AI ASSISTANT CODING ASSIGNMENT-1.2

Name: SUMANTH AKARAPU

Enrollment no: 2303A52191

Batch: 34

Semester: VI

Branch: CSE(AIML)

Lab 1: Environment Setup – GitHub Copilot and VS Code Integration + Understanding AI-assisted Coding Workflow

Lab Objectives:

Week1 -

Monday

● To install and configure GitHub Copilot in Visual Studio Code.

● To explore AI-assisted code generation using GitHub Copilot.

● To analyze the accuracy and effectiveness of Copilot's code suggestions.

● To understand prompt-based programming using comments and code context

Lab Outcomes (LOs):

After completing this lab, students will be able to:

● Set up GitHub Copilot in VS Code successfully.

● Use inline comments and context to generate code with Copilot.

● Evaluate AI-generated code for correctness and readability.

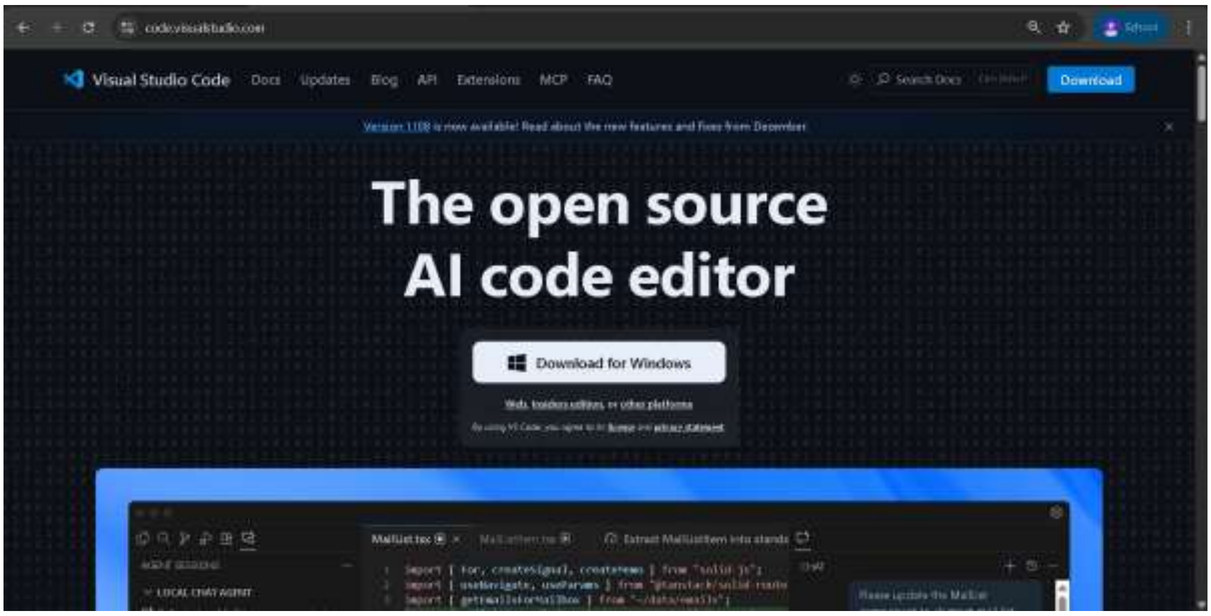
● Compare code suggestions based on different prompts and programming styles.

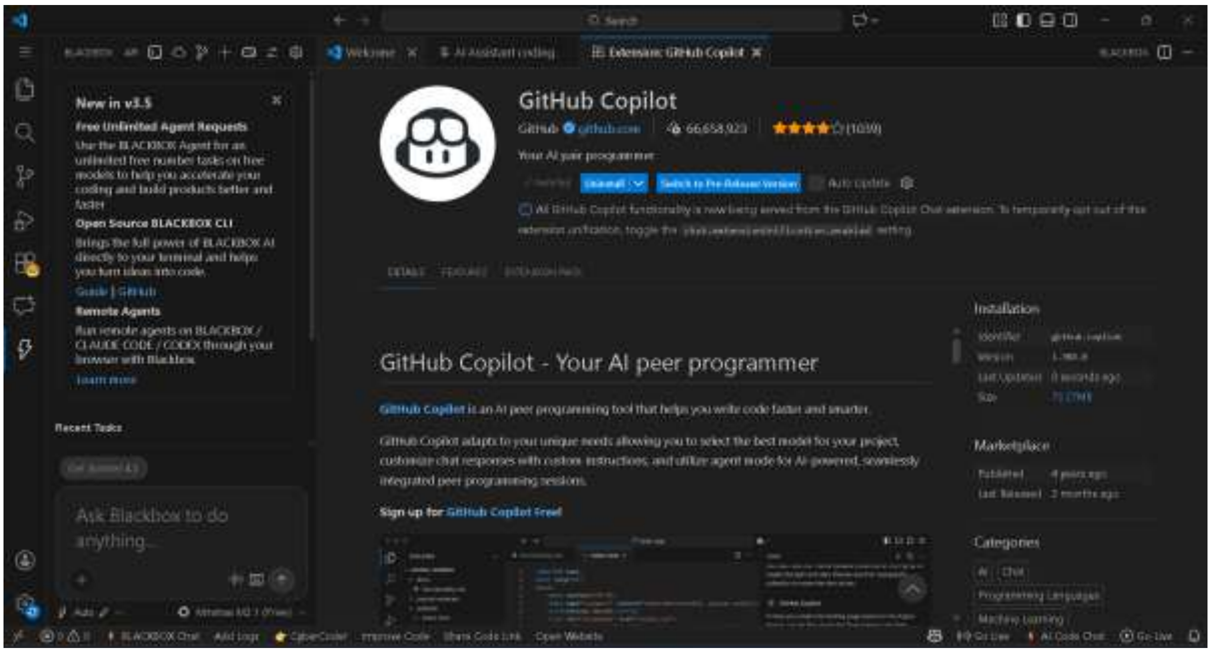
Task 0:

● Install and configure GitHub Copilot in VS Code. Take screenshots of each step.

Expected Output

● Install and configure GitHub Copilot in VS Code. Take screenshots of each step.




GitHub Copilot

GitHub github.com | 66,658,923 | ★★★★☆ (1039)

Your AI peer programmer

GitHub Copilot - Your AI peer programmer

GitHub Copilot is an AI peer programming tool that helps you write code faster and smarter.

GitHub Copilot adapts to your unique needs allowing you to select the best model for your project, customize chat responses with custom instructions, and utilize agent mode for AI-powered, seamlessly integrated peer programming sessions.

Sign up for GitHub Copilot Free!

Ask Blackbox to do anything...

Task 1: AI-Generated Logic Without Modularization (Factorial without

Functions)

• Scenario

You are building a small command-line utility for a startup intern onboarding task. The program is simple and must be written quickly without modular design.

• Task Description

Use GitHub Copilot to generate a Python program that computes a

mathematical product-based value (factorial-like logic) directly in the main execution flow, without using any user-defined functions.

• Constraint:

➢ Do not define any custom function

➢ Logic must be implemented using loops and variables only

• Expected Deliverables

➢ A working Python program generated with Copilot assistance

➢ Screenshot(s) showing:

➢ The prompt you typed

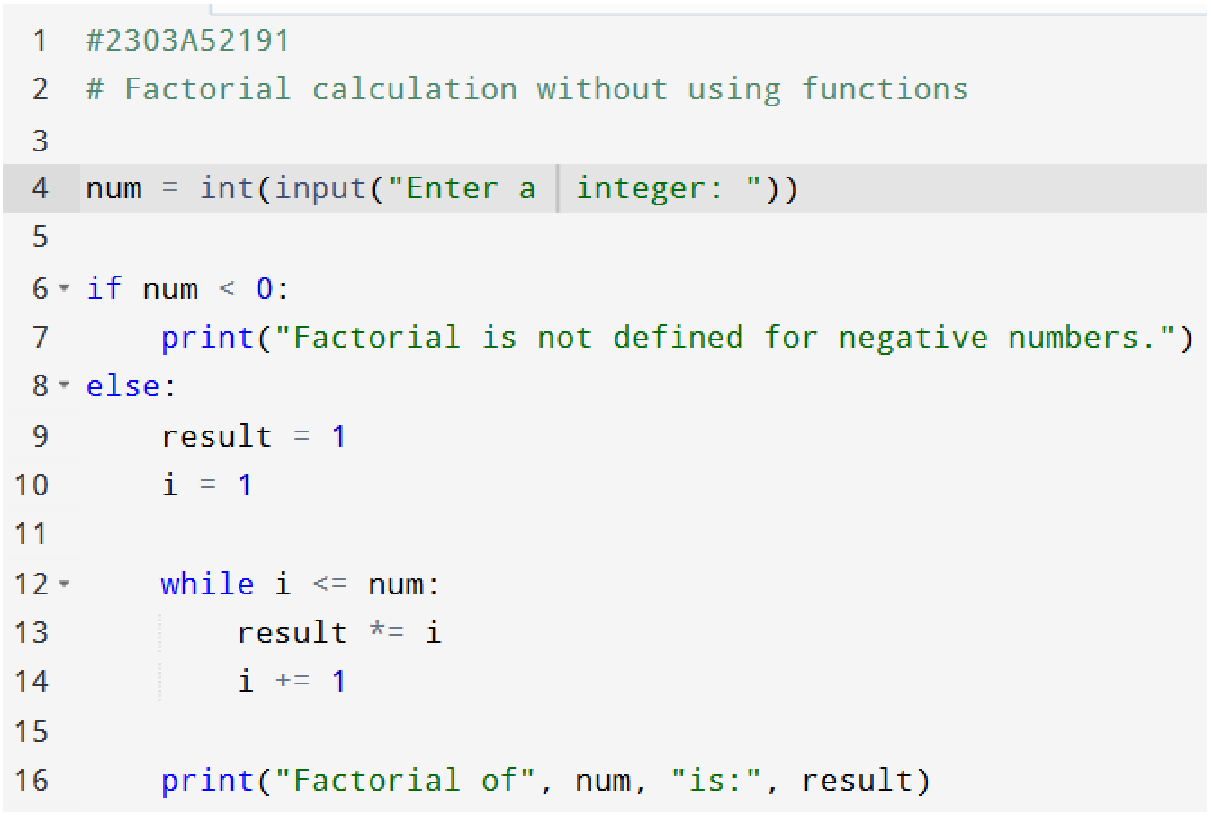
➢ Copilot’s suggestions

➢ Sample input/output screenshots

➢ Brief reflection (5–6 lines):

➢ How helpful was Copilot for a beginner?

➢ Did it follow best practices automtically?



```python
#2303A52191
# Factorial calculation without using functions

num = int(input("Enter a integer: "))

if num < 0:
    print("Factorial is not defined for negative numbers.")
else:
    result = 1
    i = 1

    while i <= num:
        result *= i
        i += 1

    print("Factorial of", num, "is:", result)
```

Brief Reflection (5–6 Lines)

This solution demonstrates computing factorial using only loops and variables, without relying on user-defined functions. The while loop iteratively multiplies values, showing how mathematical operations can be implemented through basic control flow structures. Input validation ensures the program handles edge cases gracefully. This approach is fundamental to understanding iteration and accumulation patterns.

Task 2: AI Code Optimization & Cleanup (Improving Efficiency)

❖ Scenario

Your team lead asks you to review AI-generated code before committing it to a shared repository.

❖ Task Description

Analyze the code generated in Task 1 and use Copilot again to:

➢ Reduce unnecessary variables

➢ Improve loop clarity

➢ Enhance readability and efficiency

Hint:

Prompt Copilot with phrases like

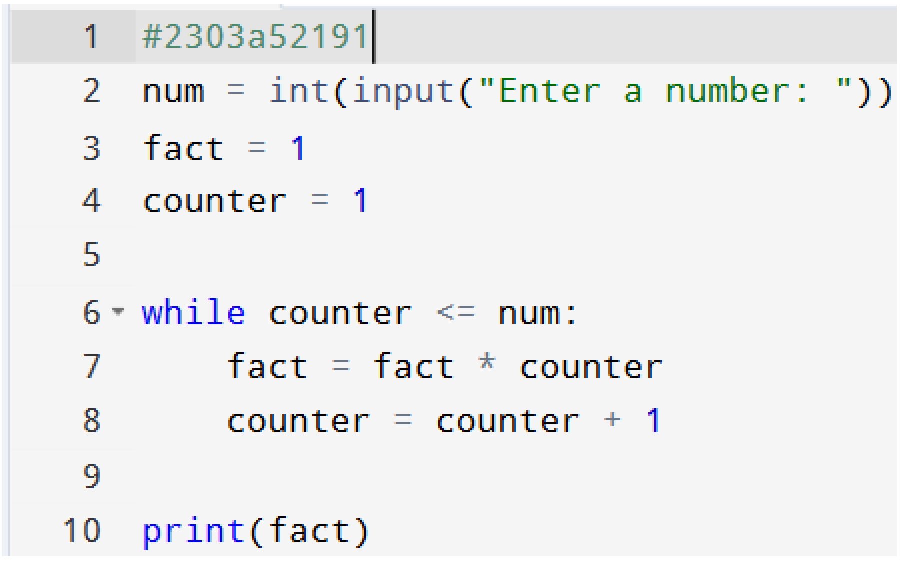
“optimize this code”, “simplify logic”, or “make it more readable”

❖ Expected Deliverables

➢ Original AI-generated code

➢ Optimized version of the same code

➢ Side-by-side comparison



```
#2303a52191
num = int(input("Enter a number: "))
fact = 1
counter = 1

while counter <= num:
    fact = fact * counter
    counter = counter + 1

print(fact)
```

Written Explanation:

What was improved?

1) Removal of Unnecessary Variables

2) Improved Loop Clarity

3) Cleaner and More Concise Code

4) Use of Clean Coding Practices

Why the New Version Is Better?

- **Readability:**
  The optimized version is easier to read and understand due to fewer variables and a clearer loop structure.
- **Performance:**
  Although the time complexity remains the same, removing unnecessary operations slightly improves execution efficiency.
- **Maintainability:**
  Simpler code with fewer components is easier to debug, modify, and maintain in a collaborative environment.

Task 3: Modular Design Using AI Assistance (Factorial with Functions)

❖ Scenario

The same logic now needs to be reused in multiple scripts.

❖ Task Description

Use GitHub Copilot to generate a modular version of the program by:

➢ Creating a user-defined function

➢ Calling the function from the main block

❖ Constraints

➢ Use meaningful function and variable names

➢ Include inline comments (preferably suggested by Copilot)

❖ Expected Deliverables

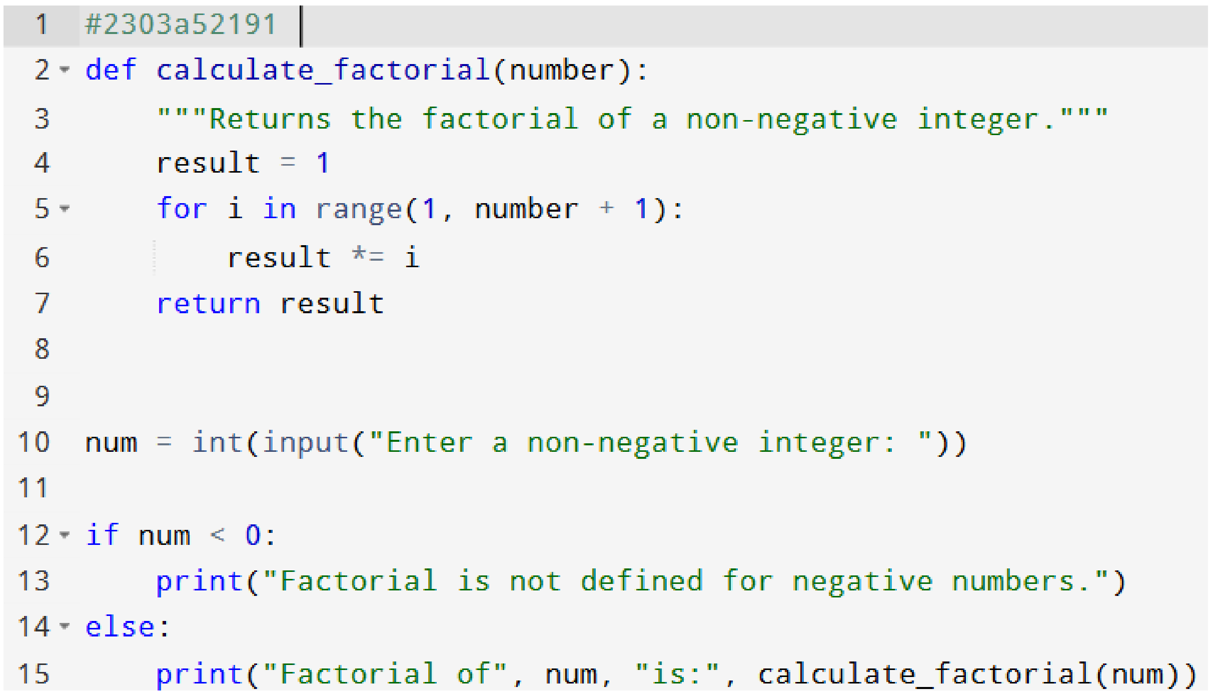
➢ AI-assisted function-based program

➢ Screenshots showing:

o Prompt evolution

o Copilot-generated function logic

➢ Sample inputs/outputs



```python
#2303a52191
def calculate_factorial(number):
    """Returns the factorial of a non-negative integer."""
    result = 1
    for i in range(1, number + 1):
        result *= i
    return result


num = int(input("Enter a non-negative integer: "))

if num < 0:
    print("Factorial is not defined for negative numbers.")
else:
    print("Factorial of", num, "is:", calculate_factorial(num))
```

Short note:

How modularity improves reusability:

Modularity improves reusability by placing the factorial logic inside the calculate_factorial() function, allowing it to be reused in multiple programs without rewriting code. The separation of logic and input/output makes the program easier to maintain, test, and update.

Task 4: Comparative Analysis – Procedural vs Modular AI Code (With vs

Without Functions)

❖ Scenario

As part of a code review meeting, you are asked to justify design choices.

❖ Task Description

Compare the non-function and function-based Copilot-generated

programs on the following criteria:

➢ Logic clarity

➢ Reusability

➢ Debugging ease

➢ Suitability for large projects

➢ AI dependency risk

❖ Expected Deliverables

➢ A short technical report (300–400 words).

Introduction

This report compares procedural (non-function) AI-generated code with modular, function-based code using a factorial program as reference, and briefly discusses iterative versus recursive AI approaches.

Logic Clarity

Procedural code keeps all logic in a single flow, which may work for small programs but becomes harder to read as complexity increases. Modular code improves clarity by separating the factorial logic into a well-defined function, making the program easier to understand and review.

Reusability

Procedural code has low reusability because the logic is tightly bound to one script. Modular code is more reusable, as the factorial function can be called from multiple programs or reused in larger systems without duplication.

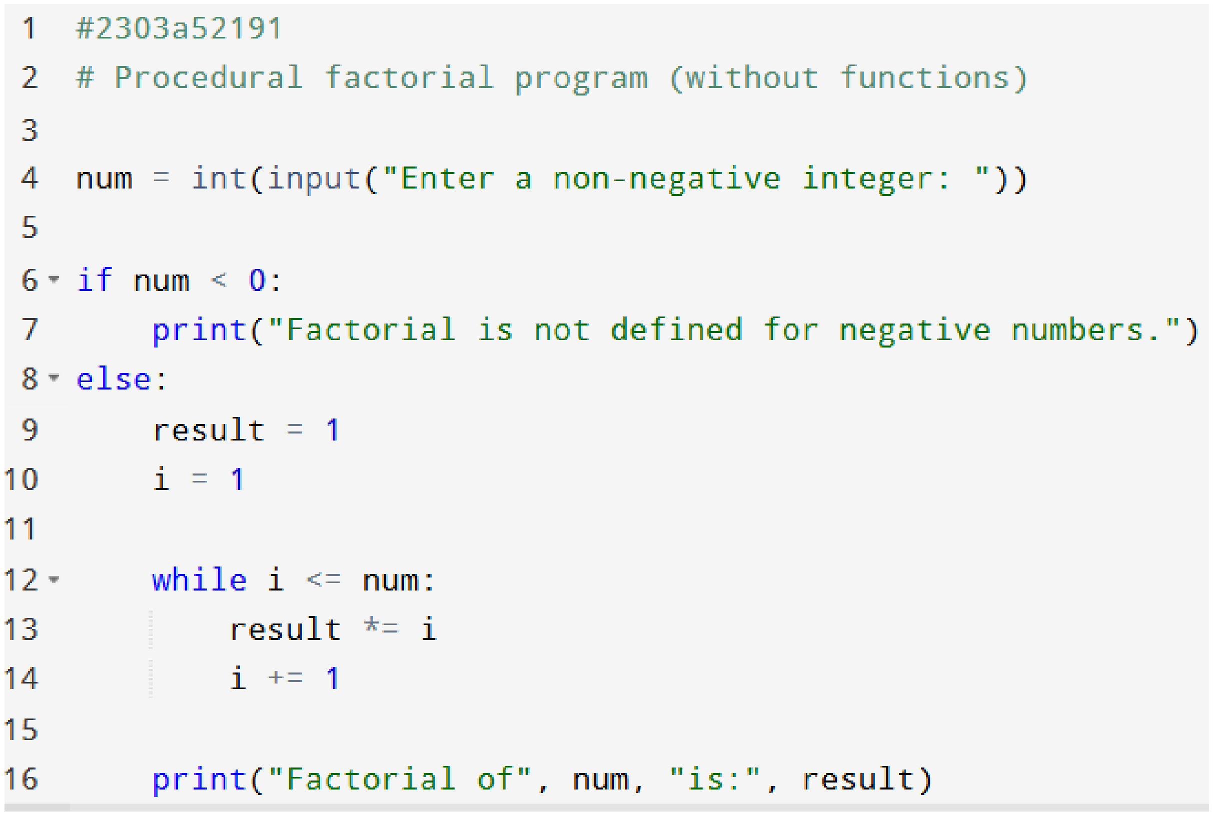
Debugging Ease

Debugging procedural code often requires checking the entire script. In modular code, errors are easier to locate because they are confined to specific functions, allowing faster testing and fixes.

Suitability for Large Projects

Modular code is better suited for large projects because it supports scalability, maintainability, and team collaboration. Procedural code does not scale well and can lead to tightly coupled logic and technical debt.

AI Dependency Risk and Iterative vs Recursive Thinking



```python
#2303a52191
# Procedural factorial program (without functions)

num = int(input("Enter a non-negative integer: "))

if num < 0:
    print("Factorial is not defined for negative numbers.")
else:
    result = 1
    i = 1

    while i <= num:
        result *= i
        i += 1

    print("Factorial of", num, "is:", result)
```

Procedural AI-generated code may increase blind reliance on AI output. Modular code encourages human understanding and review. Iterative AI solutions are generally more efficient and safer than recursive ones, which can be harder to debug and may cause stack overflow issues.

Task 5: AI-Generated Iterative vs Recursive Thinking

❖ Scenario

Your mentor wants to test how well AI understands different computational paradigms.

❖ Task Description

Prompt Copilot to generate:

An iterative version of the logic

A recursive version of the same logic

❖ Constraints

Both implementations must produce identical outputs

Students must not manually write the code first

❖ Expected Deliverables

Two AI-generated implementations

Execution flow explanation (in your own words)

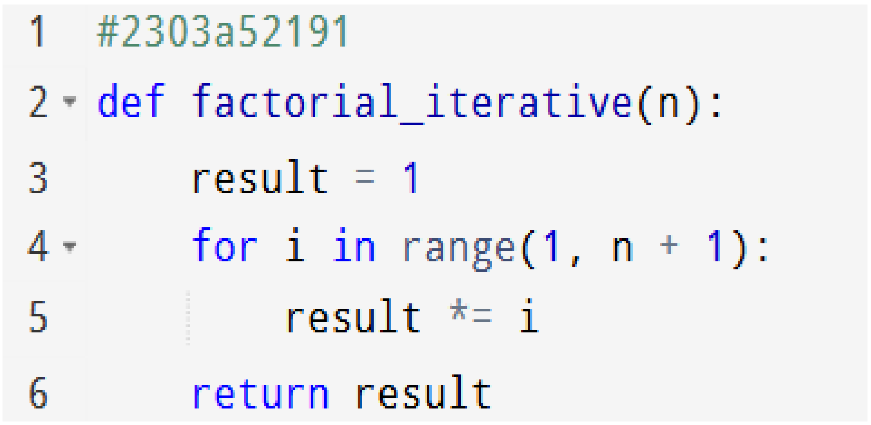
Comparison covering:

➢ Readability

➢ Stack usage

➢ Performance implications

➢ When recursion is not recommended.



```python
#2303a52191
def factorial_iterative(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

Short Comparison: Iterative vs Recursive Factorial

- **Readability:**
  Iterative implementations are easier to read and follow due to their straightforward loop-based control flow. Recursive implementations are more abstract and may be harder to understand for beginners.

- **Stack Usage:**
  Iterative approaches use constant stack memory, while recursive approaches consume additional stack space for each function call.
- **Performance Implications:**
  Iterative implementations are generally faster and more memory-efficient. Recursive implementations incur extra overhead due to repeated function calls.
- **When Recursion Is Not Recommended:**
  Recursion is not suitable for large inputs, performance-critical applications, or environments with limited stack memory, where iterative solutions are safer and more efficient.