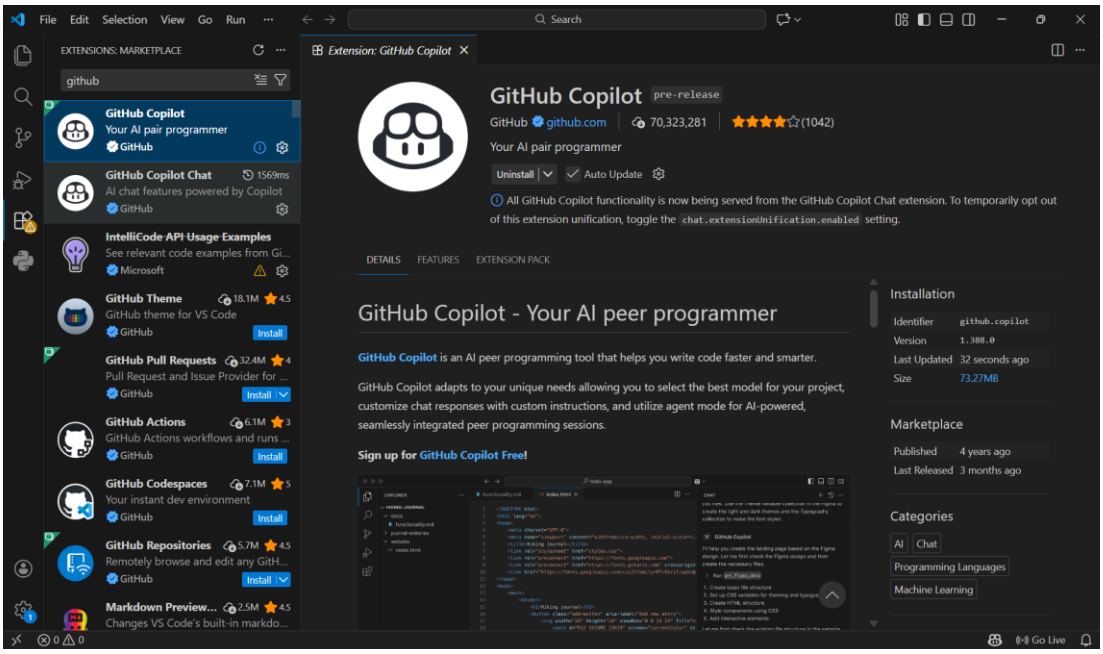
AI ASSISTED CODING ASS-1.3

2303A51642

B-28

Task 0: Install and configure GitHub Copilot in VS Code.



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

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

a = 0
b = 1

if n <= 0:
    print("Please enter a positive number")
elif n == 1:
    print(a)
else:
    print(a, end=" ")
    print(b, end=" ")

    for i in range(2, n):
        c = a + b
        print(c, end=" ")
        a = b
        b = c
```

Sample Input:

Enter the number of terms: 7

Sample Output:

0 1 1 2 3 5 8

Explanation:

- The program starts by taking user input for n
- Two variables a and b store the first two Fibonacci numbers
- A loop iteratively calculates the next term by adding the previous two
- All logic is written inline, without defining any functions

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

Optimized & Cleaned-Up Code:

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

prev = 0
curr = 1

for i in range(n):
    print(prev, end=" ")
    prev, curr = curr, prev + curr
```

Sample Input:

Enter the number of terms: 7

Sample Output:

0 1 1 2 3 5 8

Explanation:

- The original code used extra conditional checks and separate print statements, which increased complexity without improving functionality.
- A temporary variable was used unnecessarily to store intermediate Fibonacci values.
- The optimized version replaces multiple conditions with a single loop that handles all cases cleanly.

- Tuple assignment reduces redundancy, making the code shorter, more readable, and easier to maintain while keeping the same O(n) time complexity.

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

Function-Based Python Program (AI-Assisted):

```python
def fibonacci(n):
    sequence = []
    a, b = 0, 1

    for i in range(n):
        sequence.append(a)
        a, b = b, a + b

    return sequence


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

if n <= 0:
    print("Please enter a positive number")
else:
    result = fibonacci(n)
    print(result)
```

Input:

Enter the number of terms: 6

Output:

[0, 1, 1, 2, 3, 5]

Explanation:

- Logic is modular and reusable
- Code is easier to maintain and extend
- Suitable for use across multiple application modules

- Demonstrates effective AI-assisted modular design

Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code

Procedural vs Modular Fibonacci Code

Aspect	Without Functions (Task 1)	With Functions (Task 3)
Code Clarity	Logic is written inline, making the program harder to scan as it grows	Clear separation between logic and execution improves readability
Reusability	Code cannot be reused without copying and modifying it	Function can be reused across multiple files or modules
Debugging Ease	Bugs are harder to isolate since all logic is in one block	Errors can be traced and fixed within the function easily
Suitability for Larger Systems	Not suitable, as logic duplication increases maintenance effort	Well-suited for larger systems due to modular and structured design

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

1. Iterative Fibonacci Implementation:

```python
def fibonacci_iterative(n):
    a, b = 0, 1
    result = []

    for i in range(n):
        result.append(a)
        a, b = b, a + b

    return result


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

2. Recursive Fibonacci Implementation:

```python
def fibonacci_recursive(n):
    if n <= 1:
        return n
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)


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

result = []
for i in range(n):
    result.append(fibonacci_recursive(i))

print(result)
```

Execution Flow Explanation:

Iterative Approach

- Uses a loop to calculate Fibonacci numbers step by step.
- Each new value is derived from the previous two values.
- Values are stored sequentially in a list.
- Execution progresses linearly without repeated calculations.

Recursive Approach

- Each Fibonacci number is computed by calling the function for smaller values.
- The same subproblems are solved multiple times.
- Execution involves a deep call stack as n increases.
- Results are built indirectly through function return values.

Comparative Analysis:

Aspect	**Iterative Approach**	**Recursive Approach**
Time Complexity	O(n)	$O(2^n)$
Space Complexity	O(n) for result list	O(n) call stack + repeated calls
Performance for Large n	Very efficient and fast	Extremely slow
Readability	Slightly longer but clear	Short and mathematically expressive
Practical Usage	Preferred in real systems	Mostly educational

When Recursion Should Be Avoided:

- When n is large and performance matters
- When stack overflow is a risk
- In production systems where efficiency and predictability are required
- When the same subproblems are repeatedly recomputed