BRUNO FELIPE DE SOUZA ARAUJO

CDN 3 SEMESTRE

A) Memória. Use a função mode_using_counter e calcule o consumo de memória.

```python
import numpy as np
from collections import Counter
import psutil
import os

def mode_using_counter(n_integers):
    """Gera números aleatórios e retorna o mais frequente"""
    process = psutil.Process(os.getpid())  # Obter o processo atual
    mem_before = process.memory_info().rss / 1024 ** 2  # Memória antes (MB)
    random_integers = np.random.randint(1, 100_000, n_integers)
    c = Counter(random_integers)
    result = c.most_common(1)[0][0]
    mem_after = process.memory_info().rss / 1024 ** 2  # Memória depois (MB)
    print(f" Memória antes: {mem_before:.2f} MB")
    print(f" Memória depois: {mem_after:.2f} MB")
    print(f" Diferença de memória: {mem_after - mem_before:.2f} MB")
    return result

mode_using_counter(100_000)
```

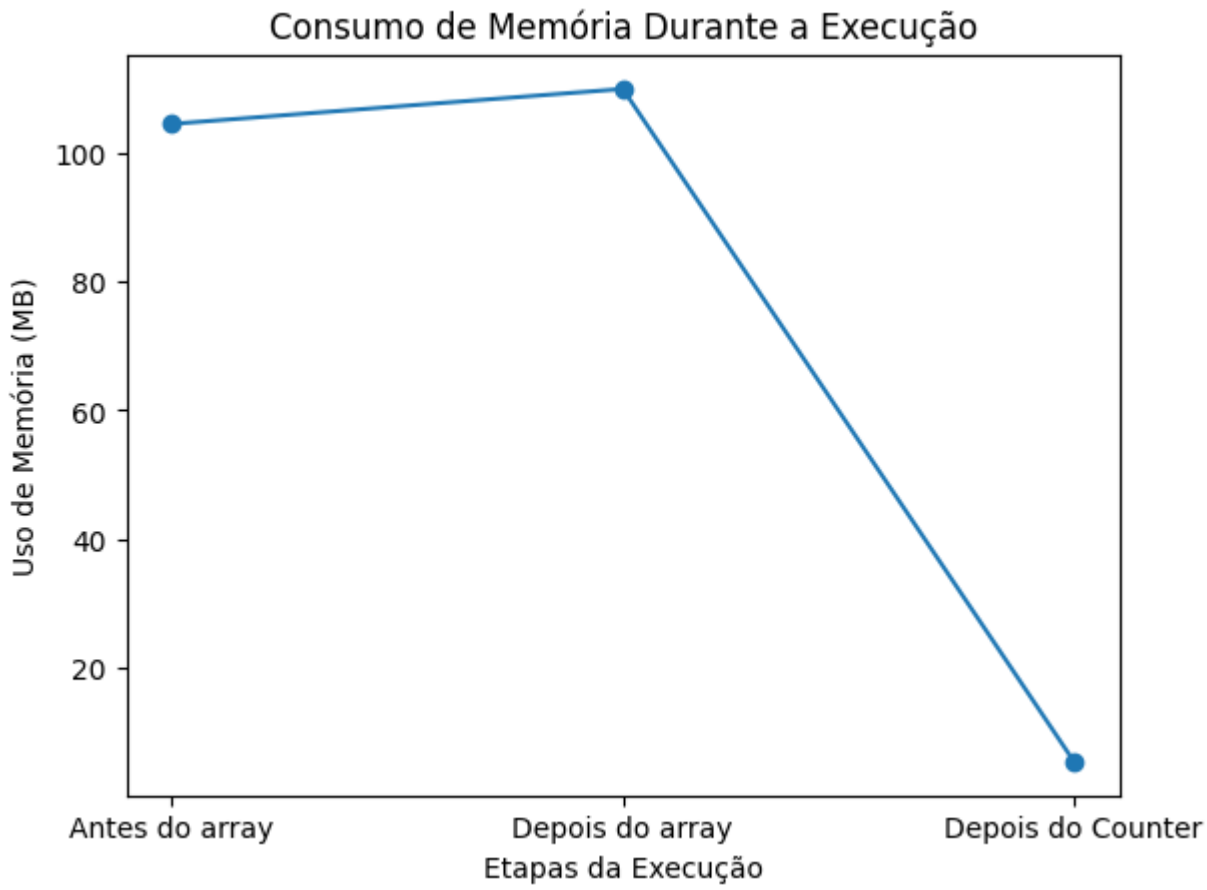
Memória antes: 104.43 MB

Memória depois: 109.86 MB

Diferença de memória: 5.44 MB

np.int32(73844)

```python
import matplotlib.pyplot as plt
memory_usage = [104.43, 109.86, 5.44]
steps = ["Antes do array", "Depois do array", "Depois do Counter"]
plt.plot(steps, memory_usage, marker="o", linestyle="-")
plt.xlabel("Etapas da Execução")
plt.ylabel("Uso de Memória (MB)")
plt.title("Consumo de Memória Durante a Execução")
plt.show()
```



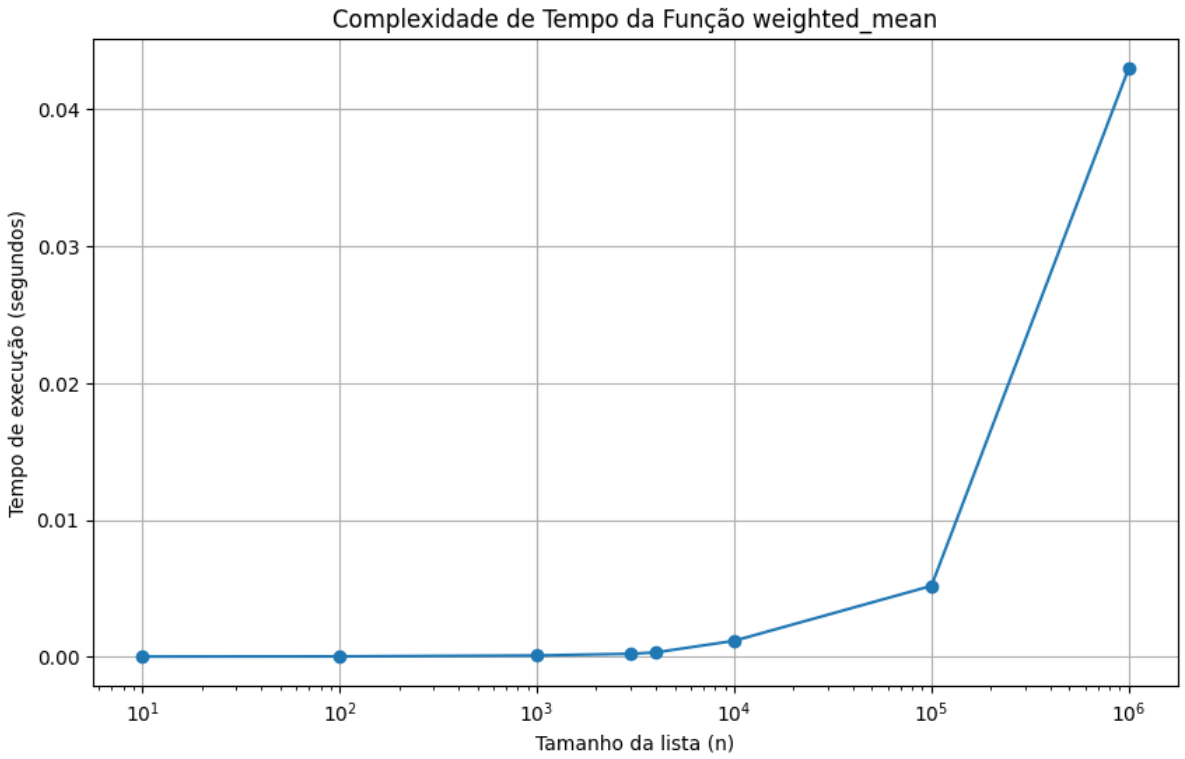
B) Função Linear - O(n) - execute a função weighted_mean e para medir o tempo

```
import time
import random
import matplotlib.pyplot as plt
def weighted_mean(list_of_numbers, weights):
    running_total = 0
    for i in range(len(list_of_numbers)):
        running_total += (list_of_numbers[i] * weights[i])
    return running_total / sum(weights)

# Função auxiliar para medir tempo
def medir_tempo(tamanho):
    numeros = [random.uniform(1, 100) for _ in range(tamanho)]
    pesos = [random.uniform(1, 10) for _ in range(tamanho)]
    inicio = time.time()
    resultado = weighted_mean(numeros, pesos)
    fim = time.time()
    return fim - inicio

# Listas para guardar os tamanhos e tempos
tamanhos = [10, 100, 1_000, 3_000, 4_000, 10_000, 100_000, 1_000_000]
tempos = []
```

```python
print("Testando função e coletando dados para o gráfico...\n")
for tamanho in tamanhos:
    tempo_execucao = medir_tempo(tamanho)
    tempos.append(tempo_execucao)
    print(f"Tamanho: {tamanho:>9} | Tempo: {tempo_execucao:.6f} segundos")

# Criando o gráfico
plt.figure(figsize=(10, 6))
plt.plot(tamanhos, tempos, marker='o')
plt.title("Complexidade de Tempo da Função weighted_mean")
plt.xlabel("Tamanho da lista (n)")
plt.ylabel("Tempo de execução (segundos)")
plt.grid(True)
plt.xscale('log')  # Escala logarítmica para melhor visualização
#plt.yscale('log')  # Opcional: mostra que o crescimento é linear em escala log
plt.show()
```

Testando função e coletando dados para o gráfico...

Tamanho: 10 | Tempo: 0.000013 segundos
Tamanho: 100 | Tempo: 0.000023 segundos
Tamanho: 1000 | Tempo: 0.000091 segundos
Tamanho: 3000 | Tempo: 0.000216 segundos
Tamanho: 4000 | Tempo: 0.000314 segundos
Tamanho: 10000 | Tempo: 0.001162 segundos
Tamanho: 100000 | Tempo: 0.005179 segundos
Tamanho: 1000000 | Tempo: 0.042998 segundos



C) Buble Sort x Insertion Sort x timSort

1. O que você observa no gráfico em relação ao tempo de execução do Bubble Sort e do Insertion Sort?

R: Pelo gráfico é possível visualizar que o tempo de execução do inserction sort é menor, sendo assim um algoritmo mais eficiência se comparado bubble e inserction sort.

2. Qual algoritmo se comportou melhor conforme o tamanho da lista aumentou? Por quê?

R: Dentre os algoritmos analisados o que teve melhor eficiencia conforme a lista aumentou é o TimSort, pois pelo gráfico vemos que conforme cresceu o tamanho da lista o tempo de execução não foi afetado, muito pelo fato da otimização de verificações repetidas em que os demais algoritmos realizam.

3. O Timsort foi mais rápido? Com base na teoria, qual sua complexidade de tempo e por que isso faz diferença?

R: O Timsort foi o mais rápido entre os algoritmos estudados, a complexidade Big O desse algoritmo é O(n log n) sendo assim comparado com uma complexidade $O(n^2)$ por exemplo do bubble, nos diz que conforme a lista cresce, a eficiência do Timsort é otimizada pois tem um menor numero de verificações ou "trocas" durante da ordenação para atingir o resultado, afetando assim memória, processamento e tempo final.

4. Comparando os algoritmos:

• Quais algoritmos têm complexidade quadrática ($O(n^2)$)?
R: Os algoritmos que tem complexidade quadrática são: Bubble Sort e Insertion Sort.

• Qual tem complexidade log-linear (O(n log n))?
R: O algoritmo que tem complexidade log-linear é o Timsort.

5. Execute o teste com uma lista de tamanho 3000 ou 4000 (adicione no vetor tamanhos).

• O que acontece com o tempo do Bubble Sort?
R: Conforme aumentamos o tamanho da lista, aumenta exponencialmente o tempo de execução.

• Por que isso ocorre?
R: Isso ocorre devido a complexidade do algoritmo ser $O(n^2)$, no qual a cada tamanho de lista temos uma curva quadrática de execução desse algoritmo.