

2303A54064 B-47B

Lab 1

Assignment Number 1.3

TASK:1

PROMPT: Write a Python program that takes user input n and prints the Fibonacci sequence up to n terms using main code only without any user-defined functions

The image shows a Windows IDE (VS Code) interface with a dark theme. The top menu bar includes File, Edit, Selection, View, Go, Run, and a search icon. The Explorer pane on the left shows the file structure with 'Task 1.py' selected. The main editor displays the code for 'Task 1.py', which prompts the user for the number of terms and prints the Fibonacci sequence. The Output pane at the bottom shows the execution of the script, displaying the input '15' and the resulting sequence '0 1 1 2 3 5 8 13 21 34 55 89 144 233 377'.

```
1 n = int(input("Enter number of terms: "))
2 a, b = 0, 1
3
4 if n <= 0:
5     print("Please enter a positive integer.")
6 else:
7     for i in range(n):
8         print(a, end=" " if i < n - 1 else "\n")
9         a, b = b, a + b
```

Output:

```
PS C:\Users\tejom\OneDrive\Desktop\AI-ASSISTED-CODING> & C:/Users/tejom/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/tejom/OneDrive/Desktop/AI-ASSISTED-CODING/Task 1.py"
Enter number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
PS C:\Users\tejom\OneDrive\Desktop\AI-ASSISTED-CODING>
```

TASK:2

PROMPT: Optimize this Fibonacci code by removing redundant variables, simplifying loop logic, avoiding unnecessary computations, and improving readability while keeping it function-free

```
File Edit Selection View Go Run ... AI-ASSISTED-CODING Welcome Task 1.py U task-2 U X Release Notes: 1.108.0 Welcome
```

```
EXPLORER AI-ASSISTED-CODING Task 1.py U task-2 U
```

```
task-2.py
```

```
1
```

```
# - Removing redundant variables
```

```
# - Simplifying loop logic
```

```
# - Avoiding unnecessary computations
```

```
# - Improving readability
```

```
Add Context...
```

```
Unexpected type
```

```
Python + -
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
```

```
PS C:\Users\tejom\OneDrive\Desktop\AI-ASSISTED-CODING> & C:/Users/tejom/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/tejom/OneDrive/Desktop/AI-ASSISTED-CODING/Task 1.py"
```

```
Enter number of terms: 15
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

```
PS C:\Users\tejom\OneDrive\Desktop\AI-ASSISTED-CODING>
```

```
Ln 1, Col 1 Spaces: 4 UTF-8 CRLF {} Plain Text
```

TASK:3

PROMPT: Write a Python program that uses a user-defined function to generate and return the Fibonacci sequence up to n terms with clear and meaningful comments

```
File Edit Selection View Go Run ... AI-ASSISTED-CODING Task 1.py U task-2.py U task 3.py U X
```

```
EXPLORER AI-ASSISTED-CODING task 3.py U Task 1.py U task-2.py U
```

```
task 3.py ...
```

```
1 # Function to generate Fibonacci sequence up to n terms
```

```
2 def generate_fibonacci(n):
```

```
3 """
```

```
4     Generate Fibonacci sequence up to n terms.
```

```
5
```

```
6     Args:
```

```
7     n: Number of Fibonacci terms to generate
```

```
8
```

```
9     Returns:
```

```
10    List containing the Fibonacci sequence
```

```
11    """
```

```
12    # Initialize list to store Fibonacci numbers
```

```
13    fib_sequence = []
```

```
14
```

```
15    # Handle edge case: if n is 0 or negative
```

```
16    if n <= 0:
```

```
17        return fib_sequence
```

```
18
```

```
19    # Initialize first two Fibonacci numbers
```

```
20    a, b = 0, 1
```

```
21
```

```
22    # Generate n Fibonacci numbers
```

```
23    for i in range(n):
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
```

```
PS C:\Users\tejom\OneDrive\Desktop\AI-ASSISTED-CODING> & C:/Users/tejom/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/tejom/OneDrive/Desktop/AI-ASSISTED-CODING/task-2.py
```

```
Enter the number of terms: 26
```

```
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
```

```
PS C:\Users\tejom\OneDrive\Desktop\AI-ASSISTED-CODING> & C:/Users/tejom/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/tejom/OneDrive/Desktop/AI-ASSISTED-CODING/task 3.py"
```

```
Enter the number of Fibonacci terms: 21
```

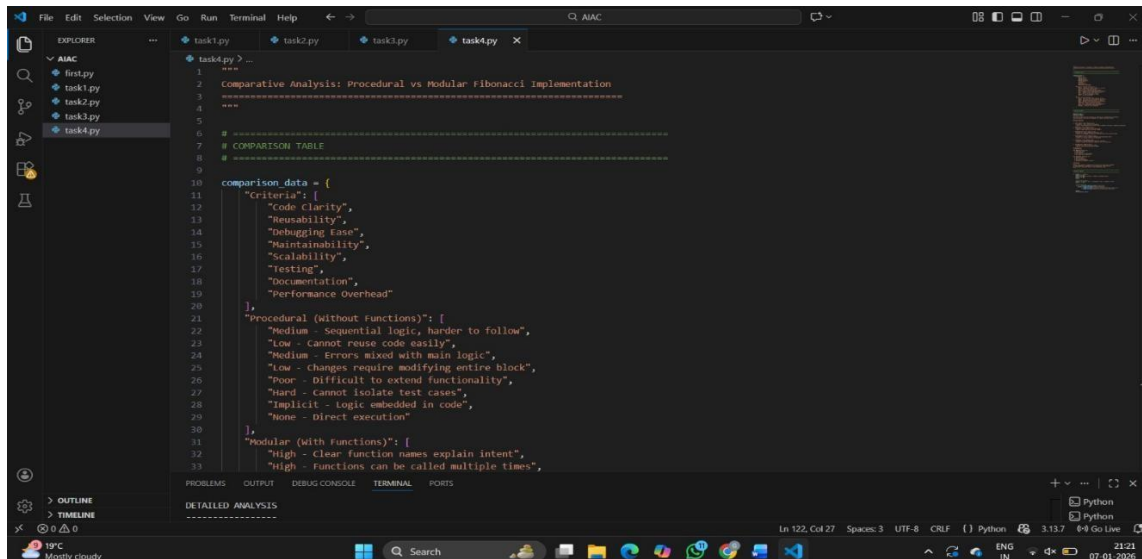
```
Fibonacci sequence (21 terms): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
```

```
PS C:\Users\tejom\OneDrive\Desktop\AI-ASSISTED-CODING>
```

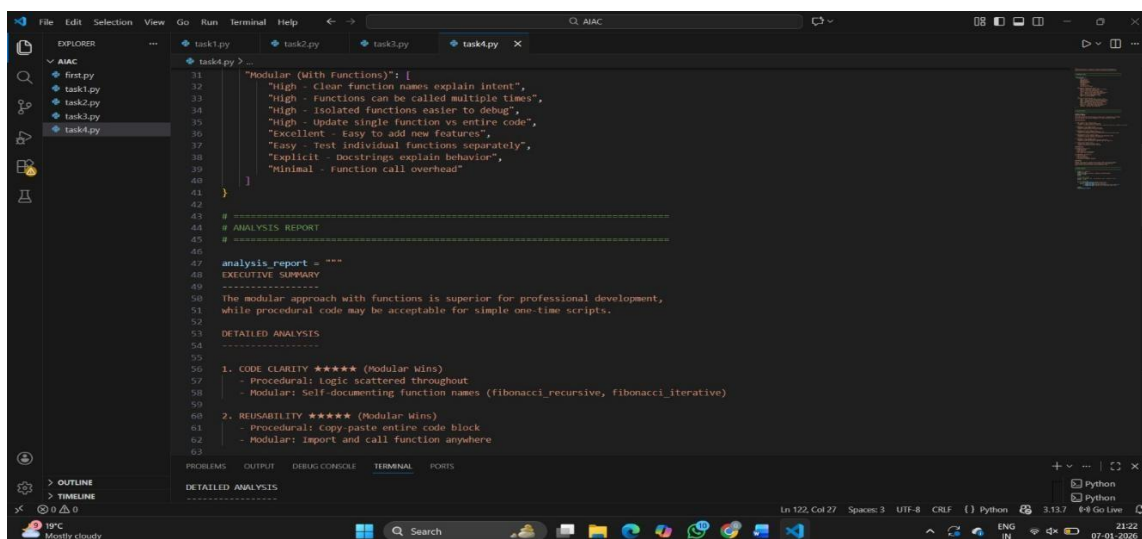
```
Ln 40, Col 63 Spaces: 4 UTF-8 {} Python 3.14.2
```

TASK:4

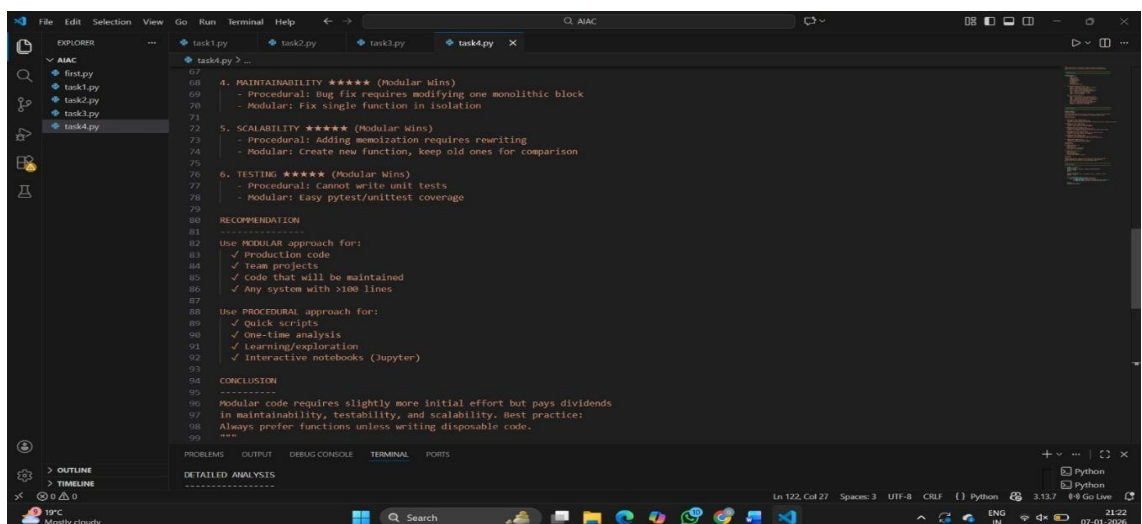
PROMPT: Create a comparative analysis between procedural Fibonacci code without functions and modular Fibonacci code with functions focusing on code clarity, reusability, debugging ease, and suitability for large systems



```
1 """
2 Comparative Analysis: Procedural vs Modular Fibonacci Implementation
3 """
4
5
6 # =====
7 # COMPARISON TABLE
8 # =====
9
10 comparison_data = {
11     "Criteria": [
12         "Code Clarity",
13         "Reusability",
14         "Debugging Ease",
15         "Maintainability",
16         "Scalability",
17         "Testing",
18         "Documentation",
19         "Performance Overhead"
20     ],
21     "Procedural (without functions)": [
22         "Medium - Sequential logic, harder to follow",
23         "Low - Cannot reuse code easily",
24         "Medium - Errors mixed with main logic",
25         "Low - Changes require modifying entire block",
26         "Poor - Difficult to extend functionality",
27         "Hard - Cannot isolate test cases",
28         "Implicit - Logic embedded in code",
29         "None - Direct execution"
30     ],
31     "Modular (With Functions)": [
32         "High - Clear function names explain intent",
33         "High - Functions can be called multiple times",
34         "High - Isolated functions easier to debug",
35         "High - Update single function vs entire code",
36         "Excellent - Easy to add new features",
37         "Easy - Test individual functions separately",
38         "Explicit - Docstrings explain behavior",
39         "Minimal - Function call overhead"
40     ]
41 }
```



```
42
43
44 # =====
45 # ANALYSIS REPORT
46 # =====
47
48 analysis_report = """
49 EXECUTIVE SUMMARY
50 -----
51 The modular approach with functions is superior for professional development,
52 while procedural code may be acceptable for simple one-time scripts.
53
54 DETAILED ANALYSIS
55 -----
56
57 1. CODE CLARITY ***** (Modular Wins)
58   - Procedural: Logic scattered throughout
59   - Modular: Self-documenting function names (fibonacci_recursive, fibonacci_iterative)
60
61 2. REUSABILITY ***** (Modular Wins)
62   - Procedural: Copy-paste entire code block
63   - Modular: Import and call function anywhere
64
65 3. MAINTAINABILITY ***** (Modular Wins)
66   - Procedural: Bug fix requires modifying one monolithic block
67   - Modular: Fix single function in isolation
68
69 4. SCALABILITY ***** (Modular Wins)
70   - Procedural: Adding memoization requires rewriting
71   - Modular: Create new function, keep old ones for comparison
72
73 5. TESTING ***** (Modular Wins)
74   - Procedural: Cannot write unit tests
75   - Modular: Easy pytest/unittest coverage
76
77 6. DOCUMENTATION ***** (Modular Wins)
78   - Procedural: No docstrings
79   - Modular: Easy to add docstrings
80
81 RECOMMENDATION
82 -----
83 Use MODULAR approach for:
84   ✓ Production code
85   ✓ Team projects
86   ✓ Code that will be maintained
87   ✓ Any system with >100 lines
88
89 Use PROCEDURAL approach for:
90   ✓ Quick scripts
91   ✓ One-time analysis
92   ✓ Learning/exploration
93   ✓ Interactive notebooks (Jupyter)
94
95 CONCLUSION
96 -----
97 Modular code requires slightly more initial effort but pays dividends
98 in maintainability, testability, and scalability. Best practice:
99 Always prefer functions unless writing disposable code.
100 """
```



```
101
102
103 RECOMMENDATION
104 -----
105 Use MODULAR approach for:
106   ✓ Production code
107   ✓ Team projects
108   ✓ Code that will be maintained
109   ✓ Any system with >100 lines
110
111 Use PROCEDURAL approach for:
112   ✓ Quick scripts
113   ✓ One-time analysis
114   ✓ Learning/exploration
115   ✓ Interactive notebooks (Jupyter)
116
117 CONCLUSION
118 -----
119 Modular code requires slightly more initial effort but pays dividends
120 in maintainability, testability, and scalability. Best practice:
121 Always prefer functions unless writing disposable code.
122 """
```

The screenshot shows a Visual Studio Code editor with a file explorer on the left containing files task1.py through task5.py. The main editor displays task4.py, which is a script for a comparative analysis of Fibonacci implementations. The terminal at the bottom shows the output of the script, which is a table comparing 'Procedural' and 'Modular' approaches across various criteria.

```
105 if __name__ == "__main__":
106     print("-" * 80)
107     print("COMPARATIVE ANALYSIS: FIBONACCI IMPLEMENTATIONS")
108     print("-" * 80)
109     print()
110
111     # Print table header
112     print(f"{'Criteria':<25} | {'Procedural':<30} | {'Modular':<30}")
113     print("-" * 88)
114
115     # Print table rows
116     for i in range(len(comparison_data["Criteria"])):
117         print(f"{'Criteria':<25} | "
118               f"{'Procedural (Without Functions)':<30} | "
119               f"{'Modular (With Functions)':<30}")
120
121     print()
122     print(analysis_report)
```

Criteria	Procedural	Modular
Code Clarity	Medium - Sequential logic, harder to follow	High - Clear function names explain intent
Reusability	Low - Cannot reuse code easily	High - Functions can be called multiple times
Debugging Ease	Medium - Errors mixed with main logic	High - Isolated functions easier to debug
Maintainability	Low - Changes require modifying entire block	High - Update single function vs entire code
Scalability	Poor - Difficult to extend functionality	Excellent - Easy to add new features
Testing	Hard - Cannot isolate test cases	Easy - Test individual functions separately
Documentation	Implicit - Logic embedded in code	Explicit - Docstrings explain behavior
Performance Overhead	None - Direct execution	Minimal - Function call overhead

TASK:5

PROMPT: Generate both iterative and recursive Fibonacci implementations in Python with comments, explain their execution flow, and compare their time and space complexity and performance for large n

The screenshot shows a Visual Studio Code editor with a file explorer on the left containing files task1.py through task5.py. The main editor displays task5.py, which contains three Python functions for calculating Fibonacci numbers: an iterative version, a recursive version, and a memoized recursive version. The terminal at the bottom shows the output of the script, which is a table comparing 'Procedural' and 'Modular' approaches across various criteria.

```
1 import time
2 # ITERATIVE FIBONACCI
3 def fibonacci_iterative(n: int) -> int:
4     if n <= 0:
5         return 0
6     elif n == 1:
7         return 1
8     a, b = 0, 1
9     for _ in range(2, n + 1):
10         a, b = b, a + b
11     return b
12
13 # RECURSIVE FIBONACCI
14 def fibonacci_recursive(n: int) -> int:
15     if n <= 0:
16         return 0
17     elif n == 1:
18         return 1
19     return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
20
21 # OPTIMIZED RECURSIVE (with memoization)
22 def fibonacci_memoized(n: int, memo: dict = None) -> int:
23     if memo is None:
24         memo = {}
25     if n in memo:
26         return memo[n]
```

Criteria	Procedural	Modular
Code Clarity	Medium - Sequential logic, harder to follow	High - Clear function names explain intent
Reusability	Low - Cannot reuse code easily	High - Functions can be called multiple times
Debugging Ease	Medium - Errors mixed with main logic	High - Isolated functions easier to debug
Maintainability	Low - Changes require modifying entire block	High - Update single function vs entire code
Scalability	Poor - Difficult to extend functionality	Excellent - Easy to add new features
Testing	Hard - Cannot isolate test cases	Easy - Test individual functions separately
Documentation	Implicit - Logic embedded in code	Explicit - Docstrings explain behavior
Performance Overhead	None - Direct execution	Minimal - Function call overhead

VS Code interface showing a Python file named `task5.py` with a Fibonacci memoization implementation. The code includes a `fibonacci_memoized` function and a `main` function that compares iterative and recursive methods for `n=35`.

```
20 def fibonacci_memoized(n: int, memo: dict = None) -> int:
21     return memo[n]
22 # COMPARISON & TESTING
23 if __name__ == "__main__":
24     n = 35
25     # Iterative Test
26     start = time.time()
27     result_iter = fibonacci_iterative(n)
28     time_iter = time.time() - start
29     print(f"Iterative (n={n}): {result_iter} | Time: {time_iter:.6f}s")
30     # Recursive Test (safe limit)
31     if n <= 30:
32         start = time.time()
33         result_rec = fibonacci_recursive(n)
34         time_rec = time.time() - start
35         print(f"Recursive (n={n}): {result_rec} | Time: {time_rec:.6f}s")
36     else:
37         print(f"Recursive: Skipped (too slow for n={n})")
38     # Memoized Test
39     start = time.time()
40     result_memo = fibonacci_memoized(n)
41     time_memo = time.time() - start
42     print(f"Memoized (n={n}): {result_memo} | Time: {time_memo:.6f}s")
```

The bottom panel displays a comparison table between Procedural and Modular code styles.

Criteria	Procedural	Modular
Code Clarity	Medium - Sequential logic, harder to follow	High - Clear function names explain intent
Reusability	Low - Cannot reuse code easily	High - Functions can be called multiple times
Debugging Ease	Medium - Errors mixed with main logic	High - Isolated functions easier to debug
Maintainability	Low - Changes require modifying entire block	High - Update single function vs entire code
Scalability	Poor - Difficult to extend functionality	Excellent - Easy to add new features
Testing	Hard - Cannot isolate test cases	Easy - Test individual functions separately
Documentation	Implicit - Logic embedded in code	Explicit - Docstrings explain behavior
Performance Overhead	None - Direct execution	Minimal - Function call overhead

The status bar at the bottom shows the file is at Line 46, Column 58, with 4 spaces, UTF-8 encoding, and CRLF line endings. The system tray at the very bottom indicates a temperature of 20°C and the date 09-01-2026.