BRUNO FELIPE DE SOUZA ARAUJO

CDN 3 SEMESTRE

1. Memória. Use a função mode\_using\_counter e calcule o consumo de memória.

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)

import matplotlib.pyplot as plt

memory\_usage = [104.43, 109.86, 5.44]

steps = ["Antes do array", "Depois do array", "Depois do Counter"]

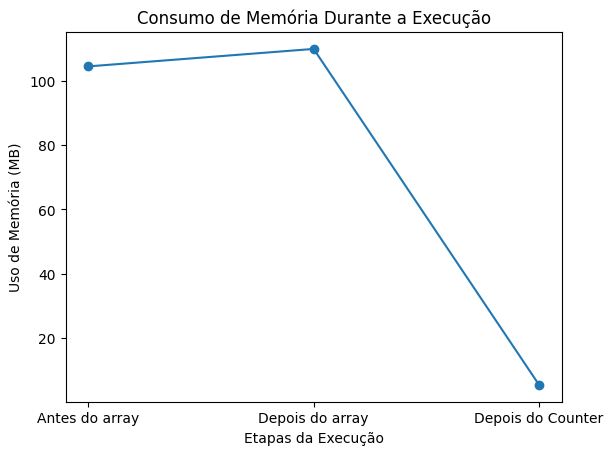
plt.plot(steps, memory\_usage, marker="o", linestyle="-")

plt.xlabel("Etapas da Execução")

plt.ylabel("Uso de Memória (MB)")

plt.title("Consumo de Memória Durante a Execução")

plt.show()



1. 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 = []

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

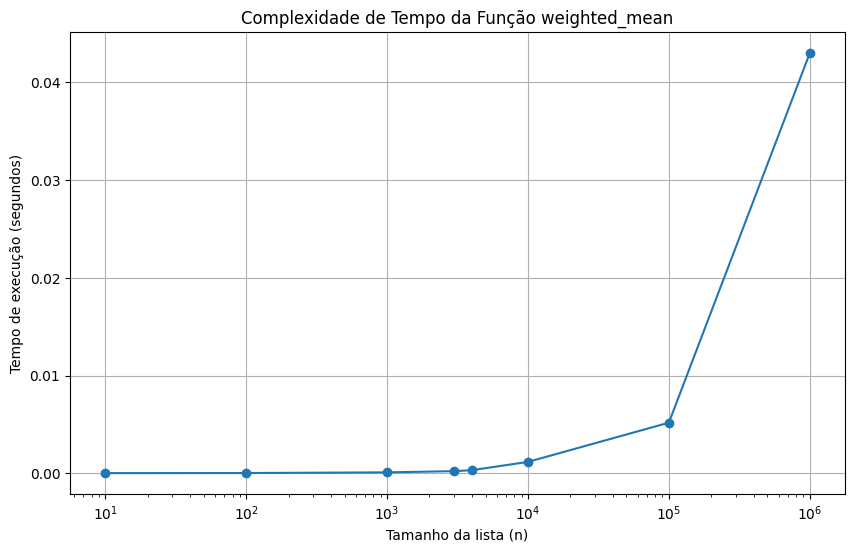
Tamanho: 3000 | Tempo: 0.000216 segundos

Tamanho: 4000 | Tempo: 0.000314 segundos

Tamanho: 10000 | Tempo: 0.001162 segundos

Tamanho: 100000 | Tempo: 0.005179 segundos

Tamanho: 1000000 | Tempo: 0.042998 segundos



1. 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(n²) 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(n²))?

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(n²), no qual a cada tamanho de lista temos uma curva quadrática de execução desse algoritmo.