

# ASSIGNMENT -1

Name : B. Hasini  
Hall ticket :2303A54013  
Batch :47\_A

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

❖ **Scenario**

You are asked to write a quick numerical sequence generator for a learning platform prototype.

❖ **Task Description**

Use GitHub Copilot to generate a Python program that:

- Prints the Fibonacci sequence up to  $n$  terms
- Accepts user input for  $n$
- Implements the logic directly in the main code
- Does not use any user-defined functions

❖ **Expected Output**

- Correct Fibonacci sequence for given  $n$
- Screenshot(s) showing Copilot-generated suggestions
- Sample inputs and outputs

CODE:

```
Lab 1.py > ...
1 # First N Fibonacci terms (no functions)
2 N = int(input("How many terms? "))
3 a, b = 0, 1
4 for _ in range(N):
5     print(a)
6     a, b = b, a + b
```

OUTPUT:

```
PS D:\3.2\AI ass Coding> & C:/Users/srimu/AppData/Local/Programs/Python/Python313/python.exe "d:/3.2/AI ass Coding/Lab 1.p
y"
How many terms? 2
0
1
PS D:\3.2\AI ass Coding>
```

PROMPT :

Give me the python code for fibonacci sequence without functions

CODE EXPLANATION :

This program prints the first N Fibonacci numbers without using functions. It first takes the number of terms (N) from the user. Then it initializes two variables a and b with values 0 and 1. A for loop runs N times, printing the current value of a each time. After printing, the values of a and b are updated so that a becomes the next Fibonacci number and b stores the following one. This loop-based approach is simple, fast, and easy to understand.

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

### ❖ Scenario

The prototype will be shared with other developers and needs optimization.

### ❖ Task Description

- Examine the Copilot-generated code from Task 1 and improve it by:
- Removing redundant variables
- Simplifying loop logic
- Avoiding unnecessary computations
- Use Copilot prompts such as:
  - “Optimize this Fibonacci code”
  - “Simplify variable usage”

Hint:

Prompt Copilot with phrases like

“optimize this code”, “simplify logic”, or “make it more readable”

### ❖ Expected Output

- Original vs improved code
- Written explanation of:
  - What was inefficient
  - How the optimized version improves performance and readability

## CODE AND OUTPUT :

The screenshot shows a code editor interface with a Python script named `Lab 1.py`. The code generates the first `N` Fibonacci numbers. The AI panel on the right provides optimization suggestions and alternative implementations.

```

19 import sys
20
21 try:
22     N = int(input("How many terms? "))
23     if N <= 0:
24         raise ValueError
25 except Exception:
26     print("Please enter a positive integer.")
27     sys.exit(1)
28
29 a, b = 0, 1
30 write = sys.stdout.write
31 for _ in range(N):
32     write(f'{a}\n')      # single write per term (fast)
33     a, b = b, a + b
  
```

**AI Panel Content:**

- optimize the code of # First N Fibonacci terms (no functions)
- `N = int(input("How many terms? "))`
- `a, b = 0, 1`
- `for _ in range(N):`
- `print(a)`
- `a, b = b, a + b`

Below the panel, a note says: "Optimized Fibonacci code and explored options".

Further down, it says: "Let me give two optimized snippets: a buffered streaming version (low memory, best for large N) and a batched join version (fast for moderate N). I'll show both and a brief note on when to use each.## Optimized Fibonacci (no functions)".

At the bottom, it lists two optimized versions:

- Use the **buffered streaming** version for very large N (low memory, minimal I/O calls).
- Use the **batched join** version for moderate N (fast because it does a single print).

The first snippet is:

```

1) Buffered streaming — low memory, fast IO
import sys
try:
    N = int(input("How many terms? "))
    if N <= 0:
        sys.exit(1)
    a, b = 0, 1
    for _ in range(N):
        write(f'{a}\n')
        a, b = b, a + b

```

**PROMPT:**

Optimize the above code

**CODE EXPLANATION :**

This program prints the first N Fibonacci numbers in an optimized way without using functions. It first takes input from the user and checks whether the value is a positive integer using a try–except block. If the input is invalid, it displays an error message and exits the program. After validation, it initializes the first two Fibonacci numbers ( $a = 0$  and  $b = 1$ ). A loop then runs N times, printing each Fibonacci number using `sys.stdout.write`, which is faster than normal printing. After each output, the values are updated to generate the next Fibonacci number. This approach is efficient and suitable for larger inputs.

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

❖ **Scenario**

The Fibonacci logic is now required in multiple modules of an application.

❖ **Task Description**

Use GitHub Copilot to generate a function-based Python program that:

- Uses a user-defined function to generate Fibonacci numbers
- Returns or prints the sequence up to  $n$
- Includes meaningful comments (AI-assisted)

❖ **Expected Output**

- Correct function-based Fibonacci implementation
- Screenshots documenting Copilot's function generation

Sample test cases with outputs

**CODE:**

```

# fib.py - function-based Fibonacci implementations (fib(0)=0, fib(1)=1)

from functools import lru_cache
from typing import Iterator, Tuple

def fib_iter(n: int) -> int:
    """Iterative O(n) nth Fibonacci (safe for moderate n)."""
    if n < 0:
        raise ValueError("n must be >= 0")
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

def fib_gen(n: int) -> Iterator[int]:
    """Yield first n Fibonacci numbers (0 .. n-1)."""
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

@lru_cache(maxsize=None)
def fib_rec(n: int) -> int:
    """Recursive with memoization (good for small-to-medium n)."""
    if n < 2:
        return n
    return fib_rec(n - 1) + fib_rec(n - 2)

def fib_fast_doubling(n: int) -> int:
    """Fast doubling O(log n) nth Fibonacci – best for large n."""
    if n < 0:
        raise ValueError("n must be >= 0")
    def _fd(k: int) -> Tuple[int, int]:
        if k == 0:
            return (0, 1)
        a, b = _fd(k // 2)
        c = a * (2 * b - a)
        return (a, c)

```

## OUTPUT:

The screenshot shows a Jupyter Notebook interface with two code cells. The left cell contains the following Python code:

```

Lab 1.py > print_first_n
73     return _fd(n)[0]
74
75 def print_first_n(n: int, method: str = "gen") -> None:
76     """Print first n terms using chosen method: 'gen'|'iter'|'fast'|'rec'."""
77     if n <= 0:
78         return
79     if method == "gen":
80         print("\n".join(str(x) for x in fib_gen(n)))
81     elif method == "iter":
82         print("\n".join(str(fib_iter(i)) for i in range(n)))
83     elif method == "fast":
84         print("\n".join(str(fib_fast_doubling(i)) for i in range(n)))
85     elif method == "rec":
86         print("\n".join(str(fib_rec(i)) for i in range(n)))
87     else:
88         raise ValueError("method must be one of 'gen', 'iter', 'fast', 'rec'")
89
90 if __name__ == "__main__":
91     N = int(input("How many terms? "))
92     print_first_n(N, method="gen")

```

The right cell contains the following Python code:

```

REVERSE MULTIPLICATION TABLES FROM 100 TO 1
if __name__ == "__main__":
    N = 50
    print("fib_iter(50) =", fib_iter(N))
    print("fib_fast_doubling(50) =", fib_fast_doubling(50))
    print("first 10 via generator:", list(fib_gen(10)))

```

Both cells show code completion dropdowns for variables like `fib_iter`, `fib_gen`, and `fib_rec`.

## PROMPT:

Give me python code for the fibonacci sequence using function

### CODE EXPLANATION :

This code uses functions to generate Fibonacci numbers in different efficient ways. It includes an iterative method, a generator for memory efficiency, a recursive method with memoization for clarity, and a fast-doubling method for very large inputs. A common print function lets the user choose any method, making the code reusable, organized, and suitable for larger programs.

### Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code

#### ❖ Scenario

You are participating in a code review session.

#### ❖ Task Description

Compare the Copilot-generated Fibonacci programs:

- Without functions (Task 1)
- With functions (Task 3)
- Analyze them in terms of:
  - Code clarity
  - Reusability
  - Debugging ease
  - Suitability for larger systems

#### ❖ Expected Output

Comparison table or short analytical report

CODE:

```
print("Comparative Analysis ┌ Fibonacci Programs")
print("-" * 75)

print("{:<25} {:<22} {:<22}".format(
    "Criteria", "Without Functions", "With Functions"
))

print("-" * 75)

print("{:<25} {:<22} {:<22}".format(
    "Code Clarity",
    "Harder to read",
    "Clear and structured"
))

print("{:<25} {:<22} {:<22}".format(
    "Reusability",
    "Not reusable",
    "Reusable function"
))

print("{:<25} {:<22} {:<22}".format(
```

```

print("{:<25} {:<22} {:<22}".format(
    "Not reusable",
    "Reusable function"
))

print("{:<25} {:<22} {:<22}".format(
    "Debugging Ease",
    "Difficult",
    "Easy to debug"
))

print("{:<25} {:<22} {:<22}".format(
    "Larger Systems",
    "Not suitable",
    "Highly suitable"
))

print("-" * 75)

```

#### OUTPUT:

Criteria	Without Functions	With Functions
Code Clarity	Harder to read	Clear and structured
Reusability	Not reusable	Reusable function
Debugging Ease	Difficult	Easy to debug
Larger Systems	Not suitable	Highly suitable

#### PROMPT:

I am participating in a code review session compare fibonacci sequence without functions and with function analyze them in terms of code clarity and reusability and debugging ease and suitability for larger systems

#### CODE EXPLANATION :

This code prints a comparison table in a clean and readable format using Python's `format()` method. Each `print` statement displays one row with properly aligned columns, making the data easy to compare. The values show differences like reusability, debugging ease, and suitability for large systems. The line printed using `"-" * 75` acts as a separator to improve table readability.

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

### ❖ Scenario

Your mentor wants to assess AI's understanding of different algorithmic paradigms.

### ❖ Task Description

Prompt GitHub Copilot to generate:

- An iterative Fibonacci implementation
- A recursive Fibonacci implementation

### ❖ Expected Output

- Two correct implementations
- Explanation of execution flow for both
- Comparison covering:
  - Time and space complexity
  - Performance for large  $n$
  - When recursion should be avoided

```
def fibonacci_iterative(n):  
    a, b = 0, 1  
    for i in range(n):  
        print(a, end=" ")  
        a, b = b, a + b  
  
n = int(input())  
fibonacci_iterative(n)
```

```
n = int(input())  
fibonacci_iterative(n)  
  
def fibonacci_recursive(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)  
  
n = int(input())  
for i in range(n):  
    print(fibonacci_recursive(i), end=" ")
```

PROMPT :

Give me the code for Iterative vs Recursive Fibonacci Approaches Different Algorithmic Approaches for Fibonacci Series

### CODE EXPLANATION :

Iterative Fibonacci uses a loop to generate Fibonacci numbers one by one by updating variables. It is fast, uses less memory, and works well for large values of n. **Recursive** Fibonacci uses a function that calls itself to calculate Fibonacci numbers based on previous values. It is easy to understand but slow and uses more memory due to repeated function calls. Therefore, iterative Fibonacci is preferred for performance, while recursion is mainly used for learning concepts.

```
Compare iterative and recursive Fibonacci implementations in terms of time complexity, space complexity, and performance.
```

```
1 import time
2 import sys
3
4 # Recursive Fibonacci (naive)
5 def fib_recursive(n):
6     if n <= 1:
7         return n
8     return fib_recursive(n - 1) + fib_recursive(n - 2)
9
10 # Recursive Fibonacci (memoized)
11 def fib_recursive_memo(n, memo=None):
12     if memo is None:
13         memo = {}
14     if n in memo:
15         return memo[n]
16     if n <= 1:
17         return n
18     memo[n] = fib_recursive_memo(n - 1, memo) + fib_recursive_memo(n - 2, memo)
19     return memo[n]
20
21 # Iterative Fibonacci
22 def fib_iterative(n):
23     if n <= 1:
24         return n
25     a, b = 0, 1
26     for _ in range(2, n + 1):
27         a, b = b, a + b
28     return b
```

```

30 # Performance comparison
31 def benchmark(func, n, name):
32     start = time.time()
33     result = func(n)
34     end = time.time()
35     print(f"{name}(n={n}): {result} | Time: {end - start:.6f}s")
36
37 # Test
38 n = 35
39 print("== Fibonacci Comparison (n=35) ==")
40 print("Recursive (naive):")
41 benchmark(fib_recursive, n, "fib_recursive")
42
43 print("\nRecursive (memoized):")
44 benchmark(lambda x: fib_recursive_memo(x), n, "fib_recursive_memo")
45
46 print("\nIterative:")
47 benchmark(fib_iterative, n, "fib_iterative")
48
49 print("\n== Complexity Summary ==")
50 print("Naive Recursive:    Time: O(2^n)    | Space: O(n)")
51 print("Memoized Recursive: Time: O(n)      | Space: O(n)")
52 print("Iterative:          Time: O(n)      | Space: O(1)")

PS C:\Users\HP\OneDrive\Documents\Website-1> & C:/Users/HP/AppData/Local/Programs/Python/Python313/python.exe c:/Users/HP/OneDrive/Documents/Website-1/Lab-01.py
fib_recursive(n=35): 9227465 | Time: 4.022568s
Recursive (memoized):
fib_recursive_memo(n=35): 9227465 | Time: 0.000054s

Iterative:
fib_iterative(n=35): 9227465 | Time: 0.000010s

== Complexity Summary ==
Naive Recursive:    Time: O(2^n)    | Space: O(n)
Memoized Recursive: Time: O(n)      | Space: O(n)
Iterative:          Time: O(n)      | Space: O(1)

```