

M.Anu Chandra

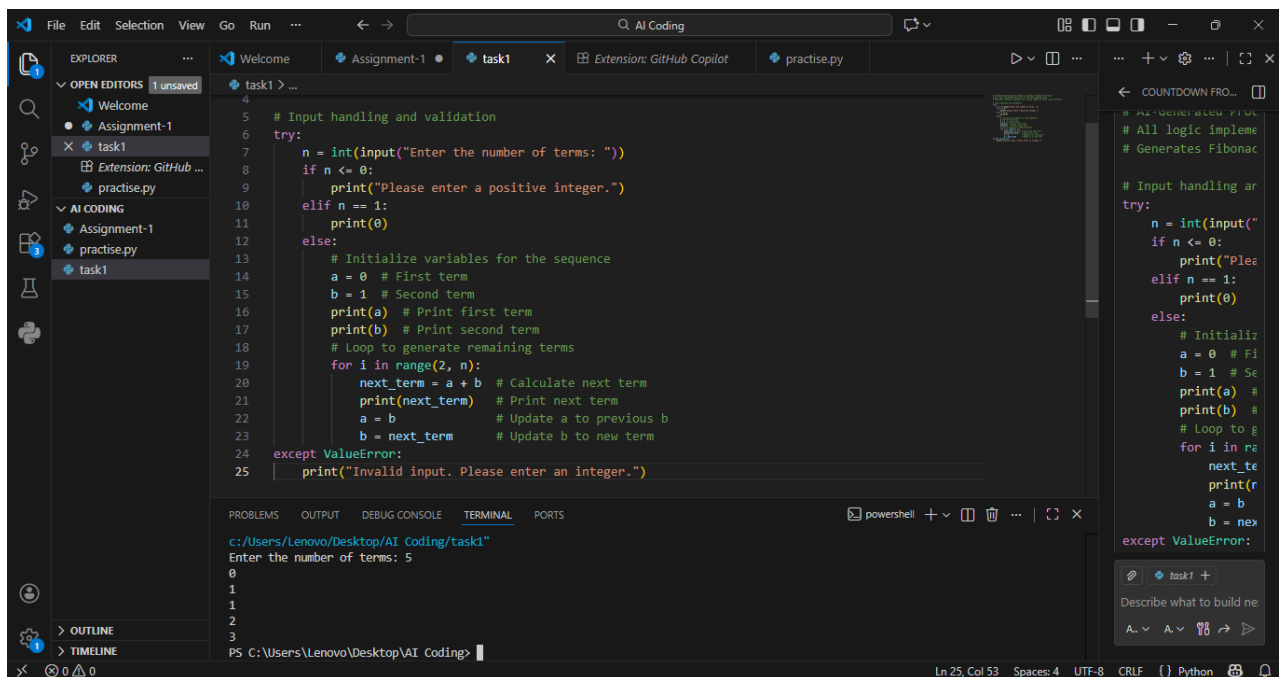
2303A54070 B-48

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



```
4
5 # Input handling and validation
6 try:
7     n = int(input("Enter the number of terms: "))
8     if n <= 0:
9         print("Please enter a positive integer.")
10    elif n == 1:
11        print(0)
12    else:
13        # Initialize variables for the sequence
14        a = 0 # First term
15        b = 1 # Second term
16        print(a) # Print first term
17        print(b) # Print second term
18        # Loop to generate remaining terms
19        for i in range(2, n):
20            next_term = a + b # Calculate next term
21            print(next_term) # Print next term
22            a = b # Update a to previous b
23            b = next_term # Update b to new term
24 except ValueError:
25     print("Invalid input. Please enter an integer.")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

c:/Users/Lenovo/Desktop/AI Coding/task1"

Enter the number of terms: 5

0

1

1

2

3

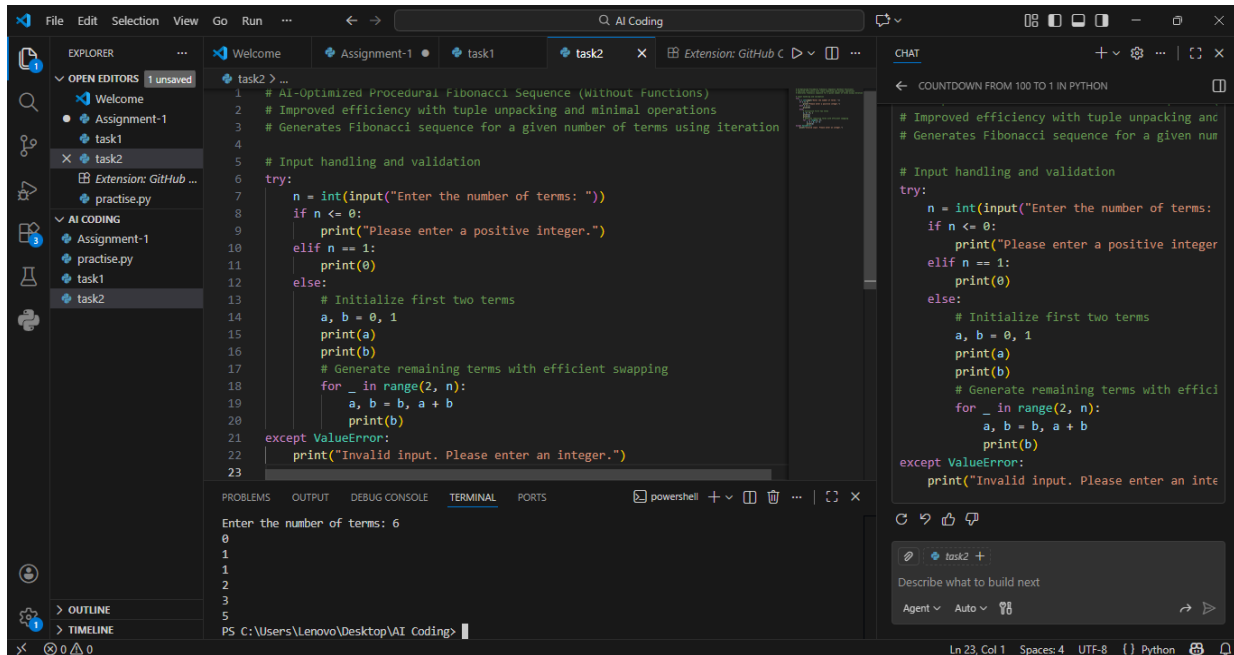
PS C:\Users\Lenovo\Desktop\AI Coding>

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)



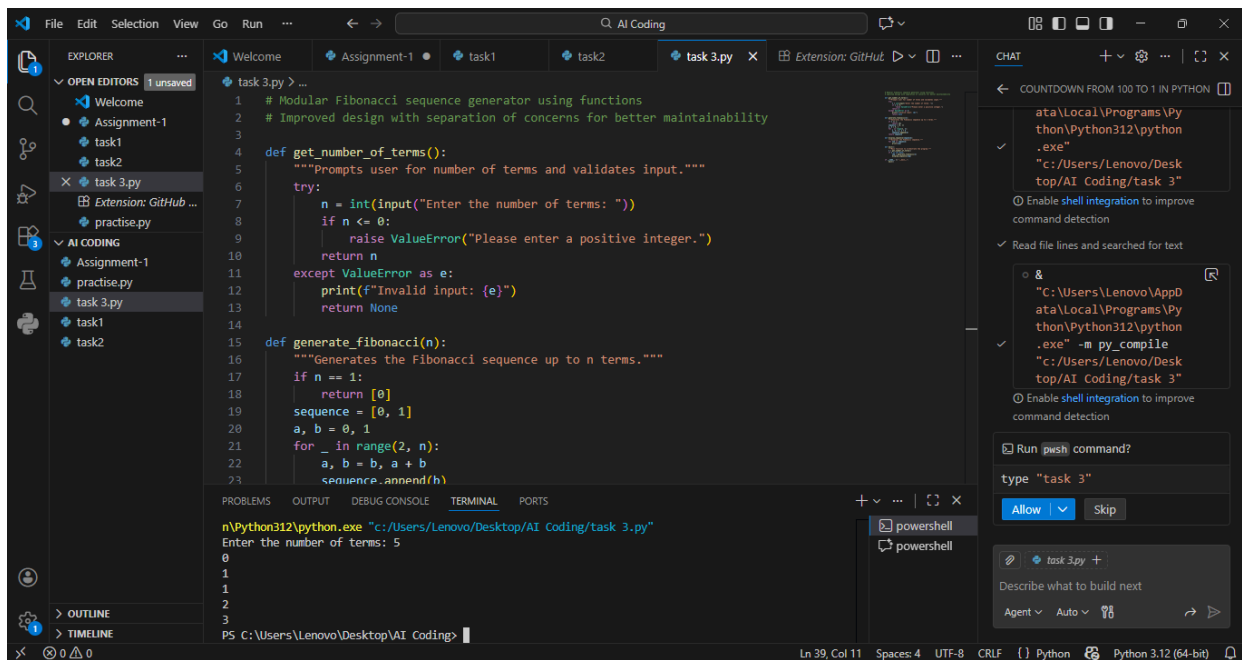
EXPLANATION: Using GitHub Copilot, the original Fibonacci code was improved by:

- 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



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

```
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 if __name__ == "__main__":
26     n = get_number_of_terms()
27     if n is not None:
28         seq = generate_fibonacci(n)
29         display_sequence(seq)
30
31 def display_sequence(sequence):
32     """Displays the Fibonacci sequence."""
33     for num in sequence:
34         print(num)
35
36 if __name__ == "__main__":
37     main()
38
39 if __name__ == "__main__":
40     main()
```

Terminal output:

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

```
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     return sequence
22
23 def display_sequence(sequence):
24     """Displays the Fibonacci sequence."""
25     for num in sequence:
26         print(num)
27
28 def main():
29     """Main function to orchestrate the program."""
30     n = get_number_of_terms()
31     if n is not None:
32         seq = generate_fibonacci(n)
33         display_sequence(seq)
34
35 if __name__ == "__main__":
36     main()
37
38 if __name__ == "__main__":
39     main()
```

Terminal output:

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

- 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

The screenshot shows a VS Code editor with a file explorer on the left containing files like 'task1', 'task2', 'task3.py', 'task4.py', 'task5.py', and 'practise.py'. The main editor displays 'task5.py' with the following code:

```
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     if n <= 1:
10         return n
11     return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
12
13 def fibonacci_iterative(n):
14     """
15     Iterative approach to calculate the nth Fibonacci number.
16     Time complexity: O(n) - linear time
17     Space complexity: O(1) - constant space (excluding output storage)
18     """
19     if n <= 1:
20         return n
21     a, b = 0, 1
22     for _ in range(2, n+1):
23         a, b = b, a + b
```

The terminal at the bottom shows the execution of the script for n=30:

```
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
```

The chat panel on the right provides a comparison of the two approaches:

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.

The screenshot shows a VS Code editor with a file explorer on the left. The main editor displays 'task5.py' with the following code:

```
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     if memo is None:
25         memo = {}
26     if n in memo:
27         return memo[n]
28     if n <= 1:
29         return n
30     memo[n] = fibonacci_memoized(n-1, memo) + fibonacci_memoized(n-2, memo)
31     return memo[n]
32
33 if __name__ == "__main__":
34     n = 30 # Test with a moderate n to see performance differences
35
```

The terminal at the bottom shows the execution of the script for n=30:

```
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
```

The chat panel on the right provides a comparison of the three approaches:

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.

```

1  # task5.py
2  """
3  Comparing Iterative and Recursive Fibonacci
4  """
5
6  def fibonacci_recursive(n):
7      """Recursive Fibonacci"""
8      if n < 0:
9          return 0
10     if n == 0:
11         return 1
12     if n == 1:
13         return 1
14     return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
15
16  def fibonacci_iterative(n):
17      """Iterative Fibonacci"""
18      if n < 0:
19          return 0
20      if n == 0:
21          return 1
22      if n == 1:
23          return 1
24      a, b = 1, 1
25      for _ in range(2, n+1):
26          a, b = b, a+b
27      return b
28
29  def fibonacci_memoized(n):
30      """Memoized Fibonacci"""
31      memo = {}
32      def fib(n):
33          if n in memo:
34              return memo[n]
35          if n < 0:
36              return 0
37          if n == 0:
38              return 1
39          if n == 1:
40              return 1
41          memo[n] = fib(n-1) + fib(n-2)
42          return memo[n]
43      return fib(n)
44
45  if __name__ == "__main__":
46      n = 30 # Test with a moderate n to see performance differences
47
48      print(f"Calculating Fibonacci number for n = {n}")
49      print()
50
51      # Recursive
52      start = time.time()
53      result_recursive = fibonacci_recursive(n)
54      end = time.time()
55      print(f"Recursive: {result_recursive}, Time: {end - start:.6f} seconds")
56
57      # Iterative
58      start = time.time()
59      result_iterative = fibonacci_iterative(n)
60      end = time.time()
61      print(f"Iterative: {result_iterative}, Time: {end - start:.6f} seconds")
62
63      # Memoized
64      start = time.time()
65      result_memoized = fibonacci_memoized(n)
66      end = time.time()
67      print(f"Memoized: {result_memoized}, Time: {end - start:.6f} seconds")

```

Terminal Output:

```

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

```

Chat Panel:

COMPARING ITERATIVE AND RECURSIV...

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.

```

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

```

Terminal Output:

```

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

```

Chat Panel:

COMPARING ITERATIVE AND RECURSIV...

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.

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.

