AI Assisted Coding

LAB 1

Name:K.Pavani

Hall-ticket:2303A51494

B-27

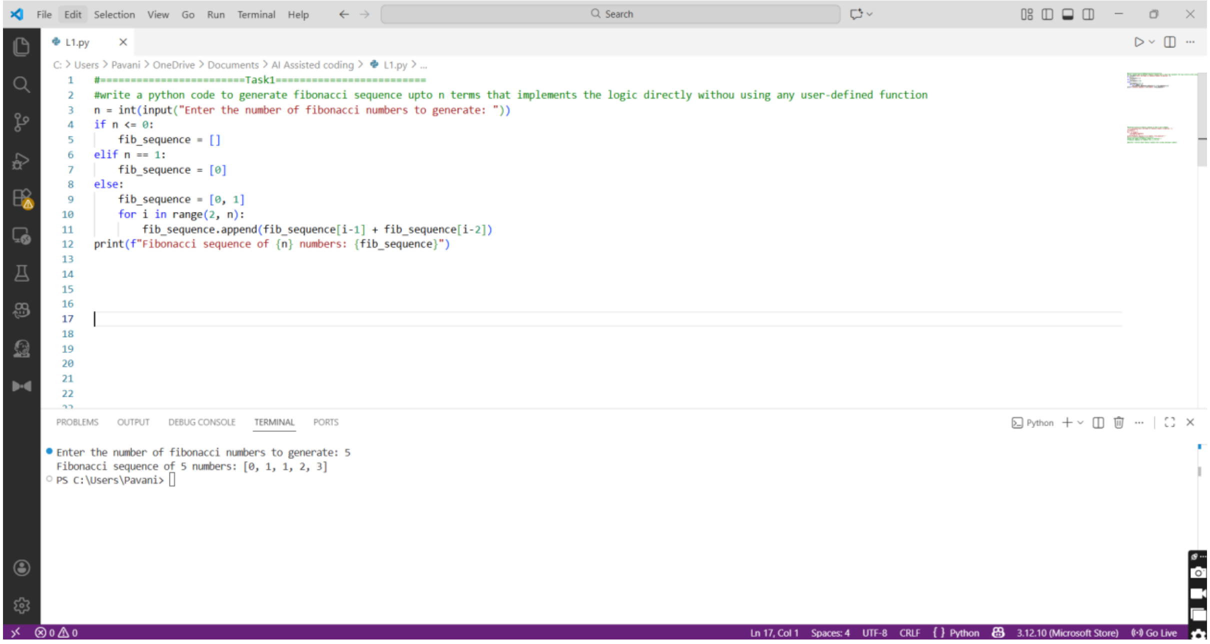
Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence

Without Functions)

PROMPT:

write a python code to generate fibonacci sequence upto n terms that implements the logic directly without using any user-defined function

CODE WITH OUTPUT:



File Edit Selection View Go Run Terminal Help
Search
L1.py
C: > Users > Pavani > OneDrive > Documents > AI Assisted coding > L1.py > ...
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python
Ln 17, Col 1 Spaces: 4 UTF-8 CRLF { } Python 3.12.10 (Microsoft Store) Go Live

```python
#=======================Task1========================
#write a python code to generate fibonacci sequence upto n terms that implements the logic directly withou using any user-defined function
n = int(input("Enter the number of fibonacci numbers to generate: "))
if n <= 0:
    fib_sequence = []
elif n == 1:
    fib_sequence = [0]
else:
    fib_sequence = [0, 1]
    for i in range(2, n):
        fib_sequence.append(fib_sequence[i-1] + fib_sequence[i-2])
print(f"Fibonacci sequence of {n} numbers: {fib_sequence}")
```

```
Enter the number of fibonacci numbers to generate: 5
Fibonacci sequence of 5 numbers: [0, 1, 1, 2, 3]
PS C:\Users\Pavani>
```

EXPLANATION:

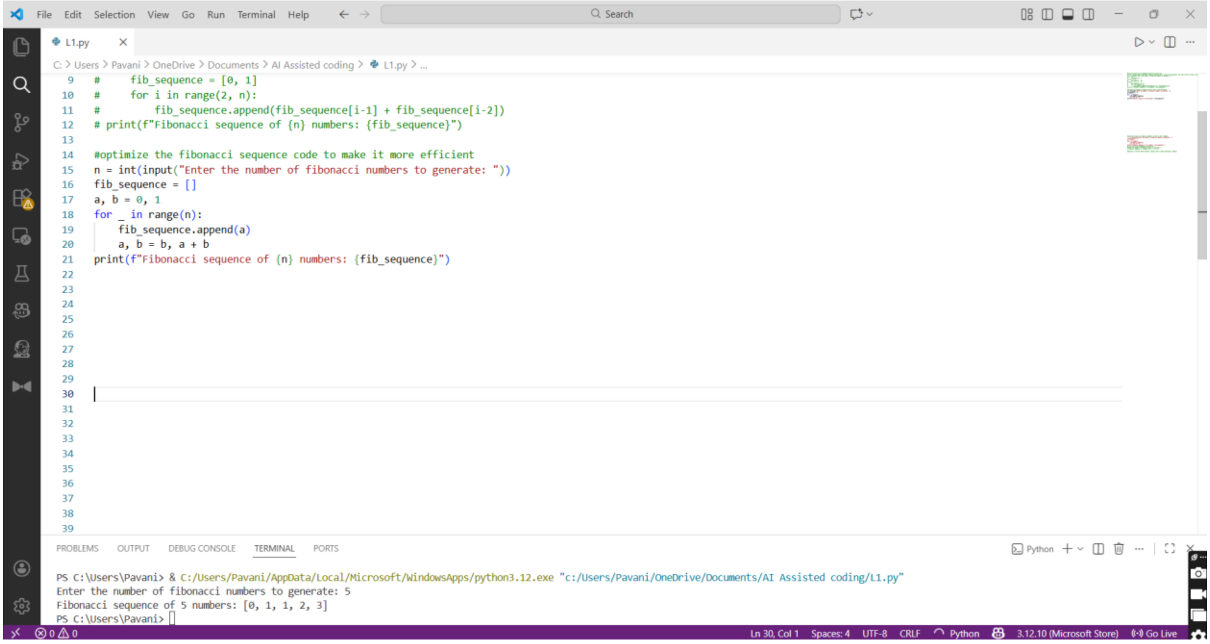
GitHub Copilot generated a Python program to print the Fibonacci sequence by directly writing logic in the main code without using functions. The program takes user input for the number of terms and uses a loop to compute each Fibonacci number. The output correctly displays the Fibonacci sequence, demonstrating procedural coding with AI assistance.

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

PROMPT:

optimize the fibonacci sequence code to make it more efficient

CODE WITH OUTPUT



```python
#     fib_sequence = [0, 1]
#     for i in range(2, n):
#         fib_sequence.append(fib_sequence[i-1] + fib_sequence[i-2])
# print(f"Fibonacci sequence of {n} numbers: {fib_sequence}")

#optimize the fibonacci sequence code to make it more efficient
n = int(input("Enter the number of fibonacci numbers to generate: "))
fib_sequence = []
a, b = 0, 1
for _ in range(n):
    fib_sequence.append(a)
    a, b = b, a + b
print(f"Fibonacci sequence of {n} numbers: {fib_sequence}")
```

```
PS C:\Users\Pavani> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/L1.py"
Enter the number of fibonacci numbers to generate: 5
Fibonacci sequence of 5 numbers: [0, 1, 1, 2, 3]
PS C:\Users\Pavani> 
```

Explanation:

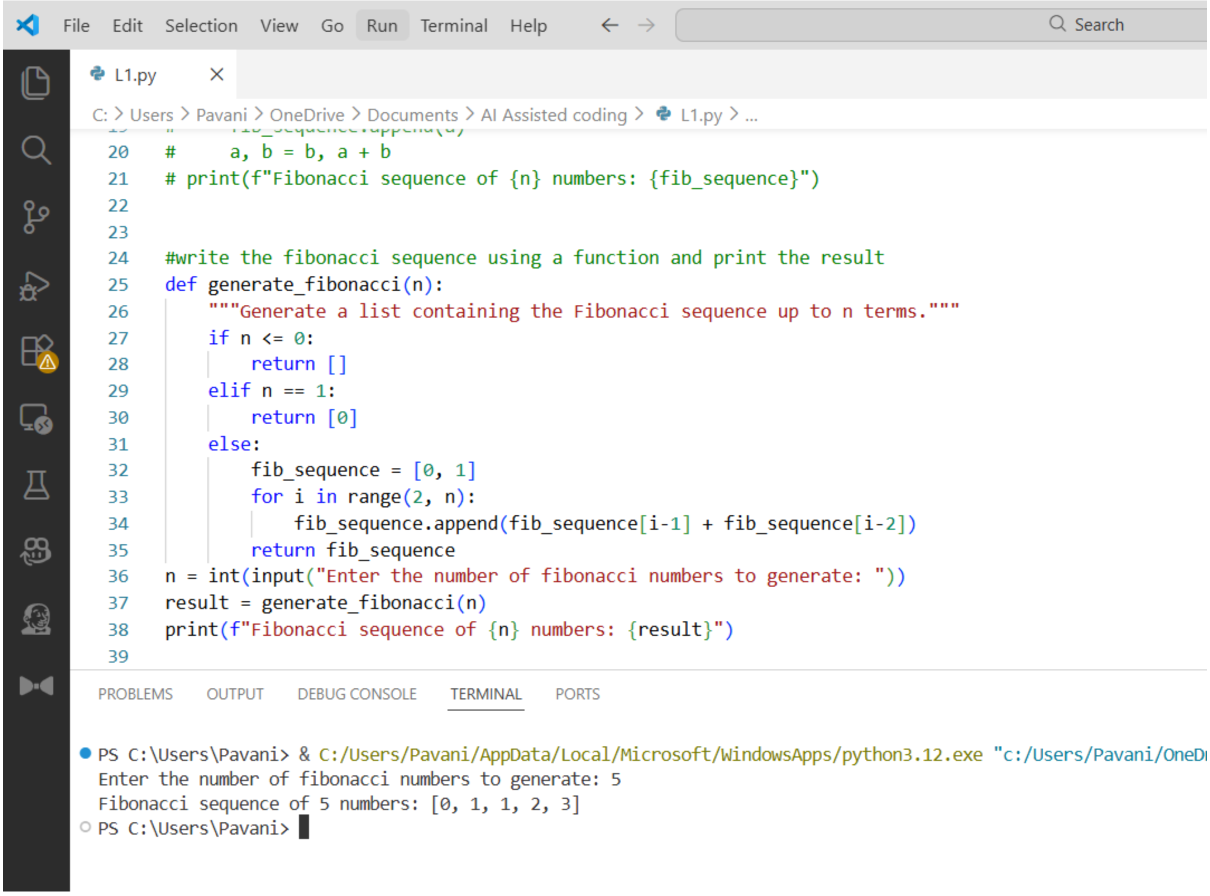
In task2 the initially generated Fibonacci code was optimized to improve efficiency and readability. Redundant variables were removed, and the logic was simplified by using only two variables to generate the sequence iteratively. This reduced unnecessary computations and made the code easier to understand. The optimized program correctly outputs the Fibonacci sequence for the given input.

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

PROMPT:

write the fibonacci sequence using a function and print the result

CODE WITH OUTPUT:



```python
#     a, b = b, a + b
# print(f"Fibonacci sequence of {n} numbers: {fib_sequence}")


#write the fibonacci sequence using a function and print the result
def generate_fibonacci(n):
    """Generate a list containing the Fibonacci sequence up to n terms."""
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    else:
        fib_sequence = [0, 1]
        for i in range(2, n):
            fib_sequence.append(fib_sequence[i-1] + fib_sequence[i-2])
        return fib_sequence
n = int(input("Enter the number of fibonacci numbers to generate: "))
result = generate_fibonacci(n)
print(f"Fibonacci sequence of {n} numbers: {result}")

```

```
PS C:\Users\Pavani> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDı
Enter the number of fibonacci numbers to generate: 5
Fibonacci sequence of 5 numbers: [0, 1, 1, 2, 3]
PS C:\Users\Pavani>
```

EXPLANATION:

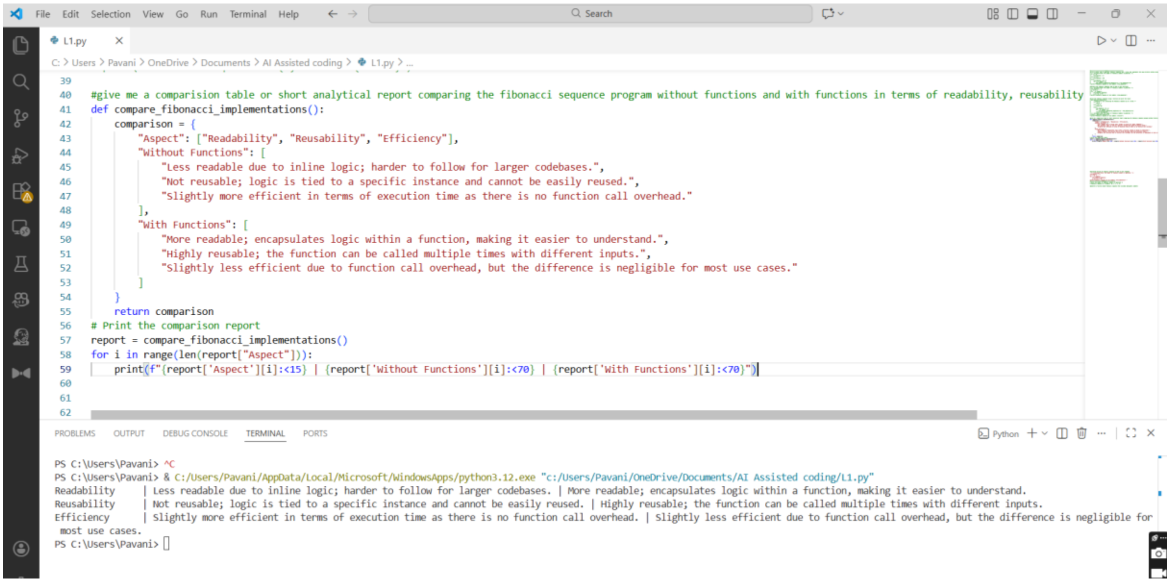
In task 3 GitHub Copilot was used to generate a modular Python program for the Fibonacci sequence using a user-defined function. The Fibonacci logic is encapsulated inside a function, improving code reusability and clarity. The function returns the sequence up to n terms, which is then printed in the main program. This modular approach makes the code easier to test, debug, and reuse in larger applications.

Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code

PROMPT:

give me a comparision table or short analytical report comparing the fibonacci sequence program without functions and with functions in terms of readability, reusability, and efficiency.

CODE WITH OUTPUT:



File Edit Selection View Go Run Terminal Help
Search
L1.py
C: > Users > Pavani > OneDrive > Documents > AI Assisted coding > L1.py > ...
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python

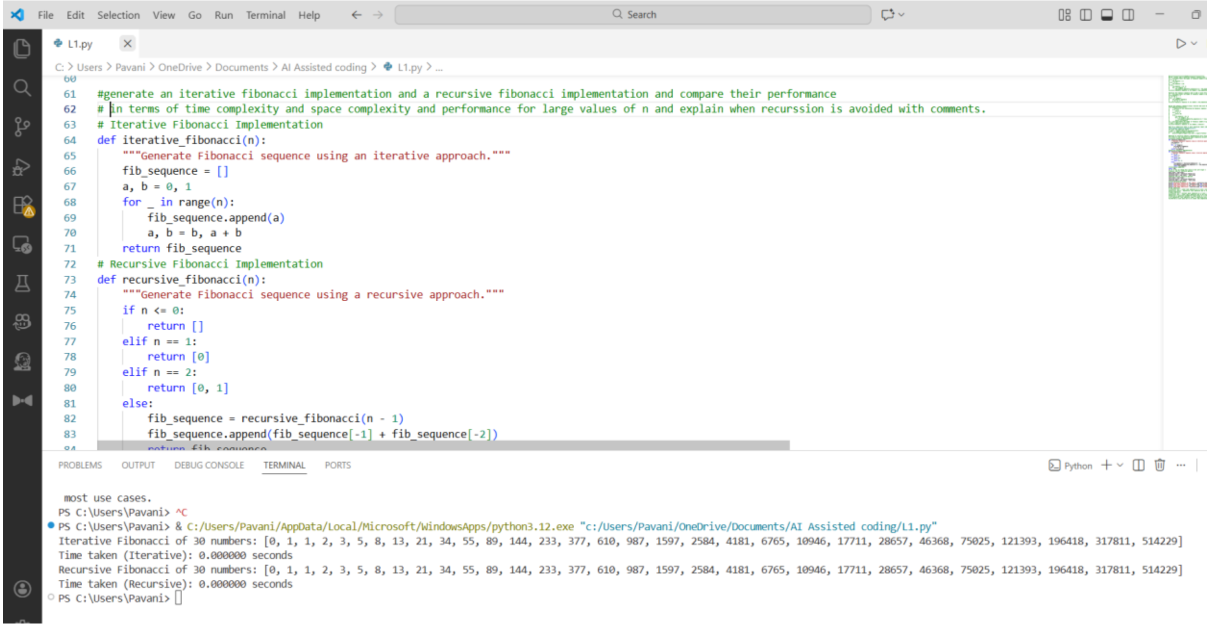
```python
#give me a comparision table or short analytical report comparing the fibonacci sequence program without functions and with functions in terms of readability, reusability
def compare_fibonacci_implementations():
    comparison = {
        "Aspect": ["Readability", "Reusability", "Efficiency"],
        "Without Functions": [
            "Less readable due to inline logic; harder to follow for larger codebases.",
            "Not reusable; logic is tied to a specific instance and cannot be easily reused.",
            "Slightly more efficient in terms of execution time as there is no function call overhead."
        ],
        "With Functions": [
            "More readable; encapsulates logic within a function, making it easier to understand.",
            "Highly reusable; the function can be called multiple times with different inputs.",
            "Slightly less efficient due to function call overhead, but the difference is negligible for most use cases."
        ]
    }
    return comparison
# Print the comparison report
report = compare_fibonacci_implementations()
for i in range(len(report["Aspect"])):
    print(f"{report['Aspect'][i]:<15} | {report['Without Functions'][i]:<70} | {report['With Functions'][i]:<70}")
```

```
PS C:\Users\Pavani> ^C
PS C:\Users\Pavani> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/L1.py"
Readability     | Less readable due to inline logic; harder to follow for larger codebases. | More readable; encapsulates logic within a function, making it easier to understand.
Reusability     | Not reusable; logic is tied to a specific instance and cannot be easily reused. | Highly reusable; the function can be called multiple times with different inputs.
Efficiency      | Slightly more efficient in terms of execution time as there is no function call overhead. | Slightly less efficient due to function call overhead, but the difference is negligible for
 most use cases.
PS C:\Users\Pavani> 
```

EXPLANATION:

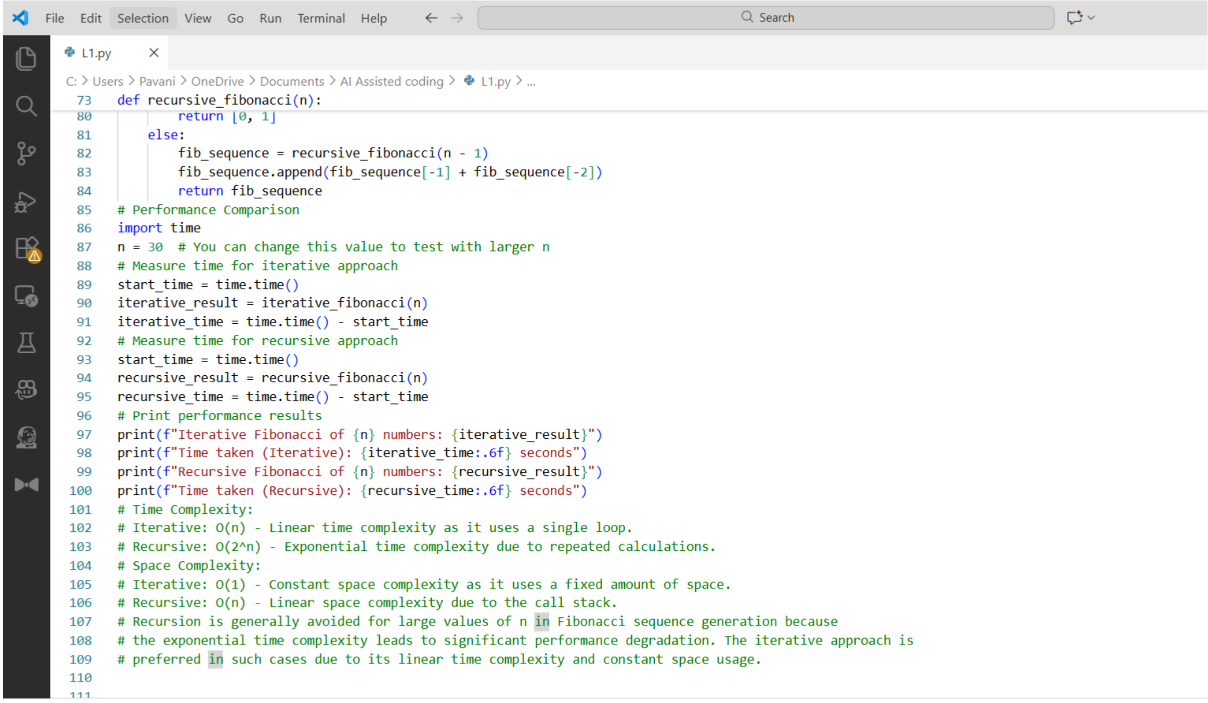
In task 4 a comparison was made between the Fibonacci program written without functions and the function-based implementation. The non-modular version is simpler but harder to read and reuse in larger codebases. The function-based version improves readability, reusability, and maintainability by encapsulating logic. This comparison shows that modular code is more suitable for debugging and large-scale applications.

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

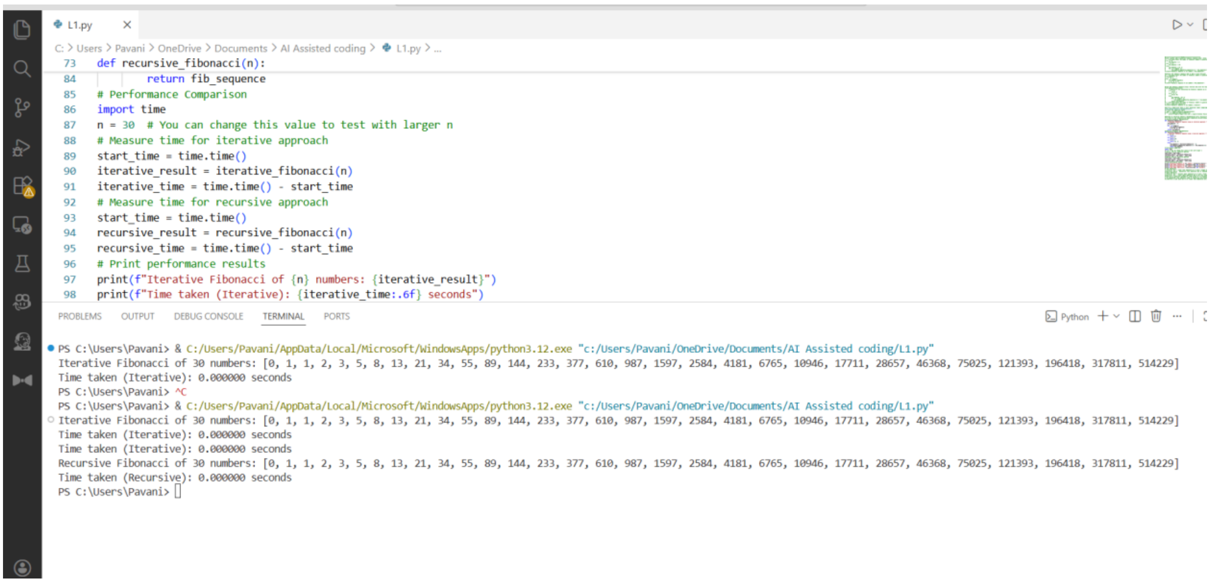

File Edit Selection View Go Run Terminal Help
Search
L1.py
C: > Users > Pavani > OneDrive > Documents > AI Assisted coding > L1.py > ...
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python

```python
#generate an iterative fibonacci implementation and a recursive fibonacci implementation and compare their performance
# in terms of time complexity and space complexity and performance for large values of n and explain when recurssion is avoided with comments.
# Iterative Fibonacci Implementation
def iterative_fibonacci(n):
    """Generate Fibonacci sequence using an iterative approach."""
    fib_sequence = []
    a, b = 0, 1
    for _ in range(n):
        fib_sequence.append(a)
        a, b = b, a + b
    return fib_sequence
# Recursive Fibonacci Implementation
def recursive_fibonacci(n):
    """Generate Fibonacci sequence using a recursive approach."""
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]
    else:
        fib_sequence = recursive_fibonacci(n - 1)
        fib_sequence.append(fib_sequence[-1] + fib_sequence[-2])
```

```
 most use cases.
PS C:\Users\Pavani> ^C
PS C:\Users\Pavani> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/L1.py"
Iterative Fibonacci of 30 numbers: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229]
Time taken (Iterative): 0.000000 seconds
Recursive Fibonacci of 30 numbers: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229]
Time taken (Recursive): 0.000000 seconds
PS C:\Users\Pavani> 
```


File Edit Selection View Go Run Terminal Help
Search
L1.py
C: > Users > Pavani > OneDrive > Documents > AI Assisted coding > L1.py > ...

```python
def recursive_fibonacci(n):
        return [0, 1]
    else:
        fib_sequence = recursive_fibonacci(n - 1)
        fib_sequence.append(fib_sequence[-1] + fib_sequence[-2])
        return fib_sequence
# Performance Comparison
import time
n = 30  # You can change this value to test with larger n
# Measure time for iterative approach
start_time = time.time()
iterative_result = iterative_fibonacci(n)
iterative_time = time.time() - start_time
# Measure time for recursive approach
start_time = time.time()
recursive_result = recursive_fibonacci(n)
recursive_time = time.time() - start_time
# Print performance results
print(f"Iterative Fibonacci of {n} numbers: {iterative_result}")
print(f"Time taken (Iterative): {iterative_time:.6f} seconds")
print(f"Recursive Fibonacci of {n} numbers: {recursive_result}")
print(f"Time taken (Recursive): {recursive_time:.6f} seconds")
# Time Complexity:
# Iterative: O(n) - Linear time complexity as it uses a single loop.
# Recursive: O(2^n) - Exponential time complexity due to repeated calculations.
# Space Complexity:
# Iterative: O(1) - Constant space complexity as it uses a fixed amount of space.
# Recursive: O(n) - Linear space complexity due to the call stack.
# Recursion is generally avoided for large values of n in Fibonacci sequence generation because
# the exponential time complexity leads to significant performance degradation. The iterative approach is
# preferred in such cases due to its linear time complexity and constant space usage.
```


L1.py
C: > Users > Pavani > OneDrive > Documents > AI Assisted coding > L1.py > ...
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Python

```python
def recursive_fibonacci(n):
        return fib_sequence
# Performance Comparison
import time
n = 30  # You can change this value to test with larger n
# Measure time for iterative approach
start_time = time.time()
iterative_result = iterative_fibonacci(n)
iterative_time = time.time() - start_time
# Measure time for recursive approach
start_time = time.time()
recursive_result = recursive_fibonacci(n)
recursive_time = time.time() - start_time
# Print performance results
print(f"Iterative Fibonacci of {n} numbers: {iterative_result}")
print(f"Time taken (Iterative): {iterative_time:.6f} seconds")
```

```
PS C:\Users\Pavani> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/L1.py"
Iterative Fibonacci of 30 numbers: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229]
Time taken (Iterative): 0.000000 seconds
PS C:\Users\Pavani> ^C
PS C:\Users\Pavani> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/L1.py"
Iterative Fibonacci of 30 numbers: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229]
Time taken (Iterative): 0.000000 seconds
Time taken (Iterative): 0.000000 seconds
Recursive Fibonacci of 30 numbers: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229]
Time taken (Recursive): 0.000000 seconds
PS C:\Users\Pavani> 
```

EXPLANATION:

In task 5 GitHub Copilot generated both iterative and recursive implementations of the Fibonacci sequence. The iterative approach uses loops and is efficient in terms of time and memory, making it suitable for large values of n. The recursive approach follows a self-calling function pattern which is simpler conceptually but consumes more time and stack memory. This comparison shows that recursion should be avoided for large inputs due to performance and space limitations.