDOM

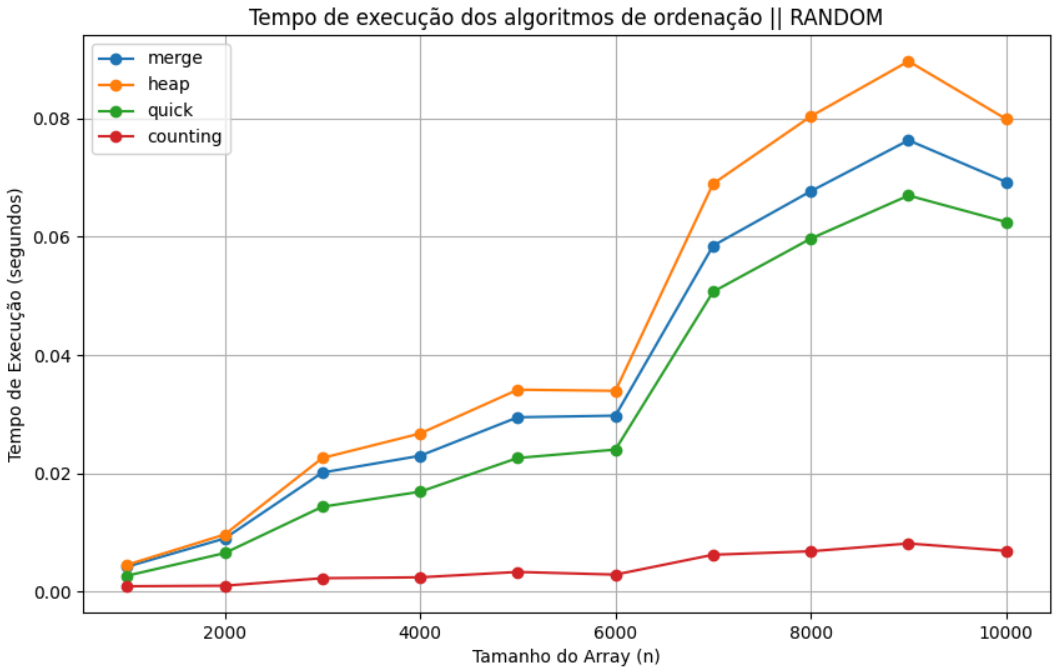
O vetor Aleatório(RANDOM) contém n números não negativos escolhidos aleatoriamente, respeitando os parâmetros inseridos. Abaixo, observa-se o tempo de execução dos algoritmos de ordenação com o vetor aleatório:



Claramente, os algoritmos BubbleSort e InsertionSort(vistos abaixo em destaque), com as suas notações $O(n^2)$, não são eficientes para a ordenação de um vetor aleatório, uma vez que, o BubbleSort, pelo modo de funcionamento do algoritmo, realiza muitas checagens e trocas repetitivas. Apesar de tomar menos tempo que sua contrapartida, o InsertionSort não é eficiente pelo mesmo motivo: múltiplas trocas. Um modo de ordenação que troca sucessivamente os números do vetor gera um excesso de trocas e, com isso, aumento do tempo de execução.

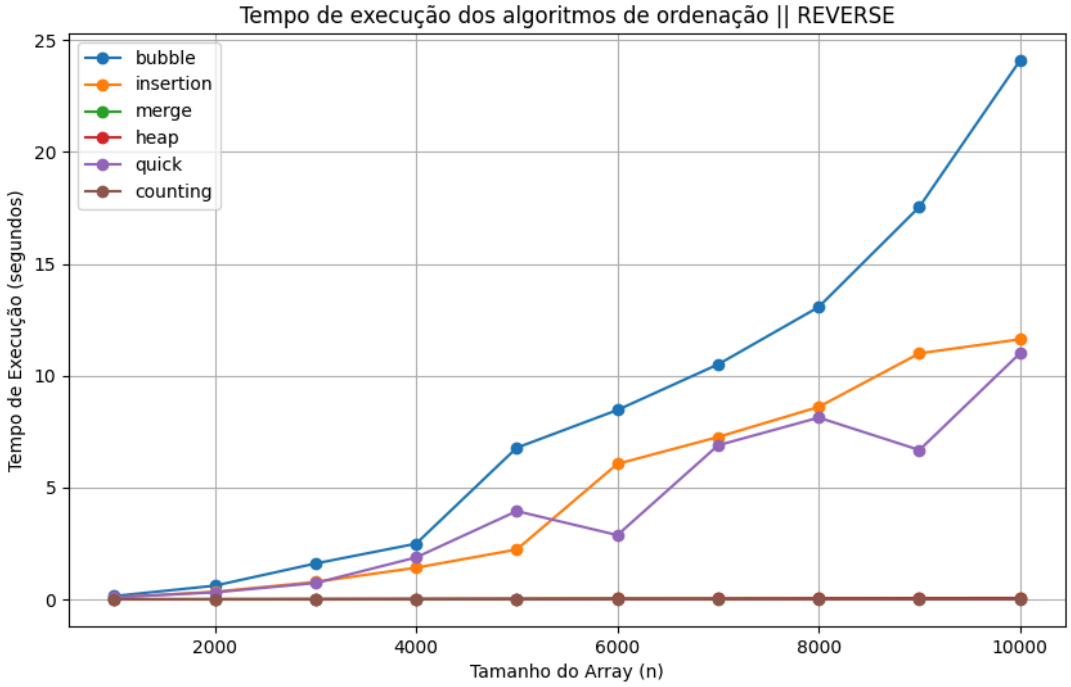


Ademais, nos casos mais eficientes, há uma certa constância nos algoritmos de eficiência média(Heapsort ,Mergesort e Quicksort), porém torna-se claro que o CountingSort seja o mais eficiente. A notação O(n+k) do algoritmo CountingSort torna-o perfeito para casos onde k(valor máximo do vetor) não seja maior que n(tamanho do vetor), por conta da criação do array que contará as repetições necessárias para o método de ordenação Counting. Logo, como mostra o gráfico, o vetor aleatório é ordenado com mais eficiência pelo CountingSort, em média, por conta dos intervalos de 2n.

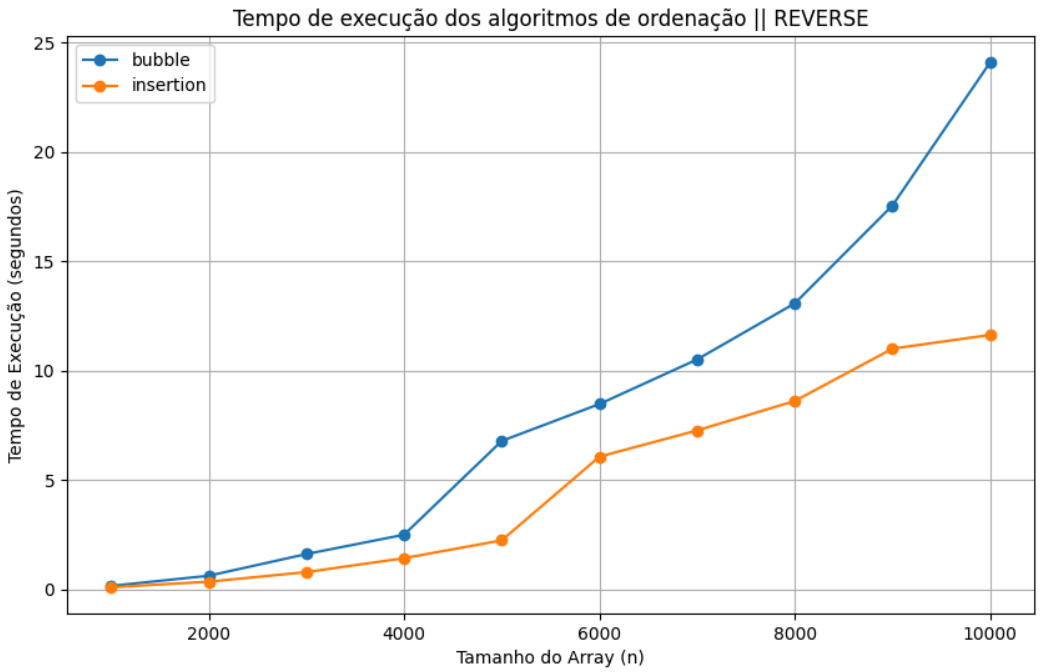


VETOR REVERSO

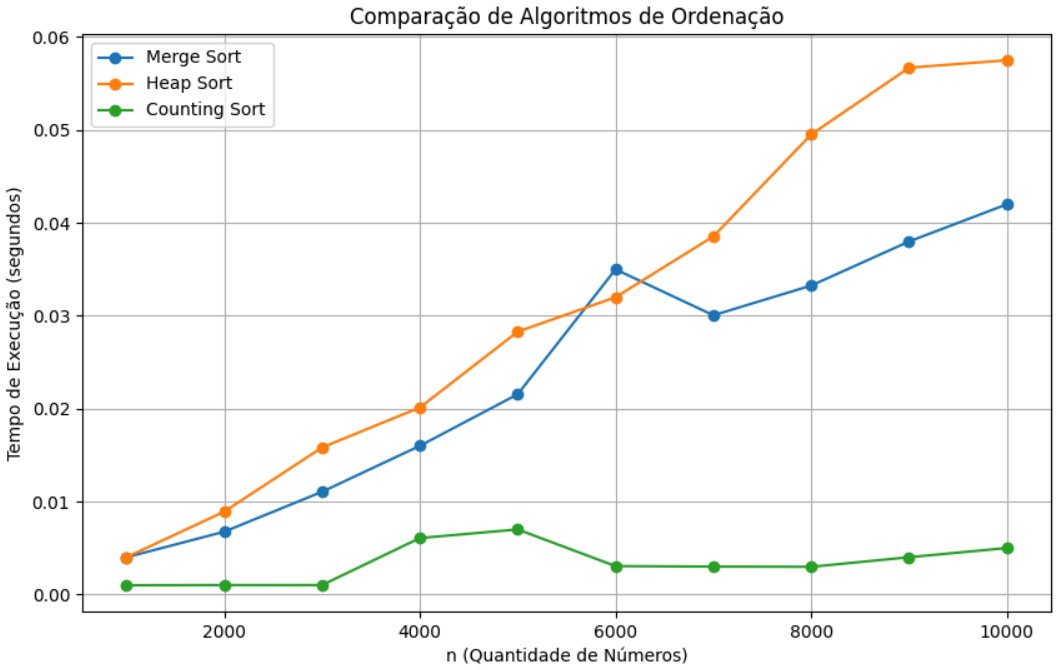
O vetor Reverso(REVERSE) contém n números não negativos escolhidos arranjados em ordem decrescente, respeitando os parâmetros inseridos, ou seja, inicia por n, é seguido por n - i, onde o “i” percorre os valores de 0 até n - 1. Abaixo, observa-se o tempo de execução dos algoritmos de ordenação com o vetor reverso:



Ao analisar o gráfico acima, novamente BubbleSort e InsertionSort tem pouca eficiência, por outro lado o algoritmo QuickSort, que geralmente tem boa rapidez com sua notação O(n log n), em médio e melhor caso, encontra seu pior caso(notação O(n^2), por conta do pivô do vetor.

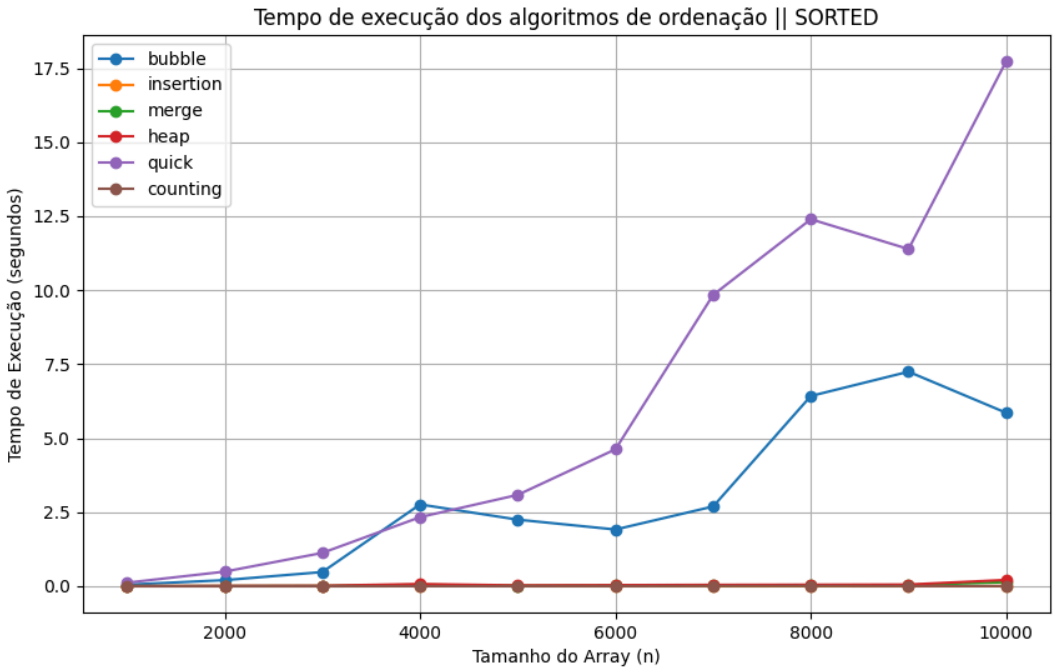


Quanto aos algoritmos eficientes, MergeSort e HeapSort mantêm suas notações O(n log n) independente da ordem inicial do vetor. Ainda assim, o CountingSort mantém como excelente opção por conta do intervalo 2*n.

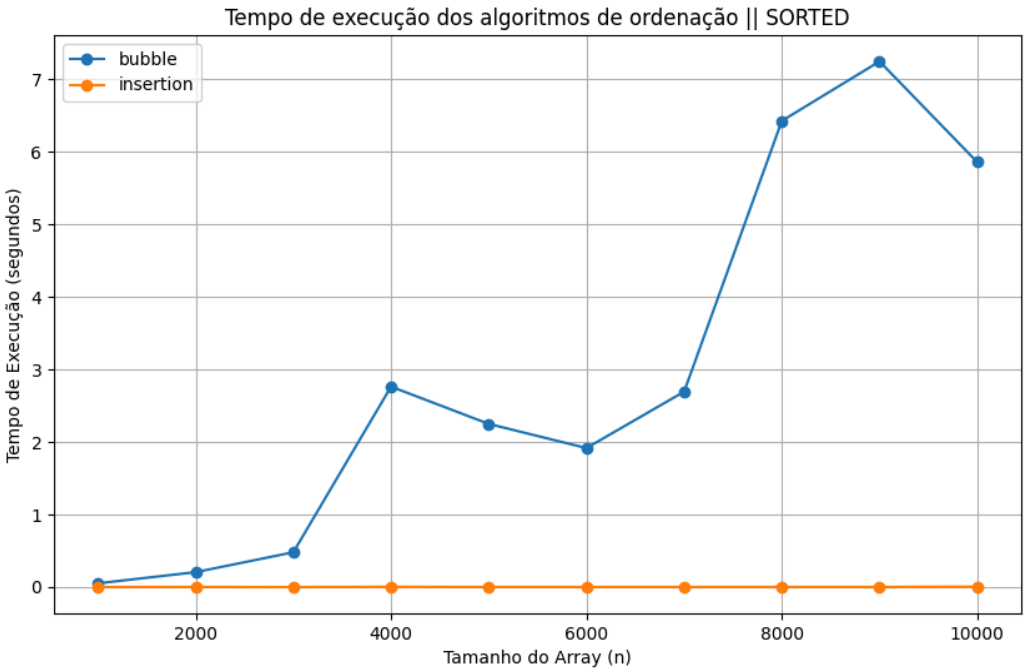


VETOR ORDENADO

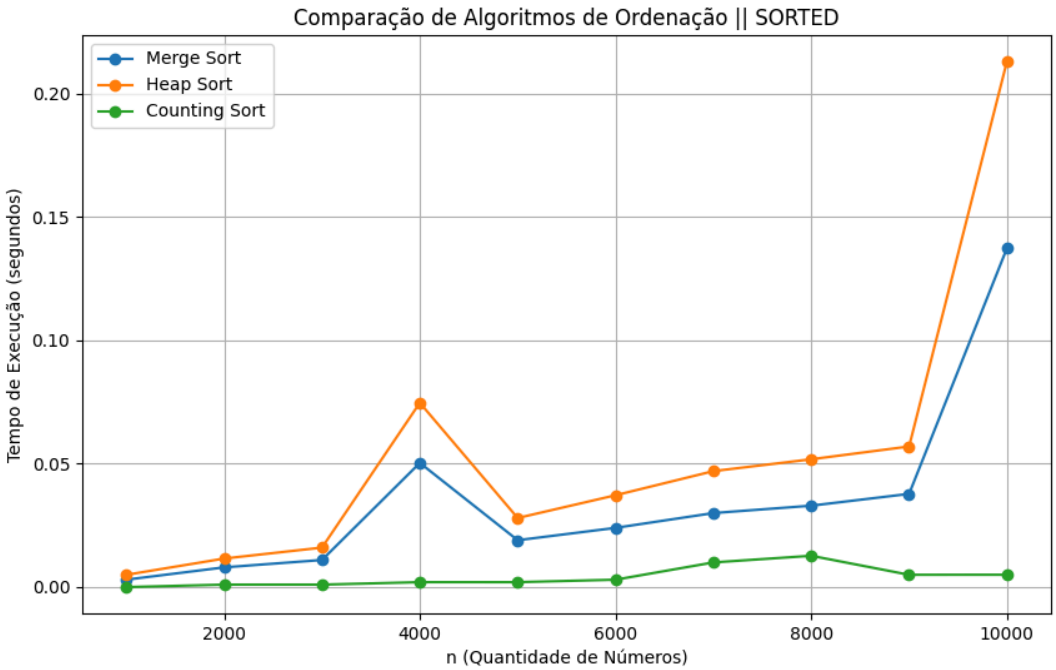
O vetor Ordenado(SORTED) contém n números não negativos escolhidos arranjados em ordem crescente, respeitando os parâmetros inseridos, ou seja, ordenado. Abaixo, observa-se o tempo de execução dos algoritmos de ordenação com o vetor ordenado:



Neste caso, surpreendentemente, InsertionSort torna-se especialmente eficiente, pois não há necessidade de trocas, então o algoritmo navega pelo vetor apenas uma vez e consegue devolvê-lo em tempo recorde. Quanto ao BubbleSort ainda há várias comparações desnecessárias, inflando o tempo de execução.

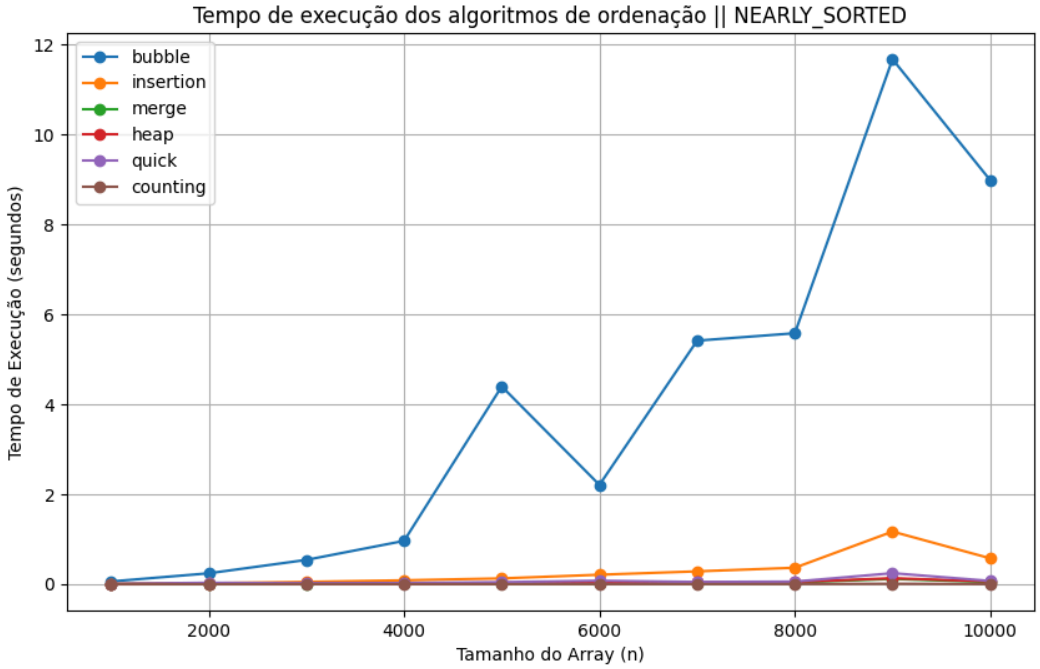


Aqui, por conta da natureza desses algoritmos de sempre realizar trocas, o InsertionSort torna-se o mais eficiente, por conta do comportamento de verificação do algoritmo, só trocando quando há necessidade, mesmo que geralmente os outros algoritmos abaixo sejam mais velozes.

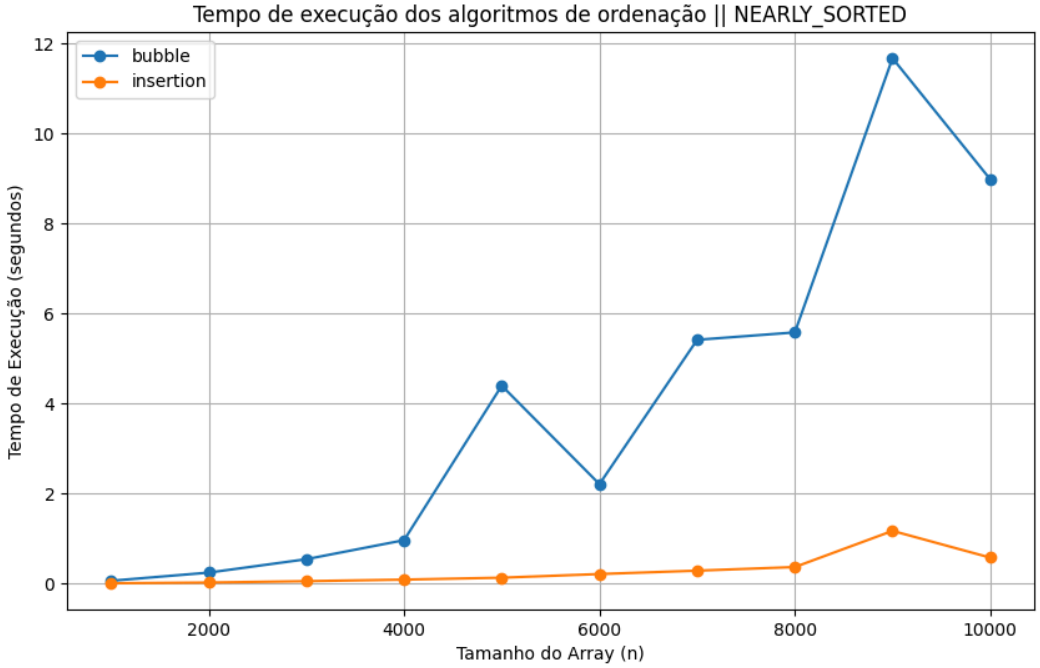


VETOR QUASE ORDENADO

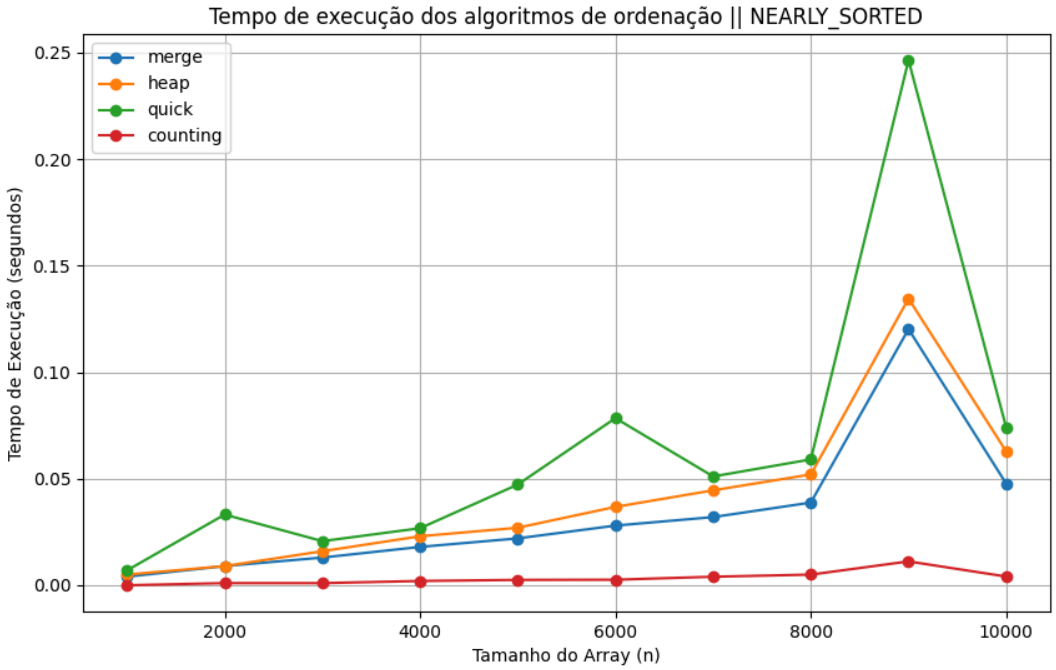
O vetor Quase Ordenado(NEARLY) contém n números não negativos escolhidos arranjados em ordem, porém ao finalizar a ordenação 10% daquele vetor é embaralhado. Abaixo, observa-se o tempo de execução dos algoritmos de ordenação com o vetor quase ordenado:



Nota-se que os resultados são semelhantes ao vetor SORTED, porém o método Insertion perde eficiência pois ainda há 10% dos valores para ordenar.



Aqui torna-se claro que os algoritmos de eficiência contínua podem superar o InsertionSort, com exceção do Quicksort, por conta desse embaralhamento de 10%. Novamente, a superioridade do CountingSort é provada por conta do intervalo constante menor à k.



RESUMO COMPARATIVO

Algoritmos	Vetor Aleatório	Vetor Reverso	Vetor Ordenado	Vetor Quase Ordenado
BubbleSort	Ineficiente ($O(n^2)$)	Pior caso ($O(n^2)$)	Melhor caso ($O(n)$)	Pior caso ($O(n^2)$)
InsertionSort	Ineficiente ($O(n^2)$)	Pior caso ($O(n^2)$)	Melhor caso ($O(n)$)	Eficiente ($O(n)$)
Mergesort	Eficiente ($O(n \log n)$)	Eficiente ($O(n \log n)$)	Eficiente ($O(n \log n)$)	Eficiente ($O(n \log n)$)
HeapSort	Eficiente ($O(n \log n)$)	Eficiente ($O(n \log n)$)	Eficiente ($O(n \log n)$)	Eficiente ($O(n \log n)$)
QuickSort	Eficiente ($O(n + k)$)	Pior caso ($O(n^2)$)	Pior caso ($O(n^2)$)	Depende do pivô ($O(n \log n)$)
CountingSort	Muito eficiente ($O(n \log n)$)	Muito eficiente ($O(n \log n)$)	Muito eficiente ($O(n \log n)$)	Muito eficiente ($O(n \log n)$)

CONCLUSÃO

Neste relatório, foi analisada a eficiência de diversos algoritmos de ordenação(BubbleSort, InsertionSort, MergeSort, HeapSort, QuickSort e CountingSort) em quatro tipos distintos de vetores: Aleatório, Reverso, Ordenado e Quase Ordenado.

Os algoritmos de complexidade O(n^2), como BubbleSort e InsertionSort, mostraram-se ineficazes nos cenários Aleatório e Reverso, devido à alta quantidade de comparações e trocas necessárias. No entanto, o InsertionSort destacou-se no vetor Ordenado, onde sua execução mostrou-se altamente eficiente, reduzindo-se para O(n) devido à ausência de trocas.

O QuickSort, conhecido por seu bom desempenho médio com complexidade O(nlogn), apresentou desafios em casos como o vetor Reverso, onde encontrou seu pior caso O(n^2) devido à escolha ineficiente de pivôs.

MergeSort e HeapSort mantiveram-se consistentemente eficientes com complexidade O(n log n), independentemente da ordenação inicial do vetor. O CountingSort foi o algoritmo mais eficiente em quase todos os casos testados, graças à sua complexidade O(n+k), que o favorece em cenários com vetores inteiros e intervalos controlados.

De maneira geral, os resultados obtidos comprovam que não existe um algoritmo que seja superior em todos os casos, sendo necessário avaliar a natureza do conjunto de dados para escolher a solução de ordenação mais adequada. Outrossim, os intervalos constantes(2n) podem levar a crer que este seja o melhor algoritmo de ordenação independente do caso, porém é necessário salientar que, em caso de k(maior valor naquele vetor) ser muito maior que n(tamanho do vetor), o tempo de execução de CountingSort pode crescer de maneira desproporcional. Pode até mesmo tornar um array de 10 elementos, que tem variação de um a um milhão, algo custoso por conta do uso excessivo da memória para comportar o valor máximo de k.

TABELAS UTILIZADAS

||RANDOM||

n	bubble	insertion	merge	heap	quick	counting
1000	0.10630858	0.04295244	0.00419509	0.00452449	0.00267532	0.00090032
2000	0.41789534	0.17234459	0.00898132	0.00964499	0.00650694	0.00099058
3000	1.31785121	0.56757782	0.02012296	0.02258875	0.01434360	0.00226204
4000	1.86220529	0.82660489	0.02296298	0.02672775	0.01688406	0.00241117
5000	3.67662578	1.19681892	0.02948070	0.03414125	0.02259402	0.00331872
6000	4.60538690	2.18571837	0.02975073	0.03394446	0.02401130	0.00286832
7000	6.98343935	3.05127456	0.05849307	0.06897380	0.05070906	0.00623124
8000	10.14684787	4.43721156	0.06770766	0.08036890	0.05969653	0.00681217
9000	12.37127688	5.27077451	0.07632248	0.08968401	0.06698012	0.00813649
10000	16.77790241	6.45306785	0.06926348	0.07989874	0.06248977	0.00687745

||REVERSE||

n	bubble	insertion	merge	heap	quick	counting
1000	0.14638042	0.08577347	0.00400019	0.00399852	0.08981705	0.00099230
2000	0.61107779	0.33844185	0.00675225	0.00891042	0.31345844	0.00101256
3000	1.60297465	0.77833533	0.01104975	0.01582336	0.72885251	0.00100899
4000	2.48801398	1.41510534	0.01599836	0.02012444	1.87447548	0.00607300
5000	6.77574706	2.22929955	0.02155280	0.02829027	3.94505858	0.00700068
6000	8.46406960	6.05644536	0.03499484	0.03199935	2.87299752	0.00305295
7000	10.50309801	7.25701380	0.03005385	0.03855133	6.88669443	0.00300074
8000	13.06905723	8.60330367	0.03325438	0.04952693	8.12409449	0.00297713
9000	17.54746938	10.99714732	0.03799987	0.05670547	6.67479992	0.00401020
10000	24.09313583	11.62353420	0.04200172	0.05750632	11.00420332	0.00499964

||SORTED||

n	bubble	insertion	merge	heap	quick	counting
1000	0.05200338	0.00100017	0.00301242	0.00497794	0.11896110	0.00000000
2000	0.20609283	0.00107884	0.00799751	0.01157331	0.49360919	0.00100565
3000	0.48024535	0.00000000	0.01098943	0.01600790	1.12641978	0.00100207
4000	2.76360703	0.00300932	0.05033803	0.07469416	2.33413935	0.00200438
5000	2.24971819	0.00099993	0.01900077	0.02800202	3.08636642	0.00199962
6000	1.91661024	0.00100040	0.02400613	0.03722715	4.62596965	0.00300026
7000	2.69429851	0.00099611	0.03000689	0.04700565	9.84041071	0.00999999
8000	6.42704129	0.00200891	0.03299904	0.05180979	12.39823461	0.01269484
9000	7.24707937	0.00101542	0.03778625	0.05699968	11.39044380	0.00499153
10000	5.85839319	0.00599909	0.13736010	0.21305728	17.73497319	0.00500655

||NEARLY_SORTED||

n	bubble	insertion	merge	heap	quick	counting
1000	0.06180453	0.00699544	0.00401187	0.00500202	0.00696087	0.00000000
2000	0.24334788	0.02268624	0.00900865	0.00899100	0.03318000	0.00100064
3000	0.54179358	0.05299544	0.01299620	0.01597500	0.02069950	0.00100040
4000	0.96503997	0.08778524	0.01800179	0.02299929	0.02676845	0.00200343
5000	4.40025711	0.13034821	0.02200007	0.02700019	0.04727554	0.00250602
6000	2.21060824	0.21176052	0.02800083	0.03678799	0.07843685	0.00259209
7000	5.41960311	0.28679085	0.03200006	0.04456234	0.05107164	0.00400257
8000	5.58442259	0.36712956	0.03880787	0.05201125	0.05906224	0.00499153
9000	11.67997146	1.17179894	0.12020659	0.13455439	0.24644256	0.01119709
10000	8.98887038	0.57687330	0.04769230	0.06279826	0.07401204	0.00408411