NAME:O.ISRAEL H.NO:2303A51825 BATCH:26

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		**DEPARTMENT OF COMPUTER SCIENCE ENGINEERING**	
Program Name: B. Tech	**Assignment Type: Lab**		**Academic Year:**2025-2026
Course Coordinator Name	Dr. Rishabh Mittal		
Instructor(s) Name	Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M.Madhuri Ms. Katherashala Swetha Ms. Velpula sumalatha Mr. Bingi Raju		
CourseCode	23CS002PC304	**Course Title**	AI Assisted Coding
Year/Sem	III/II	**Regulation**	R23
Date and Day of Assignment	**Week1 – Thursday**	**Time(s)**	23CSBTB01 To 23CSBTB52
Duration	2 Hours	**Applicable to Batches**	All batches
Assignment Number:1.3(Present assignment number)/**24**(Total number of assignments)			

Q.No.	Question	*Expected Time to complete*
1	Lab 1: Environment Setup – *GitHub Copilot and VS Code Integration + Understanding AI-assisted Coding Workflow* **Lab Objectives:**	Week1 - Monday

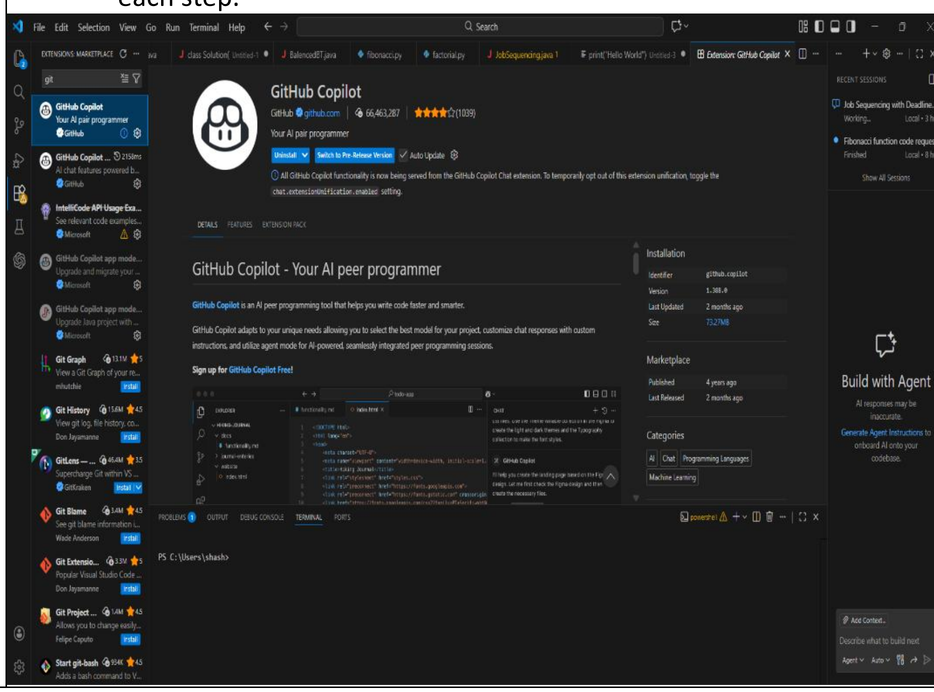
- To install and configure GitHub Copilot in Visual Studio Code.
- To explore AI-assisted code generation using GitHub Copilot.
- To analyze the accuracy and effectiveness of Copilot's code suggestions.
- To understand prompt-based programming using comments and code context

Lab Outcomes (LOs):
After completing this lab, students will be able to:

- Set up GitHub Copilot in VS Code successfully.
- Use inline comments and context to generate code with Copilot.
- Evaluate AI-generated code for correctness and readability.
- Compare code suggestions based on different prompts and programming styles.

Task 0
- Install and configure GitHub Copilot in VS Code. Take screenshots of each step.

Expected Output
- Install and configure GitHub Copilot in VS Code. Take screenshots of each step.



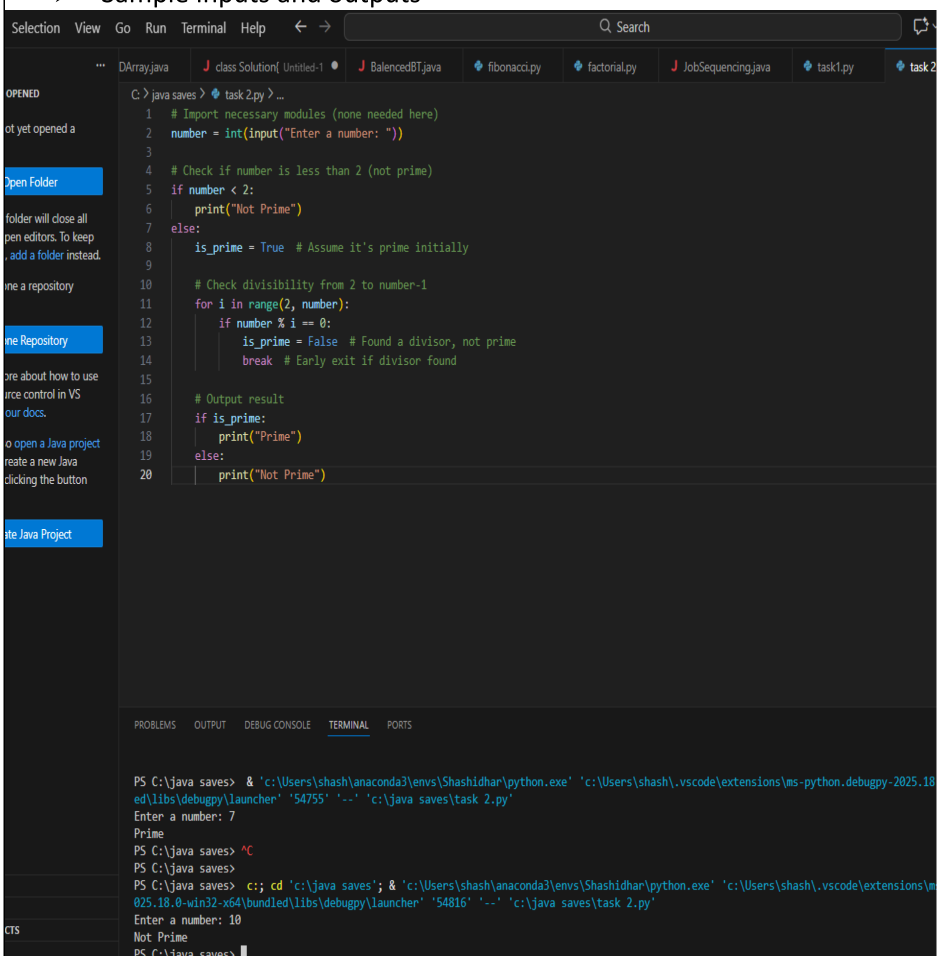
Task 1: AI-Generated Logic Without Modularization (Prime Number Check Without Functions)

- **Scenario**
  - You are developing a **basic validation script** for a numerical learning application.

- **Task Description**

  Use GitHub Copilot to generate a Python program that:
  - Checks whether a given number is **prime**
  - Accepts user input
  - Implements logic **directly in the main code**
  - Does **not** use any user-defined functions

- **Expected Output**
  - Correct prime / non-prime result
  - Screenshots showing Copilot-generated code suggestions
  - Sample inputs and outputs



Task 2: Efficiency & Logic Optimization (Cleanup)

❖ **Scenario**

The script must handle larger input values efficiently.

❖ **Task Description**

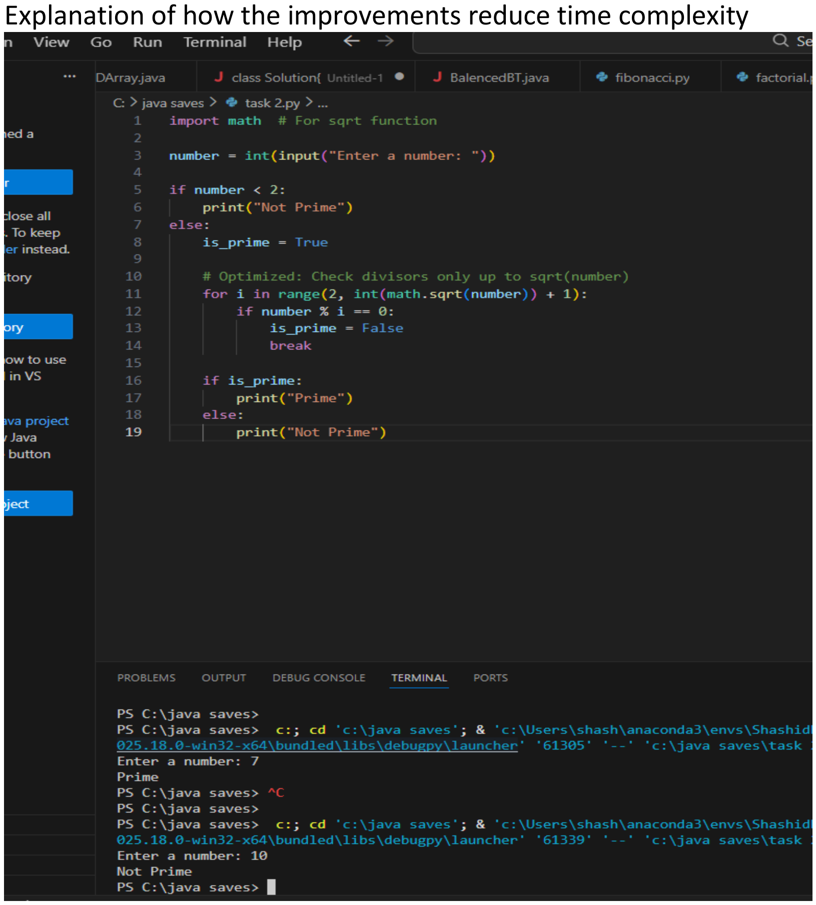
Review the Copilot-generated code from Task 1 and improve it by:

- Reducing unnecessary iterations
- Optimizing the loop range (e.g., early termination)
- Improving readability
- Use Copilot prompts like:
  - *"Optimize prime number checking logic"*
  - *"Improve efficiency of this code"*

Hint:
Prompt Copilot with phrases like
"optimize this code", *"simplify logic"*, or *"make it more readable"*

❖ **Expected Output**

- Original and optimized code versions
- Explanation of how the improvements reduce time complexity



Task 3: Modular Design Using AI Assistance (Prime Number Check Using

Functions)

❖ **Scenario**

The prime-checking logic will be reused across multiple modules.

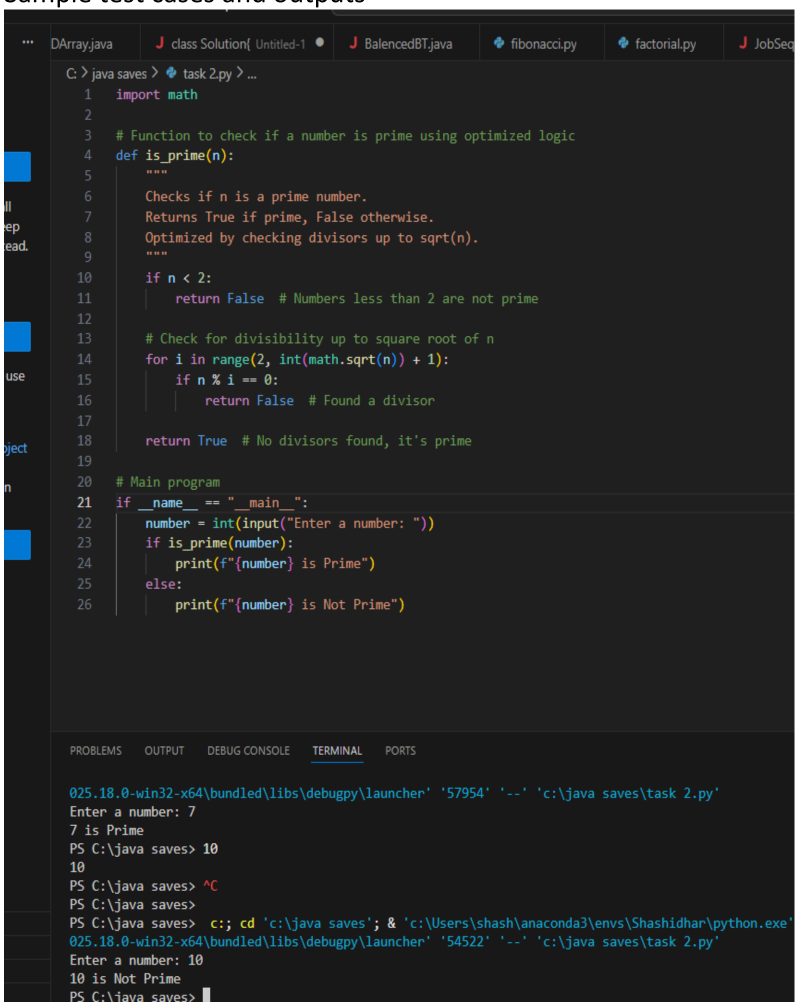
❖ **Task Description**

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

- ➢ Uses a user-defined function to check primality
- ➢ Returns a Boolean value
- ➢ Includes meaningful comments (AI-assisted)

❖ **Expected Output**

- ➢ Correctly working prime-checking function
- ➢ Screenshots documenting Copilot's function generation
- ➢ Sample test cases and outputs



Task 4: Comparative Analysis –With vs Without Functions

❖ **Scenario**

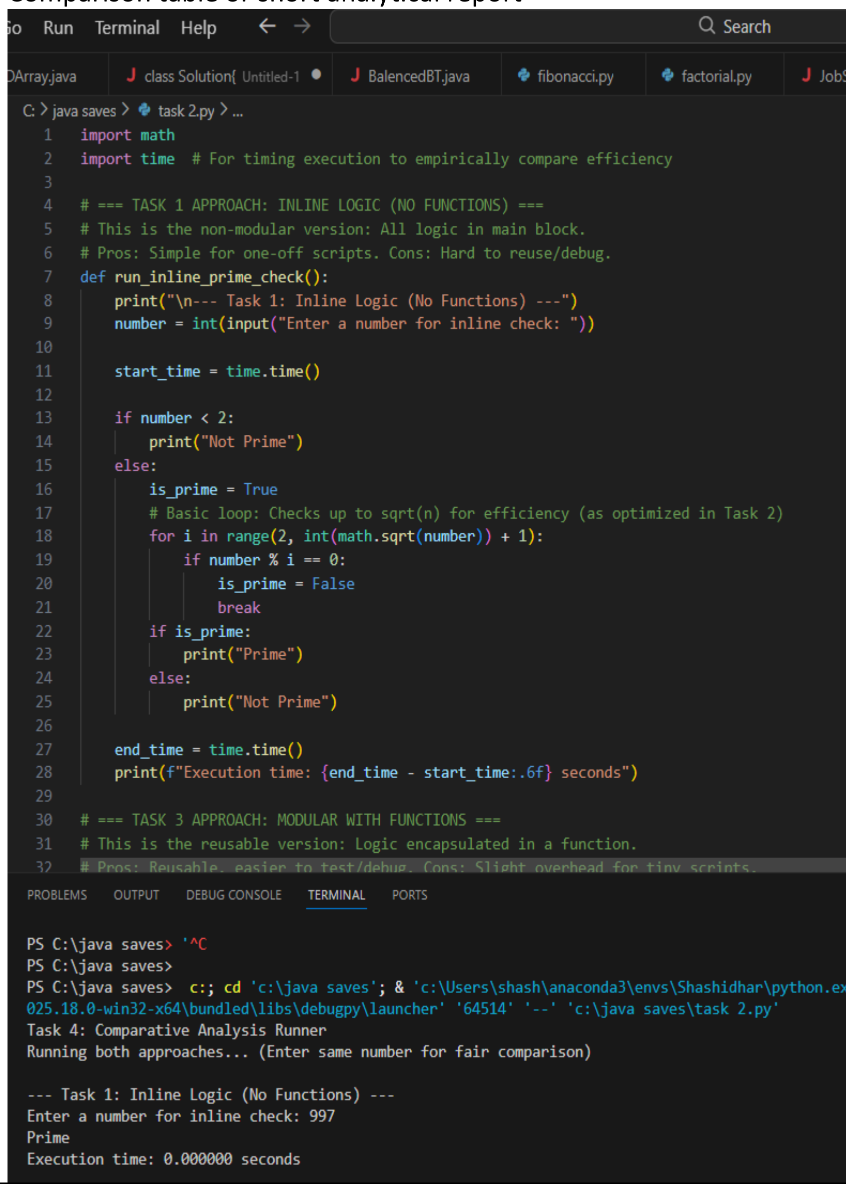
You are participating in a technical review discussion.

- **Task Description**

  Compare the Copilot-generated programs:
  - Without functions (Task 1)
  - With functions (Task 3)
  - Analyze them based on:
  - Code clarity
  - Reusability
  - Debugging ease
  - Suitability for large-scale applications

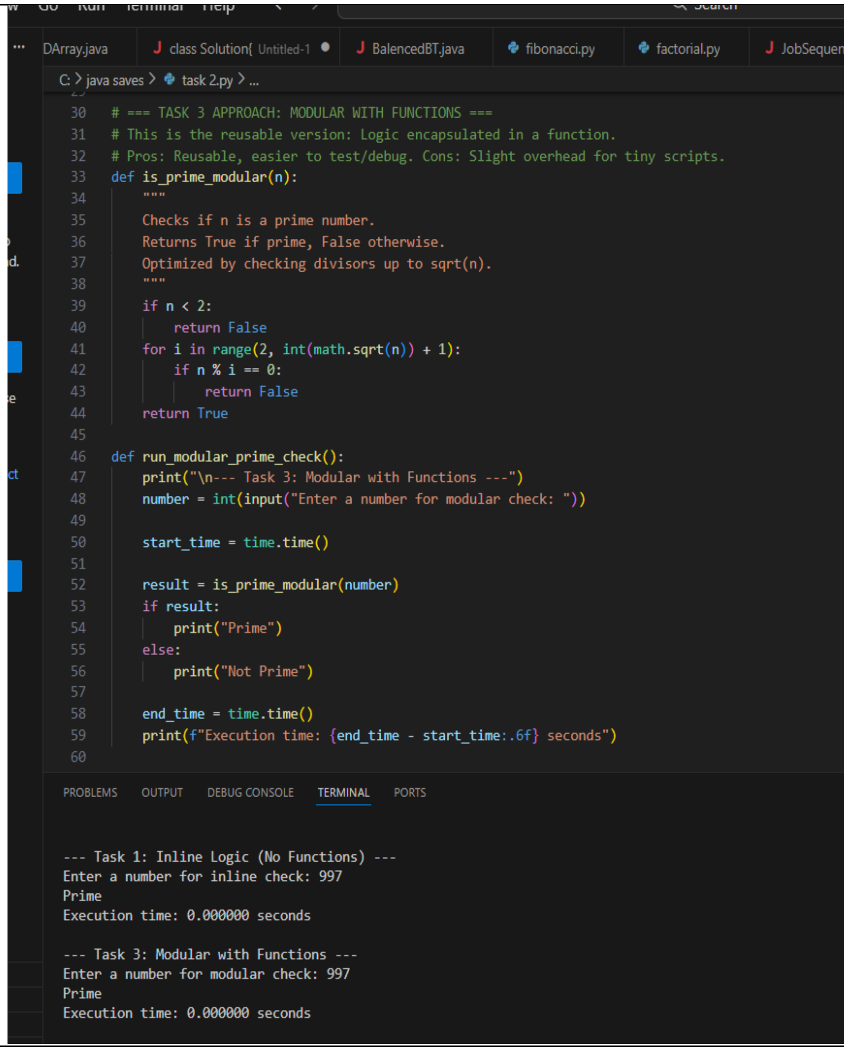
- **Expected Output**

  Comparison table or short analytical report



```python
import math
import time  # For timing execution to empirically compare efficiency

# === TASK 1 APPROACH: INLINE LOGIC (NO FUNCTIONS) ===
# This is the non-modular version: All logic in main block.
# Pros: Simple for one-off scripts. Cons: Hard to reuse/debug.
def run_inline_prime_check():
    print("\n--- Task 1: Inline Logic (No Functions) ---")
    number = int(input("Enter a number for inline check: "))

    start_time = time.time()

    if number < 2:
        print("Not Prime")
    else:
        is_prime = True
        # Basic loop: Checks up to sqrt(n) for efficiency (as optimized in Task 2)
        for i in range(2, int(math.sqrt(number)) + 1):
            if number % i == 0:
                is_prime = False
                break
        if is_prime:
            print("Prime")
        else:
            print("Not Prime")

    end_time = time.time()
    print(f"Execution time: {end_time - start_time:.6f} seconds")

# === TASK 3 APPROACH: MODULAR WITH FUNCTIONS ===
# This is the reusable version: Logic encapsulated in a function.
# Pros: Reusable, easier to test/debug. Cons: Slight overhead for tiny scripts.
```

```
PS C:\java saves> '^C
PS C:\java saves>
PS C:\java saves>  c:; cd 'c:\java saves'; & 'c:\Users\shash\anaconda3\envs\Shashidhar\python.exe
025.18.0-win32-x64\bundled\libs\debugpy\launcher' '64514' '--' 'c:\java saves\task 2.py'
Task 4: Comparative Analysis Runner
Running both approaches... (Enter same number for fair comparison)

--- Task 1: Inline Logic (No Functions) ---
Enter a number for inline check: 997
Prime
Execution time: 0.000000 seconds
```

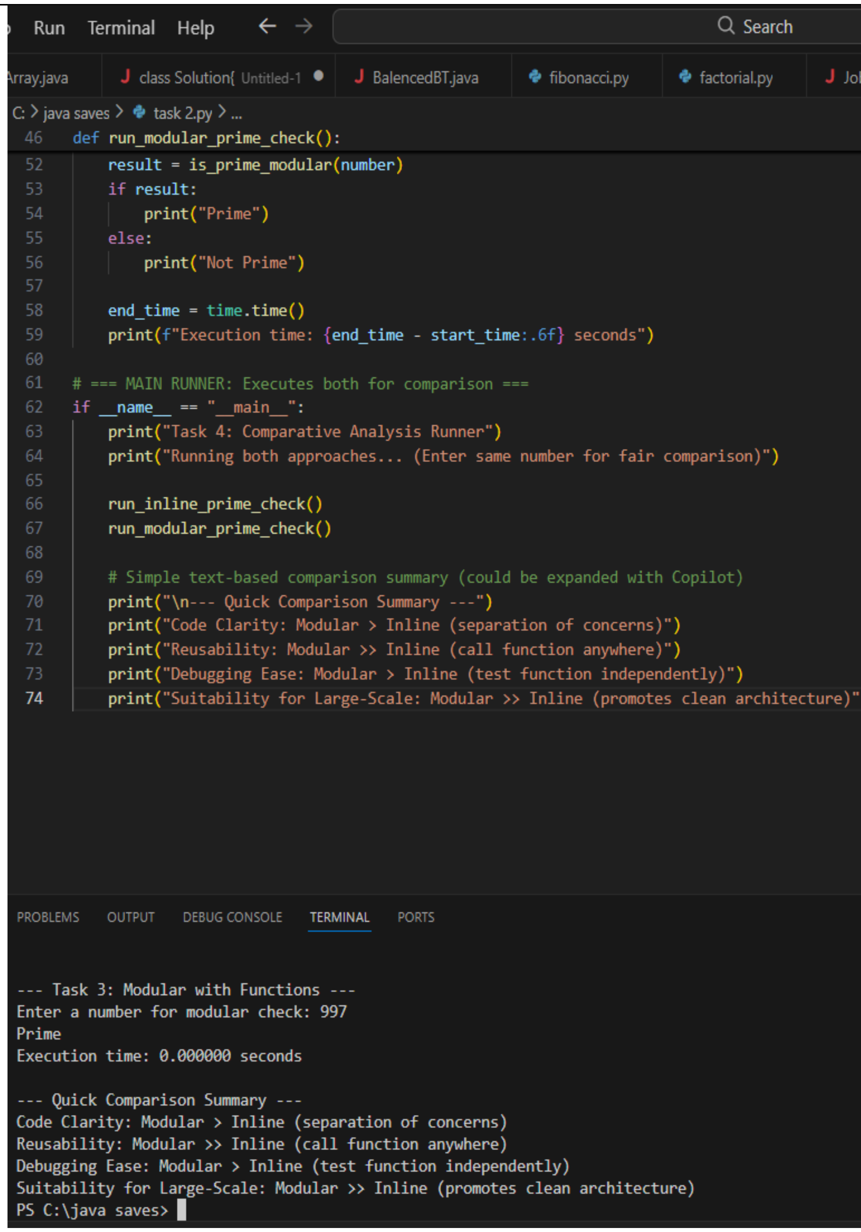

DArray.java
class Solution{ Untitled-1
BalencedBT.java
fibonacci.py
factorial.py
C: > java saves > task 2.py > ...
# === TASK 3 APPROACH: MODULAR WITH FUNCTIONS ===
# This is the reusable version: Logic encapsulated in a function.
# Pros: Reusable, easier to test/debug. Cons: Slight overhead for tiny scripts.
def is_prime_modular(n):
    """
    Checks if n is a prime number.
    Returns True if prime, False otherwise.
    Optimized by checking divisors up to sqrt(n).
    """
    if n < 2:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

def run_modular_prime_check():
    print("\n--- Task 3: Modular with Functions ---")
    number = int(input("Enter a number for modular check: "))

    start_time = time.time()

    result = is_prime_modular(number)
    if result:
        print("Prime")
    else:
        print("Not Prime")

    end_time = time.time()
    print(f"Execution time: {end_time - start_time:.6f} seconds")
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
--- Task 1: Inline Logic (No Functions) ---
Enter a number for inline check: 997
Prime
Execution time: 0.000000 seconds
--- Task 3: Modular with Functions ---
Enter a number for modular check: 997
Prime
Execution time: 0.000000 seconds



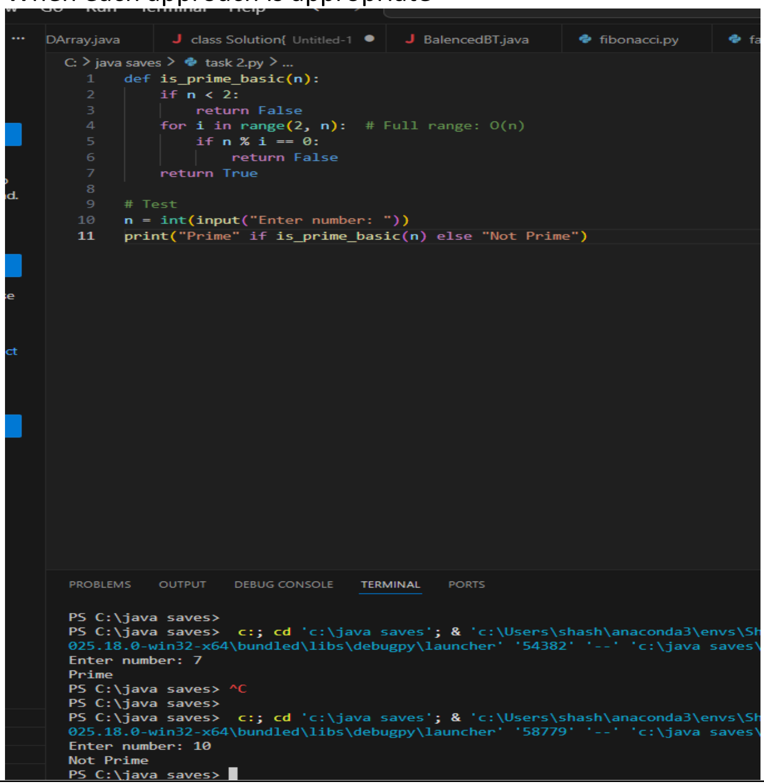
Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches to Prime Checking)

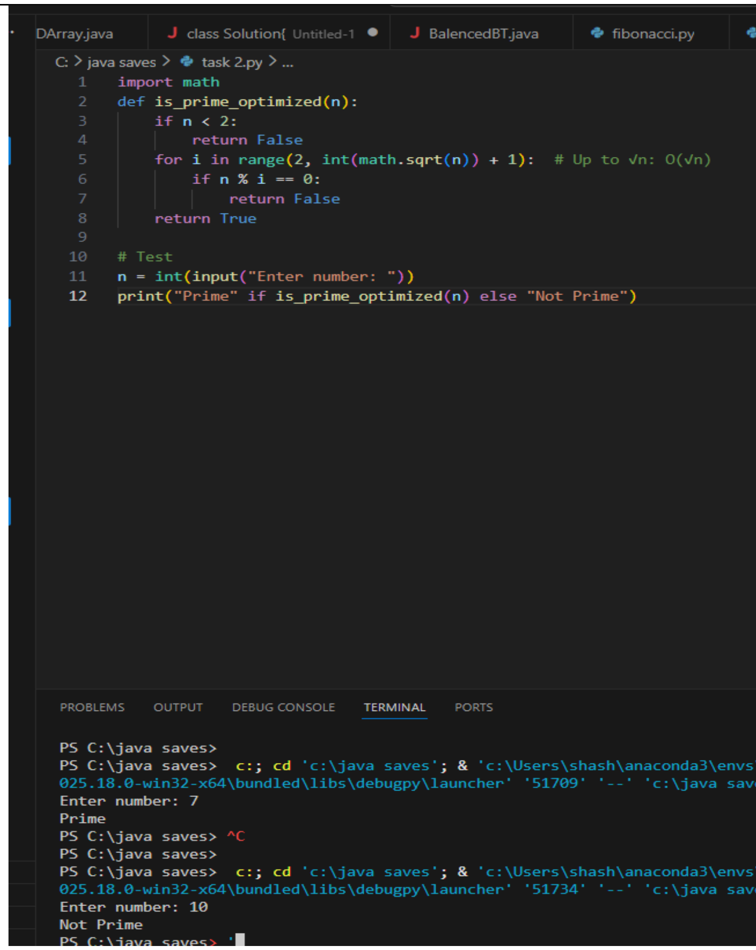
- **Scenario**
  Your mentor wants to evaluate how AI handles **alternative logical strategies**.

- **Task Description**
  Prompt GitHub Copilot to generate:
  - A **basic divisibility check** approach
  - An **optimized approach** (e.g., checking up to √n)

- **Expected Output**

- Two correct implementations
- Comparison discussing:
  - Execution flow
  - Time complexity
  - Performance for large inputs
  - When each approach is appropriate





Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots.