**Week 3 Mini-Lecture: Scaling Complexity & Algorithmic Thinking**

### **1. Overview**

This mini-lecture introduces students to the concept of algorithmic thinking and scaling complexity in AI-assisted programming. Students will explore how to structure solutions for efficiency, implement iterative improvements, and apply algorithmic patterns such as loops, conditionals, and recursion to solve increasingly complex problems.

### **2. Learning Objectives**

By the end of this session, students will:

* Understand how to break complex problems into structured steps.
* Apply AI-assisted programming to implement efficient loops, conditionals, and recursive functions.
* Optimize AI-generated scripts for efficiency and scalability.
* Develop iterative approaches for solving large-scale problems.

### **3. Key Concepts**

#### **What is Algorithmic Thinking?**

* Algorithmic thinking is the process of designing **step-by-step logical solutions** to problems.
* AI-generated code often lacks structured logic and optimization, requiring human intervention.
* Structuring problems algorithmically leads to **better performance, clarity, and scalability.**

#### **Scaling Complexity in AI-Assisted Programming**

* As data grows, inefficient algorithms slow down performance.
* AI-generated code may work for small datasets but fail at scale.
* Students will learn how to **refactor AI-generated code for larger datasets.**

### **4. Step-by-Step Approach to Algorithmic Thinking**

#### **Step 1: Breaking Down Problems**

* Identify sub-tasks within a problem.
* Example: Analyzing sales data → Load data → Clean data → Aggregate sales → Generate reports.
* Ask AI for code in **modular components** instead of one large script.

#### **Step 2: Choosing the Right Algorithm**

* Use loops for repetition, conditionals for decision-making, and recursion for nested structures.
* Example:
  + **Loops:** Processing thousands of records.
  + **Conditionals:** Filtering only relevant data points.
  + **Recursion:** Navigating hierarchical structures.

#### **Step 3: Optimizing AI-Generated Code**

* Identify **redundant calculations** and replace them with **optimized logic.**
* Example: AI-generated code using a slow loop for summing values:

# AI-generated inefficient sum function

numbers = [1, 2, 3, 4, 5]

total = 0

for num in numbers:

total += num

* Optimized version using built-in functions:

total = sum(numbers) # Faster and more efficient

### **5. Common AI-Generated Inefficiencies & Debugging Strategies**

#### **1. Unnecessary Nested Loops**

* AI often generates multiple loops when a single-pass algorithm would be better.
* **Optimization Tip:** Use dictionary lookups or list comprehensions.

#### **2. Inefficient Sorting & Searching**

* AI-generated scripts may rely on slow for loops for searching instead of set or dict lookups.
* **Optimization Tip:** Use hashing techniques for faster search operations.

#### **3. Overuse of Recursion Without Optimization**

* Recursive solutions can be elegant but may lead to **stack overflow** errors if not optimized.
* **Optimization Tip:** Use memoization or iterative approaches where applicable.

### **6. Hands-On Example: AI-Generated vs. Optimized Code**

#### **AI-Generated Code (Inefficient Recursive Fibonacci Calculation)**

def fibonacci(n):

if n == 0:

return 0

elif n == 1:

return 1

return fibonacci(n - 1) + fibonacci(n - 2)

#### **Optimized Code (Using Memoization for Efficiency)**

from functools import lru\_cache

@lru\_cache(None)

def fibonacci(n):

if n == 0:

return 0

elif n == 1:

return 1

return fibonacci(n - 1) + fibonacci(n - 2)

### **7. Wrap-Up & Takeaways**

* Algorithmic thinking is key to scaling AI-generated code efficiently.
* Debugging AI-generated scripts requires **identifying inefficiencies** and **applying structured improvements**.
* Iterative refinement leads to **better performance and maintainability**.
* Next, students will apply these techniques to real-world AI-generated projects in **Week 4**.