

**Rodrigo Isao Goto  
Kauan dos Santos Loche**

# **Relatório Trabalho Prático 1 - PAA**

Presidente Prudente  
2024

# 1 Decisões de Implementação

Dado as especificações do trabalho, foi decidido por implementá-lo em linguagem Python, devido a sua característica de ser interpretada, ou seja, neste contexto requerer vetores menores para a apresentação dos comportamentos, e o extenso catálogo de bibliotecas para a análise e plotagem de dados em gráficos. Foi utilizado a biblioteca Pandas para a geração de "dataframes" e CSVs com os dados dos testes, que foram posteriormente plotados se utilizando da biblioteca Matplotlib. Além disso, também foi se utilizado da biblioteca Timeit para a aferição dos tempos de execução dos algoritmos.

## 2 Metodologia

Todos os testes foram realizados numa máquina com as seguintes especificações de "software" e "hardware":

- Microsoft Windows 11 Pro;
- Python 3.12.7;
- Matplotlib 3.9.2;
- Pandas 2.2.3
- Intel Core i5 1145G7 (4 núcleos físicos, 8 núcleos virtuais, 8MB de cache, 2.6GHz de frequência base e 4.4GHz de frequência máxima);
- 16GB de memória RAM (2x8GB em Dual-channel, 2666MHz de frequência e CL19).

Para a execução dos algoritmos foi-se utilizado três tipos de vetores, um ordenado de maneira crescente, outro de maneira decrescente e por último um de maneira aleatória, se utilizando da função shuffle da biblioteca Random para embaralhar um vetor previamente ordenado de maneira crescente. Isto implica que cada algoritmo foi executado três vezes, à exceção do Shell sort, que foi executado se utilizando dos dois teoremas, contabilizando 6 execuções. Em cada uma das execuções com tipos de vetores diferentes, se é executado o algoritmo 10 vezes com tamanhos de vetores diferentes, que consistem em (1000, 3000, 6000, 10000, 15000, 21000, 28000, 36000, 45000 e 55000).

Em cada uma das execuções foi gerado um dataframe, posteriormente convertido num arquivo CSV, além de um gráfico de linha em formato PNG a partir do CSV.

Os dados salvos foram o nome do algoritmo, o tipo de vetor na entrada e o tempo de execução em segundos. No gráfico, pensando num sistema cartesiano bidimensional, se converte no eixo X representando o tamanho do vetor e o eixo Y o tempo de execução, além da linha representando o comportamento.

## 3 Resultados

### 3.1 Bubble Sort

#### 3.1.1 Bubble Sort Sem Melhoria

Como esperado, se nota sempre o comportamento quadrático, havendo grandes discrepâncias em tempos de execução para os vetores ordenados crescentemente devido a não necessidade de realização de trocas.

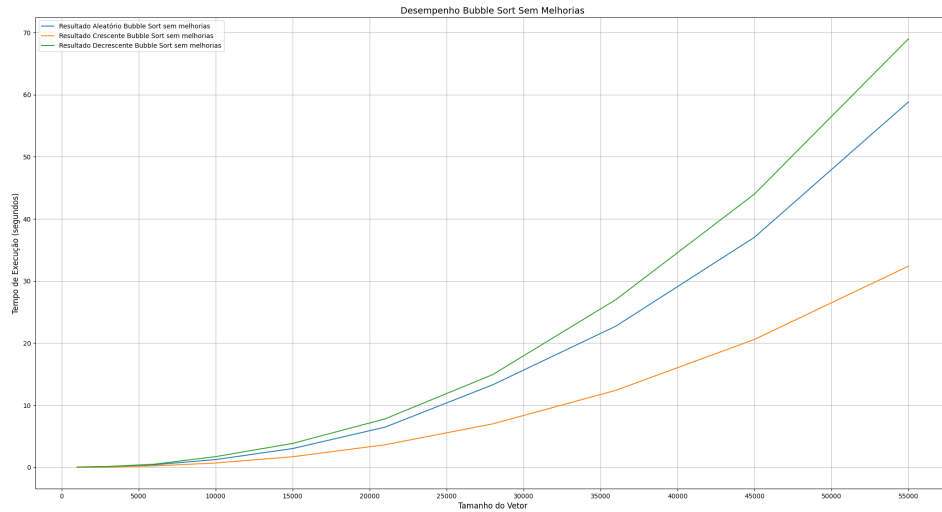


Figure 1: Resultados Bubble Sort Sem Melhoria

### 3.1.2 Bubble Sort Com Melhoria

Como esperado, continua se notando comportamento  $O(n^2)$  / quadrático para vetores ordenados aleatoriamente e decrescentemente, no entanto para vetores ordenados crescentemente tende ao linear /  $\Omega(n)$ , no entanto mascarado pelos rápidos tempos de execução.

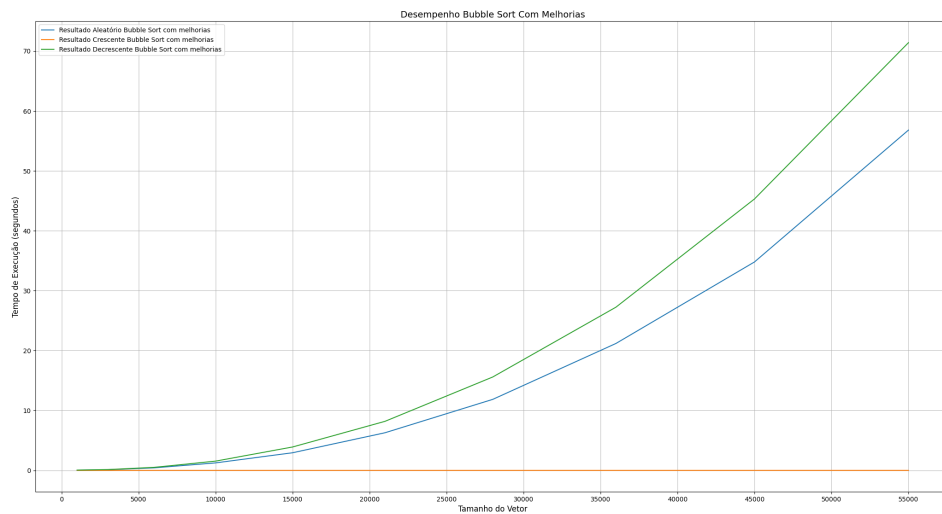


Figure 2: Resultados Bubble Sort Com Melhoria

## 3.2 Quick Sort

### 3.2.1 Quick Sort Com Pivô Elemento Inicial

Como esperado, em vetores ordenados aleatoriamente o comportamento tende a  $\Omega(n \log n)$  e em vetores ordenados crescentemente e decrescentemente apresenta comportamento quadrático /  $O(n^2)$ .

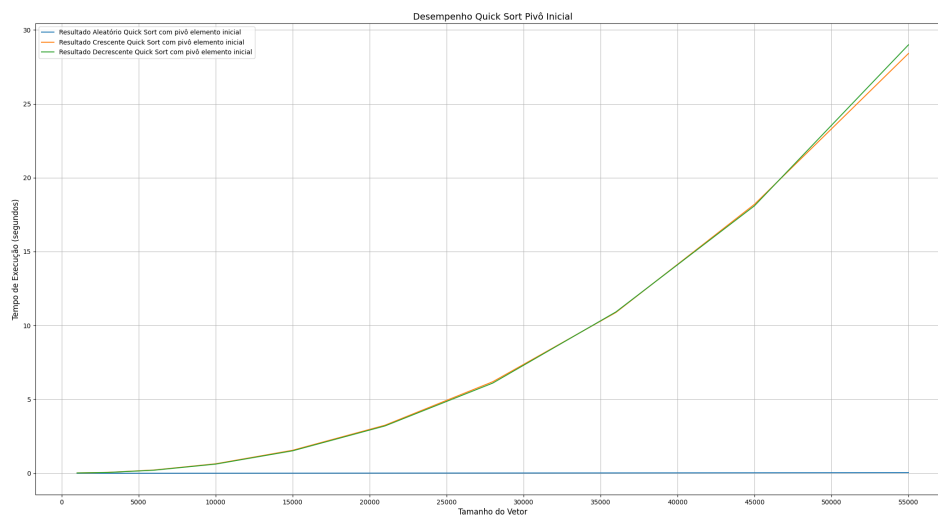


Figure 3: Resultados Quick Sort Com Pivô Elemento Inicial

### 3.2.2 Quick Sort Com Pivô Elemento Central

Como esperado, ao escolher o elemento central como pivô houve uma tendência a  $\Omega(n \log n)$  em todos os tipos de vetores, mascarados pelos rápidos tempos de execução.

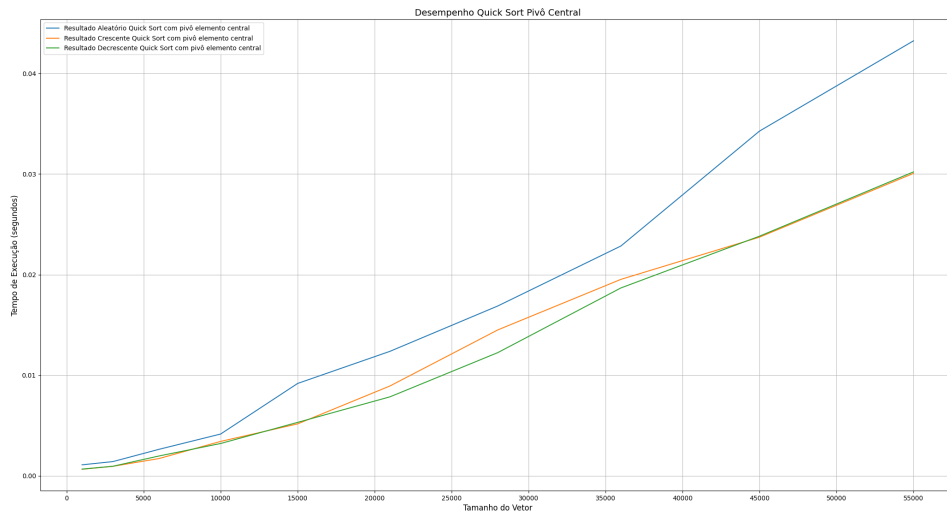


Figure 4: Resultados Quick Sort Com Pivô Elemento Central

### 3.3 Insertion Sort

Como esperado, em vetores ordenados crescentemente houve uma tendência ao linear /  $\Omega(n)$ , mascarado pelos rápidos tempos de execução, e em vetores ordenados aleatoriamente e decrescentemente apresentou comportamento quadrático /  $O(n^2)$ .

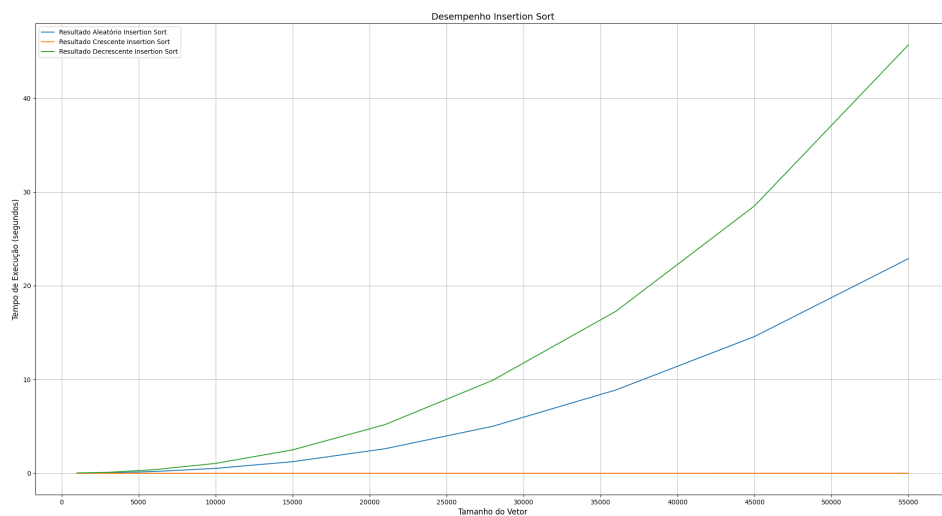


Figure 5: Resultados Insertion Sort

### 3.4 Shell Sort

#### 3.4.1 Shell Sort Teorema 1

Como esperado, em vetores ordenados crescentemente houve uma tendência à complexidade  $\Omega(n \log n)$ , mascarado pelos rápidos tempos de execução. Contudo, em vetores ordenados aleatoriamente e decrescentemente, houve uma piora nos tempos de execução, contrariando as complexidades quadrática /  $O(n^2)$  para o Insertion Sort e  $O(n^{3/2})$  para o Shell Sort se utilizando do Teorema 1  $2^k - 1$  de espaçamentos. Isto se deve aos tamanhos dos vetores, que não possibilitaram a extração do potencial dos espaçamentos.

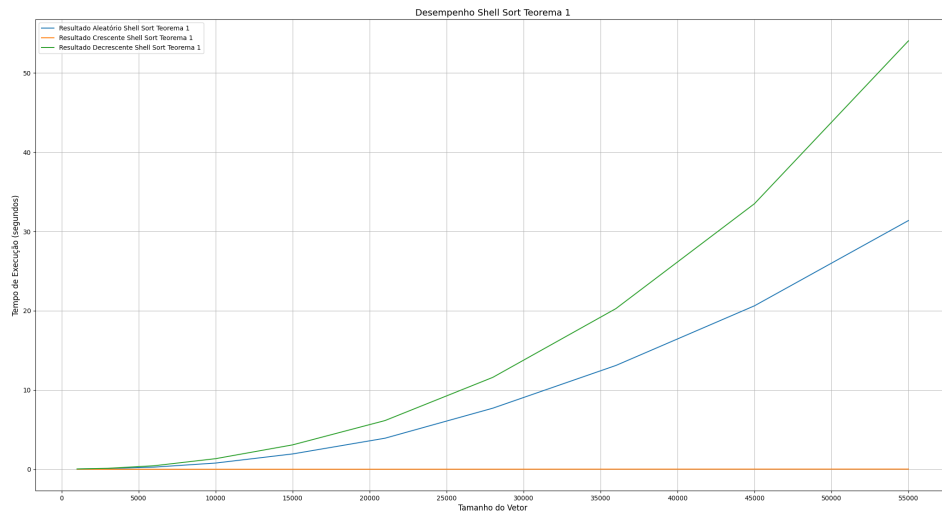


Figure 6: Resultados Shell Sort Teorema 1

#### 3.4.2 Shell Sort Teorema 2

Assim como o Shell Sort se utilizando do Teorema 1, o Shell Sort se utilizando do Teorema 2  $2^p 3^q$  de espaçamentos, obteve tempos de execução para vetores ordenados crescentemente tendendo a  $\Omega(n \log n)$ , mascarado pelos rápidos tempos de execução, e piora nos tempos de execução para os vetores ordenados de maneira aleatória e decrescente em comparação ao Insertion Sort, devido novamente aos tamanhos dos vetores impossibilitarem a exploração do potencial dos espaçamentos.

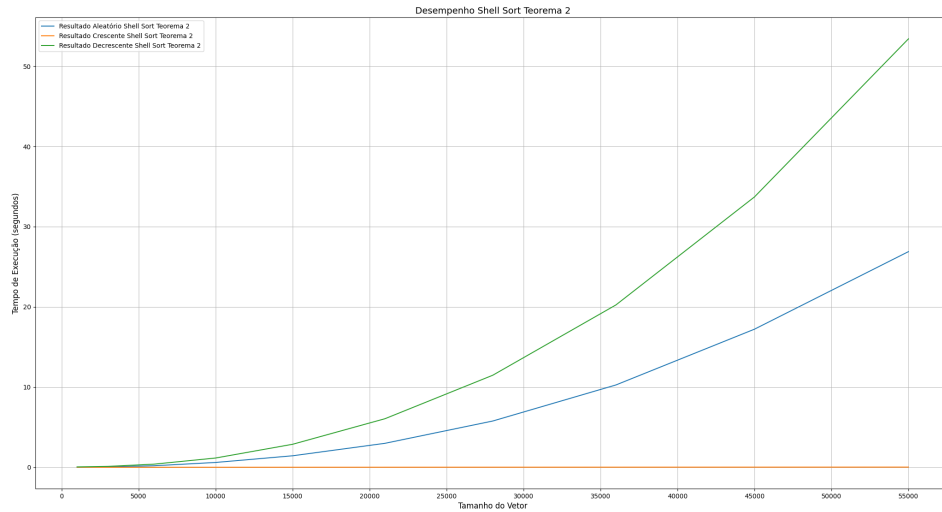


Figure 7: Resultados Shell Sort Teorema 2

### 3.5 Selection Sort

Como esperado, em todos os tipos de vetores o comportamento se mostrou quadrático /  $\Theta(n^2)$ .

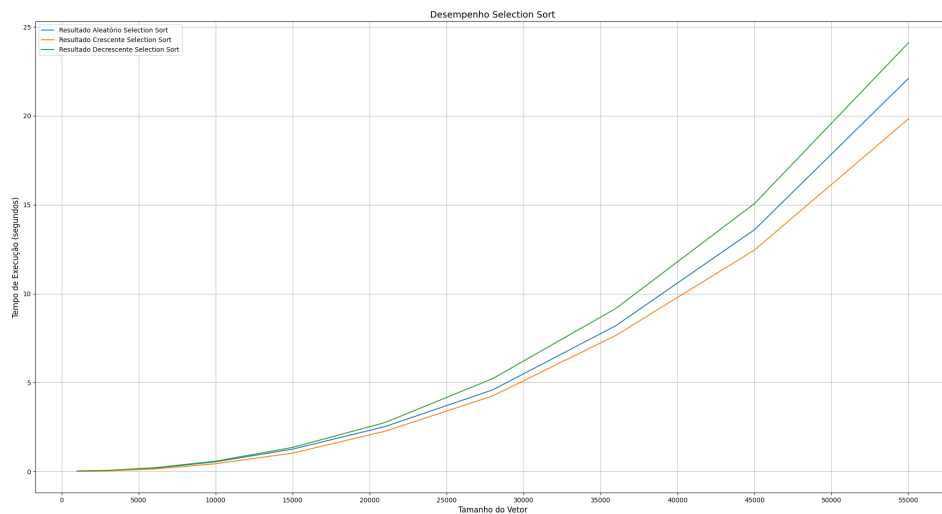


Figure 8: Resultados Selection Sort

### 3.6 Heap Sort

Como esperado, em todos os tipos de vetores o comportamento se mostrou tendendo a  $\Theta(n \log n)$ .

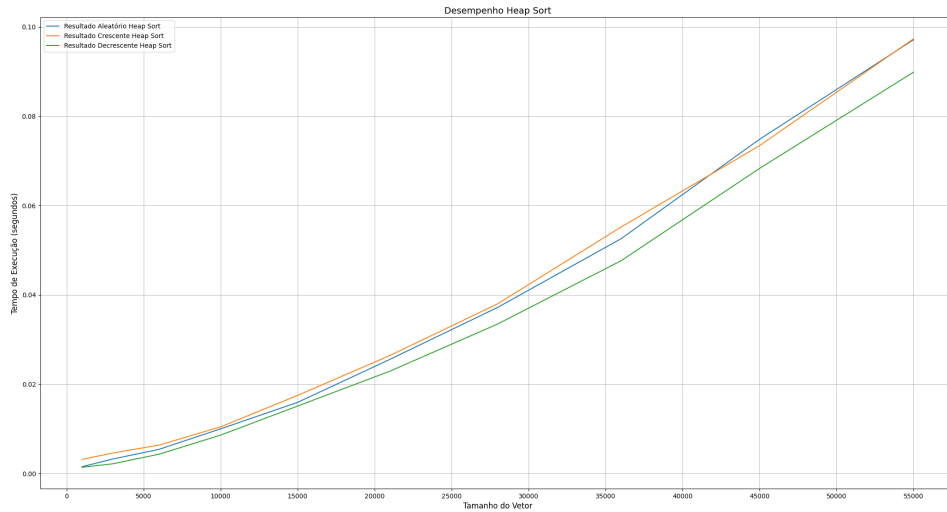


Figure 9: Resultados Heap Sort

### 3.7 Merge Sort

Como esperado, em todos os tipos de vetores o comportamento se mostrou tendendo a  $\Theta(n \log n)$ .

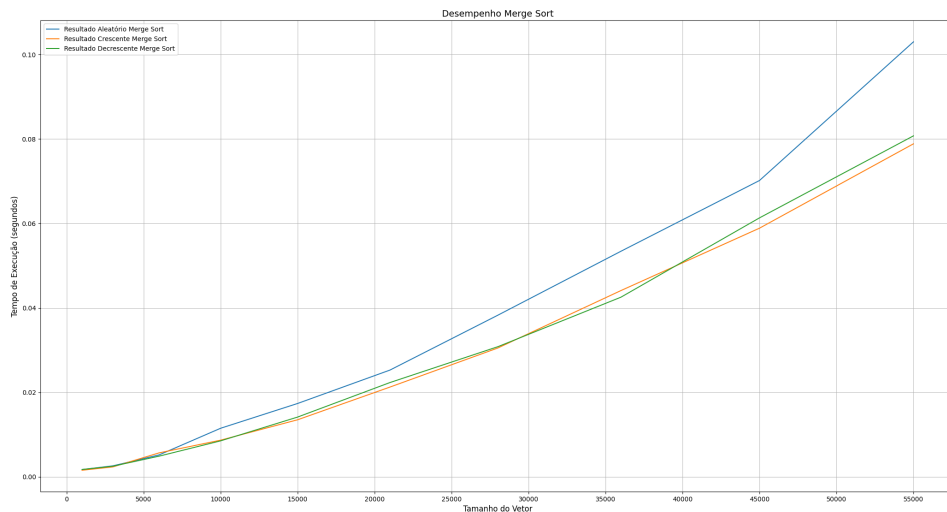


Figure 10: Resultados Merge Sort

### 3.8 Tim Sort (Python)

O Tim Sort, algoritmo de ordenação padrão do Python e um híbrido do Insertion Sort com o Merge Sort, possui como melhor caso a complexidade linear/  $\Omega(n)$  em vetores ordenados ou semi-ordenados, verificado pelos testes, e como pior caso a complexidade  $\Omega(n \log n)$  para vetores ordenados aleatoriamente



e inversamente, também verificados pelos testes. Todos os comportamentos foram mascarados pelos rápidos tempos de execução.

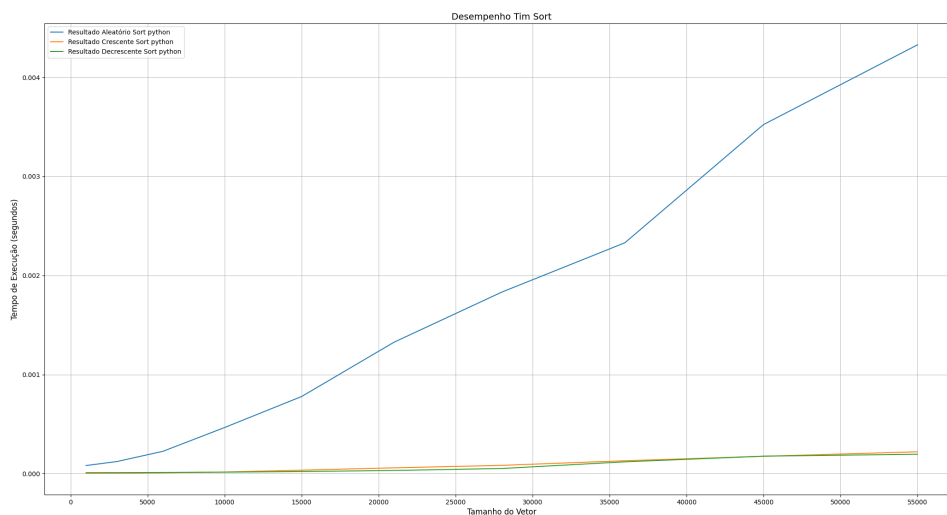


Figure 11: Resultados Tim Sort

## 3.9 Comparações

### 3.9.1 Aleatório

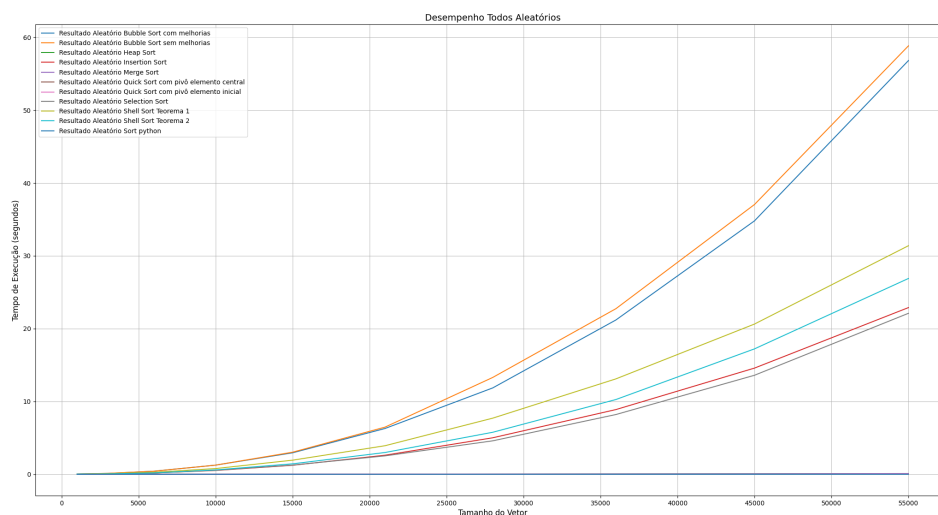


Figure 12: Comparação Todos Aleatórios

Analisando o gráfico, percebe-se um comportamento quadrático para o Bubble Sort com melhorias, Bubble Sort sem melhorias, Insertion Sort e Selection Sort. Logo, esses foram os que tiveram pior de-

sempenho, especialmente o Bubble Sort sem melhorias, por conta de todas as verificações que ele faz em sua estrutura, mesmo quando o vetor já está ordenado. Por outro lado, temos os algoritmos Sort Python, Quick Sort com pivô central, Quick Sort com pivô inicial, Heap Sort e Merge Sort, os quais tiveram melhores desempenhos em relação a tempo de execução. Contudo, como não foi possível observar as diferenças de comportamentos dos desses algoritmos, separamos eles no gráfico a seguir:

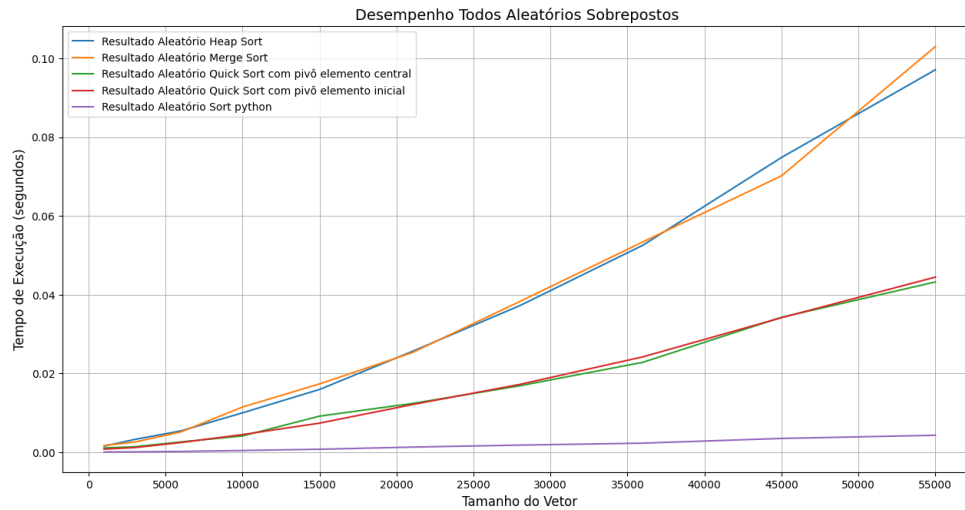


Figure 13: Comparação Todos Aleatórios Sobrepostos

Ao se observar o desempenho dos algoritmos mais rápidos que estavam sob a mesma reta no gráfico anterior, nota-se que os mais lentos foram o Heap Sort e o Merge Sort, seguido pelo Quick Sort com pivô inicial e com pivô central. Assim, dos algoritmos passados em aula, o Quick Sort possui melhor desempenho para vetores aleatórios, não importando o pivô escolhido. Contudo, verifica-se maior velocidade no algoritmo extra adicionado, o Sort Python, que utiliza como base o algoritmo Timsort. Todos esses algoritmos gravitam entre complexidades lineares e  $n \log n$ .

### 3.9.2 Crescente

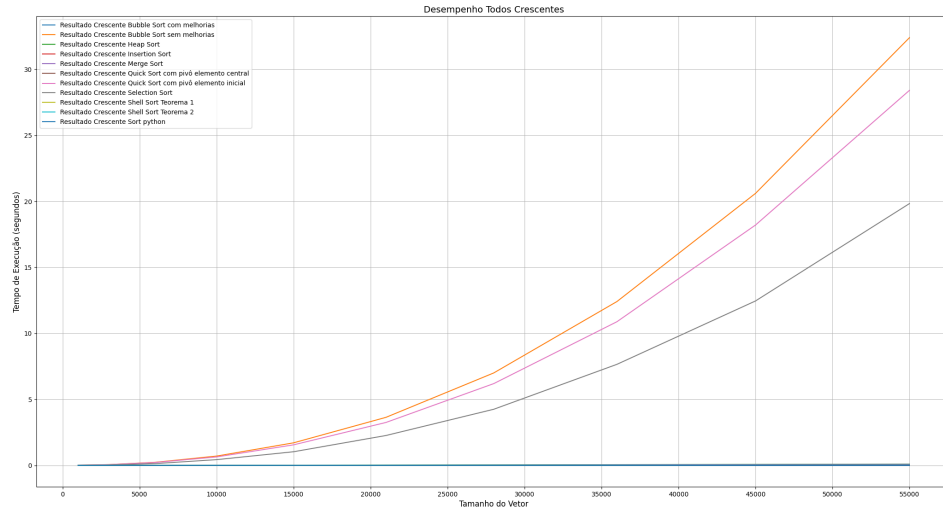


Figure 14: Comparação Todos Crescentes

Observa-se que o Bubble Sort sem melhorias e o Selection Sort continuaram com suas complexidades quadráticas, quando analisados vetores ordenados crescentemente. Contudo, verifica-se uma melhora de desempenho de diversos outros algoritmos, como: Insertion Sort, que teve complexidade quadrática para vetores aleatórios e linear para a observação atual; Bubble Sort com melhorias, apresentou complexidade linear, visto que os vetores já estão ordenados e há a verificação que interrompe o funcionamento do algoritmo; Shell Sort Teorema 1 e Teorema 2, visto que não foi preciso fazer trocas. O Quick Sort com pivô inicial teve piora de desempenho, com complexidade quadrática. Os demais algoritmos se encontram sob complexidades melhores, as quais serão analisadas no gráfico separado abaixo:

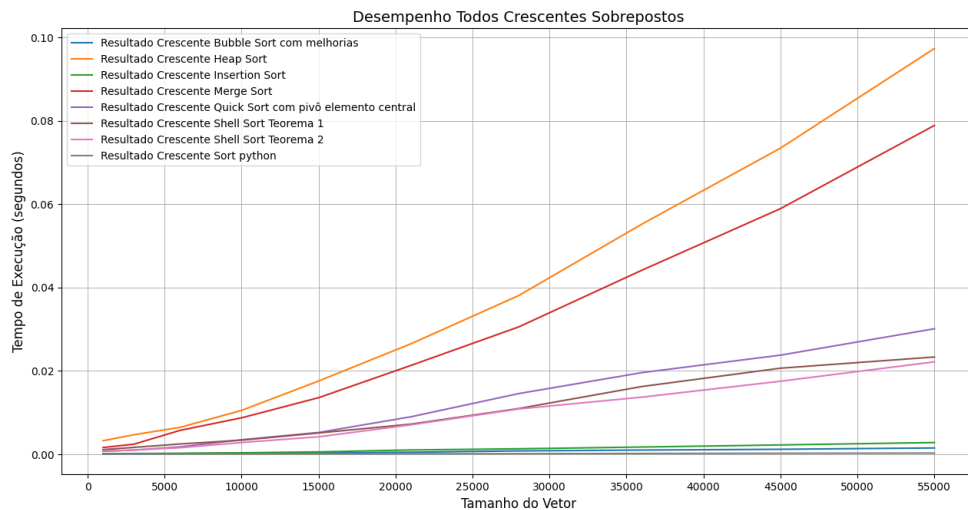


Figure 15: Comparação Todos Crescentes Sobrepostos

Considerando o gráfico de todos os algoritmos com melhores complexidades que ficaram sob uma mesma reta, vê-se que o Insertion Sort e Bubble Sort com melhorias tiveram desempenhos melhores que todos os demais algoritmos apresentados em aulas, pois têm complexidades lineares. Logo após eles, tem-se os algoritmos Shell Sort Teoremas 1 e 2 e Quick Sort com pivô central. As piores complexidades, dentre esses algoritmos mais rápidos para o atual ambiente de entrada, foram do Heap Sort e do Merge Sort. Contudo, mesmo tendo diferenças de tempo de execução entre eles, tratam-se ainda de algoritmos rápidos que possuem a complexidade  $n \log n$ , com exceção do Bubble Sort com melhorias e do Insertion Sort, anteriormente citados, os quais são lineares. Outrossim, novamente o Timsort do Python foi mais veloz que os algoritmos analisados em aula.

### 3.9.3 Decrescente

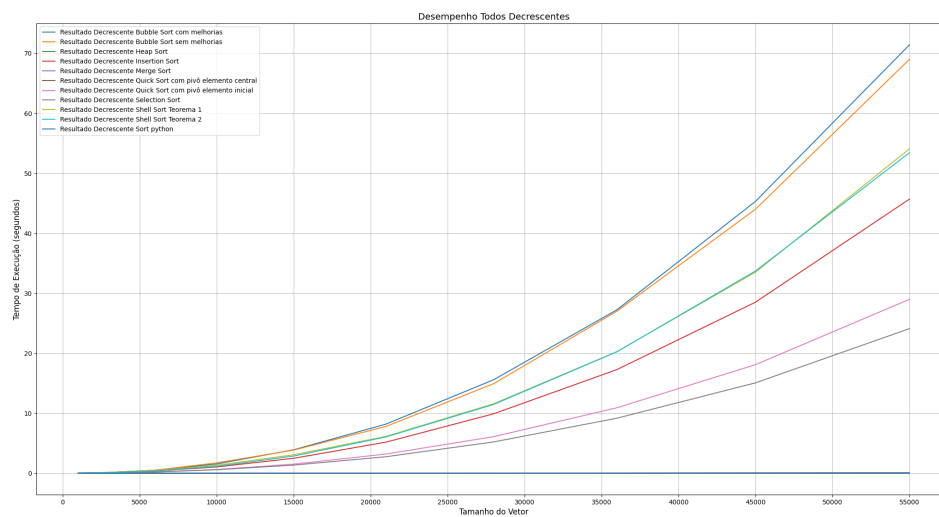


Figure 16: Comparação Todos Decrescentes

Percebe-se que, na análise do gráfico que considera os desempenhos dos algoritmos operando em vetores estruturados de forma decrescente, de forma geral, a maioria dos algoritmos tiveram seu pior desempenho nessas condições. Bubble Sort com e sem melhorias, Insertion Sort, Shell Sort Teoremas 1 e 2 e Quick Sort com pivô inicial tiveram desempenhos quadráticos. Em relação ao Shell Sort, ambos os teoremas, entende-se que esse desempenho não foi bom por conta que os vetores foram de tamanhos insignificantes para a exploração do potencial dos espaçamentos. O Quick Sort com pivô inicial obteve comportamento quadrático, assim como para com vetores crescentes, analisados anteriormente, por conta das divisões desequilibradas que ocorrem quando mediante essas duas condições de contruções dos vetores. O Quick Sort com pivô central, Heap Sort, Merge Sort e TimSort (python) ficaram sob uma mesma linha de complexidade e serão analisados separadamente no gráfico abaixo.

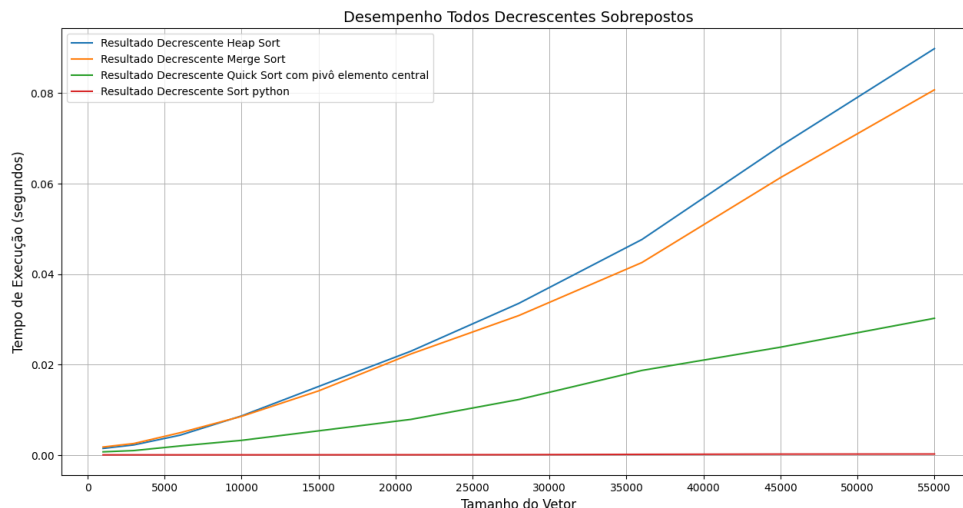


Figure 17: Comparação Todos Decrescentes Sobrepostos

Separados e comparados os algoritmos de melhores desempenhos da análise anterior, nota-se que, por mais que os algoritmos Heap Sort, Merge Sort e Quick Sort com pivô inicial tenham complexidades  $n \log n$ , o Quick se destaca como o mais rápido nessa disputa e o Heap Sort como o mais lento, considerando apenas os algoritmos estudados em aula. Contudo, de forma geral e considerando o algoritmo extra, o TimSort, é indiscutível que esse é o mais rápido em qualquer condição apresentada.

## 4 Conclusão

Como conclusão desta análise de todos os algoritmos apresentados, fica evidente que a escolha de qual usar depende do contexto e das condições do problema, pois mesmos algoritmos, em diversas partes deste relatório, obtiveram diferentes resultados para diferentes estruturas de vetores. Contudo, evidenciou-se também que, dentre os algoritmos estudados em aula, o Quick Sort com pivô central, o Merge Sort e o Heap Sort obtiveram bons desempenhos para os três tipos de vetores utilizados, enquanto que o Bubble Sort sem melhorias e o Selection Sort apresentaram em todas as análises desempenhos quadráticos. Em relação ao Sort Python (Timsort), em parte das análises ele foi deixado um pouco de lado pois, por mais que seja inegável o seu esplêndido desempenho, se trata de uma ferramenta nativa da linguagem Python e foi utilizado apenas como um algoritmo extra.

## 5 Referências Bibliográficas

- Slides do Professor Dr. Danilo Eler;
- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; (2002). Algoritmos –Teoria e Prática. Tradução da 2ª edição americana. Rio de Janeiro. Editora Campus;
- TAMASSIA, ROBERTO; GOODRICH, MICHAEL T. (2004). Projeto de Algoritmos -Fundamentos, Análise e Exemplos da Internet;
- ZIVIANI, N. (2007). Projeto e Algoritmos com implementações em Java e C++. São Paulo. Editora Thomson;
- BUBBLE Sort Algorithm. [S. l.], 6 out. 2024. Disponível em: <https://www.geeksforgeeks.org/bubble-sort-algorithm/>. Acesso em: 21 set. 2024;