Unidade II

3 ESTRUTURAS DE DADOS NÃO LINEARES

As estruturas de dados não lineares diferenciam-se das lineares pela ausência de organização sequencial, permitindo modelar relacionamentos mais complexos entre elementos. Entre essas estruturas, as árvores representam hierarquias de forma intuitiva. Cada árvore é composta por nós interligados por arestas, tendo um nó raiz que origina ramificações. Ao implementar uma árvore binária em Python, costuma-se definir uma classe que armazena o valor do nó e referências para o filho esquerdo e para o filho direito.

As travessias de árvores binárias configuram-se como procedimentos fundamentais para acesso e manipulação de dados: na travessia em pré-ordem visita-se primeiro o nó atual, após o qual ocorre a exploração recursiva do filho esquerdo e, em seguida, do filho direito; na travessia em ordem realiza-se a recursão pelo filho esquerdo antes de visitar o nó e somente depois prossegue-se para o filho direito; na travessia em pós-ordem aguarda-se a conclusão das visitas às subárvores para apenas então processar o nó. Cada abordagem atende a exigências distintas, viabilizando operações como geração de expressões prefixas ou infixas, impressão ordenada de valores e remoção controlada de nós.

Os grafos configuram outro tipo de estrutura não linear que modela conjuntos de vértices interligados por arestas, sendo empregados em cenários que vão desde redes de transporte até interações em redes sociais. Em Python, a representação pode apoiar-se em uma matriz de adjacência – uma matriz bidimensional em que cada célula indica a existência e, opcionalmente, o peso de uma aresta entre dois vértices – ou em listas de adjacência, que associam cada vértice a um conjunto de vizinhos. As listas de adjacência revelam-se eficientes em grafos esparsos, reduzindo a ocupação de memória, ao passo que a matriz simplifica consultas diretas sobre conexões entre vértices.

A exploração de grafos apoia-se em travessias por profundidade (DFS) e por largura (BFS). A primeira aprofunda-se ao máximo em cada direção antes de retroceder, sendo frequentemente implementada via recursão ou pilha; a BFS utiliza fila para garantir que todos os vértices a uma certa distância sejam visitados antes de seguir para a próxima camada do grafo. Essas técnicas fundamentam algoritmos mais sofisticados, como detecção de ciclos, busca de caminhos mínimos e análise de componentes conectados.

3.1 Árvores: conceitos, árvores binárias e travessias (pré-ordem, em ordem e pós-ordem)

A denominação árvore origina-se da semelhança entre a estrutura hierárquica do modelo computacional e a forma de uma árvore botânica. O nó raiz corresponde à raiz do vegetal, sustentando toda a estrutura; os nós intermediários atuam como galhos que se ramificam; e os nós sem filhos assemelham-se às folhas que encerram os ramos. Essa analogia facilita a compreensão intuitiva das relações de dependência e hierarquia, evidenciando

de que modo cada elemento deriva de outro e reforçando a ideia de subordinação entre níveis. A figura a seguir compara uma árvore botânica a uma árvore computacional.

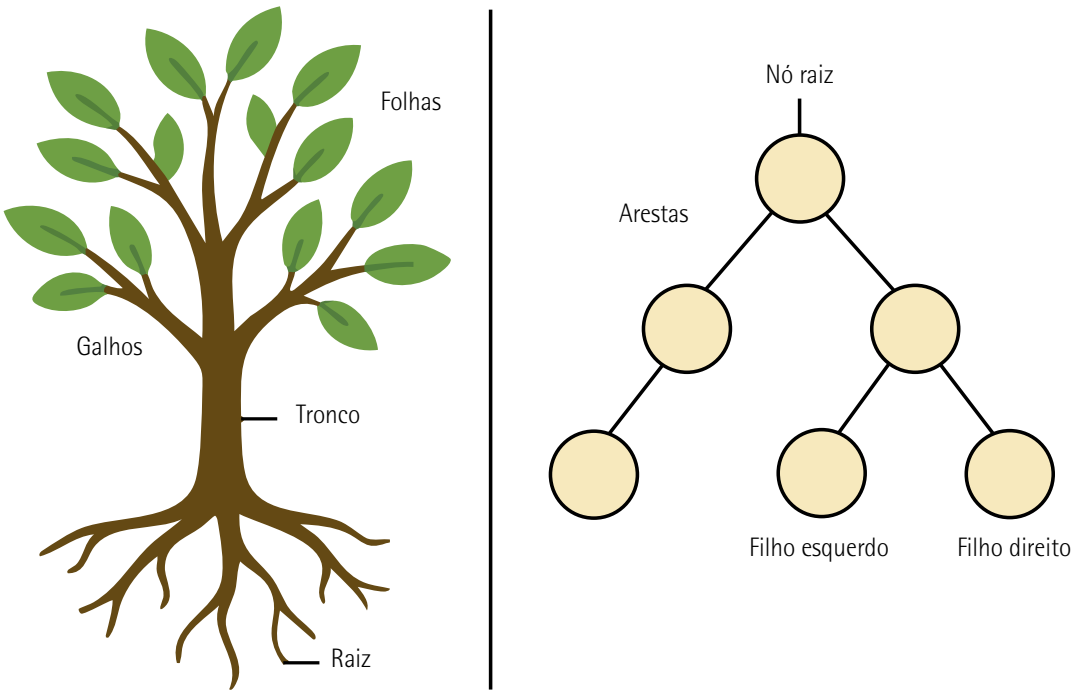

Figura 10 – Estrutura de dados do tipo árvore comparada com uma árvore natural

A representação gráfica de uma árvore de dados costuma aparecer invertida em relação à árvore botânica por atender a convenções de leitura e clareza hierárquica. No diagrama computacional, o nó raiz ocupa a posição superior, simbolizando o ponto de partida da estrutura, a partir do qual se ramificam os nós filhos em direção à base do desenho. Essa orientação top-down alinha-se ao sentido natural de leitura em diversas culturas, simplificando a interpretação do fluxo de dados ou das dependências entre os nós.

Adotar a árvore de cabeça para baixo enfatiza a ideia de descentralização progressiva: o processamento inicia-se pelo elemento raiz e avança sistematicamente pelos níveis subsequentes. Essa configuração facilita a visualização de operações como inserção, busca e remoção de nós, pois cada ramo pode ser percorrido de modo sequencial, sem necessidade de reorientar mentalmente a estrutura. Assim, o espelhamento em relação ao modelo natural atende principalmente a finalidades didáticas e à praticidade na construção de algoritmos.

No contexto da programação em Python, a metáfora da árvore ressalta que cada nó pode gerar descendentes em diferentes direções, assim como galhos dividem-se em ramos menores. A figura 11 mostra um exemplo de árvore com letras nos nós. Nesse exemplo, a raiz da árvore é o nó S, que possui um único filho, o nó E. O nó E, por sua vez, possui dois filhos: C e D. O filho C tem um filho abaixo dele (A), enquanto o filho D possui dois filhos: O e R (sendo O filho à esquerda de D e R é o filho à direita). Neste exemplo cada nó armazena apenas uma letra. Nos programas reais, diversos dados podem ser armazenados.

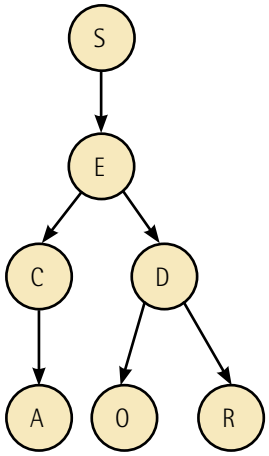

Figura 11 – Árvore com as letras A, C, D, E, O, R e S em seus nós

Nesta árvore, o percurso em pré-ordem forma a palavra "SECADOR" e o percurso em pós-ordem forma a palavra "ACEORDS":

- **Percurso em pré-ordem**: visitamos primeiro a raiz S, depois E, em seguida C, depois seu filho A; voltamos, visitamos D, depois O e R. Isso produz a sequência de letras S → E → C → A → D → O → R, formando a palavra SECADOR (a pré-ordem lê a raiz antes dos filhos).

- **Percurso em pós-ordem**: visitamos primeiro todos os descendentes, começando do fundo da árvore, temos A (folha à esquerda), depois voltamos e lemos C, então passamos para o ramo de E, lendo O seguido de R, depois subimos para ler D e, por fim, a raiz S. A ordem resultante A → C → E → O → D → R → S soletra ACEORDS (a pós-ordem lê a raiz por último).

Ao empregar o termo árvore, desenvolvedores e estudantes de computação beneficiam-se de uma imagem familiar que explica, de modo simples, como organizar, acessar e manipular dados de forma hierárquica. Essa representação visual contribui para comparações diretas com estruturas naturais, auxiliando na memorização de conceitos como nós pai, nós filho e folhas, e fornece base para entender algoritmos de inserção, remoção e percurso de forma mais clara e intuitiva.

Árvores são usadas em situações que envolvem classificações hierárquicas complexas ou sistemas de organização com múltiplos elementos relacionados a um mesmo ponto central. Um exemplo cotidiano é a estrutura de diretórios de um sistema operacional. Em um computador, uma pasta pode conter diversas subpastas e arquivos, o que exige uma árvore genérica, já que não há limite fixo de quantos filhos (itens) um nó (pasta) pode ter.

Outro exemplo é encontrado nos menus de navegação de sites ou aplicativos. Cada item de menu pode conter diversos subitens, que também podem conter outros níveis de submenu. Esse tipo de estrutura de interface é modelado como uma árvore genérica, permitindo flexibilidade na organização das opções.

Ainda em contextos do cotidiano, ontologias e classificações em bibliotecas ou catálogos de produtos utilizam árvores genéricas. Por exemplo, em uma loja virtual, a categoria "Eletrônicos" pode se ramificar em "Celulares", "Televisores", "Computadores", cada uma contendo outras subcategorias – uma estrutura não limitada a dois filhos por categoria.

Uma árvore binária é uma estrutura de dados hierárquica composta por nós, na qual cada nó possui, no máximo, dois filhos, denominados filho esquerdo e filho direito. Essa organização permite representar relações entre elementos de maneira eficiente, especialmente em algoritmos que envolvem buscas, ordenações e hierarquias. A estrutura é formada por um nó principal, chamado raiz, do qual derivam outros nós, organizando-se em subárvores. Cada subárvore, por sua vez, também é uma árvore binária.

O conceito fundamental por trás dessa estrutura é a divisão recursiva: cada nó pode ser considerado o início de uma nova árvore, composta por seus dois filhos. Se um nó não possui filhos, ele é chamado de nó folha. A profundidade de um nó é definida pela quantidade de arestas entre ele e a raiz. Já a altura da árvore corresponde à maior profundidade entre todos os seus nós.

Existem variações da árvore binária, cada uma com características específicas. A árvore binária de busca, por exemplo, mantém os dados organizados de modo que o valor de cada nó seja maior do que todos os valores de sua subárvore esquerda e menor do que todos os valores de sua subárvore direita. Essa organização facilita a realização de operações como busca, inserção e remoção, reduzindo significativamente a complexidade computacional em comparação com estruturas lineares, como listas.

Além disso, há árvores balanceadas, como a AVL, ou a árvore rubro-negra, projetadas para manter a altura da árvore controlada mesmo após múltiplas inserções e exclusões, garantindo desempenho eficiente e previsível. A eficiência dessa estrutura depende diretamente do equilíbrio entre as subárvores, já que uma árvore muito desequilibrada pode se aproximar de uma lista encadeada, comprometendo a velocidade das operações.

A árvore binária é amplamente utilizada em diversas áreas da computação. Ela está presente em algoritmos de ordenação, como o heapsort, e em estruturas mais complexas, como árvores sintáticas em linguagens de programação ou sistemas de arquivos. Seu uso contribui para a organização lógica dos dados e permite que operações sejam realizadas com maior rapidez e menor consumo de recursos, especialmente em grandes volumes de informação.

Ela também é usada em aplicações nas quais a organização e a eficiência de acesso são fundamentais, especialmente quando os dados precisam ser manipulados com rapidez e precisão. Um exemplo importante é o uso de árvores binárias de busca em bancos de dados e sistemas de indexação. Nessas situações, as árvores binárias permitem localizar registros rapidamente por meio de comparações sucessivas, otimizando consultas e inserções.

Outro uso cotidiano aparece em compiladores de linguagens de programação, os quais utilizam árvores binárias de expressão para representar operações matemáticas ou lógicas. Por exemplo, ao interpretar uma expressão como 3 + (4 * 5), o compilador organiza os operandos e os operadores em uma árvore binária, na qual cada operador conecta dois operandos (ou subexpressões), respeitando a ordem das operações.

Em jogos eletrônicos, algoritmos de inteligência artificial muitas vezes utilizam árvores binárias de decisão para avaliar cenários e tomar decisões baseadas em condições binárias (sim/não, verdadeiro/falso), estruturando o raciocínio de forma eficiente e rápida. Além disso, algoritmos de compressão de dados, como o Huffman coding, fazem uso de árvores binárias para associar símbolos a códigos binários de

tamanhos diferentes, conforme a frequência de uso. Essa técnica é empregada, por exemplo, em formatos como JPEG e MP3.

Observação

O Huffman coding é um método de compactação de dados muito utilizado para reduzir o tamanho de arquivos digitais, como imagens, vídeos e textos. A ideia principal por trás desse método é simples: em vez de usar a mesma quantidade de bits para representar todos os caracteres ou símbolos, ele atribui códigos mais curtos aos elementos que aparecem com mais frequência e códigos mais longos aos que aparecem raramente. Assim, o arquivo final ocupa menos espaço, pois os dados mais repetidos – que são justamente os mais comuns – consomem menos memória para serem armazenados.

Para fazer isso funcionar, o algoritmo cria uma espécie de árvore binária, chamada de árvore de Huffman, na qual cada símbolo é uma folha da árvore, e os caminhos da raiz até essas folhas formam os códigos binários correspondentes. A construção dessa árvore segue um critério de economia: os símbolos mais frequentes ficam mais próximos da raiz, gerando códigos mais curtos. Como os códigos são montados de forma que nenhum é prefixo de outro (ou seja, não há ambiguidade na leitura), o computador consegue decodificar os dados corretamente mesmo com códigos de tamanhos diferentes. Esse método é amplamente usado em formatos de compressão como ZIP, JPEG e MP3, contribuindo para o armazenamento e a transmissão eficiente de informações.

Além dos trajetos pré-ordem e pós-ordem, as árvores binárias permitem adicionalmente o percurso em ordem porque sua estrutura hierárquica e recursiva proporciona uma maneira natural de visitar os elementos de maneira sistemática, respeitando uma sequência lógica. O percurso em ordem (ou in-order transversal) é uma das formas clássicas de percorrer uma árvore binária e consiste em visitar recursivamente a subárvore esquerda, depois o nó atual e, em seguida, a subárvore direita.

Para determinar o percurso in order (em ordem) da árvore binária exibida na figura 11, é necessário seguir a lógica:

1. Visita-se a subárvore esquerda.

2. Depois, o nó atual.

3. Em seguida, a subárvore direita.

Aplicando essa lógica à árvore da figura 11:

Raiz: S

- Subárvore esquerda: E
  - Subárvore esquerda de E: C
    - Subárvore esquerda de C: A
    - C (nó atual)
    - C não tem subárvore direita
  - E (nó atual)
  - Subárvore direita de E: D
    - Subárvore esquerda de D: O
    - D (nó atual)
    - Subárvore direita de D: R
- S (nó atual)
- S não tem subárvore direita

Juntando os elementos na ordem em que são visitados: A → C → O → R → D → E → S ou ACEODRS.

Essa possibilidade adicional decorre do fato de que cada nó da árvore possui, no máximo, dois filhos. Com isso, é possível estabelecer uma regra clara de prioridade: sempre visitar primeiro o conteúdo do lado esquerdo, depois o conteúdo do próprio nó e, por fim, o conteúdo do lado direito. A recursividade da estrutura permite que essa mesma regra seja aplicada a cada subárvore de maneira consistente, do nível mais profundo até o mais superficial.

Quando se trata de uma árvore binária de busca, esse tipo de percurso tem um benefício adicional: ele visita os nós em ordem crescente de valores. Isso acontece porque, por definição, os valores armazenados na subárvore esquerda de um nó são menores que o valor do nó, e os valores da subárvore direita são maiores. Portanto, ao aplicar o percurso em ordem, percorrem-se todos os elementos de forma crescente, o que é útil para operações de ordenação.

Em resumo, a possibilidade de realizar um percurso em ordem em uma árvore binária deriva diretamente de sua organização em três componentes hierárquicos – subárvore esquerda, nó central e subárvore direita – e da forma como os dados estão distribuídos dentro dessa estrutura. Essa característica

oferece vantagens notáveis para tarefas que exigem acesso sequencial, análise ordenada de dados ou manipulação eficiente de estruturas hierárquicas.

A inteligência artificial e o aprendizado de máquina têm transformado radicalmente a forma como as organizações processam informações e tomam decisões automatizadas. Enquanto a inteligência artificial engloba sistemas capazes de simular a inteligência humana em tarefas específicas, o aprendizado de máquina procura extrair padrões de dados históricos para predizer comportamentos futuros. Nesse âmbito, as árvores de decisão surgem como uma técnica intuitiva e eficaz, que facilita a criação de modelos de classificação, recomendação e diagnóstico automatizado. Ao representar de maneira gráfica e lógica as regras que conduzem cada decisão, elas tornam o processo de entendimento e auditoria do modelo mais transparente.

No atual cenário corporativo, em que volumes massivos de dados precisam ser analisados em tempo real, as árvores de decisão oferecem agilidade e interpretabilidade. Elas podem ser usadas tanto para classificar clientes conforme o risco de crédito quanto para recomendar produtos ou sugerir tratamentos médicos baseados em sintomas e exames. Por exemplo, em um sistema de recomendação, basta percorrer a árvore a partir da raiz até uma folha para indicar qual produto ou serviço é mais adequado ao perfil do usuário. Já em diagnósticos automatizados, a mesma lógica permite que variáveis clínicas sejam avaliadas passo a passo, resultando em laudo ou sugestão de procedimento.

A facilidade de implementação e o baixo custo computacional tornam as árvores de decisão particularmente atraentes para projetos piloto e aplicações nos quais a explicabilidade é crucial. Diferentemente de algoritmos de caixa-preta, como redes neurais profundas, elas permitem que analistas e gestores verifiquem exatamente quais condições levaram a cada resultado. Isso favorece auditorias de conformidade e a confiança do usuário final. Em nosso primeiro exercício deste tópico, demonstraremos como criar uma árvore de decisão em Python para classificar o risco de diabetes, percorrê-la em diferentes ordens e aplicar o modelo para diagnosticar novos pacientes de forma automatizada.

Exemplo de aplicação

Exemplo 1 – Inteligência artificial e machine learning

Considere uma clínica de telemedicina que oferece um serviço automatizado de triagem de pacientes em risco de diabetes. Cada paciente fornece dois parâmetros simples: nível de glicose no sangue e índice de massa corporal (IMC).

As regras de negócio definem que, se o nível de glicose for igual ou superior a 130 mg/dL, deve-se avaliar o IMC; pacientes com IMC igual ou superior a 30 são classificados como alto risco, enquanto aqueles com IMC abaixo de 30 são risco moderado. Se o nível de glicose estiver abaixo de 130 mg/dL, o paciente é considerado baixo risco.

A solução precisa permitir a construção manual dessa árvore de decisão, oferecer métodos para percorrê-la em pré-ordem, em ordem e pós-ordem; e conter uma função que, dado um novo paciente, percorra a árvore para retornar sua classificação de risco.

Para implementar a árvore de decisão, será utilizada a definição de uma classe em Python, que encapsula cada nó como um objeto com atributos para condição de teste, valor de folha ou ponteiros para nós filhos. Como vimos, a criação de classes em Python envolve o método especial `__init__`, responsável por inicializar esses atributos, e a utilização de propriedades de instância para manter o estado de cada nó. A modularização do código em métodos facilita a reutilização e a manutenção, permitindo estender ou modificar a estrutura com poucas alterações.

O percurso da árvore em pré-ordem, em ordem e pós-ordem será realizado por meio de funções recursivas. A recursão é um conceito fundamental em Python para percorrer estruturas não lineares, possibilitando que cada chamada trate um nó e invoque a si mesma sobre os filhos esquerdo e direito. O uso de chamadas recursivas torna o código conciso e natural para operações de travessia de árvores, eliminando loops aninhados que podem se tornar complexos em estruturas mais profundas.

Saiba mais

A recursão é um conceito fundamental na ciência da computação e na matemática. Para ilustrá-la pense em um exemplo do cotidiano como calcular o fatorial de um número. O fatorial de 5, por exemplo, é o resultado de multiplicar 5 por 4 por 3 por 2 por 1, que dá 120. Usando a recursão, o fatorial de 5 é 5 vezes o fatorial de 4; o fatorial de 4 é 4 vezes o fatorial de 3; e assim por diante, até chegar ao fatorial de 1, que é simplesmente 1. Nesse processo, a tarefa de calcular o fatorial se "chama" a si mesma, reduzindo o número a cada passo, até alcançar um caso básico que não precisa de mais cálculos. Esse caso básico é essencial, pois é o ponto em que a recursão para, evitando que o processo continue indefinidamente. Outro exemplo prático é imaginar uma boneca russa, aquelas que se abrem e revelam uma boneca menor dentro, que contém outra ainda menor, até chegar à última, que não pode ser mais aberta.

A recursão é como abrir essas bonecas: cada passo lida com uma versão menor do problema original até que não há mais nada para dividir. Em programação, isso é feito com funções que se chamam a si mesmas, sempre com um cuidado para garantir que haja uma condição de parada, como a última boneca ou o fatorial de 1.

O seguinte livro aborda a recursão em detalhes no contexto de algoritmos, especialmente em capítulos sobre divisão e conquista e análise de algoritmos recursivos.

CORMEN, T. H. *et al. Algoritmos*: teoria e prática. 4. ed. Rio de Janeiro: LTC, 2024.

Segue exemplo de uma boneca russa:

Figura 12 – Representação artística de uma boneca russa. Imagem produzida pelo autor com a ferramenta de inteligência artificial generativa Grok 3

Para a classificação de novos pacientes, será empregada uma função iterativa que inicia na raiz e, com base nas condições definidas (nível de glicose e IMC), escolhe o ramo esquerdo ou direito até alcançar um nó folha. Essa função faz uso de estruturas de controle de fluxo (`if` e `else`) para avaliar as condições e retornar o valor armazenado no nó folha, que representa a classificação de risco. Assim, o exercício integra conceitos de orientação a objetos, recursão e estruturas de controle, aplicando-os a um modelo de árvore de decisão simples, porém representativo de aplicações de machine learning e diagnóstico automatizado. A seguir, consta o código-fonte da solução:

```
class NoArvore:
    def __init__(self, teste=None, valor=None, filho_esquerdo=None, filho_
direito=None):
        self.teste = teste # Tupla (atributo, limite) para nós internos
        self.valor = valor # Valor de classificação para nós folhas
        self.filho_esquerdo = filho_esquerdo # Subárvore para resposta False
        self.filho_direito = filho_direito # Subárvore para resposta True

def criar_arvore_diabetes():
    # Nós folhas
    folha_baixo = NoArvore(valor="Baixo risco")
    folha_moderado = NoArvore(valor="Risco moderado")
    folha_alto = NoArvore(valor="Alto risco")
    # Nó que testa o IMC
    no_imc = NoArvore(teste=("IMC", 30.0), filho_esquerdo=folha_moderado, filho_
direito=folha_alto)
    # Raiz que testa o nível de glicose
    raiz = NoArvore(teste=("glicose", 130.0), filho_esquerdo=folha_baixo, filho_
direito=no_imc)
    return raiz

def pre_ordem(no):
    if no is None:
```

```
        return []
    # Visita nó, em seguida subárvore esquerda e direita
    resultado = [no.teste or no.valor]
    resultado += pre_ordem(no.filho_esquerdo)
    resultado += pre_ordem(no.filho_direito)
    return resultado

def em_ordem(no):
    if no is None:
        return []
    # Visita subárvore esquerda, nó, subárvore direita
    resultado = em_ordem(no.filho_esquerdo)
    resultado.append(no.teste or no.valor)
    resultado += em_ordem(no.filho_direito)
    return resultado

def pos_ordem(no):
    if no is None:
        return []
    # Visita subárvore esquerda e direita, depois nó
    resultado = pos_ordem(no.filho_esquerdo)
    resultado += pos_ordem(no.filho_direito)
    resultado.append(no.teste or no.valor)
    return resultado

def classificar_paciente(no, paciente):
    while no.teste is not None:
        atributo, limite = no.teste
        if paciente[atributo] < limite:
            no = no.filho_esquerdo
      else:
            no = no.filho_direito
    return no.valor

def principal():
    arvore = criar_arvore_diabetes()
    print("Travessia em pré-ordem:")
    print(pre_ordem(arvore))
    print("\nTravessia em-ordem:")
    print(em_ordem(arvore))
    print("\nTravessia pós-ordem:")
    print(pos_ordem(arvore))

    # Exemplo de classificação de pacientes
    pacientes = [
        {"glicose": 120.0, "IMC": 28.0},
        {"glicose": 140.0, "IMC": 32.5},
        {"glicose": 135.0, "IMC": 25.0}
    ]
    for i, paciente in enumerate(pacientes, start=1):
        risco = classificar_paciente(arvore, paciente)
        print(f"\nPaciente {i}: glicose={paciente['glicose']} mg/dL,
IMC={paciente['IMC']} -> {risco}")

if __name__ == "__main__":
    principal()
```

O quadro 9 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições. As funções já explicitadas nos quadros anteriores foram omitidas para evitar duplicidade.

Quadro 9 – Funções e construções usadas no código-fonte do primeiro exercício prático deste tópico

Função ou construção	Explicação de uso
class NomeDaClasse:	Define uma classe em Python, que serve como modelo para criar objetos com atributos e comportamentos específicos. No código, é usada para definir nós da árvore de decisão
def __init__(self,...):	Método especial que inicializa uma nova instância da classe. Aqui, define os atributos de cada nó da árvore (`teste`, `valor`, `filho_esquerdo`, `filho_direito`)
self.atributo	Referência a um atributo pertencente à instância atual da classe. Utilizado para armazenar os dados de cada nó
or	Operador lógico que retorna o primeiro operando verdadeiro. No código, `no.teste or no.valor` seleciona o conteúdo relevante do nó (teste se for interno, valor se for folha)
while condição:	Estrutura de repetição que executa um bloco de código enquanto a condição for verdadeira. No código, é usada para percorrer a árvore até alcançar uma folha durante a classificação
dict[chave]	Acessa o valor correspondente a uma chave em um dicionário. No código, é utilizado para acessar atributos do paciente durante a decisão
enumerate(iterável, start=1)	Gera pares `(índice, elemento)` ao percorrer um iterável, começando pelo índice especificado. No código, é utilizado para numerar os pacientes ao imprimir os resultados

A classe `NoArvore` define a estrutura de um nó da árvore de decisão. Cada nó possui quatro atributos: `teste`, que armazena uma tupla (atributo, limite) para nós internos, representando uma condição de decisão (por exemplo, glicose < 130.0); `valor`, que contém a classificação final (como `Baixo risco`) para nós folhas; `filho_esquerdo`, que aponta para a subárvore correspondente a uma resposta falsa à condição do teste; e `filho_direito`, que aponta para a subárvore correspondente a uma resposta verdadeira. O método `__init__` inicializa esses atributos, permitindo que sejam nulos por padrão, o que proporciona flexibilidade na criação de nós internos e folhas.

A função `criar_arvore_diabetes` constrói uma árvore de decisão específica para avaliar o risco de diabetes. Ela cria três nós folhas, cada um com um valor de classificação: `Baixo risco`, `Risco moderado` e `Alto risco`. Em seguida, define um nó interno que testa o IMC com um limite de `30.0`, no qual o filho esquerdo aponta para o nó de `Risco moderado` (IMC < 30.0) e o filho direito para o nó de `Alto risco` (IMC ≥ 30.0). Por fim, cria o nó raiz, que testa o nível de glicose com um limite de 130.0, conectando o filho esquerdo ao nó de `Baixo risco` (glicose < 130.0) e o filho direito ao nó que testa o IMC (glicose ≥ 130.0). A função retorna a raiz da árvore, completando a estrutura.

As funções `pre_ordem`, `em_ordem` e `pos_ordem` implementam três métodos de travessia da árvore, retornando listas com os valores dos nós (teste ou valor, conforme aplicável) na ordem visitada. A travessia em pré-ordem visita primeiro o nó atual, depois a subárvore esquerda e, por fim, a subárvore direita, resultando em uma lista que começa pela raiz e segue as ramificações. A travessia em ordem visita a subárvore esquerda, o nó atual e a subárvore direita, produzindo uma lista que reflete a ordem "esquerda-raiz-direita". A travessia pós-ordem visita as subárvores esquerda e direita antes do nó atual, gerando uma lista que termina com a raiz. Em cada função, a expressão `no.teste or no.valor` seleciona o atributo não nulo do nó (teste para nós internos, valor para folhas) e a recursão é interrompida quando o nó é nulo, retornando uma lista vazia.

A função `classificar_paciente` utiliza a árvore para classificar um paciente com base em seus atributos (glicose e IMC). Ela recebe o nó inicial (raiz) e um dicionário representando o paciente, com chaves `glicose` e `IMC`. Enquanto o nó atual possui um teste (ou seja, não é uma folha), a função extrai o atributo e o limite do teste e compara o valor do atributo do paciente com o limite. Se o valor for menor que o limite, a função segue para o filho esquerdo; caso contrário, para o filho direito. Esse processo continua até alcançar uma folha, cujo atributo valor (como `Baixo risco`) é retornado como a classificação final.

A função principal integra todas as funcionalidades. Ela cria a árvore chamando `criar_arvore_diabetes` e exibe os resultados das três travessias (pré-ordem, em ordem e pós-ordem), imprimindo as listas retornadas por `pre_ordem`, `em_ordem` e `pos_ordem`. Em seguida, define uma lista de três pacientes, cada um representado por um dicionário com valores de glicose e IMC. Para cada paciente, a função `classificar_paciente` é chamada e o resultado é impresso, mostrando os valores de glicose, IMC e a classificação de risco correspondente.

Por fim, o bloco `if __name__ == "__main__":` garante que a função principal seja executada apenas se o script for o programa principal. Ao rodar o código, ele constrói a árvore, exibe as travessias e classifica os pacientes. Por exemplo, um paciente com glicose 120.0 (menor que 130.0) será classificado como `Baixo risco` diretamente pela raiz, enquanto um paciente com glicose 140.0 e IMC 32.5 passará pelo teste de glicose (direita) e pelo teste de IMC (direita), resultando em `Alto risco`.

A seguir, proporemos algumas melhorias para que você possa evoluir esse código-fonte.

- Integrar bibliotecas de machine learning, como o scikit-learn, para gerar automaticamente a árvore a partir de dados reais e exportá-la em uma estrutura de nós semelhante.
- Outra melhoria seria implementar poda (pruning) para evitar overfitting, adicionando critérios de divisão mínimos e profundidade máxima.
- A visualização gráfica da árvore, por meio de bibliotecas como Graphviz, tornaria o modelo mais legível.
- Por fim, estender o classificador para múltiplos atributos e classes, bem como adicionar técnicas de validação cruzada, configuraria um exercício completo de produção de modelos de decisão em Python.

Para nosso segundo exercício, abordaremos novamente a blockchain que consolidou-se como uma tecnologia disruptiva ao oferecer um livro-razão distribuído, imutável e auditável, no qual as transações são agrupadas em blocos interligados por hashes criptográficos. Essa descentralização elimina a necessidade de intermediários de confiança, conferindo maior transparência e segurança às operações financeiras, de cadeias de suprimentos e inúmeras aplicações emergentes. No entanto, à medida que o volume de transações cresce, surge o desafio de garantir que cada bloco possa ser verificado de forma rápida e confiável, sem a necessidade de reprocessar todo o seu conteúdo.

Para responder a essa demanda, as Merkle Trees, ou árvores de Merkle, entram em cena como uma estrutura eficiente para resumir e validar grandes conjuntos de dados. Em vez de armazenar todos os hashes das transações diretamente no cabeçalho do bloco, construiremos uma árvore binária de hashes na qual cada folha representa o hash de uma transação e cada nó pai agrega os hashes dos filhos. Dessa maneira, basta conhecer o hash raiz – o Merkle Root – e um pequeno caminho de prova para confirmar a presença e a integridade de qualquer transação no bloco, tornando o processo de verificação extremamente ágil.

Observação

A Merkle Tree leva o nome de Ralph Merkle, um cientista da computação que, na década de 1970, propôs essa estrutura em sua pesquisa sobre criptografia. Ele imaginou uma forma de organizar dados de maneira segura e eficiente, usando hashes para criar uma árvore que pudesse verificar grandes quantidades de informações rapidamente.

Para entender melhor uma árvore de Merkle, imagine que você tem uma caixa cheia de papéis, cada um com uma informação importante, como uma transação financeira. Agora, você quer ter certeza de que todas essas informações estão corretas e que ninguém mexeu nelas, mas também quer uma forma prática de verificar tudo isso sem precisar olhar cada papel individualmente. É aqui que entra a árvore de Merkle.

Pense nela como uma espécie de árvore genealógica para organizar e proteger essas informações. Vamos supor que você começa com muitos papéis, cada um com um número ou dado. Primeiro, cada papel passa por um processo matemático chamado hash, que é como criar uma impressão digital única para aquele dado. Essa impressão digital é um código curto que representa o papel de forma segura: se o papel mudar, o código também muda. Agora, em vez de lidar com todos os papéis, você trabalha somente com esses códigos.

Aqui vem a parte da árvore: você pega esses códigos (hashes) e os organiza em pares. Cada par é combinado para criar um novo hash, como se fosse um pai na árvore genealógica. Esses novos hashes são agrupados em pares novamente, gerando outros hashes pais e assim por diante, até que se chega a um único hash no topo, chamado de raiz da Merkle Tree. Essa raiz é como um resumo de todos os papéis que você começou, mas de uma forma supercompacta.

Por que isso é útil? Primeiro, porque economiza espaço e tempo. Em vez de verificar cada papel, você apenas precisa checar a raiz para saber se tudo está correto. Se alguém tentar mudar um único papel, o hash daquele papel muda, o que altera os hashes dos pais acima dele, até chegar à raiz. Então, basta olhar a raiz para perceber que algo está errado. Além disso, a Merkle Tree permite verificar somente uma parte dos dados sem precisar olhar tudo. Por exemplo, se você quer confirmar uma transação específica, pode seguir o caminho dela na árvore até a raiz, usando poucos hashes, o que é muito rápido.

Em resumo, a Merkle Tree é uma forma inteligente de organizar e proteger muitas informações, garantindo que elas não foram alteradas e permitindo verificações rápidas e eficientes. É como ter uma calculadora mágica que resume uma pilha de papéis em um único número, mas ainda lhe deixa checar

qualquer detalhe sem esforço. Essa ideia é essencial em coisas como criptomoedas, nas quais milhares de transações precisam ser verificadas de forma segura e prática.

A utilização de Merkle Trees torna-se, portanto, essencial em sistemas de blockchain que demandam escalabilidade e respostas em tempo real. Clientes leves (light clients) podem confiar em um nó completo para obter apenas o Merkle Root e, em seguida, apresentar provas de inclusão sem baixar todas as transações. Esse mecanismo reduz drasticamente o consumo de largura de banda e o tempo de computação, ao mesmo tempo em que mantém a segurança e a confiança inerentes à tecnologia blockchain. No exercício a seguir, será demonstrado como implementar em Python uma Merkle Tree simples, gerar seu Merkle Root e produzir provas de inclusão que atestem a integridade de qualquer transação.

Exemplo de aplicação

Exemplo 2 – Blockchain

Suponha que você desenvolva um sistema de blockchain para uma rede de fornecedores de componentes eletrônicos. Cada novo bloco armazena centenas de transações de pedidos e entregas, e o Merkle Root é disponibilizado em um cabeçalho publicado por nós validadores.

Um auditor externo precisa verificar rapidamente que uma transação específica (por exemplo, um pedido de 100 resistores) faz parte de um bloco sem receber todos os detalhes das demais transações.

As regras de negócio determinam que, para cada bloco, o sistema deve gerar o Merkle Root e, a pedido, fornecer uma prova de inclusão para qualquer transação, composta por um vetor de hashes irmãos e as direções (esquerda/direita) correspondentes.

Caso um hash de transação seja alterado, a validação deve falhar, impedindo adulterações e garantindo a integridade dos registros.

Para construir uma Merkle Tree em Python, faremos uso do módulo hashlib, que implementa as funções de hash SHA256 requeridas pela maioria das blockchains. Cada transação, representada como string, deve ser convertida em bytes via encode('utf-8') antes de ser processada pelo objeto `hashlib.sha256`. O uso de funções para modularizar o código facilita a leitura e a manutenção: definiremos uma função para calcular o hash de um valor, outra para construir os níveis da árvore até obter a raiz e funções dedicadas à geração e à verificação de provas de inclusão.

Na construção da árvore, utilizaremos list comprehensions para agrupar pares consecutivos de hashes e combiná-los, duplicando o último elemento caso o número de nós seja ímpar. Esse padrão de emparelhamento é repetido em um laço `while` até restar um único hash, o Merkle Root. Para gerar a prova de inclusão, mantemos uma representação em camadas (níveis) da árvore e, a partir do índice da transação, coletamos em cada nível o hash irmão e a orientação (se o irmão está à esquerda ou à direita). Essa prova é um pequeno vetor cujo tamanho é proporcional ao logaritmo do número de transações, garantindo verificações rápidas.

Finalmente, a função de verificação recebe o hash da transação-alvo e o vetor de prova, recalcando iterativamente o hash combinado com cada elemento do caminho, na ordem correta, até reproduzir o Merkle Root fornecido. Se o valor final coincidir com o Merkle Root armazenado no cabeçalho do bloco, a transação é considerada válida. Esse processo ilustra como Python, com recursos nativos de manipulação de listas, laços de repetição, funções e o módulo hashlib, possibilita a implementação de Merkle Trees que atendem a requisitos de blockchain em cenários reais de mercado. A seguir, consta o código-fonte da solução.

```
import hashlib

def calcular_hash(valor):
    """
    Retorna o hash SHA-256 em hexadecimal do texto recebido.
    """
    return hashlib.sha256(valor.encode('utf-8')).hexdigest()

def construir_arvore_merkle(transacoes):
    """
    Constrói a Merkle Tree a partir da lista de transações.
    Retorna a lista de níveis, onde o nível 0 são os hashes das folhas,
    e o último nível contém apenas o Merkle Root.
    """
    niveis = [[calcular_hash(tx) for tx in transacoes]]
    while len(niveis[-1]) > 1:
        nivel_atual = niveis[-1]
        proximo_nivel = []
        for i in range(0, len(nivel_atual), 2):
            esquerda = nivel_atual[i]
            direita = nivel_atual[i+1] if i+1 < len(nivel_atual) else esquerda
            combinado = calcular_hash(esquerda + direita)
            proximo_nivel.append(combinado)
        niveis.append(proximo_nivel)
    return niveis

def obter_raiz_merkle(niveis):
    """
    Retorna o Merkle Root, que é o único elemento do último nível.
    """
    return niveis[-1][0]

def gerar_prova_inclusao(niveis, indice):
    """
    Gera a prova de inclusão para a transação no índice informado.
    Retorna uma lista de tuplas (hash_irmão, lado) desde o nível 0 até um nível
antes da raiz.
    'lado' é 'esquerda' se o irmão estiver antes, ou 'direita' caso contrário.
    """
    prova = []
    for nivel in niveis[:-1]:
        tamanho = len(nivel)
        if indice % 2 == 0: # par, irmão é o próximo
            irmao_idx = indice+1 if indice+1 < tamanho else indice
            lado = 'direita'
        else: # ímpar, irmão é o anterior
```

```
                irmao_idx = indice-1
                lado = 'esquerda'
            prova.append((nivel[irmao_idx], lado))
            indice //= 2
    return prova

def verificar_prova_inclusao(hash_tx, prova, raiz_esperada):
    """
    Verifica a prova de inclusão. Recalcula hashes seguindo o caminho da prova
    e compara o resultado final com o Merkle Root esperado.
    """
    hash_atual = hash_tx
    for irmao_hash, lado in prova:
        if lado == 'direita':
            hash_atual = calcular_hash(hash_atual + irmao_hash)
        else:
            hash_atual = calcular_hash(irmao_hash + hash_atual)
    return hash_atual == raiz_esperada

def principal():
    transacoes = [
        "pedido#1001:10 resistores",
        "pedido#1002:5 capacitores",
        "pedido#1003:2 microcontroladores",
        "pedido#1004:20 LEDs",
        "pedido#1005:1 placa-mãe"
    ]
    niveis = construir_arvore_merkle(transacoes)
    raiz = obter_raiz_merkle(niveis)
    print(f"Merkle Root: {raiz}\n")

    # Escolhe uma transação para gerar e verificar prova
    idx = 2
    hash_alvo = calcular_hash(transacoes[idx])
    prova = gerar_prova_inclusao(niveis, idx)
    valido = verificar_prova_inclusao(hash_alvo, prova, raiz)
    print(f"Prova para transação '{transacoes[idx]}' é válida? {valido}")

    # Simula adulteração de uma transação
    transacoes_tampered = transacoes.copy()
    transacoes_tampered[idx] = "pedido#1003:99 microcontroladores"
    niveis_tampered = construir_arvore_merkle(transacoes_tampered)
    raiz_tampered = obter_raiz_merkle(niveis_tampered)
    valido_tampered = verificar_prova_inclusao(hash_alvo, prova, raiz_tampered)
    print(f"Após adulteração, prova continua válida? {valido_tampered}")

if __name__ == "__main__":
    principal()
```

O quadro 10 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições. As funções já explicitadas nos quadros anteriores foram omitidas para evitar duplicidade.

Quadro 10 – Funções usadas no código-fonte do segundo exercício prático

Função ou construção	Explicação de uso
import hashlib	Importa o módulo hashlib, que fornece funções para calcular hashes criptográficos como SHA-256
hashlib.sha256(objeto).hexdigest()	Calcula o hash SHA-256 do objeto fornecido (normalmente em bytes) e retorna o valor como uma string hexadecimal
str.encode('utf-8')	Converte uma string para bytes, necessários como entrada para funções de hashing
list.copy()	Retorna uma cópia superficial da lista original, permitindo modificações sem afetar a lista original
indice //= 2	Atualiza a variável índice com o resultado da divisão inteira por 2. Essa operação é usada para navegar da base até a raiz da árvore binária

Observação

O hash SHA-256, ou Secure Hash Algorithm 256-bit, é uma função criptográfica de hash pertencente à família SHA-2, desenvolvida pela Agência de Segurança Nacional dos Estados Unidos (NSA). Ele transforma uma entrada de dados de tamanho arbitrário, como um texto, arquivo ou transação, em uma sequência fixa de 256 bits (32 bytes), representada geralmente como 64 caracteres hexadecimais. Essa transformação é unidirecional, ou seja, não é possível reverter o hash para recuperar os dados originais, e possui propriedades que o tornam essencial em sistemas de segurança, como blockchains e verificação de integridade.

A função SHA-256 opera dividindo a entrada em blocos e aplicando uma série de operações matemáticas, incluindo deslocamentos de bits, funções lógicas e adições modulares, para produzir o hash. Uma característica fundamental é sua determinística: a mesma entrada sempre gera o mesmo hash. Além disso, pequenas alterações na entrada resultam em hashes completamente diferentes, o que garante sensibilidade a mudanças. A função também é projetada para ser resistente a colisões, significando que é extremamente improvável que duas entradas diferentes produzam o mesmo hash.

No contexto do código fornecido, o SHA-256 é usado para calcular os hashes das transações e combinar pares de hashes na construção da árvore de Merkle. Essa escolha é comum em sistemas distribuídos, pois o SHA-256 oferece um equilíbrio entre segurança, eficiência computacional e tamanho fixo do resultado, facilitando a verificação de grandes volumes de dados de forma compacta e confiável.

O código apresentado utiliza a biblioteca `hashlib` do Python para calcular hashes SHA-256 e organiza as funções de maneira modular, cada uma com uma responsabilidade específica no processo de construção, prova e verificação da árvore. A primeira função, `calcular_hash`, é responsável por gerar o hash SHA-256 de um texto fornecido como entrada. Ela recebe um valor, codifica-o no formato UTF-8 (necessário para processar strings em funções de hash) e retorna o hash em formato hexadecimal. Essa função serve como base para todas as operações subsequentes, pois os nós da árvore de Merkle são construídos a partir de hashes de transações ou combinações de hashes.

A função `construir_arvore_merkle` cria a árvore de Merkle a partir de uma lista de transações. Inicialmente, ela calcula o hash de cada transação, formando o nível mais baixo da árvore (nível 0), que contém as folhas. Em seguida, entra em um loop que constrói níveis superiores, combinando pares de hashes do nível atual para formar um novo hash no próximo nível. Para cada par, os hashes são concatenados e processados pela função `calcular_hash`. Caso o número de hashes em um nível seja ímpar, o último hash é duplicado para formar um par. O processo continua até que reste apenas um hash, conhecido como Merkle Root, que representa o topo da árvore. A função retorna uma lista contendo todos os níveis da árvore, desde as folhas até a raiz.

A função `obter_raiz_merkle` é simples, extraindo o único elemento do último nível da lista retornada por `construir_arvore_merkle`. Esse elemento é o Merkle Root, que resume todas as transações da árvore em um único valor. Ele é crucial para verificar a integridade do conjunto de transações.

A função `gerar_prova_inclusao` gera uma prova de inclusão para uma transação específica, identificada por seu índice na lista original. A prova consiste em uma lista de tuplas, cada uma contendo o hash de um irmão (o outro nó do par no mesmo nível) e a indicação de sua posição relativa (esquerda ou direita). Para construir a prova, a função percorre os níveis da árvore, começando pelo nível das folhas. Em cada nível, ela determina se o índice da transação é par ou ímpar para identificar o irmão correspondente. Se o índice é par, o irmão está à direita (próximo índice, ou o mesmo, se não houver próximo); se ímpar, está à esquerda (índice anterior). O índice é então dividido por 2 para subir ao próximo nível, refletindo a estrutura binária da árvore. A prova inclui apenas os hashes necessários para recalcular o caminho até a raiz, exceto o próprio Merkle Root.

A função `verificar_prova_inclusao` valida a prova de inclusão. Ela recebe o hash da transação-alvo, a prova gerada e o Merkle Root esperado. Começando com o hash da transação, a função recalcula os hashes subindo a árvore: para cada tupla na prova, ela combina o hash atual com o hash do irmão, respeitando a ordem indicada (esquerda ou direita). Se o irmão está à esquerda, seu hash vem antes; caso contrário, depois. O resultado final é comparado com o Merkle Root esperado, retornando `True` se forem iguais, indicando que a transação está incluída na árvore, ou `False`, caso contrário.

A função `principal` demonstra o uso dessas funções. Ela define uma lista de cinco transações fictícias, representando pedidos de componentes eletrônicos. Primeiro, constrói a árvore de Merkle chamando `construir_arvore_merkle` e obtém o Merkle Root com `obter_raiz_merkle`, exibindo-o. Em seguida, seleciona a transação de índice 2 para teste, calcula seu hash, gera a prova de inclusão com `gerar_prova_inclusao` e verifica a prova com `verificar_prova_inclusao`, imprimindo se a prova é válida. Para simular uma adulteração, cria uma cópia da lista de transações,

altera a transação de índice 2 (modificando a quantidade de microcontroladores) e reconstrói a árvore. A prova original é então verificada contra o novo Merkle Root, que, devido à alteração, resulta em uma validação falsa, demonstrando a sensibilidade da árvore a mudanças nos dados.

Lembrete

O código é executado quando o script é chamado diretamente, por meio da condição `if __name__ == "__main__"`, que invoca a função principal.

O programa ilustra de forma clara o conceito de árvores de Merkle, mostrando como elas garantem a integridade e permitem a verificação eficiente de inclusão de transações, propriedades fundamentais em sistemas como blockchains.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Integrar estruturas que otimizem a construção da árvore em paralelo, utilizando bibliotecas como multiprocessing para tratar grandes volumes de transações de forma mais rápida.

- A persistência dos níveis da árvore em um banco de dados NoSQL permitiria consultas históricas de Merkle Roots em diferentes blocos.

- Implementar uma interface de rede que responda a requisições de provas de inclusão via API REST tornaria o sistema utilizável por clientes leves, enquanto a adoção de árvores de Merkle a níveis múltiplos (Merkle Patricia Trees) aproximaria ainda mais a prática das blockchains de contrato inteligente mais sofisticadas, como as utilizadas em plataformas de criptomoedas do mercado.

Em nosso terceiro exercício, abordaremos os sistemas de arquivos que são a espinha dorsal de qualquer sistema operacional ou serviço de nuvem, pois definem a forma como os dados são organizados, armazenados e recuperados. Em ambientes locais, como Windows, Linux ou macOS, os arquivos são estruturados em diretórios aninhados, criando uma hierarquia que permite ao usuário navegar por pastas, agrupar documentos relacionados e controlar permissões de acesso. Na nuvem, plataformas como AWS S3, Google Cloud Storage e Azure Blob Storage reproduzem esse conceito de hierarquia – embora muitas vezes virtualmente – para oferecer escalabilidade, redundância e acesso global aos dados.

A organização hierárquica de diretórios e arquivos facilita a gestão manual de documentos, bem como a automação de rotinas de backup, replicação e sincronização entre diferentes ambientes. Quando se tem milhares ou milhões de arquivos, uma árvore de diretórios bem estruturada reduz drasticamente o tempo de busca, melhora o desempenho de operações em massa e torna possível aplicar políticas de retenção e versionamento

de forma seletiva. Além disso, essa estrutura permite implementar controles de segurança granulares, definindo quem pode ler, gravar ou executar cada pasta e arquivo em nível de granularidade fino.

Em serviços de nuvem, essa hierarquia assume ainda mais importância, pois muitas aplicações consomem e produzem dados de forma distribuída. APIs de armazenamento expõem métodos que aceitam caminhos para operações de upload, download e listagem de objetos, espelhando a navegação em diretórios locais. Compreender como representar essa hierarquia em memória e como percorrê-la – seja para exibir uma árvore de navegação em uma interface web, seja para calcular o espaço ocupado por cada subpasta – é uma habilidade valiosa para desenvolvedores que constroem soluções de gestão de arquivos, migração de dados ou ferramentas de monitoramento de uso de armazenamento.

Exemplo de aplicação

Exemplo 3 – Sistemas de arquivos

Imagine um serviço de backup em nuvem que sincroniza automaticamente o conteúdo de uma pasta local com um repositório remoto. Para apresentar ao usuário um painel de navegação, o sistema mantém em memória uma estrutura de árvore que espelha a organização de diretórios e arquivos.

As regras de negócio definem que, ao iniciar a aplicação, deve-se construir essa árvore a partir de uma lista de caminhos fornecida, preservando a ordem de criação.

Em seguida, o sistema precisa oferecer três modos de travessia para alimentar diferentes componentes da interface: pré-ordem (listar o diretório antes de seus filhos), em ordem (listar metade dos filhos, depois o diretório, depois o restante) e pós-ordem (listar primeiro todos os filhos, depois o diretório).

Para que o usuário possa rapidamente identificar onde um arquivo específico está localizado, o sistema também deve permitir a busca linear pelo nome, percorrendo a árvore em pré-ordem e retornando o caminho completo ao encontrar o arquivo.

Para representar a hierarquia em Python, utilizaremos uma classe `NoArquivo` que encapsula o nome do elemento, um indicador de tipo (diretório ou arquivo) e uma lista de filhos. A definição de classes em Python envolve o método `__init__`, que inicializa esses atributos, e a capacidade de armazenar referências a outros objetos da mesma classe em `self.filhos`, criando uma árvore de nós interligados. Esse padrão de composição é amplamente empregado em ferramentas de geração de árvores de diretórios e exploradores de arquivos.

O percurso da árvore em diferentes ordens será alcançado por meio de funções recursivas. Como vimos, a recursão é um recurso poderoso em Python que permite que uma função invoque a si mesma para processar cada subárvore, simplificando o código e evitando a complexidade de loops aninhados. Para a travessia em ordem adaptada a um nó com vários filhos, dividiremos a lista de filhos ao meio, chamando a função sobre a primeira metade, processando o próprio nó (adicionando seu nome ao resultado) e chamando a função sobre a segunda metade. List comprehensions serão empregadas para

coletar resultados parciais de cada nível de recursão em listas planas, tornando o código mais conciso e expressivo.

Além disso, usaremos recursos nativos de Python como o método `append` para adicionar strings a listas de resultado e a função `join` para construir caminhos completos ao exibir a localização de um arquivo encontrado. A busca linear, necessária para localizar um arquivo por nome, combinará recursão com estruturas de controle (`if` e `else`) para verificar cada nó em pré-ordem e retornar, ao primeiro encontro, a lista de diretórios que leva até ele. Dessa forma, o aluno compreenderá como integrar classes, recursão e manipulação de listas em Python para resolver problemas práticos de organização e navegação de sistemas de arquivos. A seguir, consta o código-fonte da solução:

```
class NoArquivo:
    def __init__(self, nome, tipo):
        self.nome = nome  # Nome do diretório ou arquivo
        self.tipo = tipo  # "diretorio" ou "arquivo"
        self.filhos = []  # Lista de nós filhos

    def adicionar_filho(self, filho):
        self.filhos.append(filho)
        return self

def construir_arvore(caminhos):
    raiz = NoArquivo("root", "diretorio")
    for caminho in caminhos:
        partes = caminho.strip("/").split("/")
        atual = raiz
        for parte in partes:
            # verifica se já existe um nó com esse nome
            encontrado = next((n for n in atual.filhos if n.nome == parte),
None)
            if not encontrado:
               tipo = "diretorio" if parte.find(".") < 0 else "arquivo"
               encontrado = NoArquivo(parte, tipo)
               atual.adicionar_filho(encontrado)
        atual = encontrado
    return raiz

def pre_ordem(no, resultado=None):
    if resultado is None:
        resultado = []
    resultado.append(f"{no.tipo}:{no.nome}")
    for filho in no.filhos:
        pre_ordem(filho, resultado)
    return resultado

def em_ordem(no, resultado=None):
    if resultado is None:
        resultado = []
    m = len(no.filhos) // 2
    for filho in no.filhos[:m]:
        em_ordem(filho, resultado)
    resultado.append(f"{no.tipo}:{no.nome}")
    for filho in no.filhos[m:]:
```

```
            em_ordem(filho, resultado)
    return resultado


def pos_ordem(no, resultado=None):
    if resultado is None:
        resultado = []
    for filho in no.filhos:
        pos_ordem(filho, resultado)
    resultado.append(f"{no.tipo}:{no.nome}")
    return resultado

def buscar_arquivo(no, nome_alvo, caminho=None):
    if caminho is None:
        caminho = []
    caminho_atual = caminho + [no.nome]
    if no.tipo == "arquivo" and no.nome == nome_alvo:
        return "/".join(caminho_atual)
    for filho in no.filhos:
        achado = buscar_arquivo(filho, nome_alvo, caminho_atual)
        if achado:
            return achado
    return None

def principal():
    caminhos = [
        "docs/manual.pdf",
        "docs/guia/instalacao.txt",
        "imagens/logo.png",
        "imagens/festas/ano_novo.jpg",
        "backup/2025/janeiro/data.bak"
    ]
    arvore = construir_arvore(caminhos)

    print("Travessia pré-ordem:")
    print(pre_ordem(arvore))

    print("\nTravessia em-ordem:")
    print(em_ordem(arvore))

    print("\nTravessia pós-ordem:")
    print(pos_ordem(arvore))

    arquivo_busca = "data.bak"
    caminho_encontrado = buscar_arquivo(arvore, arquivo_busca)
    print(f"\nCaminho do arquivo '{arquivo_busca}': {caminho_encontrado}")

if __name__ == "__main__":
    principal()
```

O quadro 11 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições. As funções já explicitadas nos quadros anteriores foram omitidas para evitar duplicidade.

Quadro 11 – Funções usadas no código-fonte do terceiro exercício prático

Função ou construção	Explicação de uso
str.strip(caractere)	Remove caracteres do início e do fim da string. No código, remove barras / que envolvem o caminho antes da separação
str.split(separador)	Divide uma string em partes, com base em um separador. Aqui, é usada para separar os diretórios e os arquivos com base nas barras /
next(iterador, valor_padrao)	Retorna o próximo elemento de um iterador. Se o iterador estiver esgotado, retorna o valor padrão fornecido. No código, é usado para localizar rapidamente um filho com nome correspondente
str.find(substring)	Retorna o índice da primeira ocorrência da substring. Retorna -1 se a substring não for encontrada. No código, identifica se o nome possui ponto (.), sinalizando que se trata de um arquivo
lista + [elemento]	Cria uma lista concatenando um elemento a uma lista existente. Usado para construir `caminho_atual` de forma acumulativa
"/".join(lista_de_strings)	Junta os elementos de uma lista de strings em uma única string, separando-os por barras. Utilizado para formatar o caminho completo de um arquivo
lista[:m] / lista[m:]	Fatiamento de listas. Retorna uma sublista contendo os elementos antes ou depois do índice m. No código, é utilizado na travessia em ordem para visitar metade dos filhos antes e metade depois de visitar o nó

O código Python apresentado implementa uma estrutura de dados em forma de árvore para representar um sistema de arquivos, utilizando recursos da linguagem como classes, listas, recursão, manipulação de strings e compreensão de listas. Ele define uma classe `NoArquivo` que modela nós da árvore, representando arquivos ou diretórios, e funções para construir a árvore a partir de caminhos, percorrê-la em diferentes ordens (pré-ordem, em ordem e pós-ordem) e buscar arquivos específicos. A execução ocorre na função `principal`, que demonstra o uso dessas funcionalidades.

A classe `NoArquivo` é definida com um construtor `__init__` que recebe os parâmetros `nome` e `tipo`, inicializando os atributos `self.nome` (nome do arquivo ou diretório), `self.tipo` (indicando se é arquivo ou diretorio) e `self.filhos` (uma lista vazia para armazenar nós filhos). O método `adicionar_filho` adiciona um nó à lista `filhos` e retorna o próprio objeto, permitindo encadeamento de chamadas, um padrão comum em Python para métodos que modificam o estado do objeto.

A função `construir_arvore` cria uma árvore a partir de uma lista de caminhos. Ela inicializa um nó raiz com nome `root` e tipo `diretorio`. Para cada caminho, o código remove barras iniciais ou finais com `strip("/")` e divide o caminho em partes usando `split("/")`, gerando uma lista de nomes de diretórios e arquivos. Iterando sobre essas partes, o código verifica se já existe um nó filho com o nome atual usando a função `next` combinada com uma expressão geradora, que retorna o primeiro nó cujo nome coincide ou `None` se não houver correspondência. Se o nó não existe, o código determina seu tipo verificando a presença de um ponto no nome (indicando um arquivo se presente, ou diretório se ausente) e cria um `NoArquivo`, que é adicionado como filho do nó atual. O nó atual é então atualizado para o nó encontrado ou recém-criado, permitindo a construção da hierarquia.

As funções `pre_ordem`, `em_ordem` e `pos_ordem` implementam travessias recursivas da árvore. Todas recebem um nó e uma lista opcional `resultado`, inicializada como vazia se não fornecida, um

padrão em Python para evitar problemas com mutabilidade de listas em chamadas recursivas. A função `pre_ordem` adiciona o tipo e o nome do nó atual ao resultado antes de percorrer recursivamente os filhos, seguindo a lógica de visitar a raiz antes das subárvores. A função `em_ordem` divide os filhos em duas metades, processando recursivamente a primeira metade, adicionando o nó atual e depois processando a segunda metade, o que reflete a ordem de visitar subárvores esquerdas, raiz e subárvores direitas (embora adaptada para múltiplos filhos). A função `pos_ordem` percorre todos os filhos recursivamente antes de adicionar o nó atual, visitando as subárvores antes da raiz. Cada função retorna a lista `resultado` com strings no formato "tipo:nome".

A função `buscar_arquivo` realiza uma busca recursiva por um arquivo com nome específico, construindo o caminho completo. Ela mantém uma lista `caminho` que acumula os nomes dos nós visitados. Se o nó atual for um arquivo e seu nome corresponde ao alvo, a função junta os nomes do caminho com `"/".join()` e retorna o resultado. Caso contrário, ela chama a si mesma para cada filho, passando o caminho atualizado, e retorna o primeiro caminho válido encontrado ou `None` se não houver correspondência. A recursão é usada para explorar toda a árvore, um recurso poderoso em Python para problemas hierárquicos.

A função `principal` demonstra o uso do código. Ela define uma lista de caminhos representando um sistema de arquivos, como "`docs/manual.pdf`" e "`imagens/festas/ano_novo.jpg`". Esses caminhos são passados para `construir_arvore`, que cria a árvore. Em seguida, as travessias são executadas e os resultados são impressos usando `print`. Cada travessia retorna uma lista de strings que descreve a ordem de visitação dos nós. Por fim, a função busca o arquivo "`data.bak`" usando `buscar_arquivo` e imprime seu caminho completo. O bloco `if __name__ == "__main__":` garante que `principal` seja executada apenas se o script for o programa principal, um padrão em Python para modularidade.

O código utiliza recursos como manipulação de strings (`strip`, `split`, `join`), expressões geradoras, recursão, classes e métodos de instância, além de estruturas de dados como listas. A lógica é clara e modular, aproveitando a flexibilidade da linguagem para modelar uma estrutura complexa como uma árvore de arquivos de forma eficiente.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Integrar o módulo os ou pathlib para construir a árvore a partir de um diretório real do sistema operacional, em vez de uma lista estática de caminhos.

- Implementar tratamento de exceções ao acessar caminhos não existentes e permissões negadas aumentaria a confiabilidade.

- Em cenários de alta escala, o uso de caching de resultados de travessia e busca melhoraria o desempenho, enquanto a adição de atributos como tamanho de arquivo e data de modificação em cada nó permitiria cálculos de uso de armazenamento e relatórios históricos.

- Por fim, a exposição dessa árvore por meio de uma API RESTful, utilizando frameworks como Flask ou FastAPI, viabilizaria a construção de interfaces web que exibam a estrutura em tempo real e aceitem comandos de criação, remoção e movimentação de arquivos e diretórios.

3.2 Grafos: representação (matriz de adjacência e listas de adjacência) e travessias (DFS e BFS)

Como acabamos de ver, uma árvore é uma estrutura organizada que começa com um ponto inicial, chamado raiz, no qual cada elemento (nó) se conecta a exatamente um nó pai acima, formando uma hierarquia sem caminhos que voltem ao mesmo ponto (sem loops). Como em uma árvore genealógica, cada pessoa está ligada a um ancestral direto, sem conexões cruzadas complicadas.

Por outro lado, um grafo é uma estrutura mais ampla e flexível, na qual os elementos (chamados vértices) podem se conectar de qualquer jeito, permitindo caminhos que formem loops, conexões com pesos (como distâncias) ou direções (como ruas de mão única). Em resumo, a árvore é um tipo específico de grafo: uma estrutura hierárquica e rígida sem loops, enquanto o grafo é mais livre, permitindo conexões variadas e complexas. Em outras palavras, toda árvore é um grafo, mas nem todo grafo é uma árvore.

De modo simplificado, um grafo pode ser entendido como uma forma de representar conexões entre coisas. Imagine um mapa de amizades: cada pessoa é um ponto (vértice) e as linhas que ligam as pessoas que são amigas são as conexões (arestas). Um grafo é exatamente isto: um conjunto de pontos conectados por linhas, que mostram como esses pontos se relacionam. Essas conexões podem representar qualquer coisa, como ruas entre cidades, links entre páginas da internet ou até tarefas que dependem umas das outras.

Por exemplo, pense em um grupo de amigos. João é amigo de Maria, esta é amiga de Ana, a qual é amiga de Pedro. Um grafo pode mostrar essas relações com círculos (amigos) e linhas (amizades), conforme a figura a seguir:

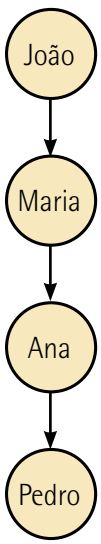

Figura 13 – Exemplo de grafo

Neste diagrama, cada nome é um vértice e as setas mostram as conexões (amizades). As setas podem ser bidirecionais (se a amizade é mútua) ou unidirecionais, dependendo do contexto. Note também que o grafo pode ser mais complexo como o da figura a seguir:

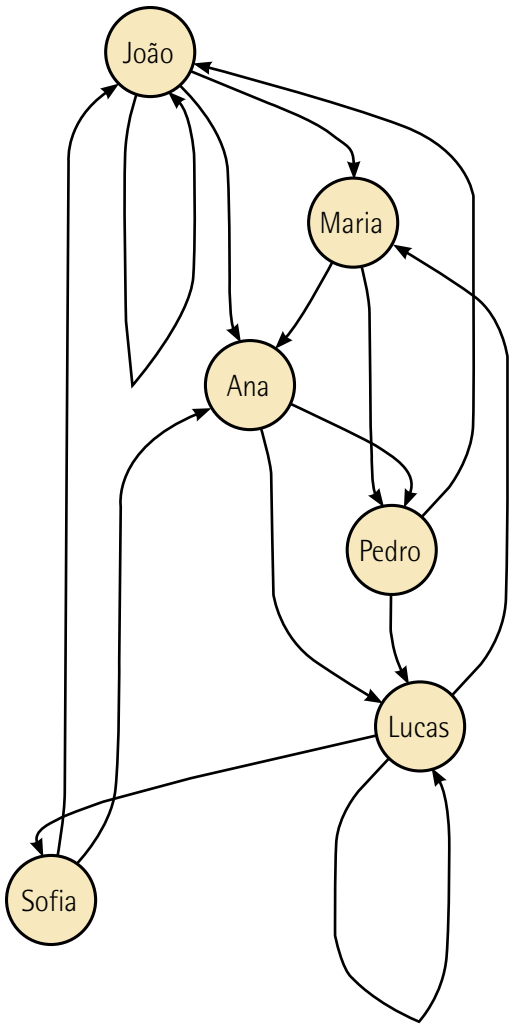

Figura 14 – Grafo mais complexo

Grafos aparecem em muitas aplicações no dia a dia. Um exemplo comum é em GPSs (Global Positioning System), como Google Maps ou Waze. As cidades ou cruzamentos são vértices e as ruas são arestas. O GPS usa grafos para calcular o caminho mais curto até o seu destino. Outro exemplo está nas redes sociais, como Facebook ou Instagram, nos quais as pessoas são vértices e as conexões de amizade ou seguidores são arestas. Essas plataformas usam grafos para sugerir novos amigos com base em quem você já conhece. Além disso, grafos são usados em logística, como na entrega de pacotes, para encontrar as rotas mais eficientes entre depósitos e clientes, e até em jogos, para mapear caminhos que um personagem pode seguir.

Em resumo, grafos são como mapas de conexões que ajudam a resolver problemas práticos, desde encontrar o melhor caminho até sugerir amigos online. Eles são uma ferramenta poderosa, mas simples de entender, como um desenho de pontos e linhas que representam relações.

A representação e a manipulação de grafos são fundamentais em ciência da computação, especialmente em problemas que envolvem relações entre entidades. Um grafo pode ser representado de diferentes maneiras, sendo as mais comuns a matriz de adjacência e a lista de adjacência.

Além disso, algoritmos de travessia, como a DFS e a BFS, permitem explorar vértices e arestas de um grafo de forma sistemática, sendo amplamente utilizados em aplicações práticas. Em Python, essas estruturas e algoritmos podem ser implementados de maneira clara e eficiente, aproveitando a flexibilidade da linguagem. A figura a seguir compara visualmente as duas buscas em um mesmo grafo.

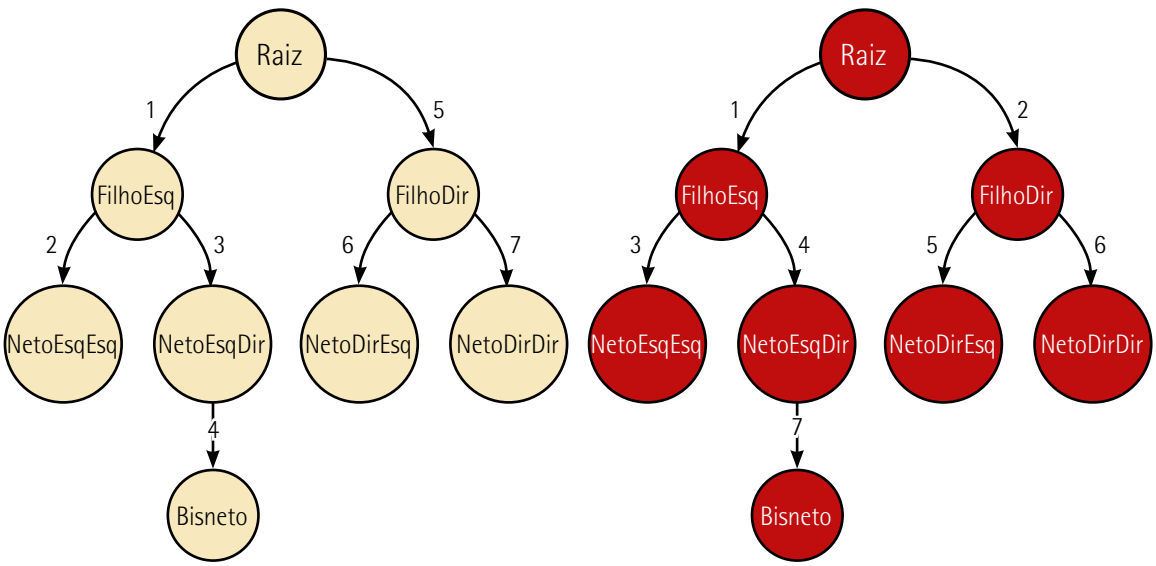

Figura 15 – Comparação entre DFS (à esquerda) e BFS (à direita). Os números indicam a ordem do percurso

A matriz de adjacência é uma representação que utiliza uma matriz quadrada para indicar as conexões entre vértices. Para um grafo com n vértices, cria-se uma matriz n x n, na qual o elemento na posição [i][j] indica a existência de uma aresta entre os vértices i e j. Em grafos não ponderados, esse elemento pode ser 1 (se há aresta) ou 0 (se não há). Consta na sequência um exemplo de matriz de adjacência não ponderada para o grafo da tabela a seguir.

Tabela 1 – Matriz de adjacência para o grafo da figura 15. Nela, "R" é o vértice Raiz, "FE" é o FilhoEsq, "FD" é o FilhoDir, "NEE" é NetoEsqEsq etc. O número 1 indica que há conexão entre os dois vértices

	R	FE	FD	NEE	NED	NDE	NDD	B
R	0	1	1	0	0	0	0	0
F	1	0	0	1	1	0	0	0
FD	1	0	0	0	0	1	1	0
NEE	0	1	0	0	0	0	0	0
NED	0	1	0	0	0	0	0	1
NDE	0	0	1	0	0	0	0	0
NDD	0	0	1	0	0	0	0	0
B	0	0	0	0	1	0	0	0

Em grafos ponderados, o valor pode representar o peso da aresta. Essa estrutura é vantajosa por sua simplicidade e acesso rápido às conexões, mas consome espaço de memória proporcional a n^2, o que a torna ineficiente para grafos esparsos, nos quais o número de arestas é muito menor que o máximo possível. Em Python, uma matriz de adjacência pode ser implementada como uma lista de listas, inicializada com zeros e preenchida conforme as arestas do grafo.

Por outro lado, a lista de adjacência oferece uma representação mais eficiente para grafos esparsos. Nessa abordagem, cada vértice está associado a uma lista que contém os vértices adjacentes a ele. Em Python, isso pode ser implementado usando um dicionário, no qual as chaves são os vértices e os valores são listas dos vértices vizinhos. Para grafos ponderados, as listas podem armazenar tuplas contendo o vértice vizinho e o peso da aresta. A lista de adjacência ocupa menos espaço em grafos com poucas arestas, pois armazena apenas as conexões existentes e permite iterações rápidas sobre os vizinhos de um vértice. Contudo, verificar a existência de uma aresta específica pode ser mais lento, exigindo uma busca na lista correspondente.

As travessias de grafos, como DFS e BFS, são usadas para visitar todos os vértices acessíveis a partir de um ponto inicial, sendo úteis em problemas como busca de caminhos, detecção de ciclos e análise de conectividade. A DFS explora o grafo seguindo um caminho o mais longe possível antes de retroceder. Em Python, ela pode ser implementada de forma recursiva ou iterativa, usando uma pilha. Na abordagem recursiva, a função chama a si mesma para cada vizinho não visitado, marcando os vértices visitados em um conjunto para evitar ciclos. A DFS é particularmente útil para encontrar componentes conectados ou caminhos em labirintos, mas sua natureza profunda pode levar a um uso significativo de memória em grafos muito profundos.

A BFS, por sua vez, explora o grafo nível por nível, visitando todos os vizinhos de um vértice antes de prosseguir para os vizinhos dos vizinhos. Em Python, a BFS é implementada usando uma fila, que armazena os vértices a serem explorados. Um conjunto de vértices visitados é mantido para evitar revisitas. A BFS é ideal para encontrar o menor caminho em grafos não ponderados, pois garante que os vértices são visitados na ordem de sua distância ao vértice inicial. Sua implementação é iterativa, o que a torna menos suscetível a problemas de estouro de pilha em comparação com a DFS recursiva.

Para ilustrar, considera-se a implementação de um grafo em Python. Uma matriz de adjacência pode ser criada como uma lista de listas, com uma função para adicionar arestas atualizando os elementos correspondentes. Para a lista de adjacência, um dicionário pode ser usado, com uma função que adiciona vértices vizinhos às listas. A DFS pode ser codificada com uma função recursiva que percorre os vizinhos de um vértice, enquanto a BFS pode usar a biblioteca `collections.deque` para gerenciar a fila de exploração. Ambas as travessias exigem um mecanismo para rastrear vértices visitados, geralmente um conjunto ou uma lista.

A escolha entre DFS e BFS ao trabalhar com grafos depende intrinsecamente da natureza do problema e das características do grafo em questão. Cada algoritmo apresenta vantagens específicas que o tornam mais adequado para determinados cenários e compreender essas particularidades é essencial para uma decisão informada.

Quando o objetivo é encontrar o caminho mais curto entre dois pontos em um grafo não ponderado, a busca em largura revela-se a escolha ideal. Isso ocorre porque a BFS explora os nós de forma nivelada, visitando todos os vizinhos de um nível antes de avançar para o próximo. Assim, garante que o primeiro caminho encontrado para um nó seja o de menor distância, medido em número de arestas. Essa propriedade é particularmente útil em problemas como determinar o menor número de movimentos em um labirinto ou calcular o grau de separação em uma rede social. Além disso, a BFS é vantajoso quando a solução está próxima do ponto de partida, especialmente em grafos amplos, pois evita explorar caminhos longos desnecessariamente. Também se destaca em situações que envolvem grafos infinitos ou muito profundos, já que sua abordagem nivelada impede que o algoritmo se perca em ramificações extensas. Contudo, o custo de memória da BFS pode ser elevado, pois ele armazena todos os nós de um nível na fila, o que talvez seja problemático em grafos com muitos vértices por nível.

Por outro lado, a busca em profundidade é mais apropriada quando o problema exige explorar todos os caminhos possíveis ou quando a solução está potencialmente localizada em um ramo profundo do grafo. A DFS segue um caminho até sua conclusão antes de retroceder, o que a torna ideal para tarefas que envolvem backtracking, como resolver quebra-cabeças ou encontrar todos os caminhos entre dois nós. Sua eficiência é notória em grafos profundos e estreitos, nos quais o uso de memória é reduzido, já que apenas o caminho atual é armazenado na pilha de recursão. Além disso, a DFS é particularmente eficaz para detectar ciclos ou identificar componentes conectadas em um grafo, sendo uma ferramenta valiosa em análises estruturais. Quando a memória é uma restrição significativa, a DFS também se sobressai, pois consome menos espaço em comparação com a BFS, especialmente em grafos com alta profundidade.

Ambos os algoritmos apresentam complexidade de tempo semelhante, processando todos os vértices e as arestas do grafo, mas diferem significativamente no uso de espaço e na ordem de visitação dos nós. Enquanto a BFS prioriza a proximidade e a exaustividade em cada nível, a DFS favorece a exploração profunda e a flexibilidade para problemas que exigem uma análise exaustiva de caminhos. Em situações que envolvem grafos ponderados, no entanto, nenhum dos dois é ideal para encontrar caminhos mais curtos, sendo necessário recorrer a algoritmos como Dijkstra ou Bellman-Ford, que consideram os pesos das arestas. Veremos esses algoritmos em detalhes no tópico 7.

Portanto, a decisão entre DFS e BFS deve ser guiada por uma análise cuidadosa do problema. Se a meta é minimizar distâncias ou explorar grafos amplos de forma sistemática, a BFS é a escolha natural. Já para problemas que demandam exploração profunda, detecção de ciclos ou lidam com restrições de memória, a DFS torna-se mais adequada. Essa escolha reflete não apenas as propriedades do grafo, mas as prioridades do contexto em que o algoritmo será aplicado.

Em resumo, a escolha entre matriz de adjacência e lista de adjacência depende das características do grafo e das operações necessárias, enquanto DFS e BFS são selecionadas com base no objetivo da travessia. Python facilita a implementação dessas estruturas e algoritmos, oferecendo ferramentas que equilibram clareza e eficiência. Essas técnicas formam a base para resolver problemas complexos em áreas como redes, inteligência artificial e análise de dados.

Para ilustrar melhor esses conceitos, faremos um exercício prático. A robótica e o desenvolvimento de veículos autônomos avançaram consideravelmente na última década, impulsionados por sensores

mais precisos, poder computacional elevado e algoritmos sofisticados de inteligência artificial. Esses sistemas precisam interpretar o ambiente, planejar trajetórias e se deslocar com segurança, em tempo real, por espaços complexos. Nessa conjuntura, a modelagem de mapas e rotas adquire papel central, pois fornece a abstração necessária para representar cruzamentos, vias e possíveis obstáculos. Sem uma estrutura de dados que descreva claramente cada ponto do percurso e suas conexões, torna-se inviável calcular trajetórias eficientes ou garantir a chegada ao destino.

A representação do ambiente em forma de grafo se revela especialmente adequada, já que vértices podem corresponder a interseções ou marcos de referência, e arestas descrevem vias livres para circulação. Sobre essa estrutura, algoritmos clássicos de travessia como DFS e BFS permitem explorar caminhos, detectar bloqueios e até verificar se existe rota alternativa quando a principal se mostra comprometida. Em veículos autônomos e robôs móveis, a capacidade de percorrer um grafo rapidamente, decidindo em tempo real qual direção seguir, é decisiva para evitar colisões e reduzir o tempo total de viagem.

Embora técnicas de roteamento mais complexas, como Dijkstra ou A*, sejam comuns para otimização de distâncias, DFS e BFS continuam indispensáveis em etapas de pré-processamento. BFS fornece o caminho mais curto em número de saltos, enquanto DFS serve para varreduras completas que avaliam conectividade e detectam componentes isolados. A compreensão dessas travessias permanece fundamental na formação de engenheiros que projetam sistemas de navegação e contribui para o entendimento de algoritmos mais avançados empregados em veículos autônomos de última geração.

Exemplo de aplicação

Exemplo 4 – Veículos autônomos e robótica

Considere uma empresa de entregas por drones que opera em um *campus* universitário. O mapa do *campus* contém diversos pontos de interesse, como laboratórios, bibliotecas e residências, conectados por corredores aéreos designados.

Ao iniciar o serviço, cada drone recebe esse mapa em formato de grafo, no qual vértices representam pontos de entrega e arestas correspondem a trajetos aéreos autorizados. As regras de negócio determinam que, antes de decolar, o sistema deve verificar se há pelo menos um caminho entre o ponto de partida e o destino; se houver mais de um, deve registrar todos os percursos possíveis para avaliação de risco. Além disso, quando um corredor aéreo se torna temporariamente indisponível, o sistema precisa explorar rotas alternativas sem recalcular todo o grafo.

Para satisfazer esses requisitos, o algoritmo empregará DFS para listar todos os caminhos entre origem e destino e BFS para descobrir rapidamente o trajeto mais curto em número de saltos, assegurando que o drone disponha de uma rota viável a qualquer momento.

A implementação em Python utilizará listas de adjacência para representar o grafo, dado que essa estrutura consome pouca memória em mapas esparsos, típicos de ambientes urbanos nos quais apenas algumas vias ligam cada interseção. Cada chave do dicionário grafo apontará para uma lista que contém

vértices vizinhos, permitindo inserção e consulta em tempo constante. A travessia BFS fará uso de uma fila (`collections.deque`) que garante complexidade O(V + E) e retorna o caminho mais curto em número de saltos (V é o número de vértices e E é o número de arestas). A DFS, implementada de forma recursiva, produzirá todas as rotas possíveis entre dois vértices, usando uma lista que registra o caminho corrente e um conjunto que evita revisitar vértices já explorados.

Durante a construção dos algoritmos, métodos auxiliares separam responsabilidades: `adicionar_aresta` cria conexões bidirecionais, `bfs_caminho` devolve o percurso mínimo e `dfs_todos_caminhos` gera uma lista de listas com todas as rotas de origem a destino. O uso de `deque` traz eficiência na remoção de elementos da frente da fila, prática comum em BFS profissionais. A manipulação de listas, cópias via `slicing` e conjuntos (`set`) para marcação de vértices visitados reflete abordagens encontradas em aplicações reais, como sistemas de gerenciamento de tráfego e simuladores de robôs móveis.

Além de funções nativas, uma classe `MapaCampus` encapsulará o grafo e exporá métodos de alto nível. Essa escolha segue padrões de mercado, pois facilita testes unitários, extensão de funcionalidades e integração com outros componentes da aplicação. A modularização possibilita, por exemplo, trocar a representação interna por matriz de adjacência em cenários densos, sem impactar o código que consome a classe. Em suma, o exercício incorpora conceitos de estruturas de dados, programação orientada a objetos e algoritmos de grafos aplicados à navegação autônoma. A seguir, consta o código-fonte da solução:

```
from collections import deque

class MapaCampus:
    def __init__(self):
        self.grafo = {} # dicionário: vértice -> lista de vizinhos

    def adicionar_aresta(self, origem, destino):
        self.grafo.setdefault(origem, []).append(destino)
        self.grafo.setdefault(destino, []).append(origem)

    def bfs_caminho(self, origem, destino):
        visitados = set([origem])
        fila = deque([[origem]])
        while fila:
            caminho = fila.popleft()
            ultimo = caminho[-1]
            if ultimo == destino:
                return caminho
            for vizinho in self.grafo.get(ultimo, []):
                if vizinho not in visitados:
                    visitados.add(vizinho)
                    fila.append(caminho + [vizinho])
        return None # sem rota

    def dfs_todos_caminhos(self, origem, destino):
        todos = []
        def dfs(atual, caminho, visitados):
            if atual == destino:
                todos.append(caminho.copy())
                return
```

```
                for vizinho in self.grafo.get(atual, []):
                    if vizinho not in visitados:
                        visitados.add(vizinho)
                        caminho.append(vizinho)
                        dfs(vizinho, caminho, visitados)
                        caminho.pop()
                        visitados.remove(vizinho)
            dfs(origem, [origem], set([origem]))
            return todos

def principal():
    mapa = MapaCampus()
    corredores = [
        ("A", "B"), ("A", "C"), ("B", "D"), ("C", "D"),
        ("C", "E"), ("D", "F"), ("E", "F"), ("F", "G")
    ]
    for u, v in corredores:
        mapa.adicionar_aresta(u, v)

    partida, destino = "A", "G"
    caminho_minimo = mapa.bfs_caminho(partida, destino)
    print(f"Caminho mínimo entre {partida} e {destino}: {caminho_minimo}")

    todos_caminhos = mapa.dfs_todos_caminhos(partida, destino)
    print(f"\nTotal de rotas alternativas encontradas: {len(todos_caminhos)}")
    for rota in todos_caminhos:
        print(rota)

if __name__ == "__main__":
        principal()
```

O quadro 12 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições. As funções já explicitadas nos quadros anteriores foram omitidas para evitar duplicidade.

Quadro 12 – Funções usadas no código-fonte do quarto exercício prático

Função ou construção	Explicação de uso
from collections import deque	Importa a classe `deque`, uma fila de duas pontas otimizada para inserções e remoções em ambas as extremidades com complexidade constante
deque([elementos])	Cria uma fila baseada em deque contendo os elementos iniciais. No código, é usada para implementar a fila da BFS
fila.popleft()	Remove e retorna o primeiro elemento da `deque`. Essa operação é mais eficiente que `list.pop(0)` em listas comuns
dict.setdefault(chave, valor_padrao)	Se a chave existir no dicionário, retorna seu valor. Caso contrário, insere a chave com o valor padrão fornecido e retorna esse valor. Usado para inicializar listas de vizinhos ao construir o grafo
set([elementos])	Cria um conjunto com os elementos fornecidos. Conjuntos não possuem elementos repetidos e permitem verificações de pertencimento rápidas. No código, são usados para rastrear vértices visitados
caminho.copy()	Cria uma cópia rasa da lista caminho. Necessário ao armazenar um caminho completo antes que ele seja modificado na recursão da DFS
lista.append(valor)	Adiciona um elemento ao final da lista. No contexto do código, serve para expandir caminhos em BFS e DFS
lista.pop()	Remove e retorna o último elemento da lista. Usado para retroceder no caminho durante a DFS

O código apresentado implementa, em Python, uma representação de um grafo não direcionado para modelar um mapa de um *campus*, utilizando a biblioteca padrão `collections` e sua estrutura `deque` para gerenciar uma fila em BFS. A linguagem Python, conhecida por sua sintaxe clara e flexibilidade, é explorada no código para criar uma classe que gerencia o grafo, realiza buscas de caminhos e apresenta resultados de forma estruturada, aproveitando construções idiomáticas e eficientes da linguagem.

A importação do módulo `deque` da biblioteca `collections` é o ponto de partida técnico. O `deque` (double-ended queue) é uma estrutura de dados otimizada para operações de adição e remoção em ambas as extremidades, com complexidade O(1), ideal para implementar a fila necessária na busca em largura. Essa escolha reflete uma preocupação com eficiência, já que uma lista padrão em Python teria custo O(n) para remoções no início devido à necessidade de realocação de elementos.

A classe `MapaCampus` é definida para encapsular a lógica do grafo e suas operações. Seu método `__init__` inicializa um dicionário vazio chamado `grafo`, que serve como estrutura central de armazenamento. Nesse dicionário, cada chave representa um vértice (um ponto no *campus*, como "A" ou "B") e o valor associado é uma lista de vértices vizinhos. O uso de um dicionário explora a eficiência de acesso O(1) em média para consultas de chaves em Python, garantindo rapidez na recuperação de vizinhos durante as buscas.

O método `adicionar_aresta` implementa a lógica para construir o grafo não direcionado. Ele recebe dois vértices, `origem` e `destino`, e adiciona cada um como vizinho do outro, garantindo a bidirecionalidade característica de grafos não direcionados. A função `setdefault` é usada de forma idiomática: ela retorna a lista associada a um vértice se ele já existe no dicionário ou cria uma lista vazia e a associa ao vértice caso ele ainda não esteja presente. Essa abordagem evita verificações explícitas de existência de chaves, simplificando o código e mantendo a eficiência. Após a chamada a `setdefault`, o método `append` adiciona o vértice destino à lista de vizinhos da origem e vice-versa.

O método `bfs_caminho` implementa a busca em largura para encontrar o caminho mais curto entre dois vértices. Ele utiliza um conjunto (`set`) chamado `visitados` para rastrear os vértices já explorados, aproveitando a eficiência O(1) de inserção e consulta em conjuntos em Python. A fila, implementada com `deque`, armazena caminhos parciais, nos quais cada elemento é uma lista representando o trajeto desde a origem até o vértice atual. Inicialmente, a fila contém apenas o caminho `[origem]`. O laço `while` continua enquanto a fila não estiver vazia, removendo o primeiro caminho com `popleft` (operação O(1) no `deque`). O último vértice desse caminho é verificado: se for o destino, o caminho é retornado; caso contrário, os vizinhos desse vértice são explorados. Para cada vizinho não visitado, ele é marcado como visitado e um novo caminho, que estende o atual com esse vizinho, é adicionado à fila. Se nenhum caminho é encontrado, o método retorna `None`. A escolha da BFS garante que o primeiro caminho encontrado seja o mais curto em termos de número de arestas, uma propriedade inerente ao algoritmo.

O método `dfs_todos_caminhos` utiliza DFS para encontrar todos os caminhos possíveis entre a origem e o destino. Ele define uma função auxiliar recursiva `dfs` que explora o grafo de forma exaustiva. A recursão mantém um caminho parcial (`caminho`) e um conjunto de vértices visitados (`visitados`). Quando o vértice atual é o destino, uma cópia do caminho atual é adicionada à lista

`todos`, que armazena todos os caminhos válidos. Para cada vizinho do vértice atual não visitado, o algoritmo marca o vizinho como visitado, adiciona-o ao caminho e chama a função recursivamente. Após a exploração, o vizinho é removido do caminho (`pop`) e do conjunto de visitados, permitindo que ele seja reconsiderado em outros caminhos. Essa abordagem de backtracking é típica da DFS para problemas de enumeração de caminhos. O uso de `copy` ao salvar o caminho evita que modificações posteriores afetem os caminhos armazenados, uma prática importante devido à mutabilidade das listas em Python.

A função `principal` demonstra a utilização da classe. Um objeto `MapaCampus` é instanciado e um conjunto de arestas, representando corredores entre pontos do *campus*, é adicionado ao grafo. As arestas são especificadas como pares de vértices em uma lista de tuplas e o método `adicionar_aresta` é chamado para cada par. Em seguida, o método `bfs_caminho` é invocado para encontrar o caminho mínimo entre os pontos "A" e "G", e o resultado é impresso. O método `dfs_todos_caminhos` é usado para listar todas as rotas possíveis entre os mesmos pontos, com o número total de rotas e cada rota sendo exibida. A estrutura da função reflete boas práticas em Python, como a separação de lógica de configuração e execução.

Como já vimos, o bloco `if __name__ == "__main__":` garante que a função `principal` seja executada apenas se o script for o ponto de entrada, uma convenção comum em Python para modularidade. O código, em sua totalidade, explora recursos da linguagem como estruturas de dados eficientes (`deque`, `set`, dicionários), métodos de manipulação de listas, recursão e encapsulamento orientado a objetos, resultando em uma implementação clara e performática para o problema de navegação em um grafo.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Atribuir pesos às arestas, refletindo distâncias ou tempos de voo e substituir BFS por Dijkstra ou A* para otimizar o trajeto em termos de custo real.
- A integração com bibliotecas como NetworkX permitiria analisar métricas avançadas, como centralidade de vértices e detecção de gargalos.
- Para cenários dinâmicos, a inclusão de algoritmos de grafos temporais possibilitaria ativar ou desativar arestas em tempo real, simulando bloqueios repentinos.
- Em aplicações de veículos autônomos de maior porte, a combinação desse módulo com sensores LIDAR e mapas de ocupação traria dados de obstáculos em tempo real, enquanto a serialização do grafo para formatos como JSON facilitaria o intercâmbio com serviços de nuvem que distribuem informações de rota entre múltiplos drones colaborativos.

Daremos outro exemplo. As redes sociais digitais e as infraestruturas de comunicação corporativas formam grafos gigantescos de interações, nos quais cada vértice representa um usuário ou dispositivo e cada aresta simboliza uma mensagem, uma solicitação ou um pacote transmitido. A cibersegurança, por sua vez, busca preservar a confidencialidade, a integridade e a disponibilidade desses fluxos de dados, identificando atividades incomuns que possam indicar violações ou tentativas de intrusão. O estudo da conectividade revela padrões de comportamento legítimo e auxilia na exposição de trajetórias

anômalas que surgem quando um invasor começa a circular pela rede tentando ampliar privilégios ou exfiltrar informações.

A importância da análise de conexões cresce à medida que a superfície de ataque se expande com milhões de dispositivos IoT e perfis sociais que permanecem online ininterruptamente. Verificar isoladamente cada log de acesso torna-se inviável, razão pela qual se recorrem a representações em grafo que condensam milhares de linhas de registro em estruturas passíveis de percursos rápidos. A DFS mapeia componentes inteiramente conectados, revelando comunidades que interagem de maneira densa, enquanto a BFS localiza o caminho mínimo entre duas entidades, permitindo avaliar quão perto um atacante se encontra de um ativo sensível. Ao comparar métricas de centralidade e distribuição de graus, analistas distinguem hubs naturais de nós que, subitamente, passaram a contatar numerosos alvos, sinalizando possível propagação de malware.

Essa abordagem ganha ainda mais relevância quando aplicada a redes de comunicação em tempo real, nas quais atrasos mínimos podem significar perda de receita ou violação de dados. Algoritmos rápidos de travessia, aliados a estruturas de dados eficientes, capacitam sistemas de detecção a apontar anomalias antes que causem danos significativos. O exercício que se segue demonstra, em Python, como modelar um grafo de conexões, percorrê-lo com DFS por meio de geradores para rastrear componentes suspeitos e empregar BFS para medir alcance de intrusão, aproveitando recursos modernos da linguagem utilizados no mercado, tais como `dataclasses`, `defaultdict`, `Counter`, geradores com `yield from` e anotações de tipo.

Exemplo de aplicação

Exemplo 5 – Redes sociais e cibersegurança

Em um centro de operações de segurança, uma equipe controla o tráfego interno de uma empresa que possui diversos servidores críticos. A cada minuto, um processo coleta pares (origem, destino) correspondentes a dispositivos que trocaram pacotes nesse intervalo.

As regras estabelecem que qualquer endereço IP cuja contagem de conexões exceda em dez vezes a mediana do período deve ser marcado para inspeção.

Além disso, se dois servidores classificados como críticos ficarem unidos por um caminho de três saltos ou menos que inclua um endereço recém-detectado como suspeito, todo o componente conectado deverá ser isolado para análise forense.

O sistema precisa, portanto, receber a lista de conexões, construir o grafo, calcular graus, identificar nós anômalos, gerar todos os componentes por DFS e empregar BFS para medir distâncias até servidores críticos, relatando se o critério de isolamento foi ativado.

Para atender ao cenário, a solução utilizará a anotação de tipos disponível em Python (typing) a fim de documentar contratos de função, prática amplamente adotada em bases de código profissionais que empregam verificadores estáticos como `MyPy`. A estrutura central será um `dataclass` chamado

`GrafoConexoes`, no qual o atributo `adjacencia` será um `defaultdict[set]`. Essa coleção proveniente do módulo `collections` evita verificações de chave ao inserir vizinhos, conferindo clareza e performance. A contagem de graus será realizada por `collections.Counter`, que consome um iterável de nós e produz rapidamente um mapeamento para frequências de ocorrência.

A identificação de componentes conectados explorará DFS implementada com geradores. A função `dfs_componentes` utilizará `yield from` para propagar valores produzidos por uma função recursiva privada, permitindo iterar sobre vértices na ordem em que são descobertos, sem armazenar todo o percurso em memória. Esse padrão aproxima o exercício de fluxos em tempo real, em que resultados parciais precisam ser processados logo que gerados. Para medir distâncias, o algoritmo BFS empregará `deque`, retirando itens de sua ponta esquerda em tempo constante. Durante a expansão, o código aplicará a atribuição com o operador "walrus" (`:=`) para atualizar o dicionário de níveis de forma concisa, recurso introduzido em Python 3.8 que aparece com frequência em soluções sintéticas de mercado.

Além das travessias, o exercício demonstrará o uso do módulo `statistics` para obter mediana e compará-la com o grau de cada vértice, alinhando-se às exigências de detecção baseadas em limiares dinâmicos. A integração desses recursos reflete padrões de produção empregados em sistemas de análise de tráfego, nos quais clareza, tipagem opcional, consumo reduzido de memória e geração incremental de resultados são determinantes para a escalabilidade. A seguir, consta o código-fonte da solução:

```
from __future__ import annotations
from collections import defaultdict, Counter, deque
from dataclasses import dataclass
from statistics import median
from typing import Dict, Iterable, List, Set, Generator, Tuple

@dataclass
class GrafoConexoes:
    adjacencia: Dict[str, Set[str]]

    @classmethod
    def a_partir_de_arestas(cls, arestas: Iterable[Tuple[str, str]]) ->
GrafoConexoes:
        adj: Dict[str, Set[str]] = defaultdict(set)
        for u, v in arestas:
            adj[u].add(v)
            adj[v].add(u)
        return cls(adj)

    def graus(self) -> Counter:
        return Counter({no: len(vizinhos) for no, vizinhos in self.adjacencia.
items()})

    def dfs_componentes(self) -> Generator[Set[str], None, None]:
        visitados: Set[str] = set()
        def dfs(no: str, comp: Set[str]):
            comp.add(no)
            for viz in self.adjacencia[no]:
                if viz not in comp:
                    dfs(viz, comp)
```

```
            for vertice in self.adjacencia:
                if vertice not in visitados:
                    componente: Set[str] = set()
                    dfs(vertice, componente)
                    visitados |= componente
                    yield componente

    def bfs_distancia(self, origem: str) -> Dict[str, int]:
        niveis: Dict[str, int] = {origem: 0}
        fila: deque[str] = deque([origem])
        while fila:
            atual = fila.popleft()
            for viz in self.adjacencia[atual]:
                if viz not in niveis and (n := niveis[atual] + 1) is not None:
                    niveis[viz] = n
                    fila.append(viz)
        return niveis

def principal():
    registros = [
        ("10.0.0.1", "10.0.0.2"), ("10.0.0.2", "10.0.0.3"),
        ("10.0.0.3", "10.0.0.4"), ("10.0.0.4", "10.0.0.5"),
        ("10.0.0.5", "10.0.0.1"), ("10.0.0.6", "10.0.0.7"),
        ("10.0.0.6", "10.0.0.2"), ("192.168.1.10", "10.0.0.4"),
        ("192.168.1.10", "10.0.0.7"), ("192.168.1.10", "10.0.0.6")
    ]
    grafo = GrafoConexoes.a_partir_de_arestas(registros)

    graus = grafo.graus()
    limiar = median(graus.values()) * 10
    suspeitos = {n for n, g in graus.items() if g >= limiar}
    print(f"Nós suspeitos por grau anômalo (limiar {limiar:.0f}): {suspeitos}")

    servidores_criticos = {"10.0.0.3", "10.0.0.5"}
    for comp in grafo.dfs_componentes():
        if servidores_criticos <= comp:
        distancia_min = min(
            grafo.bfs_distancia(s)[t]
            for s in suspeitos for t in servidores_criticos if t in grafo.bfs_
distancia(s)
        ) if suspeitos else float('inf')
        if distancia_min <= 3:
            print(f"Componente a isolar: {comp}")
            break
    else:
        print("Nenhum componente crítico precisa ser isolado.")

if __name__ == "__main__":
        principal()
```

O quadro 13 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 13 – Funções usadas no código-fonte do quinto exercício prático

Função	Explicação de uso
from __future__ import annotations	Permite que anotações de tipo se referenciem a classes ainda não definidas no momento da interpretação do código, útil em definições recursivas ou com `@dataclass`
@dataclass	Decorador que transforma uma classe comum em uma classe de dados, gerando automaticamente métodos como `__init__`, `__repr__` e `__eq__` com base nos atributos definidos
defaultdict(set)	Cria um dicionário no qual, se uma chave ainda não existir, seu valor padrão será um conjunto vazio (`set()`). Facilita a construção de grafos
Counter(dict)	Cria um contador (subclasse de `dict`) com base nos pares chave-valor fornecidos, permitindo contagem e comparação frequente de elementos
set comprehension: {x for x in algo}	Gera um conjunto a partir de uma expressão iterável. No código, é usado para construir o conjunto de nós suspeitos por grau anômalo
median(iterável)	Retorna a mediana de uma coleção numérica. No código, é utilizada para calcular o limiar de grau considerado anômalo
walrus operator (:=)	Operador de atribuição dentro de expressões. Aqui, é utilizado em (`n := niveis[atual] + 1`) para calcular e usar a nova distância em uma única linha
generator (yield)	Produz valores sob demanda em vez de retornar todos de uma vez. No código, `dfs_componentes()` usa `yield` para retornar uma componente por vez
set <= set	Verifica se um conjunto é subconjunto de outro (isto é, se todos os elementos da esquerda estão presentes na direita). No código, garante que os servidores críticos estão contidos na componente

O código fornecido é uma implementação em Python que modela e analisa um grafo não direcionado, representando conexões entre nós (como endereços IP em uma rede). Ele utiliza várias funcionalidades avançadas da linguagem Python, incluindo tipagem estática, coleções especializadas, `dataclasses`, geradores e algoritmos de busca em grafos.

O código começa com importações que revelam o uso de recursos modernos do Python. A linha `from__future__import annotations` habilita a avaliação postergada de anotações de tipo, uma funcionalidade introduzida no Python 3.7 e tornada padrão no Python 3.10. Isso permite que tipos como `Dict`, `Set` e `List` sejam usados diretamente nas anotações sem necessidade de importá-los do módulo `typing`, além de possibilitar referências a tipos definidos posteriormente no código, como a própria classe `GrafoConexoes`. Essa importação demonstra a preocupação com a legibilidade e a compatibilidade com versões futuras da linguagem.

Em seguida, são importadas estruturas de dados do módulo `collections`: `defaultdict`, `Counter` e `deque`. O `defaultdict` é uma subclasse de dicionário que fornece um valor padrão para chaves inexistentes, eliminando a necessidade de verificações manuais ao construir o grafo. No método `a_partir_de_arestas`, ele é usado para criar um dicionário em que cada chave (um nó) mapeia para um conjunto (`set`) de vizinhos, inicializado automaticamente como um conjunto vazio. O `Counter` é uma estrutura que facilita a contagem de elementos, usada no método `graus` para calcular o grau de cada nó (número de vizinhos). Já o `deque` é uma fila de dupla extremidade otimizada para operações de adição e remoção em ambas as extremidades, utilizada no algoritmo de busca em largura (`bfs_distancia`) para gerenciar os nós a serem visitados.

O módulo `dataclasses` é empregado para definir a classe `GrafoConexoes` com a decoração `@dataclass`. Essa construção simplifica a criação de classes que armazenam dados, gerando automaticamente métodos como `__init__`, `__repr__` e `__eq__`. No caso, a classe possui um único atributo, `adjacencia`, que é um dicionário mapeando strings (nós) para conjuntos de strings (vizinhos). A anotação de tipo `Dict[str, Set[str]]` reflete o uso do módulo `typing`, que proporciona tipagem estática para coleções genéricas, aumentando a clareza e permitindo verificações por ferramentas como `mypy`.

Observação

O mypy é uma ferramenta de verificação estática de tipos para Python. Ele permite adicionar anotações de tipo ao código Python (usando a sintaxe do módulo typing ou anotações online) e verifica se os tipos estão sendo usados corretamente, sem precisar executar o código. Isso ajuda a identificar erros relacionados a tipos antes da execução, aumentando a robustez e a manutenibilidade do código.

O método de classe `a_partir_de_arestas`, decorado com `@classmethod`, demonstra o uso de métodos alternativos de construção. Ele recebe um iterável de tuplas representando arestas e constrói a representação de adjacência do grafo. A expressão `defaultdict(set)` cria um dicionário em que cada nova chave é associada a um conjunto vazio. O laço `for u, v in arestas` itera sobre as tuplas, adicionando cada nó como vizinho do outro, o que reflete a natureza não direcionada do grafo. O uso de `add` para conjuntos e a construção `cls(adj)` para instanciar a classe mostram a manipulação eficiente de coleções e a flexibilidade dos métodos de classe.

O método `graus` utiliza uma compreensão de dicionário `{no: len(vizinhos) for no, vizinhos in self.adjacencia.items()}` para calcular o grau de cada nó, ou seja, o número de vizinhos. Essa construção é uma característica idiomática do Python, combinando concisão e legibilidade. O resultado é encapsulado em um `Counter`, que facilita operações subsequentes, como a análise de frequências no método `principal`.

O método `dfs_componentes` implementa uma DFS para identificar componentes conexos do grafo, retornando um gerador `Generator[Set[str], None, None]`. Geradores são uma funcionalidade poderosa do Python, permitindo iteração sob demanda e economia de memória, já que os componentes são produzidos um a um. A função interna `dfs` realiza a busca recursivamente, acumulando nós em um conjunto `comp`. O uso do operador |= para união de conjuntos `visitados |= componente` é uma operação eficiente que atualiza o conjunto de nós visitados. A anotação de tipo no gerador e o uso de conjuntos para rastrear nós visitados demonstram a integração de tipagem estática e estruturas de dados otimizadas.

O método `bfs_distancia` implementa uma BFS para calcular as distâncias mínimas de um nó de origem a todos os outros nós alcançáveis. Ele utiliza um `deque` para a fila de nós a visitar e um dicionário `níveis` para armazenar as distâncias. A expressão `if viz not in niveis and (n`

`:= niveis[atual] + 1) is not None` combina uma verificação de condição com a atribuição de expressão (introduzida no Python 3.8), conhecida como "walrus operator" (`:=`). Essa construção reduz a verbosidade ao calcular e atribuir a distância em uma única linha. O método retorna um dicionário mapeando nós para suas distâncias, ilustrando o uso de coleções para representar resultados estruturados.

Na função `principal`, o código aplica a classe `GrafoConexoes` a um problema prático: análise de uma rede representada por pares de endereços IP. A lista de registros é um exemplo de dados de entrada, na qual cada tupla representa uma conexão bidirecional. Após criar o grafo, o código calcula os graus dos nós e usa o módulo `statistics` para determinar a mediana dos graus `median(graus.values())`. A mediana é multiplicada por 10 para definir um limiar e uma compreensão de conjunto `{n for n, g in graus.items() if g >= limiar}` identifica nós com graus anômalos, considerados suspeitos. Essa construção exemplifica o uso de compreensões para filtragem de dados.

O código verifica se componentes conexos contêm servidores críticos (definidos como um conjunto) e avalia a proximidade de nós suspeitos usando `bfs_distancia`. A expressão `servidores_criticos <= comp` verifica se o conjunto de servidores está contido no componente, utilizando o operador de subconjunto. A distância mínima é calculada com uma compreensão alinhada e o uso de `float('inf')` como valor padrão para casos sem suspeitos reflete uma prática comum em algoritmos para lidar com casos extremos. A estrutura `for... else` é empregada para lidar com o caso em que nenhum componente crítico precisa ser isolado, uma construção menos comum, mas elegante, que executa o bloco `else` se o laço não for interrompido por um `break`.

Por fim, o bloco `if __name__ == "__main__":` segue a convenção do Python para executar a função `principal` apenas se o script for chamado diretamente, garantindo modularidade. O código combina recursos avançados da linguagem, como tipagem estática, coleções especializadas, geradores, dataclasses e algoritmos de grafos, para criar uma solução robusta e eficiente para análise de redes, demonstrando a expressividade e a flexibilidade do Python em aplicações práticas.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Integrar a leitura de fluxos de dados em tempo real a partir de sistemas de captura como Apache Kafka, processando janelas deslizantes e atualizando o grafo incrementalmente.

- A incorporação de algoritmos de detecção de comunidades (Louvain, Leiden) ajudaria a identificar clusters inesperados formados após a chegada de um intruso.

- Para lidar com volumes maciços, a representação poderia migrar para um banco de dados de grafos, tal como Neo4j ou TigerGraph, mantendo a interface de alto nível.

- Técnicas de aprendizado de máquina baseadas em embeddings de grafos, geradas por métodos como node2vec, permitiriam classificar automaticamente novos padrões de intrusão e recomendar ações de contenção, ampliando a eficácia do sistema de defesa.

Daremos agora um último exemplo. Telecomunicações constituem o alicerce da sociedade em rede, viabilizando desde transmissões de voz até fluxos massivos de dados que sustentam serviços de nuvem, streaming e Internet das Coisas. Redes de transporte modernas combinam enlaces ópticos, malhas de rádio e roteadores IP de alta capacidade, conectando milhares de pontos pelo mundo. A velocidade com que novas aplicações surgem – videoconferência 8K, carros conectados, cidades inteligentes – impõe pressão constante sobre essas infraestruturas, tornando imprescindível otimizar o roteamento para evitar congestionamentos e garantir qualidade de serviço.

Otimizar o roteamento significa selecionar, entre muitas rotas possíveis, aquela que conduz os pacotes de forma mais eficiente, mantendo a rede equilibrada. Diferentemente de ambientes acadêmicos, nos quais a latência é a única preocupação, operadoras precisam levar em conta sobrecarga de enlace, requisitos de redundância e critérios de engenharia de tráfego que mudam minuto a minuto. Em redes sem custo explícito definido em cada enlace – por exemplo, quando todos os circuitos têm a mesma capacidade nominal –, rotas que minimizam o número de saltos (hops) continuam sendo um padrão de mercado, pois reduzem a probabilidade de falhas intermediárias e simplificam a comutação de pacotes. A BFS encontra exatamente esse tipo de caminho mínimo em saltos, enquanto DFS percorre rotas alternativas completas para balanceamento de carga ou análises de resiliência.

Em um contexto em que o software-defined networking (SDN) proporciona reconfiguração instantânea, algoritmos leves como BFS e DFS mantêm relevância prática: executam rápido, consomem pouca memória e podem ser repetidos inúmeras vezes quando a topologia muda. Assim, profissionais de telecomunicações utilizam variações desses algoritmos para pré-processar tabelas de encaminhamento, identificar redundâncias e distribuir fluxos, antes de recorrer a estratégias de otimização mais custosas. O exercício adiante ilustra, em Python, como modelar uma rede, aplicar BFS para obter o menor número de saltos entre dois roteadores e empregar DFS para listar todas as demais rotas, escolhendo aquela que evita enlaces congestionados.

Exemplo de aplicação

Exemplo 6 – Telecomunicações

Uma operadora metropolitana administra oito roteadores-chave interligados por enlaces simétricos. A telemetria a cada cinco minutos informa quantos fluxos ativos atravessam cada enlace, classificando-o como congestionado quando esse valor ultrapassa 70% da capacidade.

Para cada solicitação de circuito entre dois roteadores, o sistema deve:

- descobrir com BFS a rota de menor número de saltos;
- se houver várias rotas empatadas nesse critério, selecionar, por DFS, aquela que evita a maior quantidade de enlaces congestionados;
- caso todos os caminhos mínimos passem por pelo menos um enlace congestionado, registrar alerta para engenharia de tráfego.

Quando um enlace falha ou volta ao ar, o sistema deve ajustar a topologia em memória e calcular novamente as rotas, sem reiniciar o serviço.

Essas regras determinam que o algoritmo mantenha lista de adjacência mutável, percorra a rede repetidamente e classifique enlaces segundo o estado de carga.

O exercício explora estruturas de dados integradas do Python para representar grafos esparsos. Usaremos `defaultdict(list)` do módulo `collections` para mapear cada roteador à sua lista de vizinhos, possibilitando inserções e remoções sem verificações explícitas de chave. A função `bfs_menor_caminho` faz uso de `deque`, também de `collections`, para retirar nós da frente da fila em tempo constante, retornando a primeira rota que atinge o destino – rota garantidamente mínima em saltos quando todos os enlaces têm custo 1.

Para enumerar rotas alternativas empatadas em saltos, a função `dfs_rotas_limitadas` percorre o grafo em profundidade, mas somente visita vértices até atingir o mesmo número de saltos do caminho mínimo. Durante a recursão, um `set` armazena vértices visitados, prevenindo ciclos, enquanto o caminho corrente é mantido em lista reutilizada com operações de `push/pop` (backtracking). A recursão usa a palavra-chave `yield from`, permitindo ao chamador iterar sobre rotas geradas sob demanda, economizando memória em redes maiores.

A classificação de enlaces congestionados faz uso de `frozenset` para armazenar pares de roteadores de forma ordenada e imutável, permitindo que enlaces sejam comparados independentemente da direção. A função `rota_mais_descongestionada` recebe todas as rotas mínimas e, com `min` combinada a uma `lambda`, seleciona a rota que contém o menor número de enlaces marcados como congestionados. Esse padrão de ordenação é comum em scripts operacionais de rede, por sua clareza e aderência às exigências de preferir caminhos menos sobrecarregados quando há empate em saltos. A seguir, consta o código-fonte da solução:

```
from collections import defaultdict, deque
from typing import Dict, List, Set, Tuple, Iterable

class Rede:
    def __init__(self):
        self.adj: Dict[str, List[str]] = defaultdict(list)
        self.congestionados: Set[frozenset] = set()

    def adicionar_enlace(self, u: str, v: str):
        self.adj[u].append(v)
        self.adj[v].append(u)

    def remover_enlace(self, u: str, v: str):
        self.adj[u] = [n for n in self.adj[u] if n != v]
        self.adj[v] = [n for n in self.adj[v] if n != u]
        self.congestionados.discard(frozenset({u, v}))

    def marcar_congestionado(self, u: str, v: str):
        self.congestionados.add(frozenset({u, v}))
```

```
    def bfs_menor_caminho(self, origem: str, destino: str) -> List[str]:
        fila = deque([[origem]])
        visit = {origem}
        while fila:
            caminho = fila.popleft()
            ultimo = caminho[-1]
            if ultimo == destino:
                return caminho
            for viz in self.adj[ultimo]:
                if viz not in visit:
                    visit.add(viz)
                    fila.append(caminho + [viz])
        return []

    def dfs_rotas_limitadas(self, origem: str, destino: str, limite: int) ->
Iterable[List[str]]:
        visit = {origem}
        caminho = [origem]
        def dfs(atual: str):
            if len(caminho) - 1 > limite:
                return
            if atual == destino and len(caminho) - 1 == limite:
                yield list(caminho)
                return
            for viz in self.adj[atual]:
                if viz not in visit:
                    visit.add(viz)
                    caminho.append(viz)
                    yield from dfs(viz)
                    caminho.pop()
                    visit.remove(viz)
        yield from dfs(origem)

    def rota_mais_descongestionada(self, rotas: Iterable[List[str]]) ->
Tuple[List[str], int]:
        def contar_cong(rota):
            return sum(frozenset({rota[i], rota[i+1]}) in self.congestionados
                       for i in range(len(rota)-1))
        return min(((rota, contar_cong(rota)) for rota in rotas), key=lambda t:
t[1])

def principal():
    rede = Rede()
    enlaces = [("A", "B"), ("B", "C"), ("C", "D"), ("A", "E"),
               ("E", "F"), ("F", "D"), ("B", "E"), ("C", "F")]
    for u, v in enlaces:
        rede.adicionar_enlace(u, v)

    # marca dois enlaces como congestionados
    rede.marcar_congestionado("B", "C")
    rede.marcar_congestionado("C", "D")

    origem, destino = "A", "D"
    rota_min = rede.bfs_menor_caminho(origem, destino)
    if not rota_min:
        print("Não há rota disponível.")
```

```
            return
        saltos = len(rota_min) - 1
        rotas_empate = list(rede.dfs_rotas_limitadas(origem, destino, saltos))
        melhor_rota, cong = rede.rota_mais_descongestionada(rotas_empate)

        print(f"Todas as rotas mínimas em saltos ({saltos}):")
        for r in rotas_empate:
            print(r)
        print(f"\nRota escolhida (menos enlaces congestionados = {cong}): {melhor_rota}")

        # simula falha em um enlace crítico
        print("\nFalha no enlace B-E")
        rede.remover_enlace("B", "E")
        nova_rota = rede.bfs_menor_caminho(origem, destino)
        print(f"Nova rota em saltos: {nova_rota or 'indisponível'}")

if __name__ == "__main__":
        principal()
```

O quadro 14 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 14 – Funções usadas no código-fonte do sexto exercício prático

Função ou construção	Explicação de uso
frozenset({a, b})	Cria um conjunto imutável e não ordenado, usado para representar enlaces entre dois nós independentemente da ordem. Ideal para identificar conexões bidirecionais
set.discard(elemento)	Remove um elemento do conjunto, se ele existir. Caso contrário, não gera erro. No código, remove enlaces congestionados da estrutura de controle
list comprehension com condicional: [x for x in lista if cond]	Gera uma nova lista contendo apenas os elementos que satisfazem a condição. No código, é usada para remover enlaces entre nós
yield from	Delegação de geração: permite que um gerador produza todos os valores de outro iterável ou gerador. No código, encadeia chamadas recursivas de `dfs`
min(iterável, key=função)	Retorna o menor elemento de um iterável com base em uma chave de comparação. Aqui, seleciona a rota com o menor número de enlaces congestionados
lambda t: t[1]	Define uma função anônima que extrai o segundo elemento de uma tupla. No código, é usada como chave de ordenação em `min()`
sum(expressão for...)	Soma os valores gerados por uma expressão dentro de um laço. Aqui, conta quantos enlaces em uma rota estão congestionados

O código Python apresentado implementa uma representação de uma rede não direcionada por meio de uma classe chamada `Rede`, que utiliza grafos para modelar conexões entre nós e realizar operações como busca de caminhos, gerenciamento de enlaces congestionados e simulação de falhas. Ele emprega bibliotecas padrão como `collections` para estruturas de dados eficientes e `typing` para anotações de tipo, garantindo clareza e robustez. A execução principal demonstra a funcionalidade da classe ao criar uma rede, configurar enlaces, marcar congestionamentos, encontrar rotas e simular falhas, tudo com uma abordagem orientada a objetos e algoritmos clássicos de grafos.

A classe `Rede` é inicializada com dois atributos principais: `adj`, um dicionário que utiliza `defaultdict(list)` para armazenar a lista de adjacência do grafo, e `congestionados`, um conjunto de `frozenset` que registra enlaces marcados como congestionados. O uso de `defaultdict` simplifica a adição de nós, pois elimina a necessidade de inicializar listas manualmente,

enquanto `frozenset` garante que enlaces sejam representados de forma imutável e não ordenada, já que a rede é não direcionada (a ordem entre dois nós não importa).

O método `adicionar_enlace` permite a inclusão de um enlace bidirecional entre dois nós u e v, adicionando cada nó à lista de adjacência do outro. Isso reflete a natureza não direcionada do grafo, em que um enlace entre u e v implica conexões em ambas as direções. Por outro lado, o método `remover_enlace` elimina um enlace bidirecional, filtrando as listas de adjacência para excluir as referências mútuas entre u e v e removendo o enlace do conjunto de congestionados, se presente. O método `marcar_congestionado` adiciona um enlace ao conjunto `congestionados`, utilizando `frozenset` para representar o par de nós de maneira consistente.

Para encontrar o menor caminho entre dois nós, o método `bfs_menor_caminho` implementa a BFS. Ele utiliza uma fila `deque` para explorar os nós nível a nível, mantendo uma lista de caminhos parciais. Cada caminho é estendido com vizinhos não visitados e o algoritmo retorna o primeiro caminho que alcança o destino, garantindo o menor número de saltos (arestas) em um grafo não ponderado. Um conjunto `visit` rastreia nós já explorados para evitar ciclos. Se nenhum caminho é encontrado, o método retorna uma lista vazia.

O método `dfs_rotas_limitadas` realiza uma DFS para encontrar todas as rotas entre a origem e o destino com um número exato de saltos, especificado pelo parâmetro `limite`. Ele usa uma abordagem recursiva, mantendo um conjunto `visit` para evitar revisitar nós no mesmo caminho e uma lista `caminho` para construir as rotas. A função interna `dfs` explora vizinhos recursivamente, adicionando e removendo nós do caminho e do conjunto de visitados conforme avança e retrocede. Rotas são geradas (`yield`) apenas quando o destino é alcançado com o número exato de saltos, garantindo que todas as rotas retornadas tenham o mesmo comprimento.

O método `rota_mais_descongestionada` avalia um conjunto de rotas para determinar qual possui o menor número de enlaces congestionados. Ele define uma função auxiliar `contar_cong` que conta quantos enlaces de uma rota estão no conjunto `congestionados`, verificando pares consecutivos de nós na rota. A função `min`, com uma chave baseada na contagem de congestionamentos, seleciona a rota com menos enlaces congestionados, retornando uma tupla com a rota escolhida e o número de enlaces congestionados.

A função `principal` demonstra a aplicação prática da classe `Rede`. Ela cria uma instância da rede e adiciona oito enlaces, formando um grafo com nós de `A` `F`. Dois enlaces, `B-C` e `C-D`, são marcados como congestionados. A função busca o menor caminho de `A` `D` usando BFS, calcula o número de saltos desse caminho e encontra todas as rotas com o mesmo número de saltos usando DFS. Entre essas rotas, a menos congestionada é selecionada e exibida, junto a todas as rotas possíveis e o número de enlaces congestionados na rota escolhida. Por fim, a função simula a falha do enlace `B-E`, removendo-o e recalcula o menor caminho de `A` `D`, exibindo o resultado.

O código é eficiente para grafos de tamanho moderado, utilizando estruturas de dados adequadas e algoritmos bem estabelecidos. A BFS garante o menor caminho em tempo O(V + E), em que V é o número de vértices e é o número de arestas, enquanto a DFS para rotas limitadas pode ser mais

custosa em grafos densos, mas é apropriada para listar caminhos específicos. O uso de `frozenset` para enlaces congestionados é uma escolha inteligente para lidar com a simetria de enlaces não direcionados, e a modularidade da classe permite fácil extensão para outras funcionalidades de redes.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Atribuir pesos dinâmicos aos enlaces, refletindo variações de latência ou utilização em tempo real, e empregar BFS multicritério que primeiro minimize saltos, depois pesos agregados.
- A introdução de algoritmos de fluxo máximo poderia balancear tráfego entre múltiplas rotas simultâneas.
- Integrações com bibliotecas de série temporal, como pandas, permitiriam ler métricas de utilização histórica e prever congestionamentos, alimentando o módulo de roteamento com pesos preditivos.
- Finalmente, expor a lógica como microsserviço REST, usando FastAPI, viabilizaria que controladores SDN requisitassem rotas otimizadas via HTTP, enquanto testes de unidade com pytest garantiriam confiabilidade em produção.

4 ALGORITMOS DE ORDENAÇÃO

Os algoritmos de ordenação constituem um pilar clássico da ciência da computação, pois fornecem metodologia essencial para organizar dados de maneira que operações subsequentes ocorram com maior eficiência. Desde as implementações mais elementares, empregadas em contextos pedagógicos, até técnicas sofisticadas concebidas para lidar com grandes volumes de informação, a ordenação permeia sistemas de bancos de dados, pipelines de processamento de arquivos e motores de busca. Compreender o comportamento de cada abordagem, suas vantagens e desvantagens, revela-se decisivo quando o desenvolvedor se depara com requisitos de desempenho, consumo de memória ou estabilidade dos resultados.

O método de ordenação Bubble Sort ilustra a ordenação por troca repetitiva de pares adjacentes. Durante cada varredura, elementos consecutivos são comparados; se estiverem fora de ordem, são trocados, de modo que o maior (ou menor, conforme o critério) borbulha gradualmente para a extremidade do vetor. Embora a implementação seja simples e utilize apenas espaço auxiliar constante, o algoritmo realiza, em média, $n^2/2$ comparações e, aproximadamente, o mesmo número de trocas, o que conduz a uma complexidade temporal $O(n^2)$. A estabilidade é preservada porque a troca ocorre somente entre elementos adjacentes e de valores distintos, recurso que mantém a ordem relativa dos registros de chave igual.

O Selection Sort, em contraste, identifica a menor chave remanescente e a posiciona no início da porção não ordenada. Realizando n-1 seleções sucessivas, cada uma custando uma busca linear, totaliza $n(n-1)/2$ comparações, igualmente situado em $O(n^2)$. Apenas uma troca por passagem é necessária, característica que reduz deslocamentos de dados em memórias nas quais a operação de escrita é onerosa. Entretanto, conservar a estabilidade demandaria adaptações adicionais, pois a simples troca de posições pode alterar a ordem preexistente entre registros iguais.

O Insertion Sort segue lógica análoga ao ato de inserir cartas em mãos de jogo. A lista é percorrida da esquerda para a direita; cada elemento corrente desloca-se para trás enquanto encontra valores maiores, colocando-se enfim na posição correta. Para listas quase ordenadas, os deslocamentos tendem a ser limitados, permitindo caso médio linear em cenários favoráveis. No entanto, no pior cenário, envolvendo sequência decrescente, ocorre novamente $O(n^2)$ operações. A natureza incremental, aliada ao acesso predominantemente sequencial, confere grande eficiência para tamanhos reduzidos ou para lotes parcialmente organizados, além de preservar a estabilidade.

A ordenação avança para técnicas de divisão e conquista quando se examina o Merge Sort. O vetor é particionado sucessivamente até que cada sublista contenha um único elemento; em seguida, sublistas vizinhas unem-se por meio de fusão estável que consome tempo linear em relação ao número total de elementos envolvidos. Como a altura da árvore de recursão equivale a $\log_2 n$, decorre complexidade $O(n \log n)$. Essa abordagem apresenta desempenho consistente independentemente da disposição inicial dos dados, aspecto valioso em sistemas que exigem previsibilidade. O custo suplementar surge no uso de memória auxiliar proporcional a n, necessária para armazenar resultados intermédios durante as fusões.

O Quick Sort compartilha o princípio de divisão, embora empregue partição em torno de um pivô. Elementos menores do que o pivô deslocam-se para seu lado esquerdo, enquanto os maiores deslocam-se para a direita, formando duas sublistas que receberão tratamento recursivo. A eficiência resulta da média de $O(n \log n)$ comparações, pois a partição divide a entrada em frações equilibradas na maioria dos casos. A escolha do pivô, entretanto, influencia sobremaneira o tempo total, visto que distribuição extremamente assimétrica conduz a profundidade máxima da recursão e, consequentemente, a $O(n^2)$ operações. A execução in-place economiza espaço, recurso que torna o algoritmo atraente para ambientes restritos, embora seu mecanismo de troca impeça estabilidade sem artifícios adicionais.

Comparar essas abordagens envolve considerar variedade de critérios. Quanto à complexidade temporal, Bubble Sort, Selection Sort e Insertion Sort compartilham degradação quadrática em instâncias de maior porte, enquanto Merge Sort e Quick Sort oferecem comportamento $O(n \log n)$ em média. No quesito espaço, Quick Sort destaca-se por demandar apenas pilha de chamadas proporcional a $\log n$ (no melhor caso), ao passo que Merge Sort traz necessidade de vetor auxiliar completo. Estabilidade permanece disponível em Bubble Sort, Insertion Sort e Merge Sort, aspecto imprescindível em aplicações que exigem preservação da ordem secundária, como ordenação de registros hierárquicos. Em dados quase ordenados, Insertion Sort alcança desempenho prático superior aos concorrentes elementares e corriqueiramente eclipsa algoritmos de complexidade assintótica menor em pequenas amostragens, situação frequente em rotinas internas de bibliotecas padrão. Quando conjuntos crescem e memórias externas entram em cena, Merge Sort sobressai devido ao padrão de acesso sequencial coerente com dispositivos de armazenamento magnético; em contraste, Quick Sort domina em ambiente de memória principal, beneficiado pela localidade de referência durante partições.

Neste tópico, examinaremos em detalhes esses algoritmos.

4.1 Ordenação simples: Bubble Sort, Selection Sort e Insertion Sort

A ordenação de dados é uma tarefa fundamental em ciência da computação, pois organiza informações de maneira que facilite buscas, análises e processamento eficiente. Entre os algoritmos de ordenação mais simples, destacam-se o Bubble Sort, o Selection Sort e o Insertion Sort, conhecidos por sua implementação intuitiva e facilidade de compreensão, embora não sejam os mais eficientes para grandes conjuntos de dados. Cada um desses métodos possui características distintas, mas todos compartilham a premissa de reorganizar elementos em uma sequência, geralmente em ordem crescente ou decrescente, por meio de comparações e trocas ou inserções.

O Bubble Sort, ou ordenação por bolha, opera de forma quase ingênua, comparando pares de elementos adjacentes em uma lista e trocando-os caso estejam na ordem errada. Esse processo se repete várias vezes, como se as trocas fizessem os elementos flutuarem como bolhas até sua posição correta. Em cada iteração, o maior (ou menor, dependendo da ordenação) elemento é gradualmente movido para o final (ou início) da lista. A simplicidade do Bubble Sort é seu maior atrativo, pois sua lógica é facilmente compreensível até para iniciantes. No entanto, sua eficiência é comprometida, especialmente em listas extensas, já que sua complexidade de tempo é da ordem de $O(n^2)$ no pior e no caso médio, em que n é o número de elementos. Em cenários com listas quase ordenadas, ele pode apresentar um desempenho ligeiramente melhor, mas ainda assim é raramente usado em aplicações práticas devido à sua ineficiência.

Por outro lado, o Selection Sort, ou ordenação por seleção, adota uma abordagem um pouco diferente, mas igualmente intuitiva. Esse algoritmo divide a lista em duas partes: uma sublista ordenada e outra desordenada. A cada iteração, ele percorre a parte desordenada em busca do menor (ou maior) elemento e o posiciona no final da sublista ordenada. Essa troca ocorre apenas uma vez por iteração, o que reduz o número de trocas em comparação com o Bubble Sort, embora o número de comparações permaneça elevado. Assim como o Bubble Sort, o Selection Sort apresenta complexidade de tempo $O(n^2)$, independentemente de a lista estar quase ordenada ou completamente desordenada. Sua vantagem reside na previsibilidade: o número de trocas é sempre limitado, o que pode ser útil em sistemas nos quais a troca de elementos é uma operação custosa. Contudo, sua simplicidade não compensa a falta de eficiência em grandes volumes de dados.

Já o Insertion Sort, ou ordenação por inserção, oferece uma abordagem que imita a forma como uma pessoa organizaria cartas em um jogo de baralho. Ele constrói a lista ordenada elemento por elemento, inserindo cada novo item na posição correta em relação aos elementos já ordenados. A cada passo, o algoritmo compara o elemento atual com os anteriores, deslocando-os se necessário, até encontrar o local adequado para a inserção. Essa estratégia faz do Insertion Sort particularmente eficiente para listas pequenas ou quase ordenadas, pois o número de comparações e deslocamentos é significativamente reduzido nesses casos. Sua complexidade de tempo também é $O(n^2)$ no pior caso, mas no melhor caso, quando a lista já está quase ordenada, ele pode se aproximar de $O(n)$. Essa característica torna o Insertion Sort uma escolha razoável em situações específicas, como em sistemas com dados parcialmente organizados ou em algoritmos híbridos que combinam diferentes técnicas de ordenação.

Embora Bubble Sort, Selection Sort e Insertion Sort sejam didaticamente valiosos por sua simplicidade, eles não competem com algoritmos mais avançados, como Quick Sort ou Merge Sort, em termos de desempenho para grandes conjuntos de dados. Sua utilidade prática é limitada a cenários

com poucos elementos ou a contextos educacionais, nos quais a clareza do código e a facilidade de implementação são prioridades. Cada um desses métodos, com suas particularidades, ilustra diferentes maneiras de abordar o problema da ordenação, revelando tanto os desafios quanto a elegância inerente à manipulação de dados. Assim, eles permanecem como ferramentas fundamentais no estudo de algoritmos, oferecendo uma base sólida para a compreensão de técnicas mais sofisticadas.

Nosso primeiro exercício prático desse tópico versa sobre Internet das Coisas (IoT) e sistemas embarcados que, como vimos, descrevem o universo de dispositivos minúsculos – sensores de temperatura, atuadores industriais, wearables, microcontroladores em drones – capazes de coletar, processar e transmitir dados sem intervenção humana direta. Diferentemente de servidores em nuvens sobrepotentes, esses dispositivos operam com poucos kilobytes de RAM, ciclos de CPU contados em megahertz e, muitas vezes, energia fornecida por baterias ou painéis solares. Ainda assim, são responsáveis por medições vitais que alimentam painéis de monitoramento, algoritmos de manutenção preditiva e serviços críticos de automação residencial.

Nesse ambiente austero, os conjuntos de dados manipulados em memória raramente excedem algumas dezenas de amostras: pacotes de telemetria coletados a cada segundo, leituras de pressão em uma válvula, registros recentes de batimentos cardíacos. Executar algoritmos sofisticados de ordenação – projetados para gigabytes em data centers – não faz sentido por exceder a quantidade de memória disponível. Em vez disso, rotinas extremamente simples, como Bubble Sort, Selection Sort e Insertion Sort, permanecem relevantes. Embora tenham complexidade $O(n^2)$, elas exibem código enxuto, exigem pouquíssima RAM e apresentam desempenho suficiente quando $n \leq 50$, cenário típico de IoT.

O processamento local desses microconjuntos traz benefícios tangíveis: reduz consumo de banda, libera gateways de tarefas triviais e responde a eventos em milissegundos. Um sensor de vibração pode ordenar rapidamente dez leituras mais recentes para eliminar valores extremos antes de enviar a média; um nó de irrigação pode ordenar leituras de umidade do solo para detectar tendências sem esperar pelo servidor. Dominar as versões micro dos algoritmos clássicos é, portanto, parte essencial da caixa de ferramentas de engenheiros que projetam firmware para dispositivos restritos.

Exemplo de aplicação

Exemplo 1 – IoT e sistemas embarcados

Imagine uma estação meteorológica autônoma alimentada por energia solar, instalada em campo remoto. Seu microcontrolador coleta, a cada minuto, a temperatura ambiente durante os últimos dez minutos, armazenando os valores em um buffer circular de dez posições.

Antes de transmitir o relatório resumido ao servidor central – operação cara em termos de energia e largura de banda –, o firmware deve ordenar o buffer, extrair a mediana e normalizar o valor.

As regras técnicas definem que:

- se o buffer ainda não está cheio (menos de dez leituras), utilizar Bubble Sort, que é fácil de implementar e requer apenas trocas adjacentes;
- quando o buffer completar dez entradas, aplicar Insertion Sort, aproveitando o fato de que nove valores já estão quase ordenados e apenas a leitura mais nova precisa ser inserida;
- se um técnico solicitar auditoria de dados e houver até cinquenta leituras armazenadas, recorrer a Selection Sort, cuja contagem mínima de trocas reduz desgaste na memória flash.

Ao final, o sistema imprime o buffer ordenado e a mediana calculada, respeitando o algoritmo escolhido segundo as regras de negócio.

O exercício demonstra três algoritmos de ordenação implementados diretamente sem recorrer à função `sorted` da biblioteca padrão, justamente para ilustrar sua mecânica em ambientes em que chamadas de alto nível podem não existir ou consumir memória excessiva. Cada rotina recebe uma lista mutável e a ordena in-place, evitando alocações adicionais. Python permite escrever esses algoritmos com poucos comandos graças a operação de troca `a[i], a[j] = a[j], a[i]`, que substitui o uso de variável temporária e espelha instruções de máquina swap encontradas em microcontroladores.

Observação

O termo in-place significa que o algoritmo de ordenação modifica diretamente a lista original fornecida como entrada, sem criar uma lista ou usar espaço adicional significativo para armazenar dados temporários. Em outras palavras, a ordenação é feita no lugar, reorganizando os elementos dentro da própria estrutura de dados original, o que economiza memória.

Para gerenciar o buffer circular, o código emprega o módulo `collections.deque`, cuja extensão máxima limitada evita estouro de RAM e elimina a necessidade de lógica manual de rotação. A conversão desse `deque` para lista – com `list(deque_obj)` – gera sequência indexável sem duplicar dados além do tamanho do buffer, pois o objeto possui no máximo cinquenta elementos. Tipagem opcional com `from__future__importannotations` e anotações estilo PEP 484 documenta a assinatura das funções, prática crescente na indústria embarcada para verificar erros em tempo de análise estática sem comprometer desempenho em execução.

Observação

A PEP 484 é uma Proposta de Aprimoramento do Python (Python Enhancement Proposal) que introduziu a sintaxe e as convenções para anotações de tipo (type hints) no Python. Publicada em 2014, ela define um padrão para especificar tipos de variáveis, parâmetros de funções e valores de retorno de forma opcional, permitindo que desenvolvedores adicionem informações de tipo ao código sem afetar sua execução, já que o Python é uma linguagem dinamicamente tipada. O estilo PEP 484 no texto indica que as anotações de tipo seguem essas diretrizes, garantindo consistência e compatibilidade com ferramentas de análise estática.

Por fim, a escolha condicional do algoritmo ilustra como aplicar regras de negócio diretamente no firmware: um simples `if/elif` decide qual rotina chamar com base no número de leituras. Esse padrão substitui sistemas de plugin pesados e mantém o binário enxuto, condição necessária para caber em chips com memória flash medida em centenas de kilobytes. A seguir, consta o código-fonte da solução:

```
from __future__ import annotations
from collections import deque
from typing import List

# -------- Algoritmos de ordenação simples --------
def bubble_sort(arr: List[float]) -> None:
    n = len(arr)
    for i in range(n - 1):
        trocou = False
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                trocou = True
      if not trocou: # já ordenado
          break

def insertion_sort(arr: List[float]) -> None:
    for i in range(1, len(arr)):
        chave = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > chave:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = chave

def selection_sort(arr: List[float]) -> None:
    n = len(arr)
    for i in range(n - 1):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
```

```
            if min_idx != i:
                arr[i], arr[min_idx] = arr[min_idx], arr[i]

# -------- Buffer circular e lógica de escolha --------
BUFFER_MAX = 50      # capacidade máxima para auditoria
BUFFER_PADRAO = 10   # tamanho regular de operação

buffer = deque(maxlen=BUFFER_MAX)

def inserir_leitura(valor: float) -> None:
    buffer.append(valor)

def ordenar_e_mostrar() -> None:
    lista = list(buffer)     # snapshot
    n = len(lista)
    if n < BUFFER_PADRAO:
        bubble_sort(lista)
        metodo = "Bubble Sort"
    elif n == BUFFER_PADRAO:
        insertion_sort(lista)
        metodo = "Insertion Sort"
    else:
        selection_sort(lista)
        metodo = "Selection Sort"
    mediana = lista[n // 2] if n % 2 else (lista[n // 2 - 1] + lista[n // 2]) / 2
    print(f"Método utilizado: {metodo}")
    print(f"Buffer ordenado: {lista}")
    print(f"Mediana: {mediana:.2f}°C\n")

# ---------- Demonstração ----------
def principal():
    leituras = [23.5, 24.1, 22.9, 23.0, 24.3, 23.8, 24.0,
                23.2, 23.7, 24.4, 25.0, 24.6, 24.8] # 13 amostras
    for temp in leituras:
        inserir_leitura(temp)
    ordenar_e_mostrar()

    # Simula mais leituras até completar 50 para auditoria
    for temp in [24.1 + i*0.02 for i in range(37)]:
        inserir_leitura(temp)
    ordenar_e_mostrar()

if __name__ == "__main__":
        principal()
```

O quadro 15 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 15 – Funções usadas no código-fonte do primeiro exercício prático

Função ou construção	Explicação de uso
deque(maxlen=n)	Cria uma fila de tamanho máximo n. Quando o limite é atingido, os elementos mais antigos são descartados automaticamente. Ideal para buffers circulares
arr[j], arr[j+1] = arr[j+1], arr[j]	Troca de elementos em uma lista de forma simultânea, sem uso de variável auxiliar. Muito comum em algoritmos de ordenação
if not trocou: break	Interrompe o laço for antecipadamente se nenhuma troca ocorreu, indicando que o array já está ordenado. Otimiza o Bubble Sort
mediana com condicional (n % 2)	Cálculo da mediana que depende da paridade do número de elementos. Se ímpar, é o valor do meio; se par, é a média dos dois valores centrais
[expressão for i in range(n)]	List comprehension para gerar listas com base em uma expressão. No código, gera leituras simuladas com incremento gradual

O código começa com importações que estabelecem a base para sua funcionalidade. A diretiva `from __future__ import annotations` habilita o uso de anotações de tipo pós-avaliadas, permitindo declarações como `List[float]` sem avaliação imediata, o que melhora a compatibilidade e legibilidade. A biblioteca `collections` fornece a estrutura `deque`, usada para criar um buffer circular com capacidade limitada, enquanto `typing` importa `List` para anotações de tipo, garantindo maior clareza sobre os tipos de dados esperados.

Os três algoritmos de ordenação implementados – Bubble Sort, Insertion Sort e Selection Sort – operam sobre listas de números de ponto flutuante (`List[float]`) e modificam a lista in-place, sem retornar um novo objeto. O Bubble Sort percorre a lista repetidamente, comparando elementos adjacentes e trocando-os se estiverem na ordem errada. Ele inclui uma otimização que interrompe a execução se nenhuma troca ocorrer em uma iteração, indicando que a lista já está ordenada. O Insertion Sort constrói a lista ordenada incrementalmente, inserindo cada elemento na posição correta ao compará-lo com os elementos anteriores. Já o Selection Sort seleciona o menor elemento restante em cada iteração e o posiciona no início da sublista não ordenada.

O buffer circular, implementado com `deque(maxlen=BUFFER_MAX)`, é o coração do sistema de armazenamento. Definido com uma capacidade máxima de 50 elementos (`BUFFER_MAX = 50`), o `deque` automaticamente descarta os elementos mais antigos quando atinge seu limite, funcionando como uma fila de tamanho fixo. A função `inserir_leitura` adiciona novos valores de temperatura ao buffer, enquanto a função `ordenar_e_mostrar` realiza a análise e a exibição dos dados. Essa função cria uma cópia da lista atual do buffer para evitar modificações no deque original, determina o tamanho da lista e escolhe o algoritmo de ordenação com base em condições definidas: Bubble Sort para listas com menos de 10 elementos (`BUFFER_PADRAO`), Insertion Sort para listas com exatamente 10 elementos e Selection Sort para listas com mais de 10 elementos. Após ordenar a lista, a função calcula a mediana – o elemento central para listas de tamanho ímpar ou a média dos dois elementos centrais para listas de tamanho par – e exibe o método utilizado, a lista ordenada e a mediana formatada com duas casas decimais.

A função `principal` simula o uso do sistema. Inicialmente, ela insere 13 leituras de temperatura predefinidas, representando valores realistas em graus Celsius, e chama `ordenar_e_mostrar` para

processá-las. Como o número de elementos (13) excede `BUFFER_PADRAO`, o Selection Sort é usado. Em seguida, a função simula a adição de mais 37 leituras, geradas com um incremento linear a partir de 24.1 °C, até atingir a capacidade máxima de 50 elementos. Nesse ponto, o buffer contém as 37 novas leituras mais as 13 mais recentes das leituras iniciais e `ordenar_e_mostrar` é chamado novamente, utilizando Selection Sort devido ao tamanho do buffer.

O código é eficiente para o propósito demonstrativo, com uma implementação clara e bem comentada. Ele equilibra simplicidade e funcionalidade, utilizando estruturas de dados adequadas (como o `deque` para o buffer circular) e algoritmos clássicos de ordenação, enquanto a lógica condicional para escolha do algoritmo adiciona um toque de dinamismo ao processamento. A formatação da saída é prática, fornecendo informações úteis como o método de ordenação, a lista ordenada e a mediana, que é uma métrica robusta para análise de temperaturas.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Para ambientes ainda mais restritivos, pode-se substituir listas Python por array('f'), economizando bytes por elemento.

- Outra evolução envolve adaptar Insertion Sort para inserção incremental sem reordenar todo o buffer, movendo apenas a última leitura até sua posição correta. Integrar compressão leve, como codificação diferencial, reduziria ainda mais a largura de banda na transmissão dos dados ordenados.

- Finalmente, portar o código para MicroPython ou CircuitPython – linguagens executadas diretamente em microcontroladores – possibilitaria testes em hardware real, reforçando a aplicabilidade do exercício em projetos de IoT profissionais.

Faremos um segundo exercício para fixar esses conceitos. Educação e prototipagem compõem duas frentes determinantes no processo de formação de novos desenvolvedores, pois permitem que conceitos fundamentais sejam experimentados em cenários de baixa complexidade antes da transição para aplicações industriais. A prática de ordenar conjuntos de dados exemplifica perfeitamente essa abordagem, visto que algoritmos simples revelam, por meio de passos visuais e facilmente rastreáveis, princípios essenciais de comparações, trocas e estados intermediários.

A prototipagem rápida, igualmente, utiliza rotinas de ordenação simples para validar pipelines de dados em aplicativos experimentais ou dispositivos de demonstração. Quando se deseja comprovar a lógica de uma interface ou o comportamento de um sensor que gera dez leituras por segundo, recorrer a uma biblioteca pesada ou dependência externa pode atrasar o processo. Implementações enxutas em Python eliminam etapas de configuração e mantêm o foco na comprovação conceitual. Consequentemente, o domínio desses algoritmos elementares tem valor pedagógico e prático, servindo tanto a estudantes quanto a engenheiros que esboçam provas de conceito.

Exemplo de aplicação

Exemplo 2 – Educação e prototipagem

Um laboratório universitário mantém um kit didático composto por placas microcontroladas conectadas a um notebook que exibe em tempo real leituras de temperatura. Cada sessão de aula dura cinquenta minutos, período no qual o professor demonstra, passo a passo, como ordenar sequências de até vinte valores coletados pelo sensor.

As regras definem que, ao receber uma nova série de leituras, o aplicativo deverá: primeiro aplicar Bubble Sort para mostrar a evolução do vetor completamente desordenado; depois realizar Insertion Sort em um vetor parcialmente organizado, destacando ganhos de desempenho; por fim exibir Selection Sort, enfatizando a contagem de trocas mínima.

Cada execução imprime o vetor a cada iteração, permitindo que estudantes acompanhem visualmente as mudanças. Ao final, o sistema apresenta o número total de comparações e trocas, reforçando a análise de custos empíricos.

O exercício emprega recursos idiomáticos de Python que, embora simples, espelham técnicas utilizadas no mercado. A instrução de desembrulhar troca (`a[i], a[j] = a[j], a[i]`) corresponde a operações de registradores em linguagens de baixo nível, encurtando o código sem sacrificar clareza. Decoradores de função com contadores internos demonstram como capturar métricas em tempo de execução, replicando práticas de profiling usadas em engenharia de desempenho. O módulo `textwrap` provê indentação uniforme para impressões de vetores ao longo das iterações, recurso que favorece legibilidade em relatórios educacionais.

Lembrete

Embora Python disponha da função sorted, as versões manuais de Bubble, Selection e Insertion exibem explicitamente laços aninhados, comparações e deslocamentos, revelando detalhes sobre ordem de complexidade quadrática.

Além disso, a utilização de parâmetros de tipo via from __future__ import annotations documenta assinaturas de função, compatibilizando o código com ferramentas de análise estática cada vez mais presentes em pipelines corporativos.

Funções geradoras com `yield` produzem o estado do vetor a cada etapa, possibilitando integração posterior com animações ou guias interativos no navegador sem reescrever o núcleo dos algoritmos. A seguir, consta o código-fonte da solução:

```
from __future__ import annotations
from typing import List, Callable, Generator
from textwrap import indent

# Decorador para contar comparações e trocas
def estatisticas(func: Callable[[List[int]],
Generator[List[int], None, None]]):
    def wrapper(lista: List[int]) -> List[int]:
        comparacoes = trocas = 0
        for estado, c, t in func(lista.copy()):
            comparacoes += c
            trocas += t
            print(indent(str(estado), " "))
        print(f"Comparações: {comparacoes}, Trocas: {trocas}\n")
        return lista
    return wrapper

@estatisticas
def bubble_sort_passo(lista: List[int]) ->
Generator[tuple[List[int], int, int], None, None]:
    n = len(lista)
    for i in range(n - 1):
        trocou = False
        for j in range(n - 1 - i):
            yield lista, 1, 0
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
                trocou = True
                yield lista, 0, 1
        if not trocou:
            break
    yield lista, 0, 0

@estatisticas
def insertion_sort_passo(lista: List[int]) ->
Generator[tuple[List[int], int, int], None, None]:
    for i in range(1, len(lista)):
        chave = lista[i]
        j = i - 1
        while j >= 0 and lista[j] > chave:
            yield lista, 1, 0
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = chave
        yield lista, 1, 1 # última comparação e inserção
    yield lista, 0, 0

@estatisticas
def selection_sort_passo(lista: List[int]) ->
Generator[tuple[List[int], int, int], None, None]:
    n = len(lista)
    for i in range(n - 1):
        min_idx = i
        for j in range(i + 1, n):
            yield lista, 1, 0
            if lista[j] < lista[min_idx]:
```

```
                    min_idx = j
            if min_idx != i:
                lista[i], lista[min_idx] = lista[min_idx], lista[i]
                yield lista, 0, 1
        yield lista, 0, 0

def principal():
    dados_originais = [18, 5, 12, 7, 1, 9, 3, 15]
    print("Bubble Sort:")
    bubble_sort_passo(dados_originais)
    dados_parciais = [1, 3, 5, 7, 12, 9, 15, 18] # quase ordenado
    print("Insertion Sort:")
    insertion_sort_passo(dados_parciais)

    dados_auditoria = [9, 4, 16, 2, 11, 6, 3, 14]
    print("Selection Sort:")
    selection_sort_passo(dados_auditoria)

if __name__ == "__main__":
        principal()
```

O quadro 16 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 16 – Funções usadas no código-fonte do segundo exercício prático

Função ou construção	Explicação de uso
@decorator	Decorador é uma função que recebe outra função como argumento e retorna uma nova função. No código, é usado para medir comparações e trocas
Callable[[T], U]	Tipo genérico da biblioteca typing, que representa uma função que recebe um argumento do tipo T e retorna um valor do tipo U
Generator[YieldType, SendType, ReturnType]	Anotação de tipo usada para indicar que uma função retorna um gerador. No código, os algoritmos de ordenação usam `Generator[List[int], None, None]`
yield valor, c, t	Pausa a execução da função e retorna múltiplos valores (no caso, estado da lista, número de comparações e trocas) para serem consumidos passo a passo
indent(texto, prefixo)	Função da biblioteca textwrap que adiciona um prefixo (como espaços) ao início de cada linha de um texto. No código, é usada para formatar a exibição dos estados intermediários
lista.copy()	Cria uma cópia rasa da lista original. Importante no decorador para evitar alterar a lista original ao passo que se avaliam os algoritmos
tupla[List[int], int, int]	Uso de uma tupla com anotação de tipos, retornando múltiplos valores com tipos diferentes de maneira explícita: a lista ordenada parcial, o número de comparações e o de trocas

O programa começa com a importação de módulos e funcionalidades específicas. A diretiva `from __future__ import annotations` habilita o uso de anotações de tipo pós-avaliadas, permitindo que tipos como `List[int]` sejam usados sem avaliação imediata, o que é útil em declarações recursivas ou complexas. O módulo `typing` fornece os tipos `List`, `Callable` e `Generator`, garantindo que as assinaturas das funções sejam claras e tipadas estaticamente. O módulo `textwrap` é usado para indentar a saída, facilitando a visualização dos estados das listas.

A função `estatisticas` é um decorador que encapsula a lógica de monitoramento das ordenações. Ela recebe uma função que retorna um gerador, o qual produz tuplas contendo o estado atual da lista, o número de comparações e o número de trocas em cada passo. O decorador inicializa contadores para

comparações e trocas, itera sobre os estados gerados pela função decorada, acumula as estatísticas e imprime cada estado com indentação. A função interna `wrapper` trabalha com uma cópia da lista de entrada para evitar modificações indesejadas na lista original e retorna a lista original (não ordenada, já que a ordenação ocorre na cópia). Esse design permite que o decorador seja reutilizado por diferentes algoritmos de ordenação.

A implementação do Bubble Sort, na função `bubble_sort_passo`, utiliza um gerador para produzir os estados intermediários. Para uma lista de tamanho `n`, o algoritmo percorre a lista repetidamente, comparando pares de elementos adjacentes. Em cada iteração interna, ele gera o estado atual da lista antes de uma comparação (com uma comparação contabilizada) e, se uma troca for necessária, realiza a troca e gera o estado atualizado (com uma troca contabilizada). A otimização clássica do Bubble Sort está presente: se nenhuma troca ocorrer em uma passagem, a lista está ordenada e o algoritmo termina cedo. O uso do gerador permite pausar a execução em cada passo, possibilitando a visualização detalhada do processo.

A função `insertion_sort_passo`, que implementa o Insertion Sort, também utiliza um gerador. Para cada elemento da lista, a partir do segundo (índice 1), o algoritmo seleciona uma chave e a insere na porção ordenada à esquerda. Durante o deslocamento dos elementos maiores que a chave, o gerador produz o estado atual antes de cada comparação. Após o deslocamento, a chave é inserida e o estado é gerado novamente, contabilizando uma comparação e uma troca. Essa abordagem reflete a natureza incremental do Insertion Sort, que constrói uma sublista ordenada passo a passo.

O Selection Sort, implementado em `selection_sort_passo`, segue uma lógica semelhante. Para cada posição `i`, o algoritmo procura o menor elemento na sublista não ordenada (de `i+1` até o final) e o troca com o elemento na posição `i` se necessário. O gerador produz o estado da lista antes de cada comparação durante a busca do mínimo e após uma troca, quando aplicável. A troca somente ocorre se o índice do menor elemento (`min_idx`) for diferente de `i`, evitando trocas desnecessárias.

A função `principal` coordena a execução dos algoritmos, criando listas de teste específicas para cada um. A lista para o Bubble Sort `[18, 5, 12, 7, 1, 9, 3, 15]` é desordenada, representando um caso geral. Para o Insertion Sort, a lista `[1, 3, 5, 7, 12, 9, 15, 18]` está quase ordenada, destacando a eficiência do algoritmo nesse cenário. Para o Selection Sort, a lista `[9, 4, 16, 2, 11, 6, 3, 14]` é outra sequência desordenada. Cada algoritmo é chamado com sua respectiva lista e os estados intermediários, com as estatísticas de comparações e trocas, são exibidos.

O uso de geradores permite eficiência na geração de estados sob demanda, enquanto o decorador abstrai a lógica de monitoramento, promovendo reutilização. As anotações de tipo aumentam a legibilidade e facilitam a verificação estática. A escolha de listas de teste variadas demonstra o comportamento dos algoritmos em diferentes cenários e a saída detalhada é útil para fins educacionais ou de depuração.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Uma extensão valiosa envolveria a integração de gráficos gerados com Matplotlib, nos quais cada barra representa um elemento do vetor e muda de cor ao ser comparada ou trocada, criando animações ao vivo.

- Outra evolução incluiria a parametrização do decorador para registrar tempos de execução em microssegundos, aproximando a experiência de benchmarks profissionais.
- Seria possível adicionar um modo de comparação randômica que gere tipos de entrada específicos – crescente, decrescente, embaralhada –, destacando o impacto no número de operações.

- A exportação dos logs para um arquivo CSV abriria espaço para análises em planilhas ou painéis interativos, aproximando o exercício de metodologias de ciência de dados adotadas no setor.

4.2 Ordenação avançada: Merge Sort, Quick Sort e análise comparativa

Entre os algoritmos de ordenação avançados, destacam-se o Merge Sort e o Quick Sort, ambos amplamente utilizados devido à sua eficiência em cenários distintos. Esses métodos superam algoritmos mais simples, como Bubble Sort ou Insertion Sort, especialmente em grandes volumes de dados, pois apresentam complexidades de tempo mais favoráveis. A análise comparativa entre Merge Sort e Quick Sort revela suas características, vantagens e limitações, permitindo uma escolha informada conforme o contexto de aplicação.

O Merge Sort é um algoritmo baseado na estratégia de divisão e conquista. Ele opera dividindo recursivamente o conjunto de dados em subconjuntos menores até que cada um contenha um único elemento. Em seguida, esses subconjuntos são combinados (ou mesclados) de forma ordenada, reconstruindo o conjunto final. Essa abordagem garante uma complexidade de tempo de O(n log n) em todos os casos, seja no melhor, médio ou pior cenário. A estabilidade do Merge Sort, que preserva a ordem relativa de elementos iguais, é uma vantagem significativa em aplicações que exigem essa propriedade. Além disso, sua previsibilidade o torna adequado para sistemas nos quais a consistência de desempenho é crucial. Contudo, o Merge Sort tem desvantagens notáveis. Ele requer espaço adicional de memória proporcional ao tamanho do conjunto de dados, o que pode ser um entrave em sistemas com recursos limitados. Ademais, a necessidade de cópias frequentes durante a mesclagem pode impactar o desempenho em termos de constantes de tempo, mesmo que a complexidade assintótica seja ótima.

Por outro lado, o Quick Sort também adota a estratégia de divisão e conquista, mas com uma abordagem distinta. Ele seleciona um elemento, chamado pivô, e particiona o conjunto de dados de modo que elementos menores que o pivô fiquem à sua esquerda e maiores à sua direita. Esse processo é aplicado recursivamente às partições resultantes até que o conjunto esteja ordenado. A eficiência do Quick Sort reside em sua complexidade média de O(n log n), que o torna extremamente rápido na prática, especialmente devido ao baixo overhead de operações. Diferentemente do Merge Sort, o Quick Sort é um algoritmo in-place, exigindo apenas uma quantidade constante de memória adicional, o que o torna mais eficiente em termos de espaço. No entanto, sua performance depende fortemente da escolha do pivô. Em cenários desfavoráveis, como quando o pivô é o menor ou o maior elemento em um conjunto quase ordenado, a complexidade pode degradar para $O(n^2)$. Estratégias como a escolha de um pivô aleatório ou a mediana de três elementos mitigam esse risco, mas não eliminam completamente a variabilidade de desempenho. Além disso, o Quick Sort não é estável, o que pode ser uma limitação em certas aplicações.

Em termos práticos, o Quick Sort tende a ser mais rápido que o Merge Sort na maioria dos casos médios devido às constantes menores em suas operações, mas sua instabilidade e risco de pior caso requerem cautela. Em implementações modernas, é comum combinar esses algoritmos com outros métodos, como o Insertion Sort para pequenos subconjuntos, criando soluções híbridas que maximizam a eficiência. O quadro 17 compara todos os algoritmos descritos neste tópico.

Quadro 17 – Algoritmos de ordenação: comparativo

Critério	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
Complexidade melhor caso	O(n)	$O(n^2)$	O(n)	O(n log n)	O(n log n)
Complexidade médio caso	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n log n)	O(n log n)
Complexidade pior caso	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n log n)	$O(n^2)$
Complexidade espacial	O(1)	O(1)	O(1)	O(n)	O(log n)
Estabilidade	Estável	Não estável	Estável	Estável	Não estável
Método principal	Comparações e trocas adjacentes	Seleção do menor elemento	Inserção incremental	Divisão e conquista com fusão	Divisão e conquista com pivô
Facilidade de implementação	Alta	Alta	Alta	Moderada	Moderada
Eficiência com listas pequenas	Moderada	Moderada	Alta	Moderada	Alta
Eficiência com listas grandes	Baixa	Baixa	Baixa	Alta	Alta
Vantagem principal	Implementação simples e estabilidade	Menos trocas de dados	Excelente desempenho com listas quase ordenadas	Complexidade previsível	Desempenho rápido em média, baixo uso de memória
Desvantagem principal	Muito lento para grandes volumes de dados	Ineficiente para listas grandes	Lento para listas grandes em pior caso	Exige memória adicional proporcional aos dados	Escolha inadequada do pivô pode gerar pior caso quadrático

BigData e DataScience constituem a espinha dorsal da economia digital, impulsionando sistemas de recomendação, detecção de fraudes e previsões de demanda baseadas em trilhões de pontos de dados. BigData concentra-se na ingestão, no armazenamento e no movimento eficientes de volumes massivos de informação, enquanto DataScience volta-se a extrair valor analítico desses conjuntos, transformando dados brutos em modelos que orientam decisões de negócio. Em ambos os campos, operações de ordenação são onipresentes: desde preparar datasets para algoritmos de aprendizado supervisionado – que exigem partições estratificadas – até acelerar buscas binárias e junções em bancos analíticos.

Quando os dados se tornam volumosos a ponto de exceder a capacidade da memória RAM disponível em um servidor, métodos avançados de ordenação, como Merge Sort e Quick Sort, ganham importância decisiva. O Merge Sort destaca-se pela complexidade consistente O(n log n) e pela capacidade eficiente de ordenar dados que estão armazenados em disco, uma característica essencial em grandes sistemas que manipulam

arquivos externos, como ocorre nas plataformas distribuídas de processamento de dados, a exemplo do Apache Spark. Nesse contexto, grandes volumes de informação são divididos em blocos menores, cada um dos quais é inicialmente ordenado separadamente e, posteriormente, combinado em etapas sucessivas até formar um conjunto final ordenado.

Por sua vez, o Quick Sort tornou-se uma opção preferencial em diversas bibliotecas populares, incluindo NumPy e pandas, devido ao excelente desempenho quando os dados permanecem próximos ao cache da CPU e à pequena quantidade de movimentações exigidas durante sua execução. Contudo, para que se evite a degradação do desempenho para o pior cenário quadrático $O(n^2)$, é necessário realizar uma escolha cuidadosa do elemento central (pivô) usado na divisão dos dados.

Compreender essas técnicas de ordenação permite que profissionais estabeleçam processos mais confiáveis para tratamento de grandes volumes de dados, definam corretamente as configurações para ambientes computacionais distribuídos e identifiquem pontos críticos que possam estar reduzindo a eficiência do processamento em etapas típicas de extração e transformação de dados (ETL).

Durante a preparação de conjuntos de dados voltados para aplicações em Machine Learning, os algoritmos de ordenação têm papel crucial ao organizar informações cronologicamente, facilitando, assim, a divisão correta de períodos de tempo. Além disso, essas técnicas ajudam a identificar e remover valores repetidos, realizar amostras mais representativas dos dados originais e agrupar registros semelhantes antes de calcular estatísticas móveis ou médias móveis ao longo do tempo. Modelos que trabalham com séries temporais dependem especialmente de dados ordenados cronologicamente, enquanto modelos baseados em algoritmos de árvore frequentemente necessitam de categorias organizadas para melhorar o desempenho.

Desse modo, o domínio dos métodos Merge Sort e Quick Sort ultrapassa o aspecto teórico, assumindo papel essencial para assegurar que os processos de preparação e transformação de dados produzam conjuntos consistentes e de alta qualidade, permitindo que o processo de criação de atributos para treinamento dos modelos ocorra de maneira eficiente e contínua.

Exemplo de aplicação

Exemplo 3 – BigData e DataScience

Uma fintech analisa transações de cartão de crédito contendo cem milhões de registros por dia. Antes de alimentar o modelo de detecção de fraude em tempo real, ela precisa garantir que cada lote esteja ordenado por timestamp.

Os requisitos estipulam:

- o pipeline local simulará a ordenação de vinte milhões de carimbos de tempo, gerados aleatoriamente, para avaliar a escalabilidade dos algoritmos;
- Merge Sort deve ser utilizado como referência para latência previsível em lote (chunk de 500.000 linhas);

- Quick Sort com escolha de pivô mediana-de-três deve ser testado para processamento inteiramente em memória, medindo desempenho e uso de RAM;

- ao final, o sistema exibirá a verificação de estabilidade (igualdade de sequência para elementos repetidos) e o tempo total de cada método, reportando qual atende ao SLA interno de ordenar dez milhões de registros em menos de trinta segundos na máquina de prototipagem.

Essas regras influenciam a implementação ao exigir tanto chunking adaptativo (para Merge Sort) quanto instrumentação de tempo e uso de memória (via `tracemalloc`).

O exercício utiliza funções geradoras e escrita temporária em disco para demonstrar o processo de ordenação externa, simulando o comportamento observado em sistemas que trabalham com grandes volumes de dados armazenados em ambientes distribuídos. No Python, a função `heapq.merge` combina sequências previamente ordenadas, empregando uma estratégia que posterga o processamento dos dados até que sejam efetivamente necessários, o que minimiza o consumo máximo de memória.

Para implementar o algoritmo Quick Sort, adota-se uma estratégia recursiva com uso mínimo de memória adicional, definindo limites para evitar profundidade excessiva nas chamadas recursivas. Além disso, seleciona-se o pivô utilizando a técnica que escolhe o elemento central com base na mediana entre três valores diferentes, garantindo maior robustez ao algoritmo. Para casos especiais de pequenas sublistas (menores que 32 elementos), o Quick Sort é substituído pelo Insertion Sort, prática comum nas principais bibliotecas disponíveis no mercado, como ocorre no Timsort, que também combina abordagens semelhantes.

O monitoramento do uso de memória é feito utilizando o módulo `tracemalloc`, que registra o estado exato da alocação antes e após o processamento, possibilitando uma análise detalhada da variação em bytes consumidos pela operação.

Ao gerar artificialmente um grande conjunto de dados composto por vinte milhões de marcas temporais, é necessário utilizar uma estrutura otimizada, como o `array('L')`, que armazena números inteiros de 32 bits de maneira compacta, permitindo acomodar grandes volumes de informação sem esgotar os recursos de memória disponíveis. A geração dos números pseudoaleatórios ocorre em grandes blocos, aproveitando funções eficientes como `random.getrandbits`, que reduz significativamente o custo computacional e evita múltiplas chamadas desnecessárias.

As medições precisas do desempenho temporal são obtidas através da função `time.perf_counter`, que fornece resultados de alta precisão, com capacidade de medir intervalos menores que um milésimo de segundo. Embora o código tenha sido escrito utilizando apenas recursos padrão do Python, ele reproduz técnicas consagradas em cenários reais, tais como o processamento em blocos sequenciais para gerenciar conjuntos de dados maiores que a capacidade da memória RAM, integração com ferramentas para monitoramento detalhado de desempenho e comparação cuidadosa entre métodos distintos, permitindo identificar e selecionar aquela abordagem que apresente o melhor desempenho e eficiência em ambientes reais de operação. A seguir, consta o código-fonte da solução:

```
from __future__ import annotations
import heapq, os, random, tempfile, time, tracemalloc
from array import array
from typing import List, Iterator

# -------- Geração de dados compactados --------
def gerar_timestamps(qtd: int) -> array:
    ar = array("L")
    bits32 = random.getrandbits
    for _ in range(qtd):
        ar.append(bits32(32))
    return ar

# -------- Quick Sort otimizado --------
def quick_sort(arr: List[int], inicio=0, fim=None):
    if fim is None:
        fim = len(arr) - 1
    while inicio < fim:
        if fim - inicio < 32: # fallback
            insertion(arr, inicio, fim)
            return
        pivo = mediana_tres(arr, inicio, fim)
        i, j = inicio, fim
        while i <= j:
            while arr[i] < pivo: i += 1
            while arr[j] > pivo: j -= 1
            if i <= j:
                arr[i], arr[j] = arr[j], arr[i]
                i, j = i + 1, j - 1
        # recursão na menor partição para limitar profundidade
        if j - inicio < fim - i:
            quick_sort(arr, inicio, j)
            inicio = i
        else:
            quick_sort(arr, i, fim)
            fim = j

def mediana_tres(a: List[int], i: int, j: int) -> int:
    k = (i + j) // 2
    if a[i] > a[k]: a[i], a[k] = a[k], a[i]
    if a[k] > a[j]: a[k], a[j] = a[j], a[k]
    if a[i] > a[k]: a[i], a[k] = a[k], a[i]
    return a[k]

def insertion(a: List[int], i: int, j: int):
    for x in range(i + 1, j + 1):
        chave, y = a[x], x - 1
        while y >= i and a[y] > chave:
            a[y + 1] = a[y]; y -= 1
        a[y + 1] = chave

# -------- Merge Sort externo com heapq.merge --------
CHUNK = 500_000

def escrever_chunks(dados: array) -> List[str]:
    arquivos = []
```

```
    for i in range(0, len(dados), CHUNK):
        parte = sorted(dados[i:i + CHUNK])
        tmp = tempfile.NamedTemporaryFile(delete=False)
        tmp.write(array("L", parte).tobytes())
        tmp.close()
        arquivos.append(tmp.name)
    return arquivos

def ler_chunk(caminho: str) -> Iterator[int]:
    with open(caminho, "rb") as f:
        bloco = array("L")
        bloco.frombytes(f.read())
        for valor in bloco:
            yield valor

def merge_externo(arquivos: List[str], destino: str):
    iteradores = [ler_chunk(p) for p in arquivos]
    with open(destino, "wb") as out:
        for valor in heapq.merge(*iteradores):
            out.write(array("L", [valor]).tobytes())
    for p in arquivos:
        os.remove(p)

# -------- Orquestração e métricas --------
def medir(func, *args, **kwargs):
    tracemalloc.start()
    ini = time.perf_counter()
    resultado = func(*args, **kwargs)
    dur = time.perf_counter() - ini
    pico = tracemalloc.get_traced_memory()[1] / 1_048_576
    tracemalloc.stop()
    return resultado, dur, pico

def principal():
    registros = 10_000_00 # 10 milhões para protótipo
    print(f"Gerando {registros:,} timestamps…")
    dados, t_gen, mem_gen = medir(gerar_timestamps, registros)
    print(f"Geração em {t_gen:.2f}s, pico {mem_gen:.1f} MiB\n")

    # Quick Sort
    lista_quick = list(dados)
    print("Quick Sort (medianadetrês)…")
    _, t_qs, mem_qs = medir(quick_sort, lista_quick)
    print(f"Tempo: {t_qs:.2f}s, Pico RAM: {mem_qs:.1f} MiB\n")

    # Merge Sort externo
    print("Merge Sort externo em chunks…")
    arqs, t_split, mem_split = medir(escrever_chunks, dados)
    merge_destino = tempfile.NamedTemporaryFile(delete=False).name
    _, t_merge, mem_merge = medir(merge_externo, arqs, merge_destino)
    tempo_total_merge = t_split + t_merge
    pico_merge = max(mem_split, mem_merge)
    print(f"Tempo total: {tempo_total_merge:.2f}s, Pico RAM: {pico_
merge:.1f} MiB")
    print(f"Arquivo mesclado armazenado em {merge_destino}")

if __name__ == "__main__":
        principal()
```

O quadro 18 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 18 – Funções usadas no código-fonte do terceiro exercício prático

Função ou construção	Explicação de uso
import heapq	Importa o módulo que fornece algoritmos de fila de prioridade e mesclagem ordenada eficiente. No código, é usado para `heapq.merge`
import os, tempfile	`os` fornece funções para manipulação de arquivos e diretórios. `tempfile` permite criar arquivos temporários de forma segura
import tracemalloc	Módulo que mede o uso de memória durante a execução do programa, útil para diagnósticos de desempenho
from array import array	Importa a estrutura array, usada para armazenar grandes volumes de números de forma compacta e eficiente em memória
array("L")	Cria um array de inteiros não negativos de 32 bits (formato 'L'). Usado para armazenar timestamps
random.getrandbits(n)	Gera um inteiro com exatamente n bits aleatórios. Aqui, gera timestamps simulados de 32 bits
array.tobytes() / frombytes()	Converte um array para bytes (para salvar em arquivo) ou reconstrói um array a partir de bytes lidos
tempfile.NamedTemporaryFile (delete=False)	Cria um arquivo temporário com nome acessível e impede sua exclusão automática ao fechar. Útil para simular armazenamento externo
heapq.merge(iteradores)	Mescla vários iteradores ordenados em um único fluxo ordenado. Usado no merge sort externo para processar os chunks já ordenados. Observação: chunks são os segmentos ordenados do conjunto de dados que o algoritmo processa e mescla
os.remove(caminho)	Remove um arquivo do sistema de arquivos. Utilizado para limpar os arquivos temporários criados nos chunks
time.perf_counter()	Retorna o tempo de alta resolução em segundos. Ideal para medições de desempenho mais precisas
tracemalloc.get_traced_ memory() [1]	Obtém a memória máxima alocada desde o início do rastreamento. O valor é convertido para mebibytes (MiB). Observação: mebibytes refere-se a uma unidade de medida de memória igual a 2^{20} bytes (1.048.576 bytes). Diferentemente de megabytes (MB), que são baseados no sistema decimal (10^6 bytes), mebibytes usam o sistema binário, sendo mais precisos em contextos computacionais
max(a, b)	Retorna o maior valor entre a e b. Aqui, compara os picos de memória de duas fases do merge sort

O programa começa importando módulos essenciais. A instrução `from __future__ import annotations` habilita anotações de tipo postergadas, permitindo maior flexibilidade em definições de tipos. Bibliotecas como `heapq` (mesclagem eficiente), `os` (manipulação de arquivos), `random` (geração de números aleatórios), `tempfile` (arquivos temporários), `time` (medição de tempo), `tracemalloc` (rastreamento de memória) e `array` (estruturas de dados compactas) são usadas para suportar as funcionalidades do código. O módulo `typing` fornece tipos como `List` e `Iterator` para anotações precisas.

A primeira função, `gerar_timestamps`, cria um vetor de timestamps aleatórios usando o tipo `array` com o código "`L`" (inteiros longos sem sinal de 32 bits). A função recebe a quantidade desejada de registros e utiliza `random.getrandbits(32)` para gerar valores aleatórios de 32 bits, que são armazenados no array. Essa abordagem é eficiente, pois o tipo `array` consome menos memória que listas Python padrão, sendo ideal para grandes volumes de dados.

O Quick Sort otimizado é implementado nas funções `quick_sort`, `mediana_tres` e `insertion`. A função `quick_sort` ordena uma lista de inteiros in-place, utilizando a estratégia de partição com pivô. Para evitar a recursão excessiva, que pode causar estouro de pilha em grandes conjuntos, a função itera sobre a maior partição e chama recursivamente apenas a menor, reduzindo a profundidade da pilha. Para conjuntos pequenos (menos de 32 elementos), ela alterna para o `insertion`, um algoritmo de inserção, que é mais eficiente em casos com poucos elementos devido à baixa sobrecarga. A escolha do pivô é feita pela função `mediana_tres`, que seleciona a mediana entre os elementos nas posições inicial, final e central do intervalo; ordenando-os para garantir uma escolha robusta que minimize casos patológicos, como arrays quase ordenados. A função `insertion` realiza a ordenação por inserção, movendo elementos maiores para abrir espaço para a chave atual, garantindo estabilidade em pequenos subconjuntos.

O Merge Sort externo é projetado para lidar com grandes volumes de dados que não cabem na memória RAM, sendo implementado nas funções `escrever_chunks`, `ler_chunk` e `merge_externo`. A constante `CHUNK` define o tamanho de cada bloco (500 mil elementos), balanceando uso de memória e eficiência de E/S. A função `escrever_chunks` divide o array de entrada em blocos menores, ordena cada um usando a função `sorted` (internamente utiliza Timsort, eficiente para dados parcialmente ordenados) e escreve os blocos ordenados em arquivos temporários criados com `tempfile.NamedTemporaryFile`. Esses arquivos são armazenados em formato binário usando o método `tobytes` do tipo `array`, otimizando espaço e velocidade de leitura/escrita. A função `ler_chunk` lê um arquivo temporário e retorna um iterador que gera os valores do bloco, minimizando o uso de memória ao evitar carregar todo o bloco de uma vez. A função `merge_externo` combina os blocos ordenados usando `heapq.merge`, que mescla iteradores de forma eficiente, escrevendo o resultado em um arquivo de destino. Após a mesclagem, os arquivos temporários são excluídos, liberando espaço em disco.

A função `medir` é uma utilidade que monitora o desempenho das funções principais, medindo o tempo de execução com `time.perf_counter` (oferece alta precisão) e o pico de uso de memória com `tracemalloc`, convertendo o resultado para MiB. Ela retorna uma tupla com o resultado da função, o tempo decorrido e o pico de memória, permitindo análises detalhadas.

A função `principal` orquestra a execução do programa. Inicialmente, ela define a geração de 10 milhões de timestamps, chamando `gerar_timestamps` e medindo seu desempenho. Em seguida, testa o Quick Sort, convertendo o array para uma lista (Quick Sort opera in-place em listas) e aplicando a função `quick_sort`. Para o Merge Sort externo, utiliza `escrever_chunks` para criar os blocos ordenados e `merge_externo` para combiná-los em um arquivo final, cujo caminho é informado ao usuário. Os tempos e os picos de memória de cada etapa são exibidos, permitindo uma comparação direta entre os algoritmos.

O código é executado apenas se o módulo for o principal `if __name__ == "__main__"`, chamando `principal`. A implementação é robusta, com otimizações como o uso de arrays compactos, pivôs bem escolhidos no Quick Sort, e divisão em blocos no Merge Sort externo, tornando-o adequado para grandes conjuntos de dados. As medições de desempenho fornecem insights valiosos sobre o comportamento dos algoritmos em termos de tempo e memória, destacando a eficiência do

Merge Sort externo para cenários com restrições de RAM e a rapidez do Quick Sort para dados que cabem na memória.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Uma extensão natural seria paralelizar a fase de ordenação dos chunks usando `concurrent.futures.ProcessPoolExecutor`, simulando nós de cluster.
- Outra ampliação envolveria adicionar compressão opcional (gzip, zstd) nos arquivos temporários, avaliando trade-offs entre CPU e I/O.
- Para demonstrar estabilidade, incluir campos extras nos registros (por exemplo, `id_transação`) e verificar se Merge Sort preserva ordem relativa.
- Finalmente, exportar as métricas para Prometheus ou DataDog facilitaria o monitoramento contínuo de pipelines, aproximando o exercício de práticas DevOps habituais em equipes de Data Engineering.

Além dos algoritmos mais comuns apresentados neste tópico – Bubble Sort, Selection Sort, Insertion Sort, Merge Sort e Quick Sort – existem técnicas de ordenação com propriedades próprias e aplicabilidades específicas, particularmente em contextos nos quais características especiais dos dados possam ser exploradas para ganhos significativos de eficiência.

Entre essas técnicas adicionais, destaca-se inicialmente o Heap Sort, fundamentado na estrutura de dados heap (veremos com mais detalhes no tópico 6). Esse algoritmo estrutura o conjunto inicial em um heap máximo, no qual o maior elemento situa-se sempre no topo, facilitando sua retirada e subsequente reorganização eficiente do heap restante. O Heap Sort apresenta complexidade temporal garantidamente O(n log n) em todos os casos, sem depender da disposição inicial dos dados, além de operar in-place (sem memória adicional proporcional ao tamanho da entrada). Contudo, ele não é estável, ou seja, não garante manutenção da ordem relativa de registros com chaves iguais.

Outro método relevante é o Counting Sort, aplicável especificamente a contextos em que os dados assumem valores inteiros em um intervalo relativamente pequeno e conhecido. O Counting Sort não usa comparações diretas entre os elementos. Ao invés disso, conta quantas vezes cada valor ocorre e reconstrói a sequência ordenada diretamente a partir dessas frequências. Como consequência, atinge complexidade linear O(n + k), sendo k o valor máximo possível dos elementos. A abordagem revela-se excepcionalmente rápida, particularmente quando o intervalo de valores é reduzido. Além disso, esse algoritmo é estável quando implementado corretamente. No entanto, sua aplicabilidade limita-se essencialmente a dados inteiros ou chaves facilmente enumeráveis.

Complementando o Counting Sort, existem também o Radix Sort e o Bucket Sort, técnicas relacionadas, mas distintas. O Radix Sort organiza dados numéricos com base na representação posicional (dígitos), realizando múltiplas passagens, cada uma ordenando os elementos por um dígito específico, começando

pelo menos significativo até alcançar o mais significativo. Com o uso interno de Counting Sort estável ou outro método similar, o Radix Sort obtém complexidade linear $O(d \times (n + k))$, sendo d o número de dígitos máximos dos valores. A estabilidade é característica natural desse método, favorecendo seu uso em contextos que exigem preservação das relações originais.

Já o Bucket Sort divide os dados em baldes (intervalos) com base em alguma propriedade dos elementos (frequentemente uma distribuição uniforme dos valores). Cada balde recebe uma quantidade limitada de elementos que podem ser posteriormente ordenados por outra técnica, geralmente mais simples (Insertion Sort). O desempenho depende criticamente da uniformidade da distribuição, podendo alcançar complexidade média linear $O(n)$ em condições favoráveis. Esse método é estável desde que a técnica de ordenação interna dos baldes também seja estável.

Existem também métodos híbridos, como o Tim Sort, empregado pelo Python em sua biblioteca padrão, que combina Merge Sort com Insertion Sort para aproveitar a eficiência deste último em pequenos blocos ou em listas parcialmente ordenadas, garantindo desempenho otimizado em contextos práticos variados. O Tim Sort apresenta complexidade $O(n \log n)$ no pior caso, estabilidade garantida e excelente desempenho empírico em situações reais.

Ainda mais específico é o Shell Sort, que representa uma extensão conceitual do Insertion Sort. O Shell Sort realiza comparações e trocas distantes inicialmente e reduz progressivamente essa distância, tornando-se cada vez mais semelhante ao Insertion Sort tradicional até a ordenação total. Embora a complexidade teórica seja difícil de determinar precisamente, estudos indicam comportamento empírico em torno de $O(n^{(3/2)})$ a $O(n \log^2 n)$, dependendo da sequência de intervalos adotada. Sua vantagem é a combinação entre simplicidade de implementação e desempenho superior ao Insertion Sort em muitos casos práticos.

Saiba mais

O clássico livro a seguir cobre algoritmos de ordenação (Quick Sort, Merge Sort e Heap Sort) em detalhes, com explicações teóricas, pseudocódigo e análise de complexidade. Ele é ideal para estudantes e profissionais que buscam uma base sólida.

CORMEN, T. H. *et al. Algoritmos*: teoria e prática. 4. ed. Rio de Janeiro: LTC, 2024.

Outra referência importante é a seguinte obra, que oferece uma abordagem prática e acessível aos algoritmos de ordenação. Ela é amplamente utilizada em cursos de ciência da computação e equilibra teoria com aplicações práticas.

SEDGEWICK, R.; WAYNE, K. *Algorithms*. 4. ed. Reading: Addison-Wesley Professional, 2011.

Esses algoritmos adicionais ampliam significativamente o arsenal disponível para lidar com diferentes tipos de conjuntos de dados e necessidades de desempenho. Cada método possui contextos particulares nos quais sua eficiência se destaca, seja por explorar peculiaridades estruturais dos dados, garantir estabilidade, consumir pouca memória adicional ou possuir complexidade previsível. Portanto, o estudo dessas técnicas complementares auxilia desenvolvedores e pesquisadores a tomar decisões conscientes na escolha do método mais adequado, considerando especificidades do problema, recursos computacionais e características intrínsecas aos dados envolvidos.

Resumo

Nesta unidade, exploramos detalhadamente as estruturas de dados não lineares, com ênfase especial nas árvores e nos grafos, esclarecendo suas particularidades, aplicações práticas e implementações em Python.

As árvores foram introduzidas como estruturas intuitivas, que permitem representar dados hierarquicamente, comparadas visualmente às árvores botânicas. Essa analogia auxilia no entendimento da organização dos elementos, compostos por um nó raiz e sucessivas ramificações, formando subárvores. As árvores binárias, especificamente, são destacadas pela propriedade de possuírem no máximo dois filhos por nó, classificados como esquerdo e direito. Detalhamos também as formas básicas de percorrer árvores binárias: pré-ordem (visitando a raiz antes das subárvores), em ordem (visitando primeiro a subárvore esquerda, depois a raiz e, em seguida, a direita) e pós-ordem (visitando subárvores antes da raiz).

Outro tópico relevante são os grafos, definidos como estruturas mais flexíveis, capazes de modelar relações complexas, incluindo ciclos e conexões bidirecionais. Grafos podem ser representados por matrizes ou listas de adjacência, sendo a primeira adequada para grafos densos e a segunda ideal para grafos com menos conexões. A exploração dos grafos ocorre principalmente por DFS e BFS. DFS percorre o grafo explorando cada caminho completamente antes de recuar, sendo útil para detectar ciclos e componentes conectados. Já a BFS avança nível por nível, identificando o caminho mais curto em grafos não ponderados e sendo empregado em problemas como redes sociais e logística.

Os algoritmos de ordenação desempenham papel essencial na ciência da computação, por proporcionarem formas eficientes de reorganizar dados para otimizar operações subsequentes, como buscas e análises. Entre os métodos simples, destacam-se Bubble Sort, Selection Sort e Insertion Sort. Para conjuntos maiores, algoritmos avançados como Merge Sort e Quick Sort são preferidos devido à sua maior eficiência.

Por fim, descrevemos como, em aplicações práticas específicas, a escolha correta desses algoritmos pode ser fundamental para o desempenho e a eficiência operacional. Em ambientes com limitações severas de recursos, como sistemas embarcados ou dispositivos IoT, os algoritmos simples de ordenação continuam sendo amplamente utilizados devido ao baixo consumo de memória e facilidade de implementação. Já em contextos que manipulam grandes volumes de dados, como aplicações em Big Data e Data Science, métodos avançados tornam-se indispensáveis, destacando-se o Merge Sort pela capacidade de lidar bem com armazenamento externo e o Quick Sort pela eficiência no uso de memória cache.

Exercícios

Questão 1. (Iades 2019, adaptada) As árvores são estruturas de dados multidimensionais que permitem a representação de hierarquias. Uma árvore é formada por um conjunto de elementos que armazenam informações, chamados de nós. Toda árvore apresenta um elemento denominado raiz, que tem ligações para outros elementos chamados filhos ou ramos. Esses ramos, por sua vez, podem estar conectados a outros elementos que também podem ter ramos. O elemento que não dispõe de ramos é conhecido como nó folha. O elemento que dá origem a outro pode ser chamado de nó pai.

Uma árvore binária é um caso especial de árvore, em que cada nó pai tem, no máximo, dois filhos. Em uma árvore binária, quando um nó tem ramificações, cada uma delas recebe um nome. A ramificação esquerda receberá o nome de subárvore esquerda (SAE) e a ramificação direita será a subárvore direita (SAD).

Nesse contexto, considere a árvore binária apresentada a seguir.

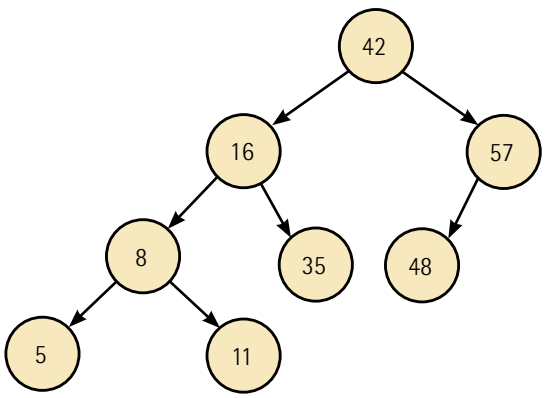

Figura 16

Assinale a alternativa que apresenta a sequência resultante ao percorrermos essa árvore utilizando o algoritmo de pre-order.

A) 11, 5, 48, 35, 8, 57, 16, 42.

B) 11, 5, 8, 35, 16, 42, 48, 57.

C) 5, 11, 8, 35, 16, 48, 57, 42.

D) 42, 16, 8, 5, 11, 35, 57, 48.

E) 5, 8, 11, 16, 35, 42, 48, 57.

Resposta correta: alternativa D.

Análise da questão

Ao percorrer uma árvore binária para realizar operações, precisamos adotar um critério que ditará como será a lógica do percurso. Uma das possibilidades é adotarmos um percurso em pré-ordem (destacado no enunciado como pre-order).

Adotaremos que a operação executada em cada nó, pelo algoritmo, é uma operação de leitura de dados. Para adotarmos o percurso em pré-ordem, fazemos, recursivamente e partindo da raiz, a sequência de passos listados a seguir.

- Lemos o nó.
- Vamos à SAE.
- Vamos à SAD.
- Retornamos.

A sequência de nós lidos é numerada na figura a seguir.

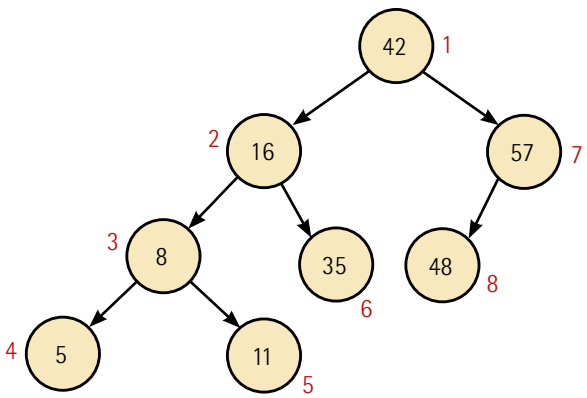

Figura 17

Partindo da raiz, lemos primeiro o próprio nó 42 (que representa a leitura 1). Como ele tem um filho à esquerda, vamos à SAE, para lermos o nó 16 (2). Como esse nó 16 também tem um filho à esquerda, vamos novamente à SAE, até o nó 8 (3). Seguindo o mesmo raciocínio, vamos novamente à SAE, até o nó 5 (4). Como o nó 5 é um nó folha, retornamos até o nó 8 (sem nova leitura, já que a operação de leitura nesse nó já havia sido executada) e vamos à SAD, para lermos o nó 11 (5). Retornamos para o nó 16 e vamos à SAD, até o nó 35 (6). Retornamos à raiz e vamos à SAD, até o nó 57 (7). O último passo é ir à SAE, até o nó 48 (8). Temos, portanto, a sequência de: 42, 16, 8, 5, 11, 35, 57, 48.

Questão 2. (UFG 2025, adaptada) Considere o seguinte código Python, que implementa um método de ordenação.

```
1.  def ordenar(lista):
2.       n = len(lista)
3.       for i in range(n):
4.          trocado = False
5.          for j in range(0, n - i - 1):
6.             if lista[j] > lista[j + 1]:
7.                lista[j], lista[j + 1] = lista[j + 1], lista[j]
8.                trocado = True
9.       if not trocado:
10.         break
11.    return lista
12.
13. #Exemplo de uso
14. numeros = [64, 34, 25, 12, 22, 11, 90]
15. print(ordenar(numeros))
```

Qual método de ordenação foi implementado?

A) Quick Sort.

B) Merge Sort.

C) Bubble Sort.

D) Insertion Sort.

E) Selection Sort.

Resposta correta: alternativa C.

Análise da questão

O algoritmo de ordenação Bubble Sort percorre uma coleção de dados múltiplas vezes, comparando e trocando elementos adjacentes, caso estejam desordenados.

O algoritmo da questão traz dois laços for aninhados (`for i e for j`), que fazem a comparação de elementos adjacentes (`if lista[j] > lista[j + 1]`), que trocam de posição se estiverem desordenados (`lista[j], lista[j + 1] = lista[j + 1], lista[j]`).

A variável trocado é utilizada para a otimização do código, pois, se nenhuma troca ocorrer em uma passada, a lista está ordenada e o laço é interrompido.