AI Assisted Coding - Assignment 1.2

Name: Goutham

HT No: 2303A52010

Batch: 31

Question 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.

Constraints

- Do not define any custom function
- Logic must be implemented using loops and variables only

Prompt:

```
Write a Python program to calculate factorial of a number using loops.

Do not use any user-defined functions.

Handle negative input properly.
```

Code Screenshot:

```
n = int(input("Enter a number: "))


if n < 0:
```

```
    print("Factorial is not defined for negative numbers")
else:
    result = 1
    for i in range(1, n + 1):
        result *= i
    print("Factorial:", result)
```

Output (O/P):

```
Enter a number: 5
Factorial: 120
```

Explanation:

Copilot proved to be especially useful for a beginner by quickly producing accurate logic from simple prompts. It significantly cut down development time by offering ready-to-use solutions. The suggested code adhered to best practices, including proper input validation and handling of edge cases. It also helped prevent syntax issues and logical errors, making the overall coding process smoother, faster, and more approachable.

Question 2: Code Optimization and Readability

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”.

Prompt:

```
Optimize this factorial code.

Reduce unnecessary variables.

Make the logic more readable and efficient.
```

Code Screenshot:

```
n = int(input("Enter a number: "))


fact = 1

for i in range(2, n + 1):

    fact *= i


print("Factorial:", fact)
```

Output (O/P):

```
Enter a number: 5

Factorial: 120
```

Explanation:

Optimized the code by reducing it to the minimum number of lines and using concise variable names.

Question 3: Modularization 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)

Prompt:

```
Convert this factorial program into a modular version.

Use a user-defined function.

Call the function from the main block.

Add inline comments.
```

Code Screenshot:

```
def factorial(n):

    # returns factorial of n

    if n < 0:

        return None



    result = 1

    for i in range(1, n + 1):

        result *= i

    return result




if __name__ == "__main__":

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

    ans = factorial(num)



    if ans is None:
```

```
        print("Factorial not defined for negative numbers")
    else:
        print("Factorial:", ans)
```

Output (O/P):

```
Enter a number: 5
Factorial: 120
```

Explanation:

Copilot refactored the code into a function-based structure with a proper main block and handled edge cases effectively.

Question 4: Comparison of Non-function and Function-based Programs

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

Explanation:

Aspect	Non-function based	Function-based
Logic clarity	Logic is in one block. Harder to understand	Logic is separated. Easy to understand
Reusability	Code cannot be reused	Code can be reused easily
Debugging ease	Debugging is difficult	Debugging is easy
Suitability for large projects	Not suitable for large projects	Suitable for large projects
AI dependency risk	High risk of blind copying	Lower risk due to clear structure

Question 5: Iterative vs. Recursive Implementations

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

Prompt 1 (Iterative):

```
Generate an iterative Python program to compute factorial.
```

Code (Iterative):

```
n = int(input("Enter a number: "))

fact = 1


for i in range(1, n + 1):

    fact *= i


print("Factorial:", fact)
```

Output (Iterative):

```
Enter a number: 5

Factorial: 120
```

Prompt 2 (Recursive):

```
Generate a recursive Python program to compute factorial.
```

Code (Recursive):

```
def factorial(n):

    if n == 0 or n == 1:

        return 1

    return n * factorial(n - 1)




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

print("Factorial:", factorial(num))
```

Output (Recursive):

```
Enter a number: 5
```

```
Factorial: 120
```

Iterative version:

The program starts with an initial value and repeatedly updates it using a loop until the condition is satisfied. The control flow stays within a single function and finishes once the loop ends.

Recursive version:

The program calls itself with a smaller input each time. Each call waits on the stack until the base condition is reached. After that, results are returned step by step.

Comparison:

Aspect	**Iterative**	**Recursive**
Readability	Easy to follow for beginners	Slightly harder due to function calls
Stack usage	Uses constant memory	Uses call stack for each function call
Performance implications	Faster and memory efficient	Slower for large inputs due to stack overhead
When recursion is not recommended	Always safe for large inputs	Not recommended when input size is large or stack overflow is possible