ASSIGNMENT 1
AI ASSISTED CODING

2303A52490
Adithya

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

- **Scenario**
  You are asked to write a quick numerical sequence generator for a learning platform prototype.

- **Task Description**
  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

1.py > ...

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

a = 0
b = 1
count = 0
if n <= 0:
    print("Please enter a positive integer")
elif n == 1:
    print(a)
else:
    print(a, end=" ")
    while count < n - 1:
        print(b, end=" ")
        a, b = b, a + b
        count += 1
```

Problems Output Debug Console Terminal

```
PS C:\Users\x\Downloads> & C:/Users/x/AppData/Local/Program
Enter the number of terms: 8
0 1 1 2 3 5 8 13
PS C:\Users\x\Downloads>
```

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

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

- **Task Description**
  - 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"*

IMPROVED CODE

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

if n <= 0:
    print("Please enter a positive integer")
else:
    a, b = 0, 1
    for i in range(n):
        if i == n - 1:
            print(a, end="")
        else:
            print(a, end=" ")
        a, b = b, a + b
```

roblems Output Debug Console Terminal

```
PS C:\Users\x\Downloads> & C:/Users/x/AppData/Local/Programs/Python/Python314/python.exe c:/Users/x/Downloads/1.py
Enter the number of terms: 10
0 1 1 2 3 5 8 13 21 34
PS C:\Users\x\Downloads>
```

EXPLANATION

This version keeps everything in the main code (no functions) but:

- Removes the extra count variable
- Uses a for loop directly over the number of terms
- Avoids a separate n == 1 branch
- Uses a single print location for all terms

What was inefficient / less clean in the original

- Redundant loop counter

  - count was tracking iterations, but Python already gives you loop counters with range .
  - Managing count manually (count = 0 , count += 1) adds noise.
- Unnecessary branching for n == 1

  - Special-case branch elif n == 1: printed a alone.
  - Then for larger n, the code printed a once before the loop and b inside the loop.
  - Logic is split across multiple places, which reduces readability.
- Two separate print patterns

  - First term (a) is printed before the loop.
  - Remaining terms (b) are printed inside the loop.
  - This makes it harder to reason about "what exactly is printed each iteration".
- Less direct mapping to "print n Fibonacci numbers"

  - The condition while count < n - 1 is slightly less intuitive than "repeat n times".

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

❖ **Scenario**

The Fibonacci logic is now required in multiple modules of an application.

❖ **Task Description**

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)

❖ **Expected Output**

➢ Correct function-based Fibonacci implementation
➢ Screenshots documenting Copilot's function generation

Sample test cases with outputs

```python
def fibonacci(n):
    """
    Return a list containing the first n Fibonacci numbers.

    n: number of terms to generate (non-negative integer)
    """
    # Handle non-positive n by returning an empty sequence
    if n <= 0:
        return []

    # Start the sequence with the first Fibonacci number
    sequence = [0]

    # If only one term is requested, return [0]
    if n == 1:
        return sequence

    # Add the second Fibonacci number
    sequence.append(1)

    # Generate remaining terms from index 2 to n-1
    for _ in range(2, n):
        # Next term is sum of the last two terms
        next_value = sequence[-1] + sequence[-2]
        sequence.append(next_value)

    return sequence


def main():
    """Read n from the user and print the Fibonacci sequence up to n terms."""
    try:
        n = int(input("Enter the number of terms: "))
    except ValueError:
        print("Invalid input. Please enter an integer.")
        return

    fib_seq = fibonacci(n)

    if not fib_seq:
        print("No terms to display (n should be > 0).")
    else:
        print("Fibonacci sequence:")
        # Join all numbers with spaces for nice output
        print(" ".join(str(x) for x in fib_seq))


if __name__ == "__main__":
    main()
```

Test: n = 0

- Call:

Python
```python
fibonacci(0)
```

- Output (return value):

Python
```python
[]
```

- Running the script and entering `0`:

PlainText
```
Enter the number of terms: 0
No terms to display (n should be > 0).
```

Test: n = 1

Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code

- **Scenario**
  You are participating in a code review session.

- **Task Description**
  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

Comparison Overview

- Procedural version (Task 1) : all logic in main , no functions
- Modular version (Task 3) : fibonacci(n) function + main() wrapper

Aspect	Procedural (no functions)	Modular (with functions)
Code clarity	Short and linear, but mixes input, logic, and printing in one block	Separates "compute Fibonacci" from "interact with user"; intention of each part is clearer
Reusability	Hard to reuse: logic is tied to `input()` and `print()` in the same place	Easy to reuse: `fibonacci(n)` can be imported and called from any module, including tests and other components
Debugging ease	Must rerun whole script or add ad-hoc prints inside main flow	Can test `fibonacci(n)` in isolation (REPL, unit tests); easier to pinpoint bugs in the pure computation
Suitability for larger systems	Poor: scales badly as more features are added; main becomes a "god block"	Good: Fibonacci logic lives in a small, focused function that fits into layered or modular architectures
Testability	Awkward to unit-test; you test via I/O behavior or refactor first	Straightforward: write tests against `fibonacci(n)` return values without dealing with console I/O
Extensibility	Adding options (e.g., different starting values, returning vs printing) complicates main	You can extend by adding parameters or new functions while keeping existing callers mostly unchanged
Abstraction level	Low: works at the "how" level (loop and updates) only	Higher: exposes "what" (get sequence of n terms) as an interface; hides details of how it's computed
Readability for new developers	Understandable for small scripts, but logic is packed together	More readable in a team: clear entry points, docstring, and separation of concerns

Short Analytical Report

- Code clarity

  - The procedural version is fine for very small, one-off scripts: everything is visible in one place.
  - However, it mixes three concerns: reading input, computing Fibonacci, and printing results. As soon as you add more features (validation, logging, different output formats), this block gets harder to read.
  - The modular version clearly separates concerns:
    - fibonacci(n) is about the sequence calculation.
    - main() is about user interaction and display.
  - This makes it easier for a reviewer to answer "where is the Fibonacci logic?" and "where is the I/O?".
- Reusability

  - In the procedural version, the only way to "reuse" the logic is to copy and paste the loop, or to call the script as a separate process. Both are fragile.
  - The modular version provides a reusable unit: fibonacci(n) can be imported by other modules (from fib import fibonacci) and used in GUI apps, APIs, other scripts, or tests.
- Debugging ease

  - Debugging the procedural code often means sprinkling extra prints into the main block or stepping through the entire input/print flow in a debugger.
  - With the function-based design, you can:
    - Call fibonacci(5) directly in a REPL and inspect the list.
    - Add unit tests for edge cases (n = 0 , n = 1 , large n) without touching I/O.
  - When a bug appears, you can quickly determine whether it lives in the pure logic (fibonacci) or in the user interaction (main).
- Suitability for larger systems

  - Larger applications benefit from modularity: clear APIs, small functions, and layers (UI, business logic, data).
  - The procedural version doesn't scale well because as more requirements are added, the main block grows and becomes tightly coupled to everything else.
  - The modular version maps naturally into a larger architecture:
    - fibonacci(n) is part of a "domain" or "core" module.
      - Different front-ends (CLI, web, GUI) can all reuse the same core function.

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

- **Scenario**

Your mentor wants to assess AI's understanding of different algorithmic paradigms.

- **Task Description**
  Prompt GitHub Copilot to generate:
    An iterative Fibonacci implementation
    A recursive Fibonacci implementation

ITERATIVE

```
def fib_iterative(n):
    if n < 0:
        raise ValueError("n must be non-negative")

    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a
```

RECURSIVE

```
def fib_recursive(n):
    if n < 0:
        raise ValueError("n must be non-negative")

    if n == 0:
        return 0
    if n == 1:
        return 1

    return fib_recursive(n - 1) + fib_recursive(n - 2)
```

Execution flow

- Iterative

  - Initialize a = 0 , b = 1 .
  - Loop n times:
    - Set (a, b) to the next Fibonacci pair (b, a + b) .
  - After the loop, a holds fib(n) .
  - Control flow is straightforward: one loop, no call stack growth.
- Recursive

  - For each call fib_recursive(n) :
    - If n is 0 or 1, return the base value directly.
    - Otherwise, call itself twice:
      - fib_recursive(n - 1)

- fib_recursive(n - 2)
  - Sum the two results.
- The call tree branches until all paths hit n == 0 or n == 1 .

Complexity and performance comparison

- Time complexity

  - Iterative:
    - Each loop iteration does O(1) work.
    - Loop runs n times.
    - Time complexity: O(n) .
  - Naive recursive:
    - Each call branches into two more calls (except at base cases).
    - Number of calls grows roughly like 2^n .
    - Time complexity: O(2^n) (exponential).
- Space complexity

  - Iterative:
    - Uses a constant number of variables (a , b , loop index).
    - Space complexity: O(1) auxiliary space.
  - Naive recursive:
    - Each call waits on the result of two child calls.
    - Maximum recursion depth is n (for the longest path down to 0/1).
    - Space complexity: O(n) due to call stack.
- Performance for large n

  - Iterative:
    - Scales well up to very large n (e.g., hundreds of thousands) until integers or time become the bottleneck.
    - Work grows linearly; practical for most uses.
  - Naive recursive:
    - Becomes unusable quickly:
      - For n ~ 30–40 , it is already noticeably slow.
      - For n > 40–45 , runtime explodes because of exponential growth in calls.

when recursion should be avoided

- You need to handle large n .
- You are in a performance-sensitive path (e.g., called many times in a loop or server endpoint).
- The recursive formulation introduces overlapping subproblems (like Fibonacci), causing repeated work and exponential behavior.
- Your runtime has a strict recursion depth limit that the algorithm might hit.