RELATÓRIO – ATIVIDADE 01

Diogo de Lima Menezes, Marcos Vinicius Medeiros

Universidade Federal do Mato Grosso Do Sul
Implementação Algorítmica; T01 – 2023-2

1. Introdução

Algoritmos de ordenação são conjuntos de instruções sistemáticas e critérios utilizados para rearranjar elementos em uma determinada sequência desejada. Estudos relacionados a esses algoritmos remontam aos primórdios da computação. O algoritmo de ordenação por intercalação (*merge sort*), por exemplo, foi detalhado em um relatório de análise de performance produzido no ano de 1948, por Herman Heine Goldstine e John von Neumann [1]. Não obstante, apesar da distância temporal, algoritmos de ordenação, e outros conceitos e produtos, frutos de pesquisas da época, são utilizados até os dias de hoje.

Este relatório tem como objetivo expor e discutir os resultados obtidos a partir de um experimento de comparação de performance entre os algoritmos de ordenação *selection sort*, *insertion sort*, *merge sort*, *heap sort*, *quick sort* e *counting sort.* Onde considerou-se, como principal fator de medida de performance, o tempo médio necessário para a ordenação de diferentes vetores, gerados com distribuição, ordem e tamanhos variados.

2. Metodologia

Para cada tipo de algoritmo de ordenação, realizou-se um conjunto de testes, com o objetivo de obter um tempo médio de execução mais confiável e estável para a comparação entre os demais algoritmos. Isso porque, apesar de o algoritmo ser o principal fator determinante de performance, circunstâncias ambientais e técnicas difíceis de mensurar ou mitigar podem interferir nos processos de execução. Portanto, a partir da repetição dos testes, com vetores de tamanho variáveis, espera-se que os resultados, embora possam sofrer com alguns vieses causados por variáveis externas, sejam, em média, precisos e úteis para a tarefa de comparação entre os algoritmos.

Utilizou-se quatro tipos diferentes de ordem de elementos nos vetores, vetores aleatórios, vetores ordenados de forma crescente, vetores ordenados de forma decrescente

e vetores crescentes semi-ordenados. Para cada tipo de vetor, realizou-se a geração de conjuntos de números pseudoaleatórios de tamanho entre ***inc*** (tamanho inicial) e ***fim*** (tamanho final); com um intervalo de ***stp*** entre os tamanhos do vetor. Em seguida, para cada vetor gerado, efetuou-se a chamada de todos os métodos de ordenação. Esse processo foi realizado um número de ***rpt*** vezes, para cada tamanho de vetor. Com exceção dos vetores ordenados crescentemente ou decrescentemente; esses foram testados apenas uma vez cada.

O software responsável pela execução dos experimentos, assim como os algoritmos de ordenação, foi realizado a partir da linguagem de programação Python. A aplicação é responsável por receber as entradas de configuração de experimento do usuário, chamar os métodos de ordenação, organizar os dados e apresentá-los ao usuário em forma de tabela impressa no console da aplicação.

3. Resultados de experimentações

Para a realização dos testes, que produziram os resultados demonstrados nas tabelas e gráficos deste relatório, informou-se os seguintes parâmetros de execução:

***Início do vetor**: 2000; **Final do Vetor**: 20000; **Intervalo**: 2000; **Repetições**: 10**;**

A seguir, são apresentados os resultados de execução, em forma de tabela, para cada um dos tipos de vetores. Cada coluna, com exceção da primeira, que indica o tamanho dos vetores utilizados, representa o conjunto de tempos médios de um algoritmo de ordenação. Já nas linhas, encontra-se o conjunto de resultados em relação ao tempo para a ordenação de vetores de tamanho '*n*'.

3.1. RANDOM – Vetores pseudoaleatórios

[[RANDOM]]

n	Selection	Insertion	Merge	Heap	Quick	Counting
2000	0.054431	0.049627	0.003291	0.000474	0.001627	0.000698
4000	0.222293	0.203635	0.007119	0.000966	0.003634	0.001659
6000	0.499742	0.450264	0.011034	0.001465	0.005652	0.002600
8000	0.886544	0.800899	0.015063	0.001911	0.007524	0.003393
10000	1.405431	1.262237	0.019481	0.002402	0.010068	0.004379
12000	2.040065	1.832178	0.023764	0.002895	0.011978	0.005514
14000	2.789841	2.533294	0.027974	0.003404	0.014509	0.006131
16000	3.617994	3.335597	0.032399	0.003918	0.017309	0.006910
18000	4.637029	4.261398	0.037230	0.004369	0.019096	0.008559
20000	5.741525	5.296055	0.041456	0.004889	0.021947	0.008995

3.2. SORTED – Vetores ordenados crescentemente

[[SORTED]]

n	Selection	Insertion	Merge	Heap	Quick	Counting
2000	0.005985	0.000013	0.000295	0.000050	0.013555	0.000064
4000	0.023795	0.000028	0.000666	0.000104	0.055420	0.000156
6000	0.053911	0.000041	0.001036	0.000154	0.125143	0.000220
8000	0.095367	0.000055	0.001384	0.000207	0.224585	0.000303
10000	0.153525	0.000076	0.001788	0.000261	0.356052	0.000396
12000	0.226021	0.000094	0.002199	0.000313	0.521397	0.000451
14000	0.312286	0.000096	0.002610	0.000375	0.715222	0.000560
16000	0.413887	0.000112	0.002967	0.000418	0.943251	0.000643
18000	0.532182	0.000130	0.003394	0.000485	1.204525	0.000718
20000	0.666057	0.000141	0.003733	0.000524	1.493694	0.000808

3.3. REVERSE – Vetores ordenados decrescentemente

[[REVERSE]]

n	Selection	Insertion	Merge	Heap	Quick	Counting
2000	0.005638	0.009554	0.000298	0.000043	0.008797	0.000067
4000	0.022774	0.039368	0.000654	0.000089	0.035729	0.000146
6000	0.052166	0.090652	0.001017	0.000133	0.081051	0.000245
8000	0.091441	0.159490	0.001381	0.000176	0.143336	0.000329
10000	0.146806	0.255299	0.001788	0.000224	0.227564	0.000443
12000	0.214456	0.376938	0.002173	0.000265	0.334462	0.000485
14000	0.293098	0.513042	0.002569	0.000311	0.460584	0.000535
16000	0.386908	0.682645	0.002983	0.000361	0.611667	0.000622
18000	0.494891	0.870071	0.003361	0.000404	0.772035	0.000662
20000	0.618174	1.098515	0.003753	0.000448	0.972366	0.000745

3.4. NEARLY SORTED – Vetores com dez porcento de elementos desordenados

[[NEARLY SORTED]]

n	Selection	Insertion	Merge	Heap	Quick	Counting
2000	0.054816	0.011859	0.003184	0.000474	0.003199	0.000687
4000	0.222748	0.049243	0.006957	0.000980	0.007596	0.001598
6000	0.498696	0.106968	0.010669	0.001444	0.010443	0.002495
8000	0.887197	0.186786	0.014738	0.001949	0.014885	0.003291
10000	1.402943	0.298595	0.018990	0.002422	0.020747	0.004318
12000	2.041326	0.434172	0.023121	0.002987	0.023446	0.005254
14000	2.800967	0.596970	0.027243	0.003436	0.028725	0.005908
16000	3.663757	0.788180	0.031519	0.003950	0.034414	0.006459
18000	4.668581	0.998338	0.036185	0.004460	0.039219	0.007527
20000	5.795244	1.245232	0.040543	0.004947	0.036969	0.008451

4. Conclusão

Nessa seção, para cada tipo de vetor de entrada (pseudoaleatório, crescente, decrescente e quase ordenado), serão discutidos os resultados apresentados brevemente nas imagens anteriores. Duas imagens são apresentadas em cada subparte dessa seção. A primeira imagem demonstra a comparação, a partir de um gráfico de funções, de todos os

algoritmos de ordenação. Já a segunda imagem é utilizada para corrigir a baixa visibilidade de resultados (da primeira imagem) de algoritmos que apresentaram menor tempo médio de ordenação – que ocorre devido a diferença exponencial de tempo para ordenação entre os algoritmos mais e menos eficientes, conforme o crescimento do tamanho dos vetores utilizados como entrada nos testes.

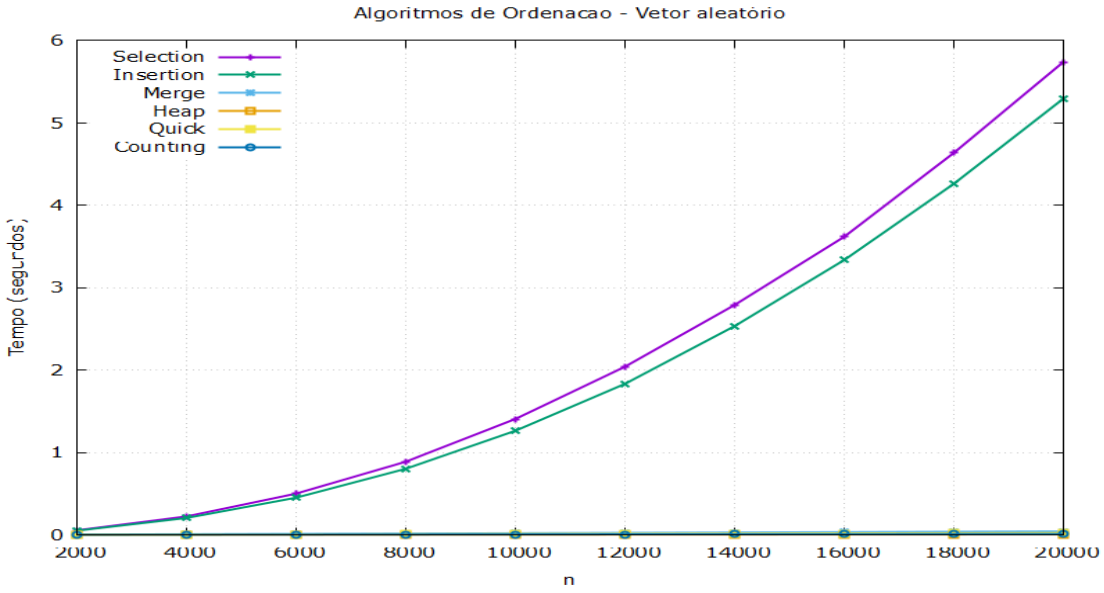
4.1 RANDOM – Vetores pseudoaleatórios

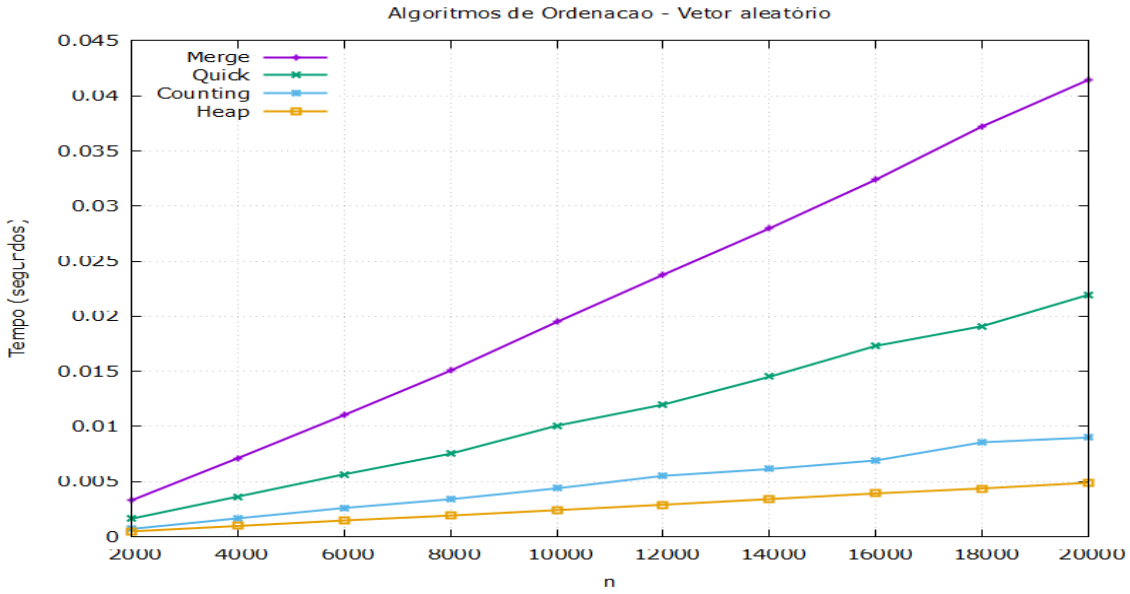
Nos testes com vetores pseudoaleatórios, percebeu-se que esse é o tipo de vetor que mais corrobora com os tempos médios de cada algoritmo. Muito provavelmente devido à distribuição dos valores, qualquer vantagem ou desvantagem que um algoritmo em teste possa apresentar, devido à um intervalo de números que possua determinada ordenação, é (em média) 'balanceada' devido à presença de uma ordenação oposta, ou parte majoritariamente desordenada de números no vetor.

Na primeira imagem apresentada, pode-se perceber que os algoritmos de ordenação que possuem complexidade média quadrática (s*election* e *insertion sort)* desempenham um tempo médio tão exponencialmente maior, em relação às funções com crescimento logarítmico ou linear, que se quer é possível observar os gráficos das demais. Com dois mil elementos, os algoritmos *selection* e *insertion sort* já se mostraram, em média, dez vezes mais lentos que os demais.

Ocorreu, contudo, uma situação não esperada nos resultados de testes dos algoritmos com complexidade média *O(n log(n))* e *O(n).* Como esperado, o *merge* performou de forma ligeiramente pior que o *quick sort,* e o c*ounting sort* apresentou um tempo de ordenação linear – uma vez que o valor máximo dos números pseudoaleatórios foi definido como '*4n'* onde '*n'* representa o tamanho do vetor testado; de forma a preservar o comportamento de crescimento linear do algoritmo, impedindo que os valores máximos sejam exponencialmente maiores que '*n'*. Mas, por algum motivo, o *heap sort* apresentou tempo médio menor que todos os outros algoritmos, inclusive o *counting sort*. Isso era esperado para vetores extremamente pequenos, mas é curioso que esse comportamento tenha persistido em todo o decorrer do gráfico, apesar do crescimento dos vetores de teste.

O que se presume em relação a esse comportamento é que o Python tenha gerado uma diferença linear maior que a esperada, devido à má optimização do processo para alocação de endereços de memória. Dessa forma, o tamanho de vinte mil não foi suficiente para demonstrar a diferença de performance entre os algoritmos *counting sort* e *merge sort.*



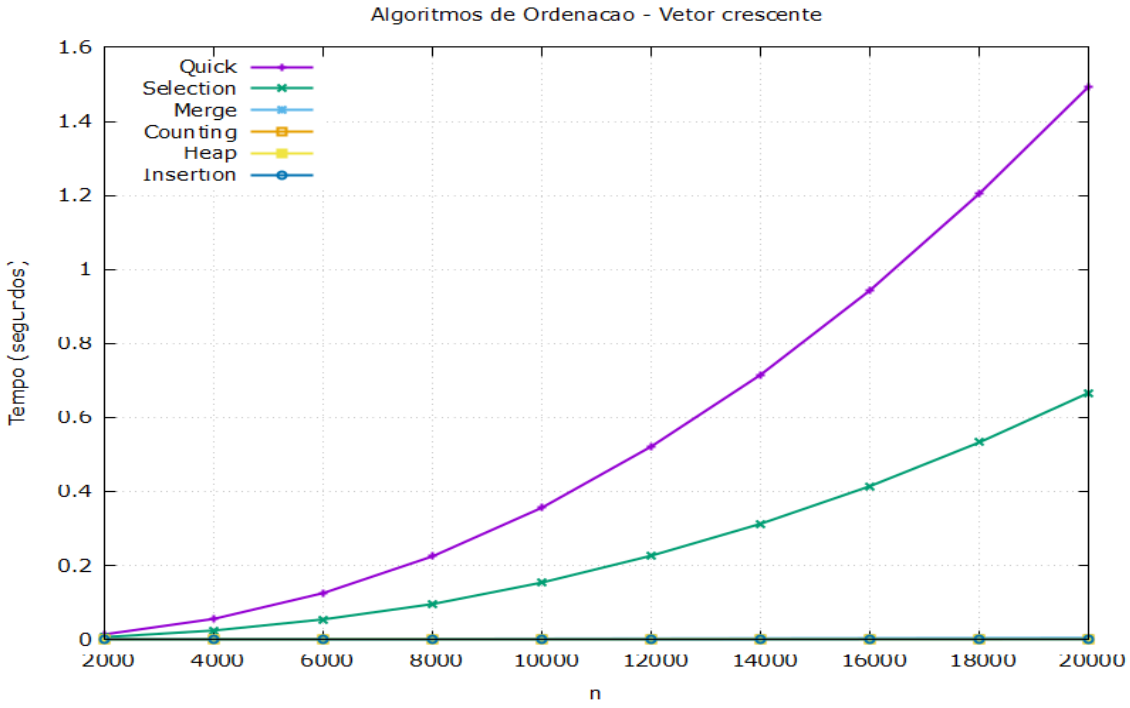


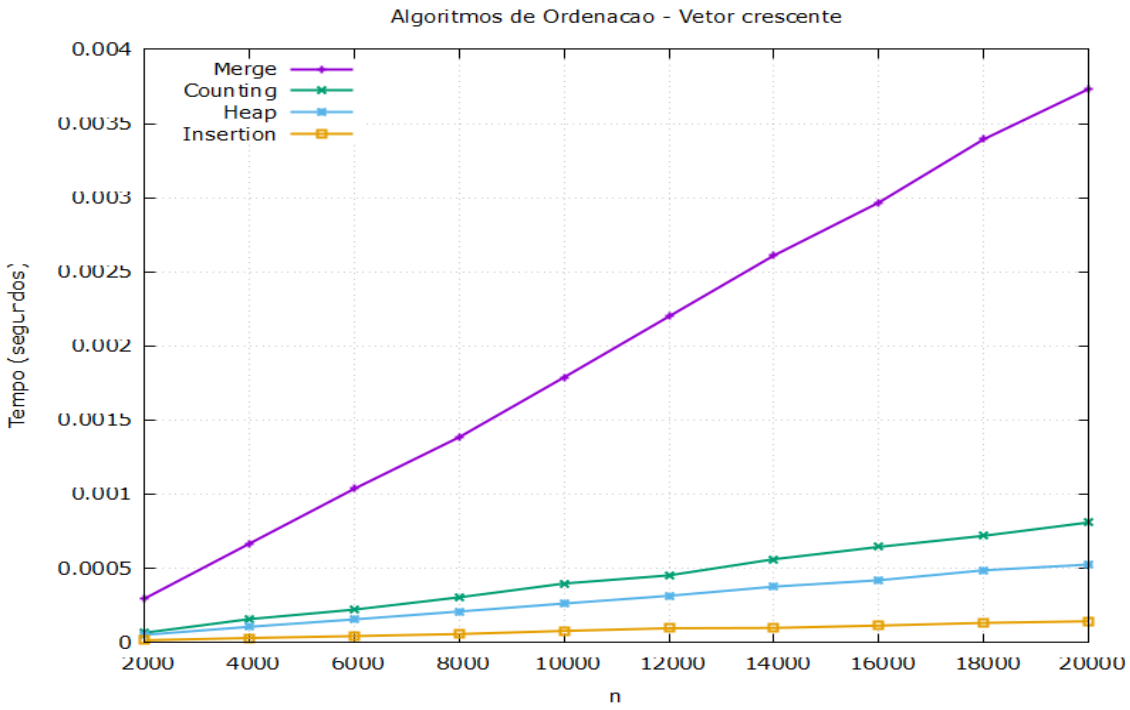
4.2 SORTED – Vetores ordenados crescentemente

Nos testes realizados com vetores ordenados de forma crescente, o *quick sort,* não tendo uma distribuição minimamente desordenada de elementos para realizar partições do vetor, apresentou o pior tempo médio de ordenação entre os algoritmos, com uma complexidade assintótica de $O(n^2)$, ultrapassando até mesmo o *selection sort.*

Já entre os algoritmos que apresentaram tempo médio com crescimento linear ou logarítmico ($O(n\ log(n))$ ou $O(n)$, e, portanto, $o(n^2)$), *merge*, *counting* e *heap sort* permanecem com um comportamento semelhante ao apresentado com vetores aleatórios.

Não obstante, o *insertion sort* se encontrando no melhor caso possível para seu funcionamento, apresenta um tempo médio com crescimento linear ($\Theta(n)$). Diametralmente distante da forma que se comportou nos testes com vetores aleatórios. Isso ocorre por uma verificação realizada pelo algoritmo antes de comparar um elemento *v[j]* com seus antecessores. Caso *v[j-1]* seja menor que *v[j]* ($j \geq 1$) – o que sempre ocorrerá em um vetor ordenado de forma ascendente – o *loop* responsável por percorrer de *j-1* até *0* é interrompido. Como efeito disso, o algoritmo não faz muito mais que percorrer os itens do vetor, e comparar o item corrente com seu antecessor.

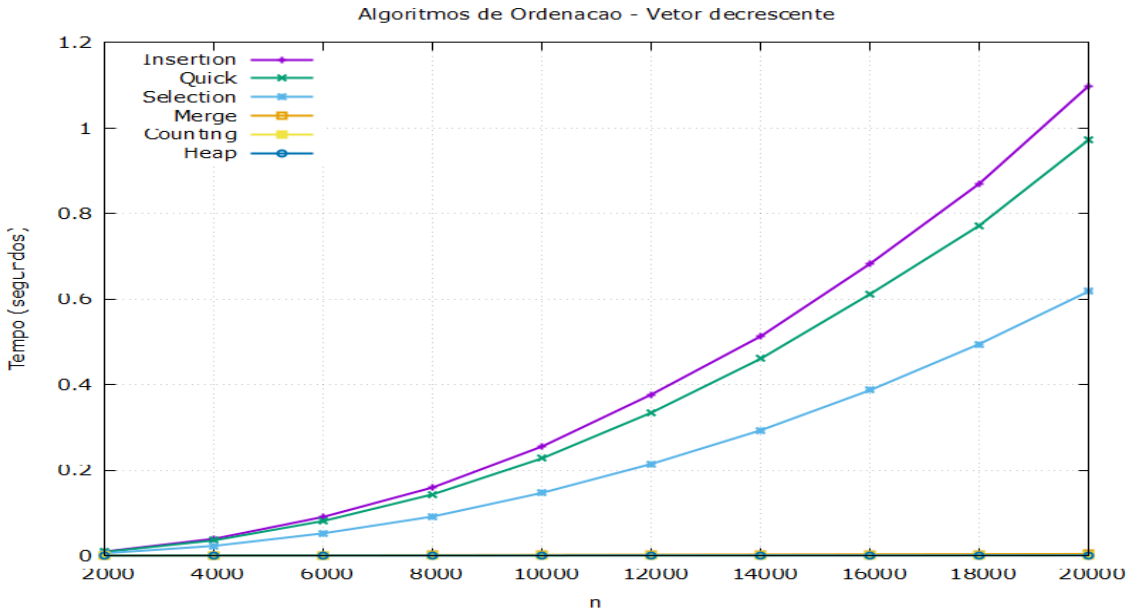


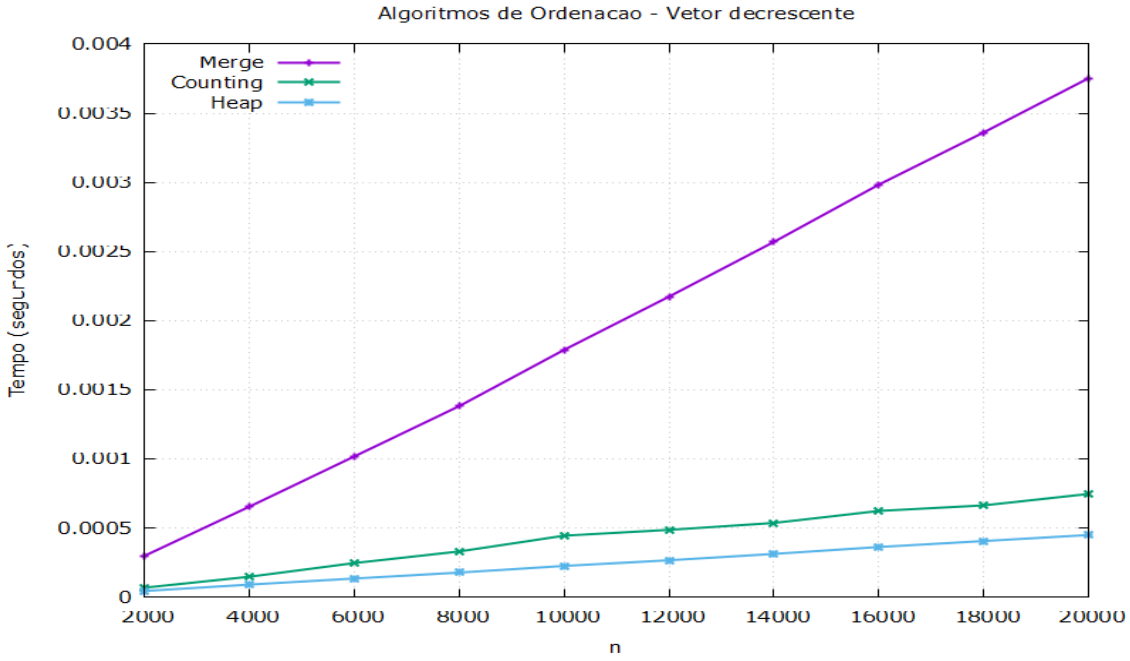


4.3 REVERSE – Vetores ordenados decrescentemente

Ao utilizar vetores ordenados de forma decrescente como entrada dos algoritmos de ordenação, nota-se que todos, com exceção de *insertion sort*, se comportam de maneira muito semelhante ao verificado nos testes com vetores ordenados de maneira crescente. O *selection sort* realizou menos atribuições à variável que armazena a posição do maior elemento do vetor; as partições do *quick* devem estar tendendo à recursão para o lado esquerdo dos vetores; e algumas outras diferenças mínimas devem ter ocorrido em relação aos cenários de experimento com vetores crescentes. Mas a disparidade mais notória em efeito, entre o cenário atual e o anterior, é percebida no desempenho do algoritmo *insertion sort*.

Nota-se que, pelo mesmo mecanismo citado na subseção anterior (4.2), que serviu de grande vantagem para entradas de vetores ordenados, o *insertion sort* passa a apresentar um tempo médio para ordenação com crescimento quadrático ($\Theta(n^2)$). Isso porque, para cada item do vetor reverso, o algoritmo deve agora verificar todos os seus antecessores. Uma vez que, em nenhum caso de verificação, *v[j-m]* será menor que *v[j]* ($j \geq 1;\ 1 \leq m \leq j;\ j-m \geq 0$).



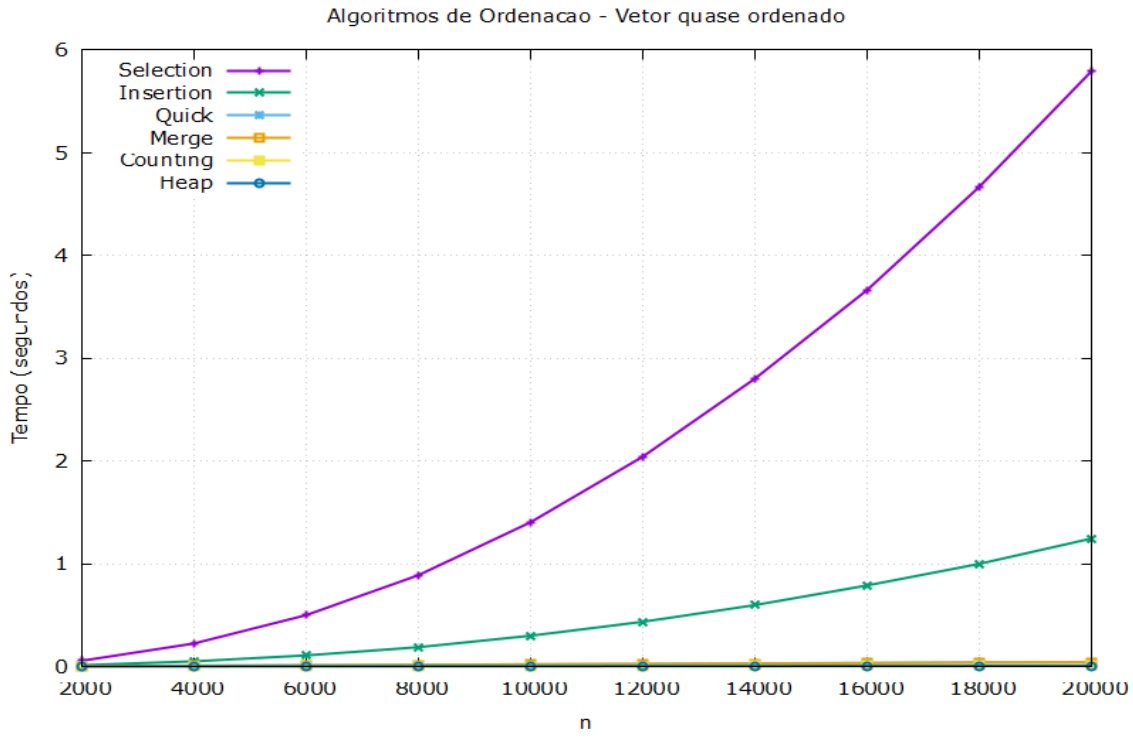


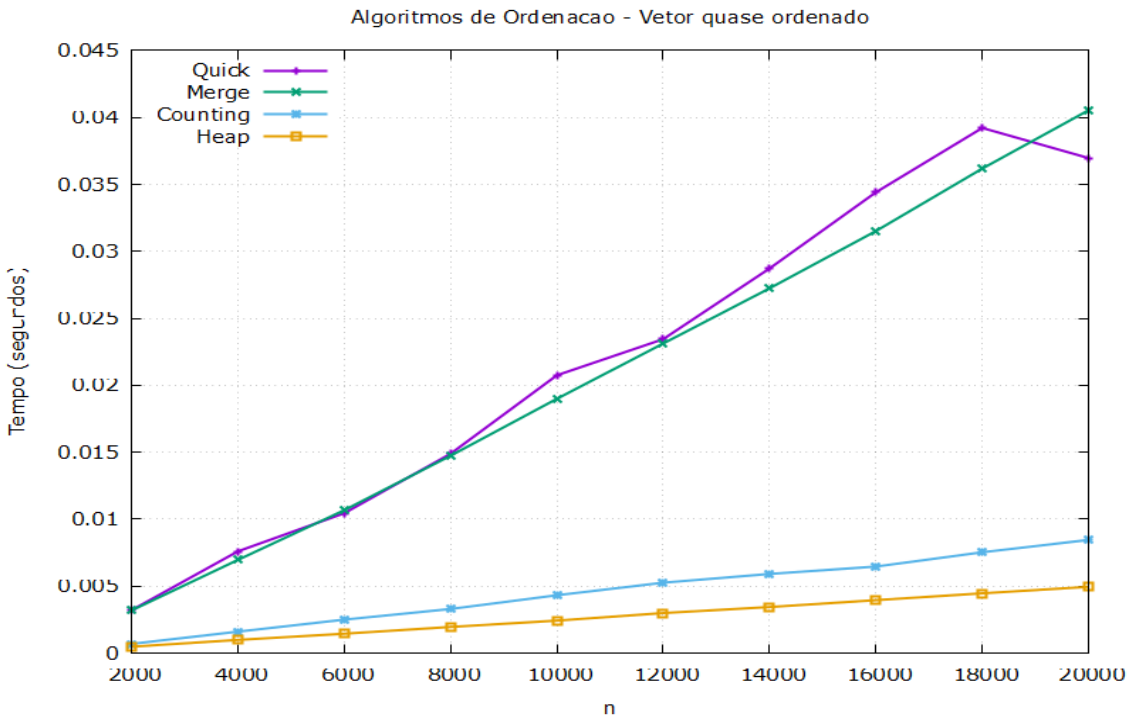
4.4 NEARLY SORTED – Vetores com dez porcento de elementos desordenados

Por fim, os resultados dos experimentos com vetores quase ordenados, os resultados se mostram muito semelhantes aos observados nos casos em que vetores gerados de maneira pseudoaleatória foram utilizados como entrada dos algoritmos. Nota-se, contudo, que por possuir grande parte ordenada de forma ascendente, o *insertion sort* ordenou os itens dos vetores de maneira ligeiramente mais rápida, em comparação ao performado com vetores não ordenados. Não assumindo, contudo, uma função de crescimento linear, como ocorreu com os vetores ordenados de forma crescente.

Também, é importante notar que os dez porcento de 'desordem' no vetor foram suficientes para fazer com que o *insertion sort* apresentasse um crescimento quadrático, e o *quick sort* retornasse a realizar as ordenações, em média, em um tempo com tendência de crescimento logarítmica (O*(n log(n))* – apesar da grande parcela ordenada do vetor (90%).

Curiosamente, o *quick sort* – muito provavelmente devido às diferentes distribuições de 'desordem', em cada vetor, gerados nas repetições de testes – apresenta um gráfico de crescimento um pouco instável. No entanto, exibindo um tempo de execução médio muito semelhante ao do *merge sort*. De forma que o gráfico da função do *merge sort* assemelhe-se em forma com o que se esperaria de uma função de ajuste à apresentada pelo algoritmo *quick sort*.





5. Considerações finais

A partir dos experimentos realizados, foi possível verificar um comportamento e performance mutável, na maioria dos algoritmos, dependendo da ordenação e distribuição dos números presentes nos vetores ao quais eram expostos. A partir dessa observação, torna-se perceptível a importância sobre a escolha do formato dos algoritmos que se escolhe para utilizar em cada cenário, e que não existe um único que atenda, de maneira ótima, todos os tipos de vetores. Havendo, contudo, nos testes realizados, alguns algoritmos com uma maior estabilidade aparente – *merge sort*, *counting sort* e *heap sort*. E mesmos esses apresentam suas limitações e cenários de menor eficiência. O *counting sort*, por exemplo, não seria uma boa escolha – em sua implementação padrão [2] – para ordenar sequências cujo maior elemento seja exponencialmente maior ao tamanho da sequência; muito menos para ordenar elementos numéricos que não sejam inteiros.

A escolha correta de um algoritmo de ordenação, a partir da paciente análise do problema e sequências que se deseja ordenar, é essencial para obter o melhor desempenho possível. Podendo ser a diferença entre um *software* cuja execução perdura por horas do que perdura por segundos.

6. Referências

1. GOLDSTINE, Herman Heine; VON NEUMANN, John. **Planning and coding of problems for an electronic computing instrument**. 1947.

2. KUMAR, Sanjeev et al. **A Novel Counting Sort for Real Numberswith Linear Time Complexity**. In: 2022 International Conference on Computational Intelligence and Sustainable Engineering Solutions (CISES). IEEE, 2022. p. 60-64. DOI: 10.1109/CISES54857.2022.9844325.