AI Assistant Coding

<u>Assignment-1</u>

Name of Student **: V.Shiva krishna** **Batch:41**

Enrollment No. **: 2303A52151**

Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence Without Functions)

Use GitHub Copilot to generate a Python program that:

➢ **Prints the Fibonacci sequence up to *n* terms**

➢ **Accepts user input for *n***

➢ **Implements the logic directly in the main code**

➢ **Does not use any user-defined functions**

Prompt:

Write a code for printing a Fibonacci series up to n terms without using a function.

Code:

```
As1.py > ...
# Write a code for printing a Fibonacci series up to n terms with
n = int(input("Enter the number of terms in Fibonacci series: "))
a, b = 0, 1
print("Fibonacci series up to", n, "terms:")
for _ in range(n):
    print(a, end=' ')
    a, b = b, a + b
```

Output:

```
/As1.py"
Enter the number of terms in Fibonacci series: 10
Fibonacci series up to 10 terms:
0 1 1 2 3 5 8 13 21 34
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

Justification:

Generated code using inline prompt of GitHub Copilot in VS Code for the logic of Fibonacci series up to n terms without using a function. It takes input from the user to iterate till n using for loop and print them in a sequence.

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

- ➢ **Examine the Copilot-generated code from Task 1 and improve it by:**
- ➢ **Removing redundant variables**
- ➢ **Simplifying loop logic**
- ➢ **Avoiding unnecessary computations**
- ➢ **Use Copilot prompts such as:**
  - ▪ **“Optimize this Fibonacci code”**
  - ▪ **“Simplify variable usage”**

Prompt:

Optimize the above code by removing unnecessary variables, simplifying the loop, and improving readability.

Code:

```
# Optimize the above code by removing unnecessary variables, simplifying the loop, and improving readability
n = int(input("Enter the number of terms in Fibonacci series: "))
print("Fibonacci series up to", n, "terms:")
a, b = 0, 1
for _ in range(n):
    print(a, end=' ')
    a, b = b, a + b
```

Output:

```
/As1.py"
Enter the number of terms in Fibonacci series: 9
Fibonacci series up to 9 terms:
0 1 1 2 3 5 8 13 21
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

Justification:

The program correctly generates Fibonacci numbers using an iterative approach starting from 0 and 1. It computes exactly *n* terms as input (9), ensuring no extra or missing values. The output format is clean, space-separated, and suitable for debugging or result verification

Task 3: Modular Design Using AI Assistance (Fibonacci Using Functions)

Use GitHub Copilot to generate a function-based Python program that:

- ➢ **Uses a user-defined function to generate Fibonacci numbers**
- ➢ **Returns or prints the sequence up to n**
- ➢ **Includes meaningful comments (AI-assisted)**

Prompt:

Write a code for printing the Fibonacci series up to n terms using a function and include meaningful comments.

Code:

```
# Write a code for printing the fibonacci series up to n terms using a function and include meaningful comments
def fibonacci_series(n):
    """
    This function prints the Fibonacci series up to n terms.

    Parameters:
    n (int): The number of terms in the Fibonacci series to be printed.
    """

    a, b = 0, 1  # Initialize the first two terms of the Fibonacci series
    print("Fibonacci series up to", n, "terms:")
    for _ in range(n):
        print(a, end=' ')  # Print the current term
        a, b = b, a + b  # Update to the next two terms

# Get the number of terms from the user
num_terms = int(input("Enter the number of terms in Fibonacci series: "))
fibonacci_series(num_terms)  # Call the function to print the series
```

Output:

```
/As1.py"
Enter the number of terms in Fibonacci series: 6
Fibonacci series up to 6 terms:
0 1 1 2 3 5 Enter the number of terms in Fibonacci series:
```

Justification:

In the above code Fibonacci Series logic is written in a Modular design with useful comments that explain what that line does. The “ fibonacci_series ” function can be called by the user as many times as needed.

Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code

Compare the Copilot-generated Fibonacci programs:

- ➢ **Without functions (Task 1)**
- ➢ **With functions (Task 3)**
- ➢ **Analyze them in terms of:**
  - ▪ **Code clarity**
  - ▪ **Reusability**
  - ▪ **Debugging ease**
  - ▪ **Suitability for larger systems**

Prompt:

Non-Modular:

Write a code for printing a Fibonacci series up to n terms without using a function.

Modular:

Write a code for printing the Fibonacci series up to n terms using a function and include meaningful comments.

Code:

Procedural:

Assign 1.py

Assign 1.py > ...

```python
# write a code for printing a fibonacci series up to n terms without using a function
n = int(input("Enter the number of terms in Fibonacci series: "))
a, b = 0, 1
print("Fibonacci series up to", n, "terms:")
for i in range(n):
    print(a, end=' ')
    a, b = b, a + b

```

Modular:

```python
# Write a code for printing the fibonacci series up to n terms using a function and include meaningful comments
def fibonacci_series(n):
    """
    This function prints the Fibonacci series up to n terms.

    Parameters:
    n (int): The number of terms in the Fibonacci series to be printed.
    """

    a, b = 0, 1  # Initialize the first two terms of the Fibonacci series
    print("Fibonacci series up to", n, "terms:")
    for _ in range(n):
        print(a, end=' ')  # Print the current term
        a, b = b, a + b  # Update to the next two terms

# Get the number of terms from the user
num_terms = int(input("Enter the number of terms in Fibonacci series: "))
fibonacci_series(num_terms)  # Call the function to print the series
```

Output:

```
/As1.py"
Enter the number of terms in Fibonacci series: 6
Fibonacci series up to 6 terms:
0 1 1 2 3 5 Enter the number of terms in Fibonacci series: 
```

Justification:

Procedural code is simple but becomes harder to maintain as system size grows. Modular code improves clarity by isolating logic, making intent easier to understand. Modular code is reusable, which can be later integrated into pipelines and projects. Also, debugging is easier in modular code.

Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches for Fibonacci Series)

Prompt GitHub Copilot to generate:

- **An iterative Fibonacci implementation**
- **A recursive Fibonacci implementation**

Prompt:

Write a code for printing a Fibonacci series up to n terms without using a function.

Write a code for printing the Fibonacci series up to n terms using recursion.

Code:

```python
# write a code for printing a fibonacci series up to n terms without using a function
n = int(input("Enter the number of terms in Fibonacci series: "))
a, b = 0, 1
print("Fibonacci series up to", n, "terms:")
for i in range(n):
    print(a, end=' ')
    a, b = b, a + b

# Write a code for printing the fibonacci series up to n terms using recursion
def fibonacci_recursive(n):
    """
    This function returns the nth term of the Fibonacci series using recursion.

    Parameters:
    n (int): The position of the term in the Fibonacci series.

    Returns:
    int: The nth term of the Fibonacci series.
    """
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)

# Get the number of terms from the user
num_terms = int(input("Enter the number of terms in Fibonacci series: "))
print("Fibonacci series up to", num_terms, "terms:")
for i in range(num_terms):
    print(fibonacci_recursive(i), end=' ')
```

Output:

```
/As1.py"
Enter the number of terms in Fibonacci series: 8
Fibonacci series up to 8 terms:
0 1 1 2 3 5 8 13 Enter the number of terms in Fibonacci series: 12
Fibonacci series up to 12 terms:
0 1 1 2 3 5 8 13 21 34 55 89 Enter the number of terms in Fibonacci series:
```

Justification:

Since the iterative approach is fast and uses a fixed amount of memory, it works well for large ***n*** value. The recursive method is mathematically neat, but it takes exponential time unless you use memorization. Recursion should be avoided for large inputs due to the risk of stack overflow and slow performance. Iteration is the practical choice for scalable systems and high-performance needs.