

M.Anu Chandra

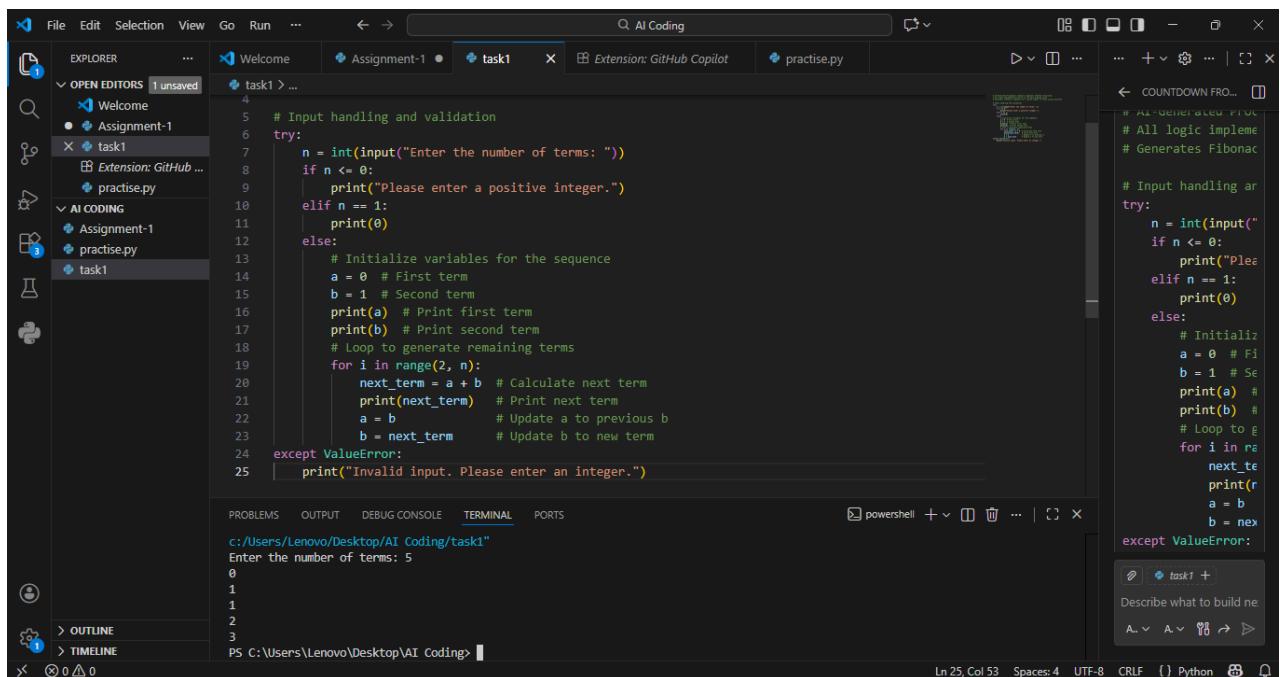
2303A54070 B-48

Lab1

AssignmentNumber1.3

TASK:1

PROMPT: Write a Python program that asks the user for a number and prints the Fibonacci sequence up to that number, without using any functions



The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows "task1 > ...", "Welcome", "Assignment-1", and "practise.py".
- Code Editor:** Displays a Python script named "task1". The code is as follows:

```
# Input handling and validation
try:
    n = int(input("Enter the number of terms: "))
    if n <= 0:
        print("Please enter a positive integer.")
    elif n == 1:
        print(0)
    else:
        # Initialize variables for the sequence
        a = 0 # First term
        b = 1 # Second term
        print(a) # Print first term
        print(b) # Print second term
        # Loop to generate remaining terms
        for i in range(2, n):
            next_term = a + b # Calculate next term
            print(next_term) # Print next term
            a = b # Update a to previous b
            b = next_term # Update b to new term
except ValueError:
    print("Invalid input. Please enter an integer.")
```

- Terminal:** Shows the command "powershell" and the output of the script execution.
- Status Bar:** Shows "Ln 25, Col 53" and other file-related information.

EXPLANATION: Using GitHub Copilot, I was able to generate a Python program that prints the Fibonacci sequence.

The program follows all the given rules and works correctly for any valid input.

TASK:2

PROMPT: AI Code Optimization & Cleanup (Improving Efficiency)

The screenshot shows the Visual Studio Code interface with the "AI Coding" extension open. The left sidebar shows the Explorer with files like "Welcome", "Assignment-1", "task1", and "task2". The main editor shows a Python script named "task2.py" with the following code:

```
# AI-Optimized Procedural Fibonacci Sequence (Without Functions)
# Improved efficiency with tuple unpacking and minimal operations
# Generates Fibonacci sequence for a given number of terms using iteration

try:
    n = int(input("Enter the number of terms: "))
    if n <= 0:
        print("Please enter a positive integer.")
    elif n == 1:
        print(0)
    else:
        # Initialize first two terms
        a, b = 0, 1
        print(a)
        print(b)
        # Generate remaining terms with efficient swapping
        for _ in range(2, n):
            a, b = b, a + b
            print(b)
except ValueError:
    print("Invalid input. Please enter an integer.")
```

The right panel shows the "CHAT" tab with a message from GitHub Copilot suggesting improvements. The message reads:

```
# Improved efficiency with tuple unpacking and
# Generates Fibonacci sequence for a given num

# Input handling and validation
try:
    n = int(input("Enter the number of terms: "))
    if n <= 0:
        print("Please enter a positive integer")
    elif n == 1:
        print(0)
    else:
        # Initialize first two terms
        a, b = 0, 1
        print(a)
        print(b)
        # Generate remaining terms with effici
for _ in range(2, n):
    a, b = b, a + b
    print(b)
except ValueError:
    print("Invalid input. Please enter an inte")
```

The bottom status bar indicates the code is in Python mode.

EXPLANATION: Using GitHub Copilot, the original Fibonacci code was improved by:

- Removing extra variables
- Simplifying logic
- Making the code more readable

This task shows how **AI can help not only write code but also improve and clean existing code**, which is very useful in real-world development.

TASK:3

PROMPT: Write a Python program that uses a function to print the Fibonacci sequence and asks the user for a number

The screenshot shows the Visual Studio Code interface with the following details:

- EXPLORER**: Shows open editors: Welcome, Assignment-1 (task1, task2), task 3.py (Extension: GitHub, practise.py).
- CODE EDITOR**: The active file is task 3.py, containing a modular Fibonacci sequence generator using functions. The code includes comments explaining the purpose of each function and a try-except block for input validation.
- TERMINAL**: Shows the command `nPython312>python.exe "c:/Users/Lenovo/Desktop/AI Coding/task 3.py"` and its output: "Enter the number of terms: 5", followed by the sequence 0, 1, 1, 2, 3.
- CHAT**: A sidebar with AI-generated completion suggestions for the command line, including options like "Run pwsh command?", "Allow", and "Skip".

The screenshot shows the Visual Studio Code interface with the following details:

- EXPLORER**: Shows open editors: task 3.py (Extension: GitHub, practise.py).
- CODE EDITOR**: The active file is task 3.py, containing a more complex modular Fibonacci program. It includes functions for generating the sequence, displaying it, and a main orchestrator function. It also handles the `__name__ == "__main__"` condition.
- TERMINAL**: Shows the command `nPython312>python.exe "c:/Users/Lenovo/Desktop/AI Coding/task 3.py"` and its output: "Enter the number of terms: 5", followed by the sequence 0, 1, 1, 2, 3.
- CHAT**: A sidebar with AI-generated completion suggestions for the command line, including options like "Run pwsh command?", "Allow", and "Skip".

EXPLANATION: Using GitHub Copilot, a function-based Fibonacci program was generated. This task shows how AI helps in writing **modular, reusable, and well-commented code**, which is important for large applications.

TASK:4

PROMPT:Print Fibonacci code with functions and without functions

```

File Edit Selection View Go Run ... ← → Q AI Coding OPEN EDITORS 1 unsaved EXPLORER task 4.py > ...
task 4.py > ...
1 # Modular Fibonacci sequence generator using functions
2 # Improved design with separation of concerns for better maintainability
3
4 def get_number_of_terms():
5     """Prompts user for number of terms and validates input."""
6     try:
7         n = int(input("Enter the number of terms: "))
8         if n <= 0:
9             raise ValueError("Please enter a positive integer.")
10        return n
11    except ValueError as e:
12        print(f"Invalid input: {e}")
13        return None
14
15 def generate_fibonacci(n):
16     """Generates the Fibonacci sequence up to n terms."""
17     if n == 1:
18         return [0]
19     sequence = [0, 1]
20     a, b = 0, 1
21     for _ in range(2, n):
22         a, b = b, a + b
23         sequence.append(b)
24
25 return sequence
26
27 def display_sequence(sequence):
28     """Displays the Fibonacci sequence."""
29     for num in sequence:
30         print(num)
31
32 def main():
33     """Main function to orchestrate the program."""
34     n = get_number_of_terms()
35     if n is not None:
36         seq = generate_fibonacci(n)
37         display_sequence(seq)
38
39 if __name__ == "__main__":
40     main()

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

hon.exe "C:/Users/Lenovo/Desktop/AI Coding/task 4.py"
Enter the number of terms: 5
0
1
1
2
3

PS C:\Users\Lenovo\Desktop\AI Coding> |

Ln 39, Col 11 Spaces: 4 UTF-8 CRLF {} Python Python 3.12 (64-bit) Q

```

File Edit Selection View Go Run ... ← → Q AI Coding OPEN EDITORS 1 unsaved EXPLORER task 4.py > ...
task 4.py > ...
15 def generate_fibonacci(n):
16     sequence = [0, 1]
17     a, b = 0, 1
18     for _ in range(2, n):
19         a, b = b, a + b
20         sequence.append(b)
21
22 return sequence
23
24 def display_sequence(sequence):
25     """Displays the Fibonacci sequence."""
26     for num in sequence:
27         print(num)
28
29 def main():
30     """Main function to orchestrate the program."""
31     n = get_number_of_terms()
32     if n is not None:
33         seq = generate_fibonacci(n)
34         display_sequence(seq)
35
36 if __name__ == "__main__":
37     main()

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

hon.exe "C:/Users/Lenovo/Desktop/AI Coding/task 4.py"
Enter the number of terms: 5
0
1
1
2
3

PS C:\Users\Lenovo\Desktop\AI Coding> |

Ln 39, Col 11 Spaces: 4 UTF-8 CRLF {} Python Python 3.12 (64-bit) Q

- EXPLANATION:** **Procedural Fibonacci code** is good for learning basics and small programs.
- Modular Fibonacci code** is better for real projects, teamwork, and large systems.
- Using functions improves **readability, reusability, and maintainability**.

This comparison shows why **modular programming is preferred in professional software development**.

TASK:5

PROMPT: Generate an iterative Fibonacci implementation ,recursive Fibonacci implementation compare their time complexity, performance for large n

```

File Edit Selection View Go Run ... < > Q: AI Coding
EXPLORER OPEN EDITORS 2 unsaved task2 task3.py task4.py task5.py 2 practise.py ...
task5.py > ...
1 import time
2
3 def fibonacci_recursive(n):
4     """
5         Recursive approach to calculate the nth Fibonacci number.
6         Time complexity: O(2^n) - exponential due to redundant calculations
7         Space complexity: O(n) - recursion stack depth
8     """
9
10    if n <= 1:
11        return n
12    return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
13
14 def fibonacci_iterative(n):
15     """
16         Iterative approach to calculate the nth Fibonacci number.
17         Time complexity: O(n) - linear time
18         Space complexity: O(1) - constant space (excluding output storage)
19     """
20
21    if n <= 1:
22        return n
23    a, b = 0, 1
24    for _ in range(2, n+1):
25        a, b = b, a + b
26
27 PS C:\Users\Lenovo\Desktop\AI Coding & "C:/Users/Lenovo/Desktop/AI Coding/.venv/Scripts/python.exe" task5.py
28 Calculating Fibonacci number for n = 30
29
30 Recursive: 832040, Time: 1.163015 seconds
31 Iterative: 832040, Time: 0.000000 seconds
32 Memoized: 832040, Time: 0.000000 seconds

```

CHAT

<- COMPARING ITERATIVE AND RECURSIVE... 1.163015 seconds

- **Iterative:** 832040, took 0.000000 seconds
- **Memoized:** 832040, took 0.000000 seconds

This clearly demonstrates why the iterative and memoized approaches are much more efficient than the basic recursive implementation for larger values of n. The recursive approach becomes exponentially slower as n increases due to redundant calculations.

To run the script yourself in the future:

1. Open a terminal in your workspace
2. Use the command: &

```
"C:/Users/Lenovo/Desktop/AI Coding/.venv/Scripts/python.exe" task5.py
```

You can also modify the value of `n` in the script to test with different inputs and see how the performance scales.

```

File Edit Selection View Go Run ... < > Q: AI Coding
EXPLORER OPEN EDITORS 2 unsaved task2 task3.py task4.py task5.py 2 practise.py ...
task5.py > ...
13 def fibonacci_iterative(n):
14     for _ in range(2, n+1):
15         a, b = b, a + b
16     return b
17
18 def fibonacci_memoized(n, memo=None):
19     """
20         Memoized recursive approach to calculate the nth Fibonacci number.
21         Time complexity: O(n) - each subproblem solved once
22         Space complexity: O(n) - memo dictionary and recursion stack
23     """
24
25     if memo is None:
26         memo = {}
27     if n in memo:
28         return memo[n]
29     if n <= 1:
30         return n
31     memo[n] = fibonacci_memoized(n-1, memo) + fibonacci_memoized(n-2, memo)
32     return memo[n]
33
34 if __name__ == "__main__":
35     n = 30 # Test with a moderate n to see performance differences
36
37 PS C:\Users\Lenovo\Desktop\AI Coding & "C:/Users/Lenovo/Desktop/AI Coding/.venv/Scripts/python.exe" task5.py
38 Calculating Fibonacci number for n = 30
39
40 Recursive: 832040, Time: 1.163015 seconds
41 Iterative: 832040, Time: 0.000000 seconds
42 Memoized: 832040, Time: 0.000000 seconds

```

CHAT

<- COMPARING ITERATIVE AND RECURSIVE... 1.163015 seconds

- **Iterative:** 832040, took 0.000000 seconds
- **Memoized:** 832040, took 0.000000 seconds

This clearly demonstrates why the iterative and memoized approaches are much more efficient than the basic recursive implementation for larger values of n. The recursive approach becomes exponentially slower as n increases due to redundant calculations.

To run the script yourself in the future:

1. Open a terminal in your workspace
2. Use the command: &

```
"C:/Users/Lenovo/Desktop/AI Coding/.venv/Scripts/python.exe" task5.py
```

You can also modify the value of `n` in the script to test with different inputs and see how the performance scales.

```

task5.py > ...
41  if __name__ == "__main__":
42      n = 30 # Test with a moderate n to see performance differences
43
44      print(f"Calculating Fibonacci number for n = {n}")
45      print()
46
47      # Recursive
48      start = time.time()
49      result_recursive = fibonacci_recursive(n)
50      end = time.time()
51      print(f"Recursive: {result_recursive}, Time: {end - start:.6f} seconds")
52
53      # Iterative
54      start = time.time()
55      result_iterative = fibonacci_iterative(n)
56      end = time.time()
57      print(f"Iterative: {result_iterative}, Time: {end - start:.6f} seconds")
58
59      # Memoized
60      start = time.time()
61      result_memoized = fibonacci_memoized(n)
62      end = time.time()
63      print(f"Memoized: {result_memoized}, Time: {end - start:.6f} seconds")

```

PS C:\Users\Lenovo\Desktop\AI Coding> & "C:/Users/Lenovo/Desktop/AI Coding/.venv /Scripts/python.exe" task5.py
Calculating Fibonacci number for n = 30
Recursive: 832040, Time: 1.163015 seconds
Iterative: 832040, Time: 0.000000 seconds
Memoized: 832040, Time: 0.000000 seconds

```

task5.py > ...
52
53      # Iterative
54      start = time.time()
55      result_iterative = fibonacci_iterative(n)
56      end = time.time()
57      print(f"Iterative: {result_iterative}, Time: {end - start:.6f} seconds")
58
59      # Memoized
60      start = time.time()
61      result_memoized = fibonacci_memoized(n)
62      end = time.time()
63      print(f"Memoized: {result_memoized}, Time: {end - start:.6f} seconds")
64
65      print()
66      print("Comparison:")
67      print("- Recursive: Simple but inefficient for large n due to exponential growth in time and space")
68      print("- Iterative: Efficient with linear time and constant space")
69      print("- Memoized: Recursive with caching, efficient but uses more memory")

```

PS C:\Users\Lenovo\Desktop\AI Coding> & "C:/Users/Lenovo/Desktop/AI Coding/.venv /Scripts/python.exe" task5.py
Calculating Fibonacci number for n = 30
Recursive: 832040, Time: 1.163015 seconds
Iterative: 832040, Time: 0.000000 seconds
Memoized: 832040, Time: 0.000000 seconds

EXPLANATION : The iterative Fibonacci approach is efficient, scalable, and suitable for real-world applications. The recursive approach is mainly useful for learning and understanding recursion concepts but is not ideal for performance-critical systems.

