

AI Assistant Coding

Assignment -1

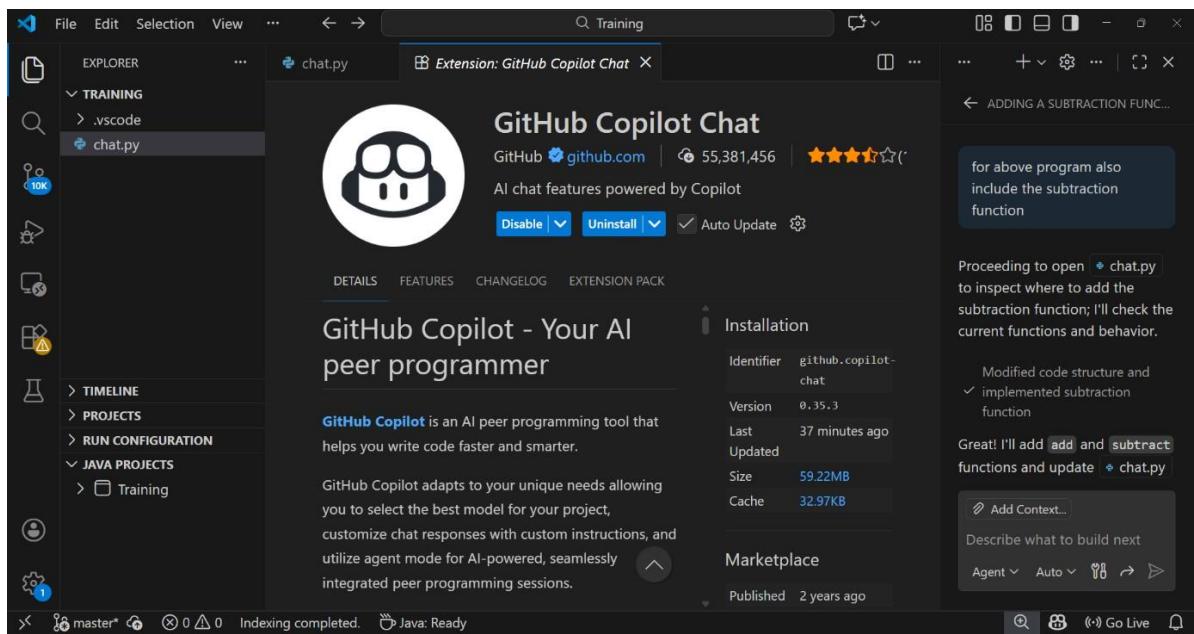
Name: T. Swetha

Roll no:2303a51317

Batch:05

1. Lab 1: Environment Setup – GitHub Copilot and VS Code Integration + Understanding AI-assisted Coding Workflow.





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

- File Bar:** File, Edit, Selection, View, ...
- Search Bar:** Training
- Code Editor:** The file "chat.py" is open, containing the following Python code:

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

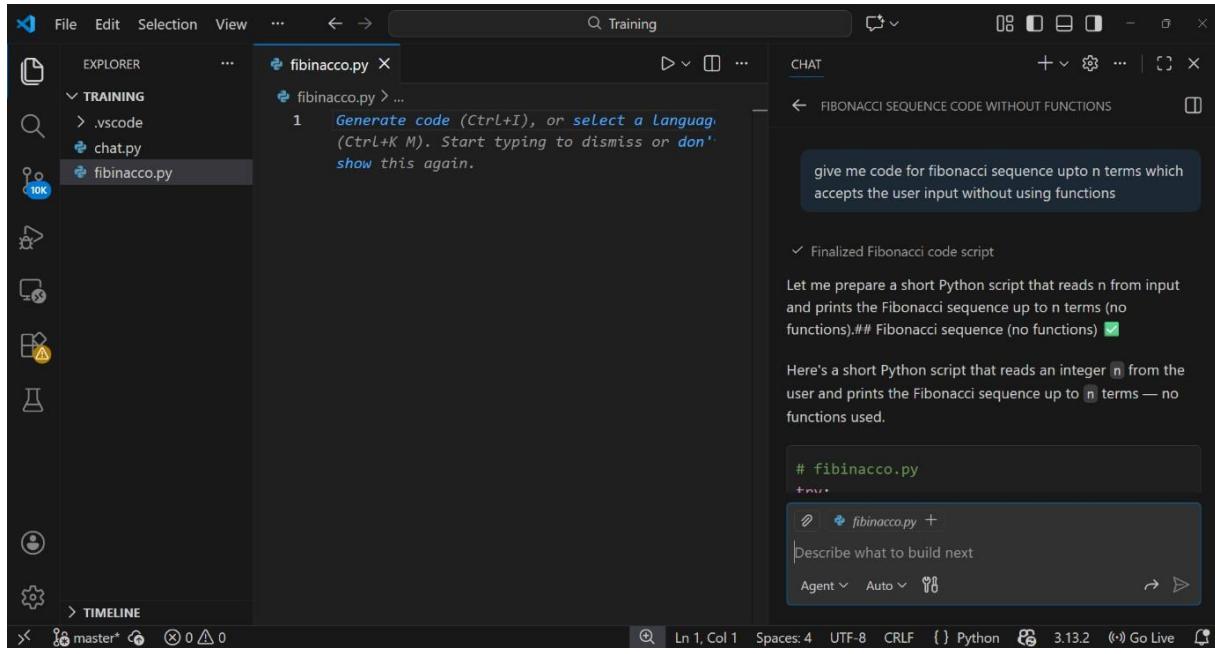
if __name__ == "__main__":
    a = 10
    b = 20
    print("Sum:", add(a, b))
    print("Difference:", subtract(a, b))
```
- Explorer:** Shows a project named "TRAINING" with a ".vscode" folder and "chat.py".
- Sidebar:** Includes sections for Timeline, Projects, Run Configuration, and Java Projects.
- Bottom Status Bar:** master*, 0△0, Indexing completed, Java: Ready, Ln 11, Col 41, Spaces: 4, UTF-8, CRLF, Python 3.13.2, Go Live, and a bell icon.

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

- File Bar:** File, Edit, Selection, View, ...
- Search Bar:** Training — chat.py - Training - Visual Studio Code
- Code Editor:** The file "chat.py" is open, showing the same Python code as the first screenshot.
- CHAT:** A sidebar on the right displays a message: "← ADDING A SUBTRACTION FUNCTION TO A PROGRAM added".
- What I changed:** A list of changes:
 - Added `add(a, b)` and `subtract(a, b)` functions to `chat.py`.
 - Updated the script to print both sum and difference.
- File updated:** `chat.py`
- Updated code (short):** A preview of the updated code.
- Terminal:** Shows the output "Difference: -10" and the path "C:\Users\thaku\OneDrive\Desktop\Training>".
- Bottom Status Bar:** cmd, cmd, Python, and other standard status bar items.

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

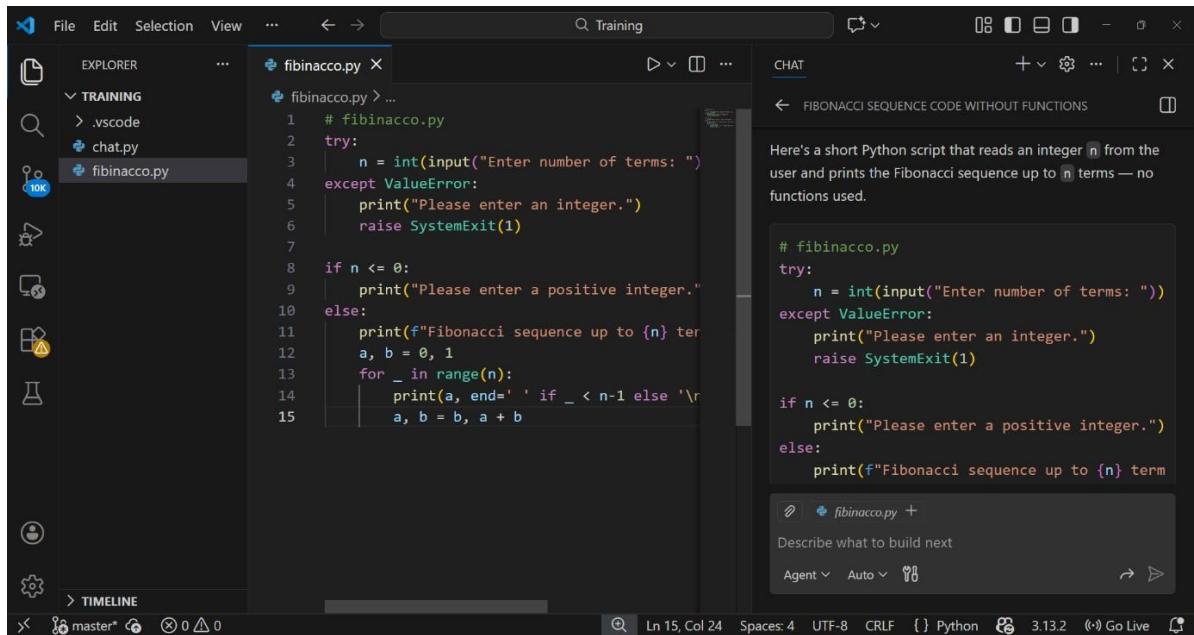
Step 1: Giving prompt: Give me code for fibonacci sequence up to n terms which accepts the user input without using functions



```
# fibinacco.py
try:
    n = int(input("Enter number of terms: "))
except ValueError:
    print("Please enter an integer.")
    raise SystemExit(1)

if n <= 0:
    print("Please enter a positive integer.")
else:
    print(f"Fibonacci sequence up to {n} term")
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ' if _ < n-1 else '\n')
        a, b = b, a + b
```

Step 2: Code



```
# fibinacco.py
try:
    n = int(input("Enter number of terms: "))
except ValueError:
    print("Please enter an integer.")
    raise SystemExit(1)

if n <= 0:
    print("Please enter a positive integer.")
else:
    print(f"Fibonacci sequence up to {n} term")
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ' if _ < n-1 else '\n')
        a, b = b, a + b
```

Step 3: Output

The screenshot shows a VS Code interface with the following details:

- File Explorer:** Shows a folder named "TRAINING" containing files ".vscode", "chat.py", and "fibinacco.py".
- Code Editor:** Displays the "fibinacco.py" file with the following code:

```
# fibinacco.py
try:
    n = int(input("Enter number of terms: "))
except ValueError:
    print("Please enter an integer.")
    raise SystemExit(1)

if n <= 0:
    print("Please enter a positive integer.")
else:
    print(f"Fibonacci sequence up to {n} term")
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ')
        a, b = b, a + b
```
- Terminal:** Shows the output of running the script: "Enter number of terms: 4" followed by "Fibonacci sequence up to 4 terms: 0 1 1 2".
- Status Bar:** Shows the path "C:\Users\thaku\OneDrive\Desktop\Training", the file name "fibinacco.py", and the line number "Ln 15, Col 24".
- Bottom Right:** Includes a snippet of code for generating a Fibonacci sequence without functions, a "Describe what to build next" input field, and other VS Code status indicators.

Step 4: Explanation of Code

Input Handling:

- Uses try-except to ensure the user enters an integer.
- Exits gracefully if input is invalid.

Validation:

- Checks if $n \leq 0$.
- Prompts user to enter a positive integer if invalid.

Output Message:

- Prints a header with correct pluralization (term vs terms).

Initialization:

- Starts with $a = 0, b = 1$ (first two Fibonacci numbers).

Loop Logic:

- Runs for $_ \text{ in } \text{range}(n)$ to generate n terms.
- Prints current value of a .
- Updates values with $a, b = b, a + b$.

Printing Style:

- Numbers are printed on the same line separated by spaces.
- Last number ends with a newline.

Result:

- Produces the Fibonacci sequence up to n terms.

Example:

Input: 4 → Output: 0 1 1 2

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

Step 1: Giving Prompt: Using the above code improve it by removing abundant variables and reduce to small loop logic and avoid unnecessary computations / optimize the code

The screenshot shows a Microsoft Visual Studio Code (VS Code) interface. On the left is the Explorer sidebar with files like '.vscode', 'chat.py', and 'fibinacco.py'. The main editor area contains Python code for generating a Fibonacci sequence. A Chat window on the right has a message from an AI agent asking to improve the code by removing redundant variables and optimizing the loop logic. The AI also suggests updating the code and refining it. The status bar at the bottom shows the file is 'fibinacco.py', line 19, column 16, with Python 3.13.2 and Go Live features.

```
9     print("Please enter a positive integer.")
10    elif n == 1:
11        print("Fibonacci sequence up to 1 term:")
12        print(0)
13    else:
14        # build sequence in-place (only n-2 additions)
15        seq = [0, 1]
16        for _ in range(n - 2):
17            seq.append(seq[-1] + seq[-2])
18        print(f"Fibonacci sequence up to {n} terms")
19        print(*seq)
```

Step 2: Code

The screenshot shows the VS Code interface with the following details:

- EXPLORER** sidebar: TRAINING folder containing .vscode, chat.py, and fibinacco.py.
- Code Editor**: The file fibinacco.py is open, showing Python code to generate a Fibonacci sequence. The code includes a try-except block for integer input and a loop that prints the sequence up to n terms.
- CHAT** panel: A conversation with an AI about optimizing the code. The AI has suggested removing the per-iteration conditional and adding a single newline after the loop.
- Bottom Status Bar**: master*, Ln 16, Col 12, Spaces: 4, UTF-8, Python, 3.13.2, Go Live.

Step 3: Output

The screenshot shows the VS Code interface with the following details:

- EXPLORER** sidebar: TRAINING folder containing .vscode, chat.py, and fibinacco.py.
- Code Editor**: The file fibinacco.py is open, showing the same Python code as before.
- TERMINAL** tab: Active, showing the command line output of running the script. It shows the user entering '4' and seeing the output 'Fibonacci sequence up to 4 terms: 0 1 1 2'.
- CHAT** panel: The AI continues to provide optimization tips, mentioning that the minimal optimization is implemented.
- Bottom Status Bar**: master*, Ln 17, Col 5, Spaces: 4, UTF-8, CRLF, Python, 3.13.2, Go Live.

Step 4: Code Explanation

Input Handling

- o Uses try-except to ensure the user enters an integer.
- o Exits safely if input is invalid.

Validation

- Checks if $n \leq 0$.
- Prompts user to enter a positive integer if invalid.

Output Message

- Prints a header with correct pluralization (term vs terms).

Initialization

- Starts with $a = 0, b = 1$ (first two Fibonacci numbers).

Loop Logic

- Runs for $_$ in $\text{range}(n)$ to generate n terms.
- Prints current value of a .
- Updates values with $a, b = b, a + b$.

What Was Inefficient in the Original Code

The original code had stuff that wasn't needed to generate the Fibonacci sequence.

A volatile memory variable was used to hold the sum and added extra length.

It's because you're putting the fibonacci sequence into a list as well a program like this will just output the result, but store it in memory somewhere.

The loop was constructed in more than one step, which made the code less readable

How the optimized/python version brings performance improvement and improved readability.

Unnecessary variables were discarded and the values were updated by tuple assignment in an efficient manner.

Your code also writes to disc, I assume you want to speed the codeup, so this is not good. Your optimized version that this saved memory.3) that it prints the number instead of storing a whole vector with all solutions.

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

Step 1: Prompt: Give code for Fibonacci sequence using functions.

The screenshot shows the Visual Studio Code interface. In the Explorer sidebar, there are files: .vscode, chat.py, and fibinacco.py. The main editor window contains the following Python code:

```
# fibinacco.py
try:
    n = int(input("Enter number of terms: "))
except ValueError:
    print("Please enter an integer.")
    raise SystemExit(1)

if n <= 0:
    print("Please enter a positive integer.")
else:
    print(f"Fibonacci sequence up to {n} terms")
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ')
        a, b = b, a + b
```

The terminal below shows the output of running the script:

```
C:\Users\thaku\OneDrive\Desktop\Training> C:/Users/thaku/AppData/Local/Programs/Python/Python313/python.exe c:/Users/thaku/OneDrive/Desktop/Training/fibinacco.py
Enter number of terms: 4
Fibonacci sequence up to 4 terms:
0 1 1 2
```

The Chat panel on the right has a message from the AI: "give code for fibonacci series using functions". The AI is thinking and preparing a response.

Step 2: Code:

The screenshot shows the Visual Studio Code interface. The Explorer sidebar shows files: .vscode, chat.py, and fibinacco.py. The main editor window contains the following Python code:

```
def fib(n: int) -> list:
    """Return first n Fibonacci terms as a list."""
    if n <= 0:
        return []
    if n == 1:
        return [0]
    seq = [0, 1]
    for _ in range(n - 2):
        seq.append(seq[-1] + seq[-2])
    return seq

def main():
    try:
        n = int(input("Enter number of terms: "))
    except ValueError:
        print("Please enter an integer.")
        return
    seq = fib(n)
    if not seq:
        print("Please enter a positive integer.")
    else:
        print(*seq)

if __name__ == "__main__":
    main()
```

The Chat panel on the right shows the AI providing feedback: "Provided and presented Fibonacci code" and "I'm going to provide a concise implementation with a reusable function and a generator variant; next I'll show the code and a short note on usage.## Fibonacci using functions". It also shows a snippet of iterative function code and a generator function code.

Step 3: Output

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows files .vscode, chat.py, and fibinacco.py.
- Code Editor:** Displays the content of fibinacco.py. The code defines two functions: fib(n) which returns a list of Fibonacci terms, and main() which prompts the user for input and calls fib(n).
- Terminal:** Shows the command line output of running the script: "C:\Users\thaku\Desktop\Training> C:/Users/thaku/AppData/Local/Programs/Python/Python313/python.exe c:/Users/thaku/OneDrive/Desktop/Desktop/Training/fibinacco.py". It then asks "Enter number of terms: " and prints the sequence "0 1 1 2 3".
- Output Panel:** Shows the output of the fib(n) function call, resulting in the list [0, 1, 1, 2, 3].
- Chat Panel:** Shows a note from the AI: "I'm going to provide a concise implementation with a reusable function and a generator variant; next I'll show the code and a short note on usage.## Fibonacci using functions".

Step 4: Explanation of code

Function Design

- fib(n) cleanly separates Fibonacci logic from input/output.
- Returns a list of the first n Fibonacci terms.

Validation

- If $n \leq 0$: returns an empty list.
- If $n == 1$: returns [0].
- Handles edge cases gracefully.

Sequence Generation

- Starts with [0, 1].
- Uses a loop for $_ \in \text{range}(n - 2)$ to generate remaining terms.
- Each new term = sum of last two ($\text{seq}[-1] + \text{seq}[-2]$).

Main Function

- Prompts user for input.
- Uses try-except to catch invalid input (ValueError).
- Calls fib(n) to generate sequence.
- Prints a warning if input is invalid or non-positive.

Output

- Prints the sequence neatly with `print(*seq)` (space-separated values).

Practices

- Includes a docstring for clarity.
- Uses `__name__ == "__main__"` guard for modularity.
- Keeps logic reusable and testable.

Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code

Feature	Without Functions	With Functions
Code Clarity	Logic is written in one block; harder to read when long	Logic is separated into a named function → easier to understand
Reusability	Cannot reuse Fibonacci logic without rewriting	Can call the function anywhere in program
Debugging Ease	Bugs must be traced in main logic, mixed with other code	Errors isolated in function → easier to test & fix
Suitability for Larger Systems	Poor; not scalable, becomes messy with added features	Good; fits into bigger systems, easier to maintain
Testing	Hard to unit test a part of code independently	Function can be tested separately with multiple inputs
Maintainability	Low; changes affect entire code block	High; changes only in function, no impact on main flow
Performance Impact	No function call overhead (very small benefit)	Minimal overhead but worth it for structure & scaling

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

Step 1: Prompt: Give code for fibonacci series using iterative and recursive functions

```
File Edit Selection View ... < > Q Training CHAT + < FIBONACCI SEQUENCE CODE WITHOUT FUNCTIONS Raptor mini (Preview) * 1x Give code for fibannoci dequence using iterative and recursive functions Thinking... Implementing Fibonacci Functions I'm setting out to provide both an iterative function, fib_iter(n), that returns a list of the first n terms, and a recursive function, fib_rec(n), that returns the nth
```

```
fibonacci.py > ...
def fib(n: int) -> list:
    """Return first n Fibonacci terms as a list."""
    if n <= 0:
        return []
    if n == 1:
        return [0]
    seq = [0, 1]
    for _ in range(n - 2):
        seq.append(seq[-1] + seq[-2])
    return seq

def main():
    try:
        n = int(input("Enter number of terms: "))
    except ValueError:
        print("Please enter an integer.")
    return
    seq = fib(n)
    if not seq:
        print("Please enter a positive integer.")
    else:
        print(*seq)

if __name__ == "__main__":
    main()
```

Describe what to build next
Agent Auto

Step 2: Code

Iterative:

```
File Edit Selection View ... < > Q Training CHAT + < FIBONACCI SEQUENCE CODE WITHOUT FUNCTIONS if n < 0: raise ValueError("n must be non-negative") if n < 2: return n return fib_rec_memo(n - 1) + fib_rec_memo(n - 2)

Example usage (prompt + print) def main(): try: n = int(input("Enter number of terms: ")) except ValueError: print("Please enter an integer.") return

if n <= 0: print("Please enter a positive integer.") return

# iterative sequence print("Iterative:", *fib_iter(n))

# naive recursive (only for very small n) if n <= 10: print("Naive recursive:", [fib_rec(i) for i in range(n)]) else: print("Naive recursive: (skipped for n > 10 due to time complexity)")

# memoized recursive (efficient) print("Memoized recursive:", [fib_rec_memo(i) for i in range(n)])
```

```
fibonacci.py 3 > ...
def main():
    try:
        n = int(input("Enter number of terms: "))
    except ValueError:
        print("Please enter an integer.")
    return
    seq = fib(n)
    if not seq:
        print("Please enter a positive integer.")
    else:
        print(*seq)

if __name__ == "__main__":
    main()
```

Describe what to build next
Agent Auto

Step 3: Output

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows a folder named "TRAINING" containing ".vscode", "chat.py", and "fibinacco.py".
- Code Editor:** Displays the "fibinacco.py" file with three methods:
 - `def fib_iter(n: int) -> list:` An iterative function that builds a list from 0 to n.
 - `def fib_rec(n: int) -> int:` A naive recursive function that returns the nth Fibonacci number.
 - `def fib_rec_memo(n: int) -> int:` A recursive function with memoization using `@lru_cache`.
- Terminal:** Prints the output of the code:

```
Enter number of terms: 4
Iterative: 0 1 1 2
Naive recursive: [0, 1, 1, 2]
Memoized recursive: [0, 1, 1, 2]
```
- Status Bar:** Shows the current file path: C:\Users\thaku\OneDrive\Desktop\Training\fibinacco.py, and other details like line 60, column 11, and Python 3.13.2.

Step 4: Explanation of code

`fib_iter(n)` → Iterative list builder, efficient $O(n)$.

- `fib_rec(n)` → Naive recursion, exponential cost, only practical for very small n .
- `fib_rec_memo(n)` → Recursive with `@lru_cache`, efficient $O(n)$ via memoization.
- Input Handling → Validates integer input, rejects negatives/zero.
- Main Function →
 - Prints iterative sequence always.
 - Prints naive recursive only if $n \leq 10$.
 - Prints memoized recursive for any n .
- Good Practices → Clear separation of methods, input validation, modular design, efficient memoization.

Example ($n=6$):

Iterative: 0 1 1 2 3 5

Naive recursive: [0, 1, 1, 2, 3, 5]

Memoized recursive: [0, 1, 1, 2, 3, 5]

Comparison covering:

Aspect	Iterative	Recursive
Time Complexity	$O(n)$	$O(2^n)$ (very slow due to repeated calls)
Space Complexity	$O(1)$	$O(n)$ (stack memory for calls)
Performance for Large n	Excellent (can handle 10^7+ if needed)	Poor ($\text{fib}(50)$ may take seconds/minutes)
Memory Usage	Very low	High because of recursion stack
Scalability	Best for real systems	Not scalable without optimization
Risk	No crash risk	Stack Overflow for large n