AI - Assisted Coding-Assignment -1

HTNO:2303A52228
Batch:37

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

Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence Without Functions)
You are asked to write a quick numerical sequence generator for a learning platform prototype.

Prompt: Generate a python code of Fibonacci Sequence without Functions upto n terms.

CODE AND OUTPUT:

```
# Program to print Fibonacci sequence up to n terms

# Accept user input
n = int(input("Enter the number of terms: "))

# Initialize the first two terms
a, b = 0, 1

print("Fibonacci sequence:")
for i in range(n):
    print(a, end=" ")
    a, b = b, a + b
```

Example Run:

Code Copy

```
Enter the number of terms: 10
Fibonacci sequence:
0 1 1 2 3 5 8 13 21 34
```

Explanation: The Fibonacci program is like teaching the computer a simple counting game. First, you tell it how many numbers you want to see, and it starts with 0 and 1•the first two building blocks of the sequence. Then, with each step, the computer adds the last two numbers together to create the next one, almost like stacking blocks on top of each other. By repeating this process, it prints out the series of numbers in order, showing how the pattern grows naturally without needing any extra functions or complicated rules.

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

The prototype will be shared with other developers and needs optimization.

Prompt: Optimize the above code by removing redundant variables and Simplifying Loop Logic.

CODE AND OUTPUT:

```
Python

# Program to print Fibonacci sequence up to n terms

n = int(input("Enter the number of terms: "))

a, b = 0, 1

print("Fibonacci sequence:")
for i in range(n):
    print(a, end=" ")
    a, b = b, a + b
```

Improved code:

```
# Optimized Fibonacci sequence program

n = int(input("Enter the number of terms: "))

fib = [0, 1]

for i in range(2, n):
    fib.append(fib[-1] + fib[-2])

print("Fibonacci sequence:")
print(" ".join(str(x) for x in fib[:n]))
```

Explanation: The original Fibonacci code worked but was a bit messy, constantly juggling two variables and printing inside the loop, which made it harder to read and less flexible. It also did extra work even for small inputs. The optimized version cleans this up by storing numbers in a list and simply appending new terms based on the last two values. This makes the loop shorter, the logic clearer, and the output neatly printed all at once. In short, the improved code avoids redundancy and is easier to understand and maintain.

Task 3: Modular Design Using AI Assistance (Fibonacci Using Functions).
The Fibonacci logic is now required in multiple modules of an application.

Prompt: Generate a python code for Fibonacci Sequence upto n terms with user defined functions.
CODE AND OUTPUT:

```python
def fibonacci(n):
    if n<=0:
        return []
    elif n==1:
        return [0]
    seq=[0,1]
    for i in range(2,n):
        seq.append(seq[-1]+seq[-2])
    return seq
n=int(input("Enter the number of terms: "))
print("Fibonacci sequence:")
print(" ".join(str(x) for x in fibonacci(n)))
```

```
Enter the number of terms: 5
Fibonacci sequence:
0 1 1 2 3

=== Code Execution Successful ===
```

Explanation : Think of this program as a little helper that builds the Fibonacci sequence step by step. You tell it how many numbers you want, and the function starts with the basics—0 and 1. From there, it keeps adding the last two numbers together to grow the sequence, almost like stacking blocks where each new block is made from the two before it. The main part of the program simply asks you for a number, calls the helper function, and then shows you the sequence neatly in one line. It’s organized, easy to follow, and feels much cleaner than juggling variables directly in the loop.

Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code
Prompt: Comparison of code with Functions and without Functions.
CODE AND OUTPUT:

Aspect	Without Functions (Task 1)	With Functions (Task 3)
Code clarity	Logic is inline, simple for beginners but harder to follow as program grows.	Encapsulates logic in a function, making the flow cleaner and easier to read.
Reusability	Cannot be reused elsewhere; sequence generation is tied to main code.	Function can be called multiple times with different inputs, reusable across projects.
Debugging ease	Debugging is harder since logic and printing are mixed together.	Easier to debug because sequence generation is isolated in a function.
Suitability for larger systems	Not suitable; lacks modularity and scalability.	Well-suited; modular design integrates smoothly into larger applications.

Explanation: The first Fibonacci program, without functions, is like a quick sketch. It gets the job done, but everything—input, logic, and output—is crammed together. That makes it fine for small tasks but harder to reuse or fix if something goes wrong. The function-based version, on the other hand, is more like a neatly organized toolbox. The sequence logic lives inside its own function, so you can call it whenever you need, test it separately, and even plug it into bigger projects without rewriting code.

Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches for Fibonacci Series)
Prompt: Generate an Iterative Fibonacci Implementation and A Recursive Fibonacci Implementation.

CODE AND OUTPUT:

```python
def fibonacci_iterative(n):
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    seq = [0, 1]
    for i in range(2, n):
        seq.append(seq[-1] + seq[-2])
    return seq
n = int(input("Enter number of terms: "))
print("Iterative Fibonacci:", fibonacci_iterative(n))
```

```
Enter number of terms: 5
Iterative Fibonacci: [0, 1, 1, 2, 3]

=== Code Execution Successful ===
```

```python
def fibonacci_recursive(n):
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]
    seq = fibonacci_recursive(n - 1)
    seq.append(seq[-1] + seq[-2])
    return seq
n = int(input("Enter number of terms: "))
print("Recursive Fibonacci:", fibonacci_recursive(n))
```

```
Enter number of terms: 5
Recursive Fibonacci: [0, 1, 1, 2, 3]

=== Code Execution Successful ===
```

Explanation: The iterative approach runs in linear time,
$O(n)$,because each loop adds one new term. It uses
$O(n)$,space to store the sequence, but the logic is straightforward and efficient. Since everything happens step by step in a single loop, it scales well and is the preferred choice for large values of n. Debugging is also easier because the flow is simple and predictable.

The recursive approach builds the sequence by repeatedly calling itself until it reaches the base cases. While elegant and good for demonstrating the idea of breaking problems down, it has more overhead: time complexity grows to about $O(n2)$,due to repeated list operations, and space usage includes both the sequence and the call stack, which can reach depth $O(n)$. For small inputs, recursion works fine, but for larger inputs it quickly becomes slow and risks stack overflow, so it should be avoided in performance-critical or large-scale systems.