**Relatório Técnico - Comparação de Algoritmos de Ordenação**

[**https://github.com/mateusdfaria/comparacao-algoritmos-ordenacao**](https://github.com/mateusdfaria/comparacao-algoritmos-ordenacao)

**Renan Iomes, Eduardo Klug, Mateus Mautone, Mateus de Faria e Leonardo Rocha**

# **1. Código-Fonte Documentado e Organizado**

O código-fonte foi desenvolvido em Python e estruturado em dois arquivos principais:

## **sorting.py**

Contém a implementação do padrão Strategy, com classes para diferentes algoritmos de ordenação:

* Bubble Sort
* Bubble Sort Otimizado
* Insertion Sort
* Selection Sort
* Quick Sort
* Merge Sort
* Heap Sort
* Tim Sort

### **Exemplo de Documentação na Classe TimSort:**

class TimSort(SortingStrategy):

def sort(self, data):

"""Implementa o algoritmo Tim Sort, híbrido de Merge Sort e Insertion Sort.

Divide a lista em blocos (run) de tamanho 32, ordena com Insertion Sort

e combina com Merge Sort. Otimizado para dados parcialmente ordenados.

"""

def insertion\_sort(arr, left, right):

for i in range(left + 1, right + 1):

key = arr[i]

j = i - 1

while j >= left and arr[j] > key:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

def merge(arr, left, mid, right):

left\_arr = arr[left:mid + 1]

right\_arr = arr[mid + 1:right + 1]

i = j = 0

k = left

while i < len(left\_arr) and j < len(right\_arr):

if left\_arr[i] <= right\_arr[j]:

arr[k] = left\_arr[i]

i += 1

else:

arr[k] = right\_arr[j]

j += 1

k += 1

def tim\_sort(arr):

n = len(arr)

RUN = 32

for i in range(0, n, RUN):

insertion\_sort(arr, i, min((i + RUN - 1), (n - 1)))

size = RUN

while size < n:

for left in range(0, n, size \* 2):

mid = left + size - 1

right = min((left + size \* 2 - 1), (n - 1))

if mid < right:

merge(arr, left, mid, right)

size \*= 2

tim\_sort(data)

return data

## **main.py**

Este arquivo coordena a execução dos algoritmos, gera dados aleatórios e expõe métricas via prometheus\_client.

### **Exemplo de Documentação:**

def main():

"""Inicia o servidor HTTP para métricas e executa testes de ordenação.

Testa os algoritmos com tamanhos 1000 e 10000 e registra os resultados.

"""

start\_http\_server(8000)

sizes = [1000, 10000]

algorithms = [

("Bubble Sort", BubbleSort()),

("Tim Sort", TimSort()),

("Quick Sort", QuickSort()),

]

try:

while True:

for size in sizes:

data = generate\_data(size)

for name, algorithm in algorithms:

\_, elapsed\_time, comparisons, swaps = measure\_sorting\_time(algorithm, data)

print(f"{name}: {elapsed\_time:.2f} ms | Comparacoes: {comparisons} | Trocas: {swaps}")

time.sleep(30)

except KeyboardInterrupt:

print("\nParando o servidor de métricas.")

# **2. Explicação do Uso do Padrão Strategy**

O padrão **Strategy** permite encapsular diferentes algoritmos e torná-los intercambiáveis. No projeto:

* SortingStrategy define a interface comum.
* SortingContext delega a ordenação para a estratégia escolhida.

class SortingContext:

def \_\_init\_\_(self, strategy):

self.strategy = strategy

def execute\_sort(self, data):

sorted\_data = self.strategy.sort(data.copy())

return sorted\_data

# **3. Descrição do Processo de Geração dos Dados**

* **Geração:** Dados são gerados aleatoriamente com:

def generate\_data(size):

return [random.randint(0, 100000) for \_ in range(size)]

* **Execução:** Algoritmos ordenam uma cópia dos dados, e métricas são coletadas via prometheus\_client.

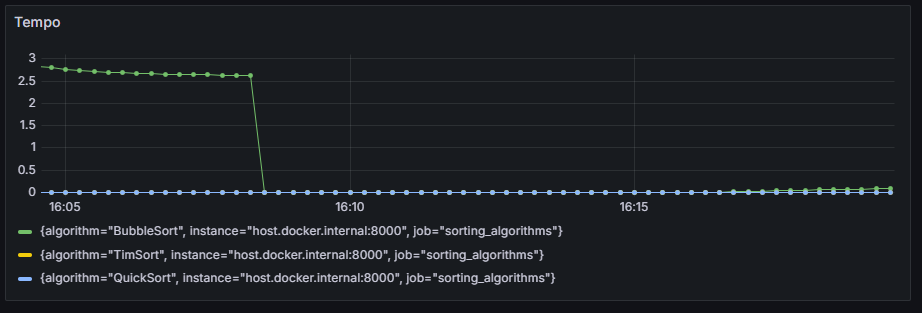
# **4. Métricas e Gráficos Comparativos de Desempenho**

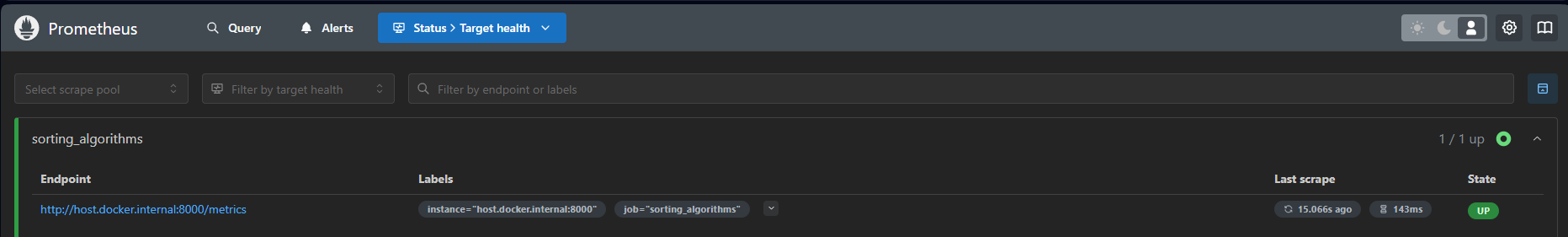
* **Métricas Coletadas:**
  + **Tempo de Execução:** Medido via Histogram.
  + **Comparacões:** Contadas via Counter.
  + **Trocas:** Contadas via Counter.

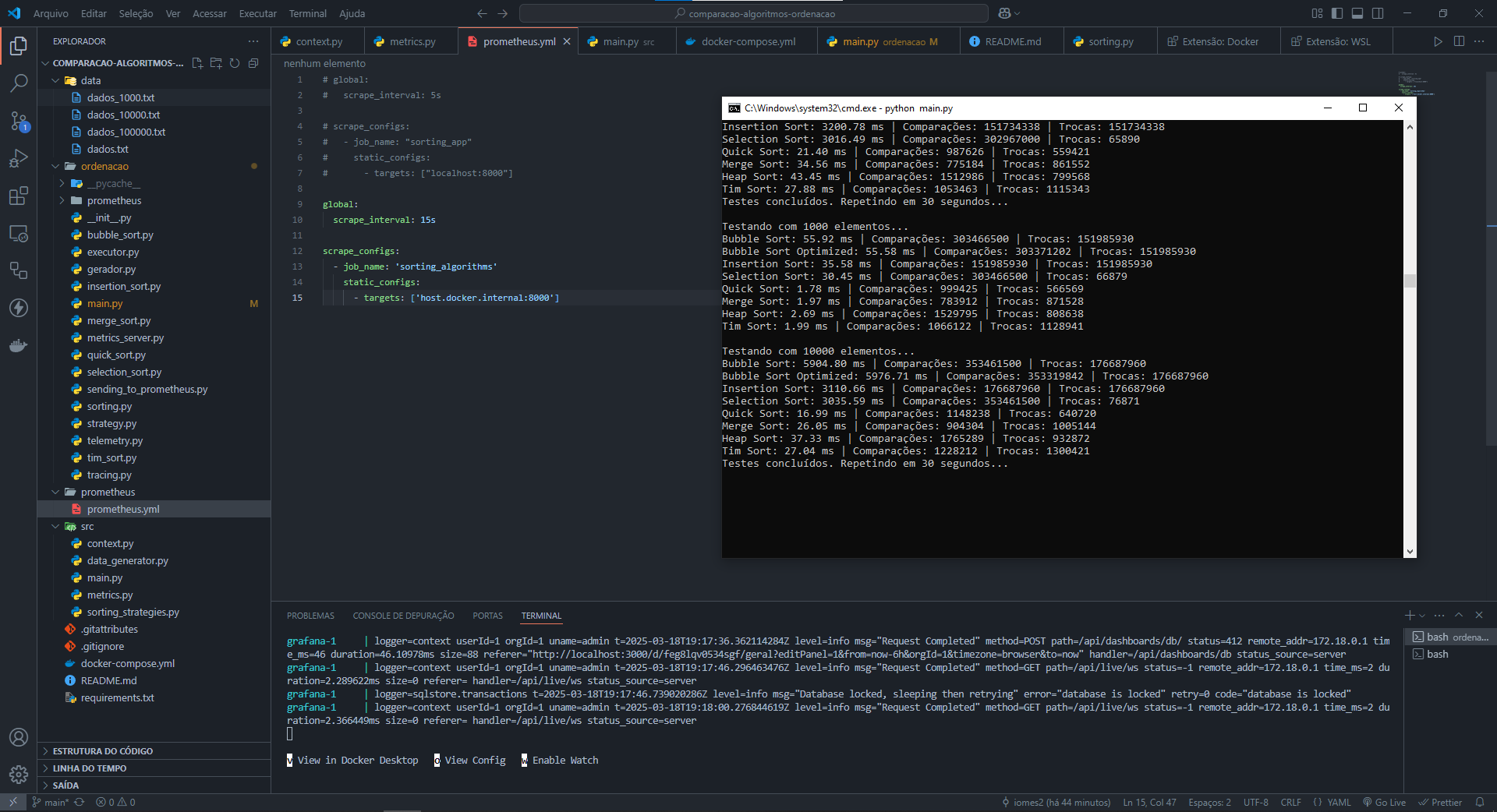
# **5. Ferramentas Utilizadas para Logs e Análise**

* **Prometheus:** Coleta e armazena métricas.
* **Grafana:** Exibe os dados do Prometheus.









# **6. Conclusão: Melhor Algoritmo e Análise**

* **Melhor Desempenho:**
  + **Tim Sort** foi o mais eficiente devido à sua abordagem híbrida.
  + **Quick Sort** foi competitivo, mas com piores casos em dados já ordenados.
  + **Bubble Sort** foi o menos eficiente.
* **Vale a Pena Usar "Dividir e Conquistar"?**
  + Sim! Quick Sort e Merge Sort (base do Tim Sort) utilizam essa técnica, reduzindo a complexidade para **O(n log n)**.
  + Para pequenos conjuntos de dados, **Insertion Sort** é mais eficiente devido ao baixo overhead.

**Conclusão Final:** Tim Sort é a melhor escolha para a maioria dos cenários, especialmente para listas parcialmente ordenadas.