





UNIP
UNIVERSIDADE PAULISTA

Roteiros

Algoritmos e Estrutura de Dados em Python

  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Algoritmos e Estrutura de Dados em Python</p> <p>Título da aula: Implementando Algoritmos em Python</p>	<p>ROTEIRO 1</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 01: IMPLEMENTANDO ALGORITMOS EM PYTHON

1. Objetivos da aula

- Reforçar a ligação entre lógica de programação e construção de algoritmos executáveis em Python.
- Apresentar exemplos que combinem variáveis, condicionais (if/elif/else) e laços (for e while) com entrada e saída básicas.
- Exercitar a análise de pequenos problemas, convertendo-os em código comentado e organizado.
- Orientar a produção de um relatório enxuto contendo fundamentação teórica, código-fonte e reflexões sobre o processo.

2. Recursos necessários

- Computadores ou dispositivos com acesso à internet (laboratório ou BYOD – *bring your own device*).
- Acesso a um interpretador Python (instalado localmente ou online, como [OnlineGDB](#), [Online Python](#) ou [OneCompiler](#)).
- Material de apoio (capítulo 1 do livro-texto).
- Editor de texto ou IDE (opcional) para organização do relatório final.

3. Estrutura da aula

1. Abertura (10 minutos)

- Introdução breve dos conceitos de algoritmo, variável, decisão e repetição.
- Conexão direta com Python, destacando indentação e legibilidade.

2. Revisão conceitual (20 minutos)

- **O que é um algoritmo:** Conceito, importância e exemplos do dia a dia (como o exemplo do leite derramado ou o da criança calculando a média de números).
- O pseudocódigo e o fluxograma como rascunhos formais.
- A sintaxe essencial de Python para decisões e laços.
- Os paralelos entre pseudocódigo e código Python (mini-trechos lado a lado).

3. Demonstração prática (30 minutos)

- **Acesso ao interpretador:** Orientar os alunos a abrir um dos interpretadores online ou uma IDE instalada localmente.
- A criação de um script "Contador Inteligente":
- A variável limite recebida via `input()`.
- O laço `for` que conta de 1 até limite.
- A condicional que, para cada número, imprime "par" ou "ímpar".
- O uso de comentários `#` para explicar cada trecho relevante.

4. Atividade prática (40 minutos)

▪ **Desafio:** Cada estudante deve desenvolver um programa que:

1. Pergunte o nome da pessoa.
2. Pergunte quantos valores numéricos serão analisados.
3. Leia os valores dentro de um laço for ou while, acumulando soma.
4. Calcule a média.
5. Utilize uma condicional para classificar a média:
 - $\geq 70 \rightarrow$ "Excelente!"
 - 50-69 \rightarrow "Bom."
 - $< 50 \rightarrow$ "Precisa melhorar."
6. Exiba mensagem final personalizada com f-string e, preferencialmente, usando três linhas (string multilinha ou múltiplos print()).
7. Exemplo mínimo esperado:

```
nome = input("Qual é o seu nome? ")
n = int(input("Quantos números você vai digitar? "))

soma = 0
for i in range(n):
    valor = float(input(f"Digite o {i+1}º valor: "))
    soma += valor

media = soma / n

if media >= 70:
    status = "Excelente!"
elif media >= 50:
    status = "Bom."
else:
    status = "Precisa melhorar."

print(f"\nOlá, {nome}!\nA média obtida foi {media:.2f}.\n{status}")
```

8. Ampliação:

- A inclusão de verificação para evitar divisão por zero.
- O uso de listas para armazenar todos os valores e apresentar o maior e o menor.
- A implementação de um laço de repetição externa que permita rodar o programa várias vezes até que o usuário escolha sair.

5. Encerramento e orientações finais (20 minutos)

- **Tempo para dúvidas:** Esclarecer eventuais problemas encontrados na implementação.
- **Proposta de continuação:** Experimentar outras estruturas de repetição e inserir funções próprias.
- **Entrega do relatório:** Orientar como deve ser o formato e o conteúdo do relatório que será entregue.

4. Relatório final

Cada aluno (ou equipe) deve produzir um relatório curto (1 a 2 páginas) contendo:

Resumo teórico:

- Explicar, com palavras próprias, o que é lógica de programação e por que ela é importante.
- Mencionar brevemente o que é pseudocódigo e fluxograma e como ajudam na organização de ideias.
- Citar as vantagens de usar Python para aprender programação.
- Definição de algoritmo, variável, condicional, laço.

Código-fonte comentado:

- Inserir o *código-fonte completo* da atividade proposta.
- Comentar as principais linhas, ressaltando o uso de `print()`, `input()`, variáveis etc.

5. Critérios de avaliação

Critério	Peso	Descrição
Qualidade do resumo teórico	2,0	Clareza e correlação correta entre conceitos.
Estrutura e organização do código e funcionamento da solução	3,0	O código deve estar indentado corretamente, usar nomes de variáveis adequados e conter comentários informativos (quando necessários).
	3,0	Execução sem erros, entrada e saída corretas, uso de condições e laços.
Criatividade e aprimoramentos	2,0	Adição de perguntas extras, uso de strings multilinha, personalização das mensagens e outras melhorias que demonstrem domínio do conteúdo.


Nota final: Será a soma dos valores obtidos em cada critério. Alunos ou equipes que não cumprirem os requisitos mínimos de funcionamento do código ou não entregarem o relatório dentro do prazo terão sua nota diminuída proporcionalmente.

6. Conclusão

Este roteiro busca oferecer uma base sólida para que os alunos compreendam a lógica de programação e apliquem-na em Python de forma interativa.



Ao final da aula, espera-se que cada estudante seja capaz de:

- a. Demonstrar domínio elementar de variáveis, decisões e repetições em Python.

- 
- b. Explicar, em linguagem acessível, como conceber um algoritmo e transpô-lo para código.
 - c. Produzir documentação sucinta, mas completa, das soluções implementadas.

Bom estudo e boa prática de programação!



  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Algoritmos e Estrutura de Dados em Python</p> <p>Título da aula: Estruturas de Dados Lineares em Python: Listas, Pilhas, Filas E Eficiência. Notação BIG-O</p>	<p>ROTEIRO 2</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 02: ESTRUTURAS DE DADOS LINEARES EM PYTHON: LISTAS, PILHAS, FILAS E EFICIÊNCIA. NOTAÇÃO BIG-O

1. Objetivos da aula

- Apresentar as principais estruturas de dados lineares (listas, pilhas e filas), destacando suas características e usos.
- Demonstrar como implementar essas estruturas em Python com listas nativas e `collections.deque`
- Introduzir a notação Big-O para análise elementar de eficiência algorítmica.
- Propor exercícios que estimulem a manipulação dessas estruturas em cenários simulados.
- Forçar a ligação entre lógica de programação e construção de algoritmos executáveis em Python.

2. Recursos necessários

- Computadores ou dispositivos com acesso à internet (laboratório ou BYOD – *bring your own device*).
- Acesso a um interpretador Python (instalado localmente ou online, como [OnlineGDB](#), [Online Python](#) ou [OneCompiler](#)).
- Módulo `collections` incluído no Python padrão.

- Material de apoio (capítulos 1 e 2 do livro-texto).
- Editor de texto ou IDE (opcional) para organização do relatório final.

3. Estrutura da aula

1. Abertura (10 minutos)

- Explicação breve sobre a finalidade das estruturas de dados.
- Apresentação dos objetivos da sessão e de aplicações concretas (histórico de navegador, fila de impressão etc.).

2. Revisão conceitual (20 minutos)

- Estrutura de lista: indexação, inserção, remoção.
- Pilha: modelo LIFO, operações push, pop, peek.
- Fila: modelo FIFO, operações enqueue, dequeue, visualização da frente.
- Comparação conceitual entre as três estruturas.
- Introdução à notação Big-O e análise das operações principais.

Estrutura	Operação	Complexidade
Lista (<code>append</code>)	inserção no fim	$O(1)$
Lista (<code>insert</code>)	inserção em posição arbitrária	$O(n)$
Pilha (<code>append/pop</code>)	inserção e remoção	$O(1)$
Fila (<code>append/pop(0)</code>)	inserção e remoção	$O(n)$
Fila com <code>deque</code>	inserção e remoção	$O(1)$

3. Demonstração prática (30 minutos)

- Criação de uma lista simples, ordenação e remoção de elementos.
- Simulação de uma pilha com list:

```
pilha = []  
pilha.append("prato 1")  
pilha.append("prato 2")  
print(pilha.pop()) # prato 2
```

- Simulação de uma fila com deque:

```
from collections import deque  
  
fila = deque()  
fila.append("cliente 1")  
fila.append("cliente 2")  
print(fila.popleft()) # cliente 1
```

- Discussão sobre o impacto da escolha da estrutura no desempenho.
- Comentários no código para explicação de cada operação.

4. Atividade prática (40 minutos)

- **Desafio:** Cada estudante deve desenvolver um programa que:
 1. Use uma fila para simular atendimento de clientes.
 2. Use uma pilha para controlar uma sequência de tarefas.
 3. Apresente estatísticas ao final (número de atendimentos concluídos e de tarefas pendentes).
 4. Exemplo mínimo esperado:

```

from collections import deque

fila = deque()
pilha = []

# Atendimento
for i in range(3):
    nome = input(f"Nome do cliente {i+1}: ")
    fila.append(nome)

print("\nIniciando atendimentos:")
while fila:
    cliente = fila.popleft()
    print(f"Atendendo {cliente}")

# Tarefas
for i in range(3):
    tarefa = input(f"Tarefa {i+1}: ")
    pilha.append(tarefa)

print("\nExecutando tarefas:")
while pilha:
    print(f"Executando: {pilha.pop()}")

```

5. Ampliação:

- Coleta de tempo de execução com o módulo time.
- Implementação de fila circular ou fila de prioridade.
- Uso de listas aninhadas ou dicionários para armazenar dados adicionais.

5. Encerramento e orientações finais (20 minutos)

- Resolução de dúvidas frequentes.
- Debate sobre a escolha da estrutura mais adequada a cada problema.
- Lembrete sobre o envio do relatório.
- Sugestão de leitura adicional sobre filas de prioridade e pilhas com limites.

4. Relatório final

Cada aluno (ou equipe) deve produzir um relatório curto (2 a 3 páginas) contendo:

Resumo teórico:

- Definição de listas, pilhas, filas e introdução à notação Big-O.
- Código-fonte comentado das soluções desenvolvidas.
- Reflexão sobre desafios encontrados e critérios para selecionar cada estrutura.

Código-fonte comentado:

- Inserir o *código-fonte completo* da atividade proposta.
- Comentar as principais linhas, ressaltando o uso das estruturas de dados.

5. Critérios de avaliação

Critério	Peso	Descrição
Qualidade do resumo teórico	2,0	Clareza e correlação correta entre conceitos.
Estrutura e organização do código e funcionamento da solução	3,0	O código deve estar indentado corretamente, usar nomes de variáveis adequados e conter comentários informativos (quando necessários).
	3,0	Execução sem erros, entrada e saída corretas.
Criatividade e aprimoramentos	2,0	Funcionalidades extras, simulações realistas.

Nota final: Será a soma dos valores obtidos em cada critério. Alunos ou equipes que não cumprirem os requisitos mínimos de funcionamento do código ou não entregarem o relatório dentro do prazo terão sua nota diminuída proporcionalmente.



6. Conclusão

Este roteiro busca oferecer uma base sólida para que os alunos compreendam as estruturas de dados lineares e apliquem-nas em Python de forma interativa.

Ao final da aula, espera-se que cada estudante seja capaz de:

- entendimento prático do funcionamento de listas, pilhas e filas em Python.
- domínio das operações fundamentais dessas estruturas com suas respectivas complexidades Big-O.
- justificar a escolha da estrutura mais eficiente em problemas comuns de programação.

Bom estudo e boa prática de programação!

  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Algoritmos e Estrutura de Dados em Python</p> <p>Título da aula: Estruturas de Dados Não Lineares - Árvores e Grafos Em Python</p>	<p>ROTEIRO 3</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 03: ESTRUTURAS DE DADOS NÃO LINEARES – ÁRVORES E GRAFOS EM PYTHON

1. Objetivos da aula

- Apresentar os fundamentos de estruturas de dados não lineares, com ênfase em árvores e grafos.
- Relacionar esses conceitos à implementação em Python por meio do módulo collections e de classes próprias.
- Praticar a criação de nós, arestas e percursos (BFS, DFS) em códigos curtos.
- Elaborar um exercício prático que resulte em um relatório com resumo teórico e código-fonte comentado.

2. Recursos necessários

- Computadores ou dispositivos com acesso à internet (laboratório ou BYOD – *bring your own device*).
- Acesso a um interpretador Python 3 (instalado localmente ou online, como [OnlineGDB](#), [Online Python](#) ou [OneCompiler](#)).
- Material de apoio (capítulo 3 do livro-texto).
- Editor de texto ou IDE (PyCharm, VS Code, Thonny) para organização do relatório final.

3. Estrutura da aula

1. Abertura (10 minutos)

- Contextualização: recordar rapidamente listas e filas (estruturas lineares vistas anteriormente) e motivar a necessidade de estruturas hierárquicas (árvores) e relacionais (grafos).
- Conexão com Python: destacar como classes e listas aninhadas viabilizam a representação de nós e arestas de forma acessível.

2. Revisão conceitual (20 minutos)

- A árvore: conjunto de nós conectados sem ciclos, com raiz única.
- O grafo: conjunto de vértices ligados por arestas, podendo ser dirigido ou não.
- Terminologia essencial: grau, altura, caminho, ciclo, conectividade.
- Representações clássicas em teoria de grafos: a lista de adjacências e a matriz de adjacências.
- Exemplos do cotidiano (roteamento de redes, organogramas) para fixar utilidade prática.

3. Demonstração prática (30 minutos)

- Configuração do ambiente: abrir o interpretador ou IDE.
- Construção de uma classe Node simplificada:

```
class Node:
    def __init__(self, valor):
        self.valor = valor
        self.filhos = []
```

- Percurso em profundidade (DFS) recursivo sobre árvore: exibição de ordem de visita.
- Introdução ao módulo collections.deque para BFS em grafo simples dirigido.
- Exemplo completo:
 - 1. O Código 1 – criação de uma árvore de personagens de "Star Wars".

```
class Node:
    """Nó simples para árvores n-árias."""
    def __init__(self, valor):
        self.valor = valor
        self.filhos = []

    def add_child(self, *novos_filhos):
        """Acrescenta um ou mais filhos ao nó atual."""
        self.filhos.extend(novos_filhos)

def imprimir_arvore(no, nivel=0):
    """Percorre a árvore em pré-ordem apenas para exibição."""
    indent = " " * (4 * nivel)
    print(f"{indent}- {no.valor}")
    for filho in no.filhos:
        imprimir_arvore(filho, nivel + 1)

# Construção da árvore
anakin = Node("Anakin Skywalker")      # raiz
luke = Node("Luke Skywalker")
leia = Node("Leia Organa")
ben = Node("Ben Solo")

leia.add_child(ben)                    # neto
anakin.add_child(luke, leia)           # filhos

# Visualização
if __name__ == "__main__":
    imprimir_arvore(anakin)
```


2. O Código 2 – BFS em um grafo de cidades conectadas por estradas.

```
from collections import deque

def bfs(grafo, origem, destino):
    """Retorna o menor caminho em número de arestas entre origem e destino."""
    fila = deque([[origem]])          # cada elemento é um caminho
    visitados = {origem}

    while fila:
        caminho = fila.popleft()
        atual = caminho[-1]

        if atual == destino:
            return caminho

        for vizinho in grafo.get(atual, []):
            if vizinho not in visitados:
                visitados.add(vizinho)
                fila.append(caminho + [vizinho])

    return None                       # não há ligação

# Grafo não dirigido com cinco cidades
grafo_cidades = {
    "São Paulo": ["Rio de Janeiro", "Curitiba"],
    "Rio de Janeiro": ["São Paulo", "Belo Horizonte"],
    "Curitiba": ["São Paulo", "Florianópolis"],
    "Belo Horizonte": ["Rio de Janeiro", "Brasília"],
    "Florianópolis": ["Curitiba"]
}

if __name__ == "__main__":
    origem = input("Cidade de origem: ").strip()
    destino = input("Cidade de destino: ").strip()

    caminho = bfs(grafo_cidades, origem, destino)

    if caminho:
        print(f"O percurso é: {caminho}")
        print(f"O número de etapas é {len(caminho) - 1}.")
    else:
        print("Não existe caminho entre as duas cidades nesse grafo.")
```

4. Atividade prática (40 minutos)

- **Desafio:** Cada estudante deve desenvolver um programa que:
 1. Crie um grafo dirigido que represente cinco cidades e distâncias em quilômetros.
 2. Implemente BFS para descobrir o caminho mais curto em número de arestas entre duas cidades informadas pelo usuário.
 3. Exiba o percurso encontrado e o número de etapas envolvidas.
 4. Exemplo mínimo esperado:

```
origem = input("Informe a cidade de origem: ")
destino = input("Informe a cidade de destino: ")
caminho = bfs(grafo, origem, destino)
print(f"O percurso de {origem} até {destino} é: {caminho}")
print(f"O número de etapas é {len(caminho) - 1}.")
```

5. Ampliação:
 - Incentivar a atribuição de pesos às arestas.

5. Encerramento e orientações finais (20 minutos)

- Resolução de dúvidas frequentes.
- Lembrete sobre o envio do relatório.

4. Relatório final

Cada aluno (ou equipe) deve produzir um relatório curto (2 a 3 páginas) contendo:

Resumo teórico:

- Definição de árvore e grafo, vantagens de cada estrutura.
- Comentário sobre a escolha de Python para ilustrar algoritmos de percursos.

Código-fonte comentado:

- Inserir o *código-fonte completo* da atividade proposta.
- Comentar as principais linhas, ressaltando criação de vértices, filas, laços e condições de parada.

5. Critérios de avaliação

Critério	Peso	Descrição
Qualidade do resumo teórico	2,0	Clareza e correlação correta entre conceitos.
Estrutura e organização do código e funcionamento da solução	3,0	O código deve estar indentado corretamente, usar nomes de variáveis adequados e conter comentários informativos (quando necessários).
	3,0	Execução sem erros, entrada e saída corretas.
Criatividade e aprimoramentos	2,0	Inclusão de pesos, tratamento de exceções ou visualização gráfica simples.

Nota final: Será a soma dos valores obtidos em cada critério. Alunos ou equipes que não cumprirem os requisitos mínimos de funcionamento do código ou não entregarem o relatório dentro do prazo terão sua nota diminuída proporcionalmente.



6. Conclusão

Este roteiro busca oferecer uma base sólida para que os alunos compreendam as estruturas de dados não lineares e apliquem-nas em Python de forma interativa.

Ao final da aula, espera-se que cada estudante seja capaz de:

- a. Explicar os conceitos fundamentais de árvores e grafos, distinguindo suas representações.
- b. Construir percursos em profundidade e em largura em Python, aplicando-os a problemas simples.
- c. Documentar suas soluções em relatório conciso, demonstrando a conexão entre teoria e prática.

Bom estudo e boa prática de programação!

  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Algoritmos e Estrutura de Dados em Python</p> <p>Título da aula: Algoritmos de Ordenação em Python (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort e Quick Sort)</p>	<p>ROTEIRO 4</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 04: ALGORITMOS DE ORDENAÇÃO EM PYTHON (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort e Quick Sort)

1. Objetivos da aula

- Apresentar os princípios que motivam a necessidade de algoritmos de ordenação.
- Discutir a análise de complexidade temporal e espacial de cada método.
- Implementar, em Python, cinco algoritmos clássicos de ordenação.
- Conduzir um experimento comparativo simples, culminando em relatório com síntese teórica e código fonte comentado.

2. Recursos necessários

- Computadores ou dispositivos com acesso à internet (laboratório ou BYOD – *bring your own device*).
- Acesso a um interpretador Python 3 (instalado localmente ou online, como [OnlineGDB](#), [Online Python](#) ou [OneCompiler](#)).
- Material de apoio (capítulo 4 do livro-texto).
- Editor de texto ou IDE (PyCharm, VS Code, Thonny) para organização do relatório final.

3. Estrutura da aula

1. Abertura (10 minutos)

- Motivação: recordar a importância de ordenar dados para busca binária, estatísticas e visualização.
- Conexão com Python: realçar que a função nativa `sorted()` usa Timsort, mas estudar os clássicos reforça pensamento algorítmico.

2. Revisão conceitual (20 minutos)

- As definições e a análise assintótica: melhor caso, pior caso, caso médio.
- Os algoritmos estudados:
 1. O Bubble Sort – varre a lista repetidamente, trocando pares adjacentes fora de ordem; complexidade $O(n^2)$.
 2. O Selection Sort – seleciona repetidamente o menor elemento restante e o coloca na posição correta; $O(n^2)$.
 3. O Insertion Sort – insere cada elemento na posição adequada de uma lista parcialmente ordenada; $O(n^2)$, mas eficiente para listas quase ordenadas.
 4. O Merge Sort – divide e conquista, estabilidade garantida; $O(n \log n)$.
 5. O Quick Sort – escolhe pivô, partitiona e recorre; $O(n \log n)$ em média, $O(n^2)$ no pior caso.
 - Observação sobre estabilidade e uso prático.

3. Demonstração prática (30 minutos)

- Preparação: abrir o interpretador ou IDE.

- Implementação guiada de cada algoritmo em funções independentes.
- Uso do módulo `timeit` para medir a duração em listas de 1 000, 5 000 e 10 000 elementos aleatórios.
- Discussão dos resultados obtidos em tela.
- Código exemplo para Bubble Sort:

```
def bubble_sort(lista):  
    n = len(lista)  
    for i in range(n - 1):  
        for j in range(n - 1 - i):  
            if lista[j] > lista[j + 1]:  
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
```

4. Atividade prática (40 minutos)

- **Desafio:** Cada estudante deve desenvolver um programa que:
 1. Implemente ao menos três dos cinco algoritmos.
 2. Leia do usuário o tamanho da lista e gere números aleatórios com `random.randint`.
 3. Meça e exiba os tempos de execução, apresentando os resultados ordenados por desempenho.
 4. Exemplo mínimo esperado:

```

import random, timeit
tam = int(input("Tamanho da lista: "))
dados = [random.randint(0, 10_000) for _ in range(tam)]
tempos = {}
for nome, func in [("Bubble", bubble_sort),
                  ("Insertion", insertion_sort),
                  ("Merge", merge_sort)]:
    copia = dados.copy()
    duracao = timeit.timeit(lambda: func(copia), number=1)
    tempos[nome] = duracao
print("Tempos:", tempos)

```

5. Ampliação:

- Introduzir Quick Sort com escolha de pivô aleatório e comparar com Timsort (sorted());

5. Encerramento e orientações finais (20 minutos)

- Resolução de dúvidas de complexidade e código.
- Sugestão de estudo em casa: estabilidade *versus* instabilidade, otimizações de Quick Sort (mediana de três).
- Lembrete sobre o envio do relatório.

4. Relatório final

Cada aluno (ou equipe) deve produzir um relatório curto (2 a 3 páginas) contendo:

Resumo teórico:

- Definir ordenação e justificar sua relevância em ciência da computação.
- Explicar diferenças conceituais entre algoritmos quadráticos e log-lineares.
- Comentar vantagens e limitações de cada método.

Código-fonte comentado:

- Inserir as implementações completas, destacando linhas decisivas (trocas, partições, fusões).
- Incluir tabela dos tempos obtidos nas medições.

5. Critérios de avaliação

Critério	Peso	Descrição
Qualidade do resumo teórico	2,0	Clareza e correlação correta entre conceitos.
Estrutura e organização do código e funcionamento da solução	3,0	O código deve estar indentado corretamente, usar nomes de variáveis adequados e conter comentários informativos (quando necessários).
	3,0	Execução correta e apresentação dos resultados de tempo.
Criatividade e aprimoramentos	2,0	Introdução de visualização gráfica simples, pivô aleatório ou otimizações.

Nota final: Será a soma dos valores obtidos em cada critério. Alunos ou equipes que não cumprirem os requisitos mínimos de funcionamento do código ou não entregarem o relatório dentro do prazo terão sua nota diminuída proporcionalmente.



6. Conclusão

Este roteiro busca oferecer uma base sólida para que os alunos compreendam os algoritmos de ordenação e apliquem-nos em Python de forma interativa.

Ao final da aula, espera-se que cada estudante seja capaz de:

- a. Discriminar quando escolher métodos quadráticos ou log-lineares.
- b. Implementar e comparar algoritmos de ordenação em Python, avaliando custo computacional.
- c. Produzir relatório conciso que demonstre integração de teoria, implementação e análise empírica.

Bom estudo e boa prática de programação!

  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Algoritmos e Estrutura de Dados em Python</p> <p>Título da aula: Algoritmos de Pesquisa – Busca Linear e Busca Binária em Python</p>	<p>ROTEIRO 5</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 05: ALGORITMOS DE PESQUISA – BUSCA LINEAR E BUSCA BINÁRIA EM PYTHON

1. Objetivos da aula

- Apresentar os princípios básicos de pesquisa em estruturas de dados ordenadas e não ordenadas.
- Explicar a diferença conceitual entre algoritmos de força bruta e algoritmos de divisão e conquista aplicados à busca.
- Implementar, em Python, a busca linear e a busca binária, avaliando a eficiência.
- Conduzir um experimento prático de medição de tempo, produzindo relatório com síntese teórica e código comentado.

2. Recursos necessários

- Computadores ou dispositivos com acesso à internet (laboratório ou BYOD – *bring your own device*).
- Acesso a um interpretador Python 3 (instalado localmente ou online, como [OnlineGDB](#), [Online Python](#) ou [OneCompiler](#)).
- O módulo `timeit` para cronometrar execuções.
- Material de apoio (capítulo 5 do livro-texto).

- Editor de texto ou IDE (PyCharm, VS Code, Thonny) para organização do relatório final.

3. Estrutura da aula

1. Abertura (10 minutos)

- Motivação: lembrar que muitas aplicações exigem localizar rapidamente elementos em coleções de dados.
- A conexão com Python: destacar que, embora in faça busca linear sobre listas, compreender a busca binária é essencial para aproveitar bibliotecas como bisect.

2. Revisão conceitual (20 minutos)

- As definições de complexidade temporal ($O(n)$ vs. $O(\log n)$) e melhor/pior caso.
- As técnicas abordadas:
 1. A busca linear – inspeção sequencial de cada posição; adequada para coleções pequenas ou mutáveis sem ordenação.
 2. A busca binária – divisão repetida do intervalo de pesquisa; exige lista previamente ordenada.
 - A ilustração gráfica: mostrar como a busca binária reduz o espaço de busca pela metade a cada iteração.
 - A discussão de limites inferiores do problema de busca em vetor ordenado ($\Omega(\log n)$) segundo a teoria da informação.

3. Demonstração prática (30 minutos)

- Configuração: abrir IDE, criar lista aleatória de 10 000 inteiros e uma cópia ordenada com sorted()).
- Implementação de funções:

```
def linear_search(seq, alvo):  
    for i, v in enumerate(seq):  
        if v == alvo:  
            return i  
    return -1  
  
def binary_search(seq, alvo):  
    ini, fim = 0, len(seq) - 1  
    while ini <= fim:  
        meio = (ini + fim) // 2  
        if seq[meio] == alvo:  
            return meio  
        if seq[meio] < alvo:  
            ini = meio + 1  
        else:  
            fim = meio - 1  
    return -1
```

- Medição com timeit para 1 000 execuções em listas de tamanhos variados.
- Observação da relação logarítmica nos tempos da busca binária.

4. Atividade prática (40 minutos)

- **Desafio:** Cada estudante deve desenvolver um programa que:
 1. Gere lista de tamanho escolhido pelo usuário (mínimo 5 000).
 2. Solicite ao usuário um valor-alvo.

3. Execute a busca linear na lista original e a busca binária na lista ordenada, cronometrando ambas.
4. Exiba índices retornados e tempos, indicando qual método foi mais eficiente.
5. Exemplo mínimo esperado:

```
import random, timeit
n = int(input("Tamanho da lista: "))
lista = [random.randint(0, 100_000) for _ in range(n)]
alvo = int(input("Valor a procurar: "))

t_lin = timeit.timeit(lambda: linear_search(lista, alvo), number=1)
ordenada = sorted(lista)
t_bin = timeit.timeit(lambda: binary_search(ordenada, alvo), number=1)

print(f"Tempo busca linear: {t_lin:.6f}s")
print(f"Tempo busca binária: {t_bin:.6f}s")
```

6. Ampliação:

- Comparar com bisect e com index() de listas, elaborar gráfico de barras em casa.
- Implementar busca binária recursiva e analisar sobrecarga de chamadas.

5. Encerramento e orientações finais (20 minutos)

- Resolução de dúvidas sobre limites de precisão de medidas pequenas.
- Lembrete sobre o envio do relatório.

4. Relatório final

Cada aluno (ou equipe) deve produzir um relatório curto (2 a 3 páginas) contendo:

Resumo teórico:

- Explicar diferença entre pesquisa exaustiva e pesquisa por divisão.
- Comparar custos de busca linear e binária em termos de complexidade e de requisitos de ordenação.

Código-fonte comentado:

- Inserir implementações completas de ambos os métodos.
- Apresentar tabela com tempos coletados para três tamanhos distintos de listas.

5. Critérios de avaliação

Critério	Peso	Descrição
Qualidade do resumo teórico	2,0	Clareza conceitual e uso correto de terminologia.
Estrutura e organização do código e funcionamento da solução	3,0	O código deve estar indentado corretamente, usar nomes de variáveis adequados e conter comentários informativos (quando necessários).
	3,0	Execução correta e apresentação dos resultados de tempo.
Criatividade e aprimoramentos	2,0	Integração de gráficos, versionamento recursivo ou utilização de bisect.

Nota final: Será a soma dos valores obtidos em cada critério. Alunos ou equipes que não cumprirem os requisitos mínimos de funcionamento do código ou não entregarem o relatório dentro do prazo terão sua nota diminuída proporcionalmente.



6. Conclusão

Este roteiro busca oferecer uma base sólida para que os alunos compreendam os algoritmos de busca linear e binária e apliquem-nos em Python de forma interativa.

Ao final da aula, espera-se que cada estudante seja capaz de:

- a. Discriminar cenários adequados para busca linear e para busca binária.
- b. Implementar ambos os algoritmos em Python, avaliando desempenho experimentalmente.
- c. Documentar a solução em relatório sintético, conectando princípios teóricos a resultados práticos.

Bom estudo e boa prática de programação!

  Instituto de Ciências Exatas e Tecnologia	<p>Disciplina: Algoritmos e Estrutura de Dados em Python</p> <p>Título da aula: Tabelas de Dispersão (Hash Tables) e os Heaps em Python</p>	<p>ROTEIRO 6</p>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 06: TABELAS DE DISPERSÃO (HASH TABLES) E OS HEAPS EM PYTHON

1. Objetivos da aula

- Apresentar os conceitos fundamentais de tabelas de dispersão (hash tables) e heaps, destacando propriedades estruturais e operações principais.
- Relacionar esses conceitos às implementações nativas de Python (dict/set e módulo heapq).
- Implementar inserção, remoção e consulta em ambas as estruturas, analisando custos assintóticos.
- Realizar um experimento prático, gerando relatório com síntese teórica e código-fonte comentado.

2. Recursos necessários

- Computadores ou dispositivos com acesso à internet (laboratório ou BYOD – *bring your own device*).
- Acesso a um interpretador Python 3 (instalado localmente ou online, como [OnlineGDB](#), [Online Python](#) ou [OneCompiler](#)).
- Material de apoio (capítulo 6 do livro-texto).
- Editor de texto ou IDE (PyCharm, VS Code, Thonny) para organização do relatório final.

3. Estrutura da aula

1. Abertura (10 minutos)

- Motivação: recordar listas e filas, apontando limites de busca neles e a necessidade de estruturas com operações próximas de $O(1)$ ou $O(\log n)$.
- A conexão com Python: salientar que dict e set utilizam hashing, enquanto heapq fornece uma fila de prioridade eficiente.

2. Revisão conceitual (20 minutos)

- As definições principais:
 1. A tabela de dispersão – estrutura que associa chaves a valores via função de hash; tempo médio de busca, inserção e remoção em $O(1)$.
 2. O heap binário – árvore completa que satisfaz propriedade de heap; operações centrais (heappush, heappop) em $O(\log n)$.
- Os detalhes analíticos:
 1. A função de hash: uniformidade, colisões e tratamento (encadeamento separado, endereçamento aberto).
 2. A propriedade de heap: pai \leq filhos (mín-heap) ou pai \geq filhos (máx-heap).
- Os exemplos práticos: armazenamento de índices invertidos (hash table) e escalonamento de tarefas (heap).

3. Demonstração prática (30 minutos)

- A preparação: abrir IDE e importar heapq e timeit.
- A construção de min-heap manual:

```
import heapq

tarefas = []
heapq.heappush(tarefas, (3, "Enviar relatório"))
heapq.heappush(tarefas, (1, "Responder e-mails"))
heapq.heappush(tarefas, (2, "Revisar código"))

while tarefas:
    prioridade, nome = heapq.heappop(tarefas)
    print(prioridade, nome)
```

- A simulação de colisões em dicionários pequenos para demonstrar tratamento interno (ilustrativo).
- A medição de tempo: comparar busca em lista *versus* dicionário em 100 000 elementos com `timeit`.

4. Atividade prática (40 minutos)

- **Desafio:** Cada estudante deve desenvolver um programa que:
 1. Gere 50 000 strings aleatórias como chaves e valores inteiros.
 2. Insira pares em dicionário (`dict`) e em lista de tuplas.
 3. Meça o tempo de busca de 1 000 chaves escolhidas aleatoriamente em ambas as estruturas.
 4. Construa um min-heap com 20 000 números aleatórios e execute 5 000 extrações, cronometrando operação.
 5. Exemplo mínimo esperado para a comparação de busca:

```

import random, string, timeit

def gera_str(tam=8):
    return "".join(random.choices(string.ascii_letters, k=tam))

n = 50_000
chaves = [gera_str() for _ in range(n)]
valores = list(range(n))
lista = list(zip(chaves, valores))
tabela = dict(zip(chaves, valores))

consulta = random.sample(chaves, 1_000)

def busca_lista():
    for c in consulta:
        next((v for k, v in lista if k == c), None)

def busca_tabela():
    for c in consulta:
        tabela.get(c)

print("Lista:", timeit.timeit(busca_lista, number=1))
print("Dict :", timeit.timeit(busca_tabela, number=1))

```

6. Ampliação:

- Implementar heap de máx-prioridade invertendo sinais ou criando classe própria, e comparar desempenho.
- Analisar impactos de funções de hash mal distribuídas e explorar heapreplace.

5. Encerramento e orientações finais (20 minutos)

- A resolução de dúvidas sobre colisões e fator de carga.
- Lembrete sobre o envio do relatório.

4. Relatório final

Cada aluno (ou equipe) deve produzir um relatório curto (2 a 3 páginas) contendo:

Resumo teórico:

- Definir tabelas de dispersão, explicar colisões e tratamentos.
- Descrever heaps binários e justificar eficiência em filas de prioridade.

Código-fonte comentado:

- Inserir implementações das medições solicitadas, com observações sobre linhas-chave (cálculo de hash, heappush, heappop).
- Apresentar tabela dos tempos obtidos em cada experimento.

5. Critérios de avaliação

Critério	Peso	Descrição
Qualidade do resumo teórico	2,0	Precisão conceitual e clareza de exposição.
Estrutura e organização do código e funcionamento da solução	3,0	O código deve estar indentado corretamente, usar nomes de variáveis adequados e conter comentários informativos (quando necessários).
	3,0	Execução sem erros, coleta e exibição confiável dos tempos.
Criatividade e aprimoramentos	2,0	Implementação de heap de máx-prioridade, visualizações simples ou análise de fator de carga.

Nota final: Será a soma dos valores obtidos em cada critério. Alunos ou equipes que não cumprirem os requisitos mínimos de funcionamento do código ou não entregarem o relatório dentro do prazo terão sua nota diminuída proporcionalmente.



6. Conclusão

Este roteiro busca oferecer uma base sólida para que os alunos compreendam as hash tables e heaps e apliquem-nos em Python de forma interativa.

Ao final da aula, espera-se que cada estudante seja capaz de:

- Explicar o funcionamento de tabelas de dispersão e heaps, indicando vantagens e limitações.
- Implementar operações básicas nessas estruturas em Python, observando o desempenho empírico.
- Elaborar relatório conciso que articule fundamentos teóricos e resultados práticos.

Bom estudo e boa prática de programação!

  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Algoritmos e Estrutura de Dados em Python</p> <p>Título da aula: Algoritmos de Grafos – Dijkstra, Bellman-Ford, Kruskal e Prim em Python</p>	<p>ROTEIRO 7</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 07: ALGORITMOS DE GRAFOS – DIJKSTRA, BELLMAN-FORD, KRUSKAL E PRIM EM PYTHON

1. Objetivos da aula

- Apresentar os problemas de caminho mínimo e de árvore geradora mínima em grafos ponderados.
- Comparar quatro algoritmos clássicos: o Dijkstra, o Bellman-Ford, o Kruskal e o Prim, destacando hipóteses de uso (pesos negativos, grafos esparsos ou densos).
- Implementar, em Python, versões básicas de cada algoritmo, utilizando listas de adjacências e o módulo `heapq` quando apropriado.
- Realizar experimento prático de medição de tempo em instâncias geradas aleatoriamente, produzindo relatório com síntese teórica e código comentado.

2. Recursos necessários

- Computadores ou dispositivos com acesso à internet (laboratório ou BYOD – *bring your own device*).
- Acesso a um interpretador Python 3 (instalado localmente ou online, como [OnlineGDB](#), [Online Python](#) ou [OneCompiler](#)).
- Material de apoio (capítulo 7 do livro-texto).
- Editor de texto ou IDE (PyCharm, VS Code, Thonny) para organização do relatório final.

3. Estrutura da aula

1. Abertura (10 minutos)

- Contextualização: relembrar representações de grafos (listas e matrizes de adjacência) e motivar problemas de roteamento e redes.
- Ligação com Python: apontar que, embora a biblioteca networkx ofereça implementações prontas, compreender a lógica interna é essencial para o uso crítico.

2. Revisão conceitual (20 minutos)

- Conceitos centrais: caminho mínimo, relaxamento de arestas, ciclos negativos, propriedade de árvore geradora.
- Análise assintótica:
 1. Algoritmo de Dijkstra – custo $O((V + E) \log V)$ com heap; restrição a pesos não negativos.
 2. Algoritmo de Bellman-Ford – custo $O(V E)$, lida com pesos negativos e detecta ciclos negativos.
 3. Algoritmo de Kruskal – custo $O(E \log E)$ com união-busca; adequado para grafos esparsos.
 4. Algoritmo de Prim – custo $O((V + E) \log V)$ com heap; preferível em grafos densos.

3. Demonstração prática (30 minutos)

- A preparação: abrir IDE, importar heapq, random e timeit.
- A implementação de Dijkstra sobre lista de adjacência:

```
import heapq

def dijkstra(grafo, origem):
    dist = {v: float('inf') for v in grafo}
    dist[origem] = 0
    fila = [(0, origem)]
    while fila:
        d, u = heapq.heappop(fila)
        if d > dist[u]:
            continue
        for v, w in grafo[u]:
            novo = d + w
            if novo < dist[v]:
                dist[v] = novo
                heapq.heappush(fila, (novo, v))
    return dist
```

- A ilustração de relaxamento no Bellman-Ford, mostrando detecção de ciclo negativo.
- A demonstração de Kruskal com união-busca (parent, rank).
- A execução de Prim com heap mínimo.

4. Atividade prática (40 minutos)

- **Desafio:** Cada estudante deve desenvolver um programa que:
 1. Gere um grafo ponderado aleatório conectado com 200 vértices e 400 arestas.
 2. Execute o Dijkstra e o Bellman-Ford a partir de vértice escolhido pelo usuário; medir tempos.
 3. Calcule árvore geradora mínima com Kruskal e com Prim; comparar pesos totais e tempos.
 4. Esboço mínimo de medição:

```
import timeit
tm_dij = timeit.timeit(lambda: dijkstra(g, 0), number=1)
tm_bf = timeit.timeit(lambda: bellman_ford(g, 0), number=1)
print(f"Dijkstra: {tm_dij:.4f}s    Bellman-Ford: {tm_bf:.4f}s")
```

5. Ampliação:
 - Adicionar pesos negativos em apenas 5 % das arestas e discutir impacto nos algoritmos.
 - Comparar implementações próprias com networkx e plotar grafos com matplotlib.

5. Encerramento e orientações finais (20 minutos)

- Elucidação de dúvidas sobre união-busca e complexidade amortizada.
- Lembrete sobre o envio do relatório.

4. Relatório final

Cada aluno (ou equipe) deve produzir um relatório curto (2 a 3 páginas) contendo:

Resumo teórico:

- Explicar diferenças entre caminhos mínimos de fonte única e árvores geradoras mínimas.
- Apontar condições de aplicabilidade (pesos negativos, denso × esperso).

Código-fonte comentado:

- Incluir implementações completas, indicando linhas de relaxamento e união-busca.
- Apresentar tabela de tempos e pesos totais das árvores.

5. Critérios de avaliação

Critério	Peso	Descrição
Qualidade do resumo teórico	2,0	Precisão conceitual e clareza de exposição.
Estrutura e organização do código e funcionamento da solução	3,0	O código deve estar indentado corretamente, usar nomes de variáveis adequados e conter comentários informativos (quando necessários).
	3,0	Execução sem erros, resultados coerentes.
Criatividade e aprimoramentos	2,0	Uso de visualizações, análise de ciclos negativos ou comparação com bibliotecas externas.

Nota final: Será a soma dos valores obtidos em cada critério. Alunos ou equipes que não cumprirem os requisitos mínimos de funcionamento do código ou não entregarem o relatório dentro do prazo terão sua nota diminuída proporcionalmente.



6. Conclusão

Este roteiro busca oferecer uma base sólida para que os alunos compreendam os algoritmos de grafos e apliquem-nos em Python de forma interativa.

Ao final da aula, espera-se que cada estudante seja capaz de:

- a. Descrever hipóteses e custos dos algoritmos de Dijkstra, Bellman-Ford, Kruskal e Prim.
- b. Implementar tais algoritmos em Python, avaliando desempenho em diferentes grafos.
- c. Redigir relatório conciso que integre teoria, código e métricas experimentais.

Bom estudo e boa prática de programação!

  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Algoritmos e Estrutura de Dados em Python</p> <p>Título da aula: Técnicas de Divisão e Conquista e de Programação Dinâmica em Python</p>	<p>ROTEIRO 8</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 08: TÉCNICAS DE DIVISÃO E CONQUISTA E DE PROGRAMAÇÃO DINÂMICA EM PYTHON

1. Objetivos da aula

- Apresentar os fundamentos das estratégias de divisão e conquista e de programação dinâmica, situando-as no panorama de desenho de algoritmos.
- Comparar as hipóteses necessárias a cada técnica (sub-problemas independentes *versus* sobrepostos).
- Implementar exemplos representativos em Python, medindo desempenho de abordagens recursivas ingênuas, com memoização e com tabelas "bottom-up".
- Elaborar um relatório que integre resumo teórico, código-fonte comentado e análise empírica.

2. Recursos necessários

- Computadores ou dispositivos com acesso à internet (laboratório ou BYOD – *bring your own device*).
- Acesso a um interpretador Python 3.9 ou superior.
- Material de apoio (capítulo 8 do livro-texto).
- Editor de texto ou IDE (PyCharm, VS Code, Thonny) para organização do relatório final.

3. Estrutura da aula

1. Abertura (10 minutos)

- Motivação: lembrar problemas ordenados por complexidade e mostrar por que a escolha da estratégia pode reduzir drasticamente o tempo de execução.
- Ligação com Python: frisar que a sintaxe clara favorece a experimentação de diferentes paradigmas.

2. Revisão conceitual (20 minutos)

- As ideias-chave de divisão e conquista:
- A fase de divisão – particionar o problema em sub-problemas independentes.
- A fase de conquista – resolver cada parte recursivamente.
- A fase de combinação – unir resultados parciais em solução global.
- A análise por recorrência – aplicação do Teorema Mestre.
- As ideias-chave de programação dinâmica:
- A sub-estrutura ótima – solução global derivada de soluções ótimas de sub-problemas.
- Os sub-problemas sobrepostos – reutilização de resultados previamente calculados.
- A memoização (top-down) e a tabulação (bottom-up).
- A comparação entre as técnicas, destacando quando usar cada uma.

3. Demonstração prática (30 minutos)

- A preparação: abrir IDE, importar `functools.lru_cache`, `timeit` e `random`.
- O exemplo Fibonacci:

```
# -----  
# 2. Sequência de Fibonacci  
#   (a) Recursão exponencial  
#   (b) Memoização com lru_cache  
#   (c) Abordagem iterativa bottom-up ( $O(n)$  tempo,  $O(1)$  espaço)  
# -----  
  
from functools import lru_cache  
  
def fib_rec(n: int) -> int:  
    """Versão recursiva pura (muito lenta para n grande)."""  
    if n <= 1:  
        return n  
    return fib_rec(n - 1) + fib_rec(n - 2)  
  
@lru_cache(maxsize=None)  
def fib_memo(n: int) -> int:  
    """Versão top-down com memoização."""  
    if n <= 1:  
        return n  
    return fib_memo(n - 1) + fib_memo(n - 2)  
  
def fib_iter(n: int) -> int:  
    """Versão bottom-up iterativa."""  
    if n <= 1:  
        return n  
    a, b = 0, 1  
    for _ in range(2, n + 1):  
        a, b = b, a + b  
    return b  
  
# Exemplo rápido  
if __name__ == "__main__":  
    n = 35  
    print("fib_rec   :", fib_rec(n))  
    print("fib_memo  :", fib_memo(n))
```

- A versão recursiva simples (exponencial).
- A versão com memoização (@lru_cache) – tempo quase linear.
- A versão iterativa bottom-up – tempo linear e espaço $O(1)$.
- A medição de tempo com timeit para $n = 35$ e $n = 1\ 000$, evidenciando ganhos.

4. Atividade prática (40 minutos)

- **Desafio:** Cada estudante deve desenvolver um programa que tenha:
 - O problema da soma de subconjunto: determinar se há subconjunto que soma S .
 - O problema do troco mínimo: calcular o número mínimo de moedas para valor V .
- Após a escolha, o estudante deve:
 - Implementar solução recursiva direta.
 - Adicionar memoização.
 - Reescrever em estilo bottom-up.
 - Medir tempos de execução para três valores crescentes de entrada e apresentar em ordem crescente de eficiência.
- Ampliação:
 - Criar gráfico de barras comparando as três abordagens (a ser desenvolvido em casa).
 - Estudar multiplicação de matrizes em corrente (Matrix Chain) e edição de sequências (Edit Distance).

5. Encerramento e orientações finais (20 minutos)

- O esclarecimento de dúvidas sobre escolha de estrutura de dados para tabelas.
- Lembrete sobre o envio do relatório.

4. Relatório final

Cada aluno (ou equipe) deve produzir um relatório curto (2 a 3 páginas) contendo:

Resumo teórico:

- A explanação das diferenças estruturais entre divisão e conquista e programação dinâmica.
- A justificativa dos ganhos obtidos com memoização ou tabulação nos problemas escolhidos.

Código-fonte comentado:

- A inclusão das três versões do algoritmo escolhido (recursiva simples, memoizada, bottom-up).
- A apresentação dos tempos medidos em tabela.

5. Critérios de avaliação

Critério	Peso	Descrição
Qualidade do resumo teórico	2,0	Precisão conceitual e clareza de exposição.
Estrutura e organização do código e funcionamento da solução	3,0	O código deve estar indentado corretamente, usar nomes de variáveis adequados e conter comentários informativos (quando necessários).
	3,0	Execução sem erros, resultados coerentes.
Criatividade e aprimoramentos	2,0	Análise gráfica, discussão sobre consumo de memória ou casos extremos.

Nota final: Será a soma dos valores obtidos em cada critério. Alunos ou equipes que não cumprirem os requisitos mínimos de funcionamento do código ou não entregarem o relatório dentro do prazo terão sua nota diminuída proporcionalmente.

6. Conclusão

Este roteiro busca oferecer uma base sólida para que os alunos compreendam as técnicas de divisão e conquista e de programação dinâmica e apliquem-nas em Python de forma interativa.

Ao final da aula, espera-se que cada estudante seja capaz de:

- Explicar quando aplicar divisão e conquista e quando preferir programação dinâmica.
- Implementar ambas as estratégias em Python, quantificando ganhos de desempenho.
- Documentar resultados em relatório sintético que una teoria, implementação e evidência empírica.

Bom estudo e boa prática de programação!