Grupo 1 - Exercícios para desenvolver processos assíncronos e paralelos, integrando asyncio e OpenMP para otimização de tarefas computacionais.

Exercício 1.1: Download Assíncrono de URLs

Objetivo: Implementar um programa em Python que use `asyncio` para baixar conteúdos de múltiplas URLs ao mesmo tempo.

Instruções:

1. Crie uma lista de URLs.
2. Utilize `aiohttp` junto com `asyncio` para realizar downloads assíncronos.
3. Varie o número de downloads assíncronos e meça o tempo total para completar todos os downloads.
4. Monte um gráfico de número de threads x tempo. Analise o gráfico.

Exercício 1.2: Cálculo Paralelo com OpenMP

Objetivo: Usar a API do OpenMP do módulo cython.parallel para calcular a soma de um grande vetor em paralelo.

Instruções:

1. Crie um vetor grande com números aleatórios com 10000 posições, com valores entre 1 e 100000.
2. Compare o tempo de execução com uma versão sequencial do cálculo.

Exercício 1.3: Processamento de Imagens Assíncrono

Objetivo: Processar múltiplas imagens de forma assíncrona em Python.

Instruções:

1. Crie um script que lê várias imagens de uma pasta.
2. Utilize a biblioteca `Pillow` para aplicar um filtro a cada imagem.
3. Use `asyncio` para aplicar o filtro a todas as imagens de forma assíncrona e salve-as em outra pasta.
4. Monte um gráfico de número de threads x tempo. Analise o gráfico.

Grupo 2 - Exercícios para implementar os algoritmos QuickSort e QuickSelect para solucionar problemas complexos.

Exercício 2.1: Implementação do QuickSort

Objetivo: Implemente o algoritmo QuickSort em uma linguagem de programação python.

Tarefas:

1. Escreva uma função que receba uma lista de números e retorne a lista ordenada usando o QuickSort.
2. Adicione a opção de escolher diferentes pivôs (por exemplo, o primeiro elemento, o último elemento ou o elemento mediano) e compare o desempenho.

Exercício 2.2: Ordenação de Estruturas de Dados

Objetivo: Aplique o QuickSort a uma lista de objetos em Python.

Tarefas:

1. Crie uma estrutura de dados (por exemplo, um objeto de estudante com nome e nota).
2. Implemente o QuickSort para ordenar uma lista de estudantes por suas notas.
3. Exiba a lista ordenada.

Exercício 2.3: Implementação do QuickSelect

Objetivo: Implemente o algoritmo QuickSelect para encontrar o k-ésimo menor elemento (em Python).

Tarefas:

1. Escreva uma função que receba uma lista de 10000 números entre 1 e 1000 e um inteiro k e retorne o k-ésimo menor elemento.
2. Teste sua implementação em 10 diferentes listas e com 5 valores de k para cada lista.

Exercício 2.4: Problemas Avançados com QuickSelect

Objetivo: Aplique o QuickSelect em problemas mais complexos (em Python)..

Tarefas:

1. Resolva o problema da mediana: dado um conjunto de números, utilize o QuickSelect para encontrar a mediana.
2. Experimente adaptá-lo para encontrar os k elementos menores de uma lista sem ordená-la completamente.

Grupo 3 - Exercícios para manipular estruturas de dados lineares, incluindo listas encadeadas, listas duplamente encadeadas, com operações de leitura, inserção e exclusão.

Exercício 3.1: Lista Encadeada Simples

Objetivo: Implementar uma lista encadeada simples com operações básicas (em Python).

Tarefas:

1. Crie uma classe `Node` que represente um elemento da lista, armazenando um valor e um ponteiro para o próximo nó.
2. Crie uma classe `LinkedList` que inclua métodos para:
   - Inserir um novo nó no início da lista.
   - Inserir um novo nó no final da lista.
   - Excluir um nó com um valor específico.
   - Exibir todos os elementos da lista.
3. Teste suas implementações com diferentes operações.

Exercício 3.2: Lista Duplamente Encadeada

Objetivo: Implementar uma lista duplamente encadeada (em Python).

Tarefas:

1. Crie uma classe `DNode` que armazene um valor, um ponteiro para o próximo nó e um ponteiro para o nó anterior.
2. Crie uma classe `DoublyLinkedList` que inclua métodos para:
   - Inserir um novo nó no início da lista.
   - Inserir um novo nó no final da lista.
   - Excluir um nó de uma posição específica.
   - Exibir todos os elementos da lista na ordem direta e reversa.
3. Realize testes abrangentes das suas implementações.

Exercício 3.3: Manipulações em Listas Encadeadas

Objetivo: Desenvolver funções adicionais para a lista encadeada (em Python).

Tarefas:

1. Adicione um método à sua classe `LinkedList` para buscar um nó pelo seu valor e retornar sua posição.
2. Implemente um método para inverter a lista encadeada.
3. Teste esses métodos com exemplos variados e diferentes estados da lista.

Exercício 3.4: Ordenação de Elementos em Lista Duplamente Encadeada

Objetivo: Implementar a ordenação em uma lista duplamente encadeada (em Python).

Tarefas:

1. Após ter sua lista duplamente encadeada funcionando, implemente um método que ordene a lista utilizando o algoritmo de Insertion Sort ou Bubble Sort.

2. Implemente também uma função de mesclagem que combine duas listas duplamente encadeadas ordenadas em uma única lista ordenada.
3. Verifique a correta funcionalidade desses métodos através de testes.

Grupo 4 - Exercícios para resolver problemas complexos empregando diferentes técnicas de algoritmos recursivos

Exercício 4.1: Fatorial de um Número (em Python).

Objetivo: Implementar a função fatorial de um número usando recursão.

Tarefas:
1. Escreva uma função recursiva que calcule o fatorial de um número inteiro positivo.
2. Teste sua função com vários valores e explique a complexidade.

Exercício 4.2: Sequência de Fibonacci (em Python).

Objetivo: Calcular a sequência de Fibonacci utilizando um algoritmo recursivo.

Tarefas:
1. Crie uma função recursiva que retorne o n-ésimo número da sequência de Fibonacci.
2. Implemente otimizações como memorização para melhorar a eficiência. Compare o tempo de execução com e sem memorização.

Exercício 4.3: Torres de Hanói *(em Python).*

Objetivo: Resolver o puzzle das Torres de Hanói com recursão.

Tarefas:
1. Desenvolva uma função que resolva o problema das Torres de Hanói e imprima os passos necessários para mover os discos.
2. Teste para números de discos entre 1 e 30, construa um gráfico com os tempos e indique a complexidade do algoritmo implementado.

Exercício 4.4: Permutações de uma String *(em Python).*

Objetivo: Gerar todas as permutações possíveis de uma string usando recursão.

Tarefas:
1. Implemente uma função recursiva que encontre e imprima todas as permutações de uma string fornecida.
2. Garanta que as duplicações sejam tratadas caso a string tenha caracteres repetidos.
3. Calcule a complexidade do algoritmo implementado.

Grupo 5 - Exercícios para orquestrar a navegação e manipulação de estruturas de dados complexas (listas e árvores) utilizando computação paralela com OpenMP

Exercício 5.1: Soma de Elementos em uma Lista (em Python).

Objetivo: Utilizar OpenMP para calcular a soma de elementos em uma lista.

Tarefas:

1. Crie um programa que use OpenMP para dividir a soma dos elementos em múltiplos threads e calcular a soma total.
2. Compare o desempenho com a versão sequencial.

Exercício 5.2: Busca em Árvore Binária*(em Python).*

Objetivo: Implementar uma busca paralela numa árvore binária.

Tarefas:

1. Crie uma árvore binária e implemente um algoritmo que busque um valor específico utilizando OpenMP para realizar a busca em paralelo.
2. Analise o tempo de execução em diferentes tamanhos de árvore, aumentando a instância em potência de 2 de nós (Ex: 1, 2, 4, 8, 16, ...).

Exercício 5.3: Ordenação Paralela de Uma Lista *(em Python).*

Objetivo: Implementar a ordenação paralela de uma lista usando OpenMP.

Tarefas:

1. Utilize o algoritmo QuickSort ou MergeSort em versão paralela para ordenar uma lista de números.
2. Compare a performance com a versão sequencial.

Exercício 5.4: Máximo de uma Lista*(em Python).*
Objetivo: Encontrar o valor máximo de uma lista utilizando OpenMP.

Tarefas:

1. Escreva um código que usa OpenMP para dividir o trabalho de encontrar o valor máximo na lista em múltiplos threads.
2. Meça e compare a eficiência entre a versão sequencial e a paralela.

Grupo 6 - Exercícios para aplicar programação dinâmica para otimizar e corrigir desafios inerentes à recursão

Exercício 6.1: Problema da Mochila*(em Python).*
Objetivo: Resolver o problema da mochila usando programação dinâmica.

Tarefas:

1. Implemente uma solução que utilize a abordagem de programação dinâmica para resolver o problema da mochila 0-1.
2. Teste com diferentes conjuntos de itens e pesos.

Exercício 6.2: Sequência Longa Comum *(em Python).*

Objetivo: Encontrar a sequência longa comum entre duas strings.

Tarefas:
1. Use programação dinâmica para implementar uma solução que calcule o comprimento da sequência comum mais longa entre duas strings dadas.
2. Mostre a sequência correspondente.

Exercício 6.3: Troco de Moedas *(em Python).*

Objetivo: Calcular o número mínimo de moedas necessárias para fazer um determinado valor com um dado conjunto de moedas.

Tarefas:
1. Implemente um algoritmo de programação dinâmica para resolver o problema do troco, garantindo que o algoritmo retorne à quantidade mínima de moedas.
2. Teste com diferentes valores e conjuntos de moedas.

Exercício 6.4: Pintura de Cadeiras*(em Python).*

Objetivo: Resolver o problema de pintura de N cadeiras utilizando programação dinâmica.

Tarefas:
1. Considere que você tem um número fixo de cores e precisa descobrir a quantidade de maneiras de pintar N cadeiras, de modo que não haja duas cadeiras adjacentes da mesma cor.
2. Implemente uma solução com programação dinâmica e discuta a eficiência.