AI ASSISTANT CODING

Assignment – 5.5

Name: G. Ushasree

H.T No.: 2303A52431
Batch: 34

Question:

Lab Objectives:

- To explore the ethical risks associated with AI-generated
- To recognize issues related to security, bias, transparency, and copyright.
- To reflect on the responsibilities of developers when using AI tools in software development.
- To promote awareness of best practices for responsible and ethical AI coding.

Lab Outcomes (LOs):

After completing this lab, students will be able to:

- Identify and avoid insecure coding patterns generated by AI tools.
- Detect and analyze potential bias or discriminatory logic in AI-generated outputs.
- Evaluate originality and licensing concerns in reused AI-generated code.
- Understand the importance of explainability and transparency in AI-assisted programming.
- Reflect on accountability and the human role in ethical AI coding practices.

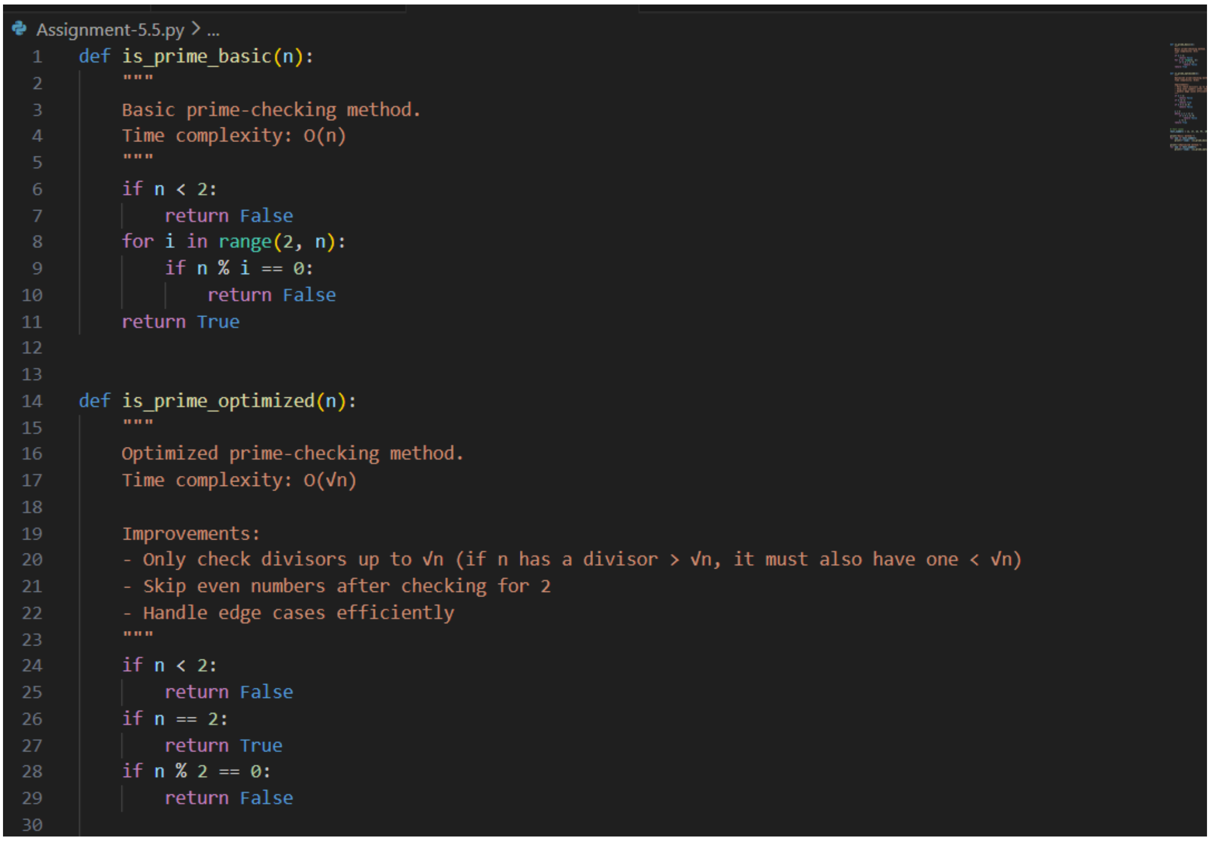
Task Description #1 (Transparency in Algorithm Optimization)

Task: Use AI to generate two solutions for checking prime numbers:

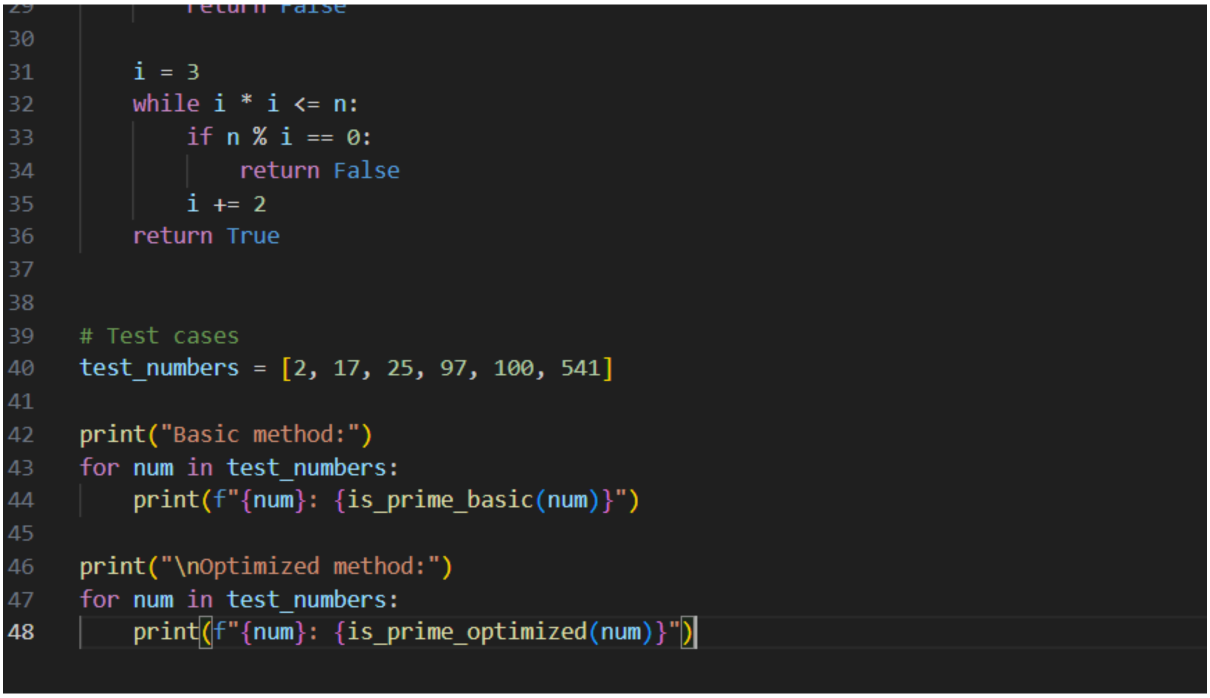
- Naive approach(basic)
- Optimized approach

Prompt:

“Generate Python code for two prime-checking methods and explain how the optimized version improves performance.”



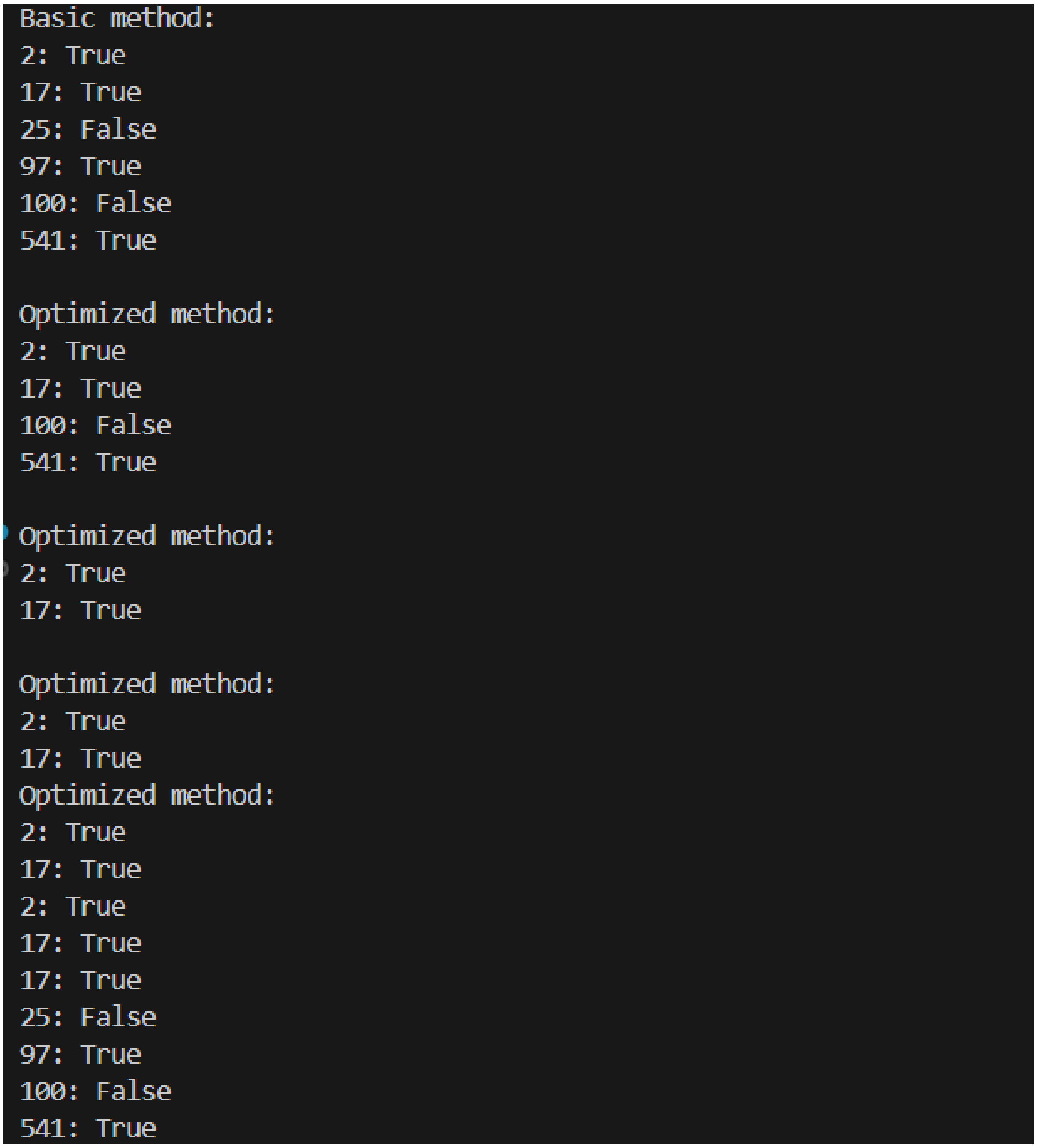
```python
def is_prime_basic(n):
    """
    Basic prime-checking method.
    Time complexity: O(n)
    """
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True


def is_prime_optimized(n):
    """
    Optimized prime-checking method.
    Time complexity: O(√n)

    Improvements:
    - Only check divisors up to √n (if n has a divisor > √n, it must also have one < √n)
    - Skip even numbers after checking for 2
    - Handle edge cases efficiently
    """
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

```



```
        return False

    i = 3
    while i * i <= n:
        if n % i == 0:
            return False
        i += 2
    return True


# Test cases
test_numbers = [2, 17, 25, 97, 100, 541]

print("Basic method:")
for num in test_numbers:
    print(f"{num}: {is_prime_basic(num)}")

print("\nOptimized method:")
for num in test_numbers:
    print(f"{num}: {is_prime_optimized(num)}")
```

Expected Output:

- Code for both methods.
- Transparent explanation of time complexity.
- Comparison highlighting efficiency improvements.



```
Basic method:
2: True
17: True
25: False
97: True
100: False
541: True

Optimized method:
2: True
17: True
100: False
541: True

Optimized method:
2: True
17: True

Optimized method:
2: True
17: True
Optimized method:
2: True
17: True
2: True
17: True
17: True
25: False
97: True
100: False
541: True
```

Task Description #2 (Transparency in Recursive Algorithms)

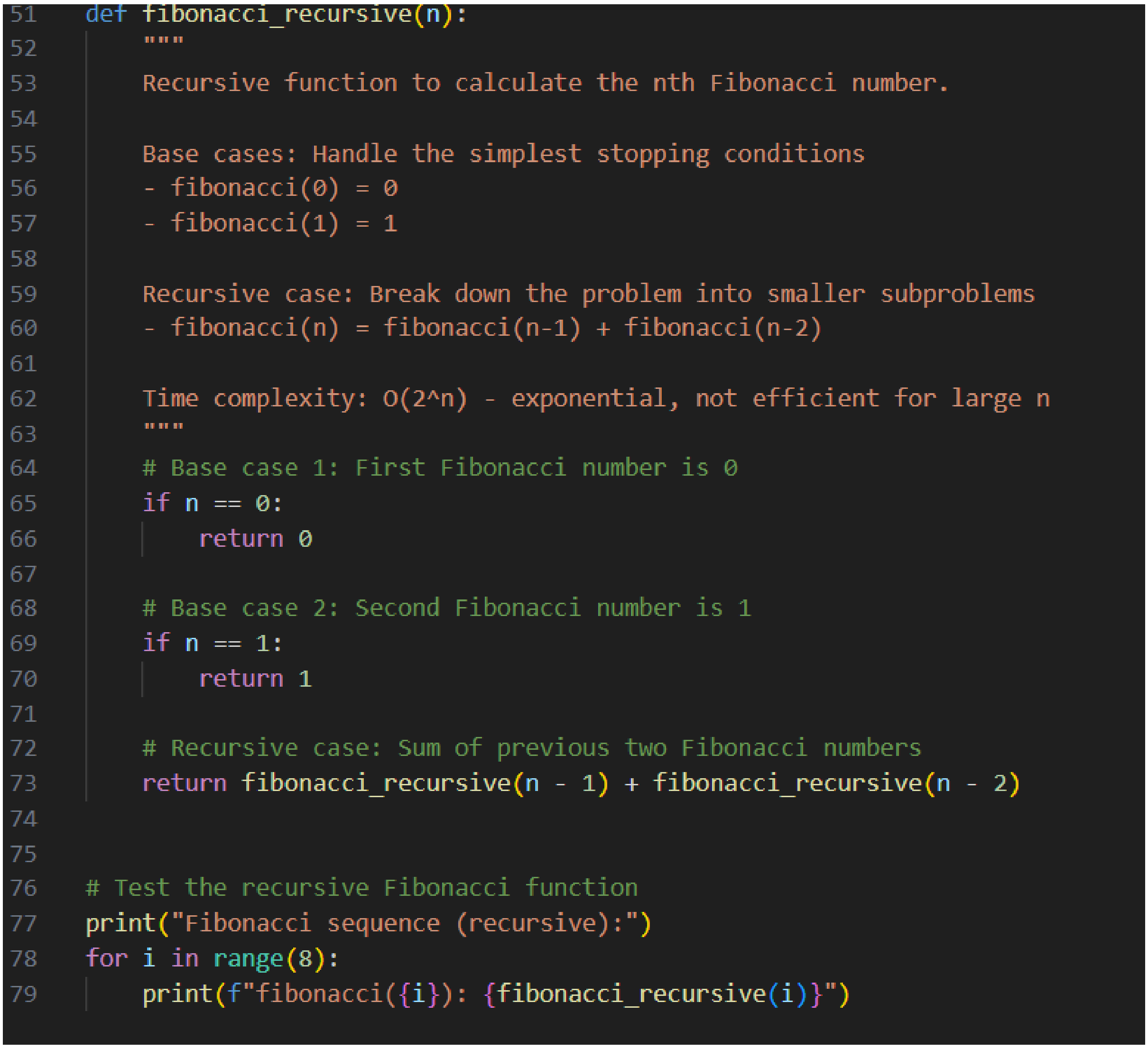
Objective: Use AI to generate a recursive function to calculate Fibonacci numbers.

Instructions:

1. Ask AI to add clear comments explaining recursion.
2. Ask AI to explain base cases and recursive calls.

Prompt:

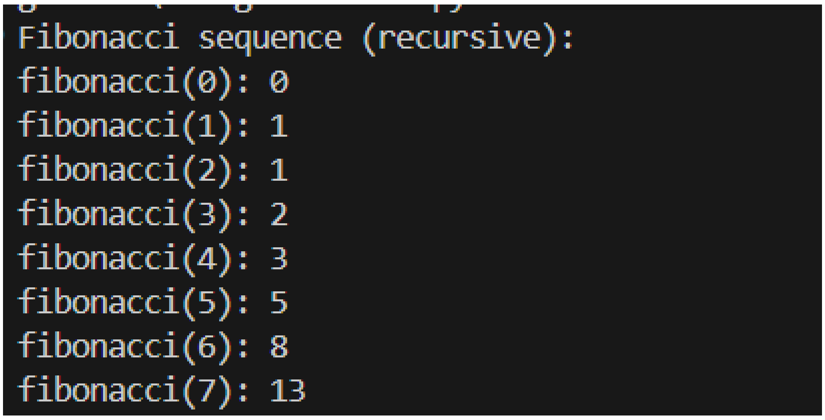
Generate a recursive function to calculate Fibonacci numbers by adding clear comments explaining recursion and by explaining base cases and recursive calls



```
def fibonacci_recursive(n):
    """
    Recursive function to calculate the nth Fibonacci number.

    Base cases: Handle the simplest stopping conditions
    - fibonacci(0) = 0
    - fibonacci(1) = 1

    Recursive case: Break down the problem into smaller subproblems
    - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)

    Time complexity: O(2^n) - exponential, not efficient for large n
    """
    # Base case 1: First Fibonacci number is 0
    if n == 0:
        return 0

    # Base case 2: Second Fibonacci number is 1
    if n == 1:
        return 1

    # Recursive case: Sum of previous two Fibonacci numbers
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)


# Test the recursive Fibonacci function
print("Fibonacci sequence (recursive):")
for i in range(8):
    print(f"fibonacci({i}): {fibonacci_recursive(i)}")
```

Expected Output:

- Well-commented recursive code.
- Clear explanation of how recursion works.
- Verification that explanation matches actual execution.



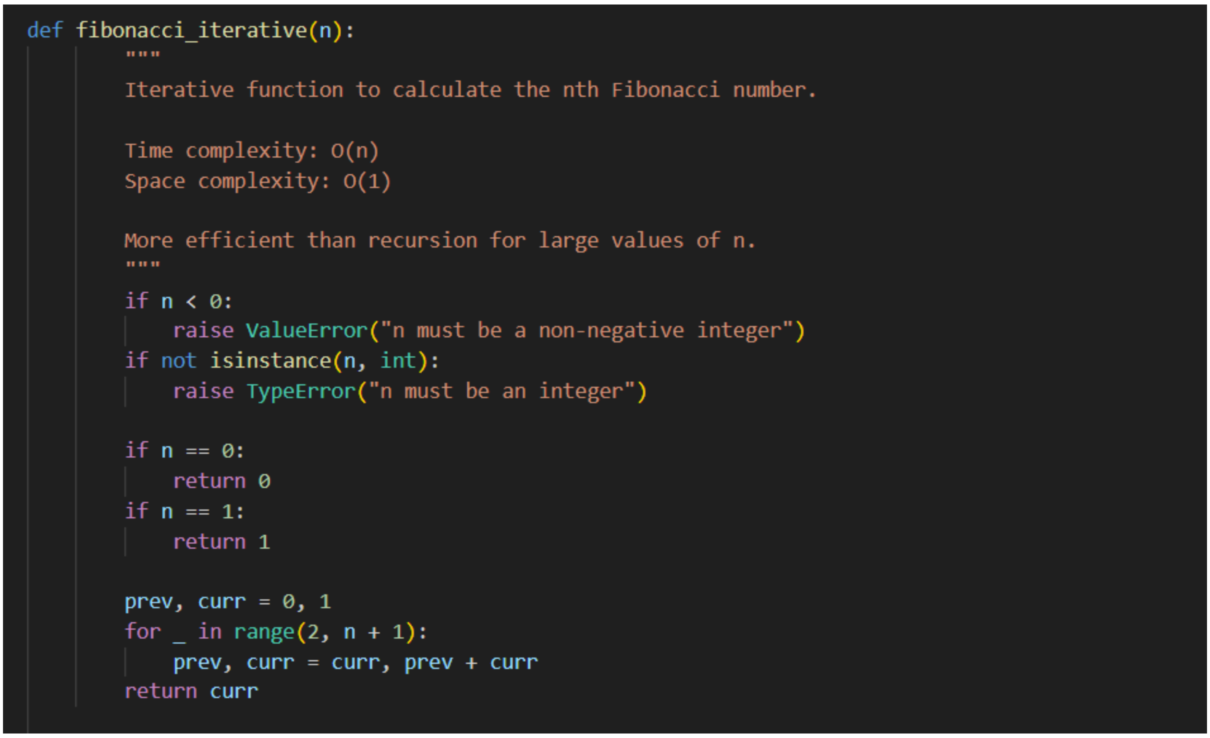
```
Fibonacci sequence (recursive):
fibonacci(0): 0
fibonacci(1): 1
fibonacci(2): 1
fibonacci(3): 2
fibonacci(4): 3
fibonacci(5): 5
fibonacci(6): 8
fibonacci(7): 13
```

Task Description #3 (Transparency in Error Handling)

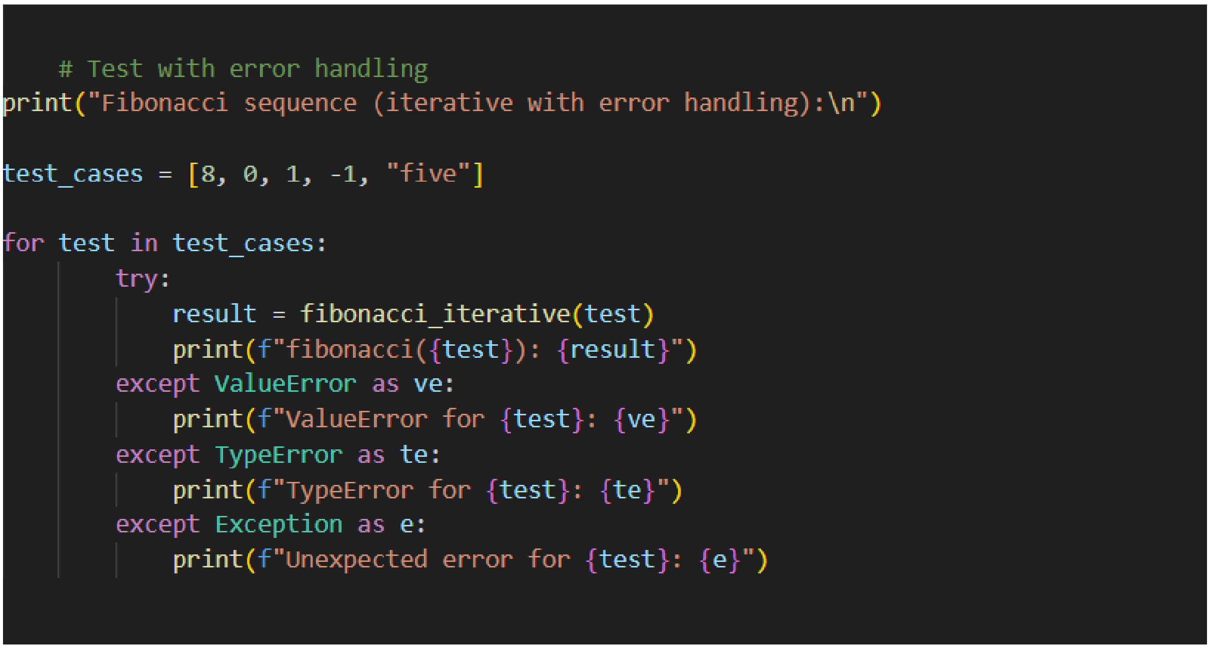
Task: Use AI to generate a Python program that reads a file and processes data.

Prompt:

“Generate code with proper error handling and clear explanations for each exception.”



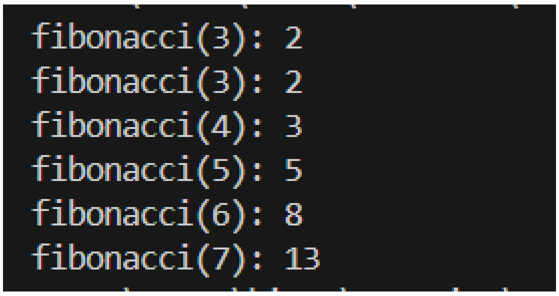
```python
def fibonacci_iterative(n):
        """
        Iterative function to calculate the nth Fibonacci number.

        Time complexity: O(n)
        Space complexity: O(1)

        More efficient than recursion for large values of n.
        """
        if n < 0:
            raise ValueError("n must be a non-negative integer")
        if not isinstance(n, int):
            raise TypeError("n must be an integer")

        if n == 0:
            return 0
        if n == 1:
            return 1

        prev, curr = 0, 1
        for _ in range(2, n + 1):
            prev, curr = curr, prev + curr
        return curr
```



```
    # Test with error handling
print("Fibonacci sequence (iterative with error handling):\n")

test_cases = [8, 0, 1, -1, "five"]

for test in test_cases:
        try:
            result = fibonacci_iterative(test)
            print(f"fibonacci({test}): {result}")
        except ValueError as ve:
            print(f"ValueError for {test}: {ve}")
        except TypeError as te:
            print(f"TypeError for {test}: {te}")
        except Exception as e:
            print(f"Unexpected error for {test}: {e}")
```

Expected Output:

- Code with meaningful exception handling.
- Clear comments explaining each error scenario.
- Validation that explanations align with runtime behavior.



```
fibonacci(3): 2
fibonacci(3): 2
fibonacci(4): 3
fibonacci(5): 5
fibonacci(6): 8
fibonacci(7): 13
```

Task Description #4 (Security in User Authentication)

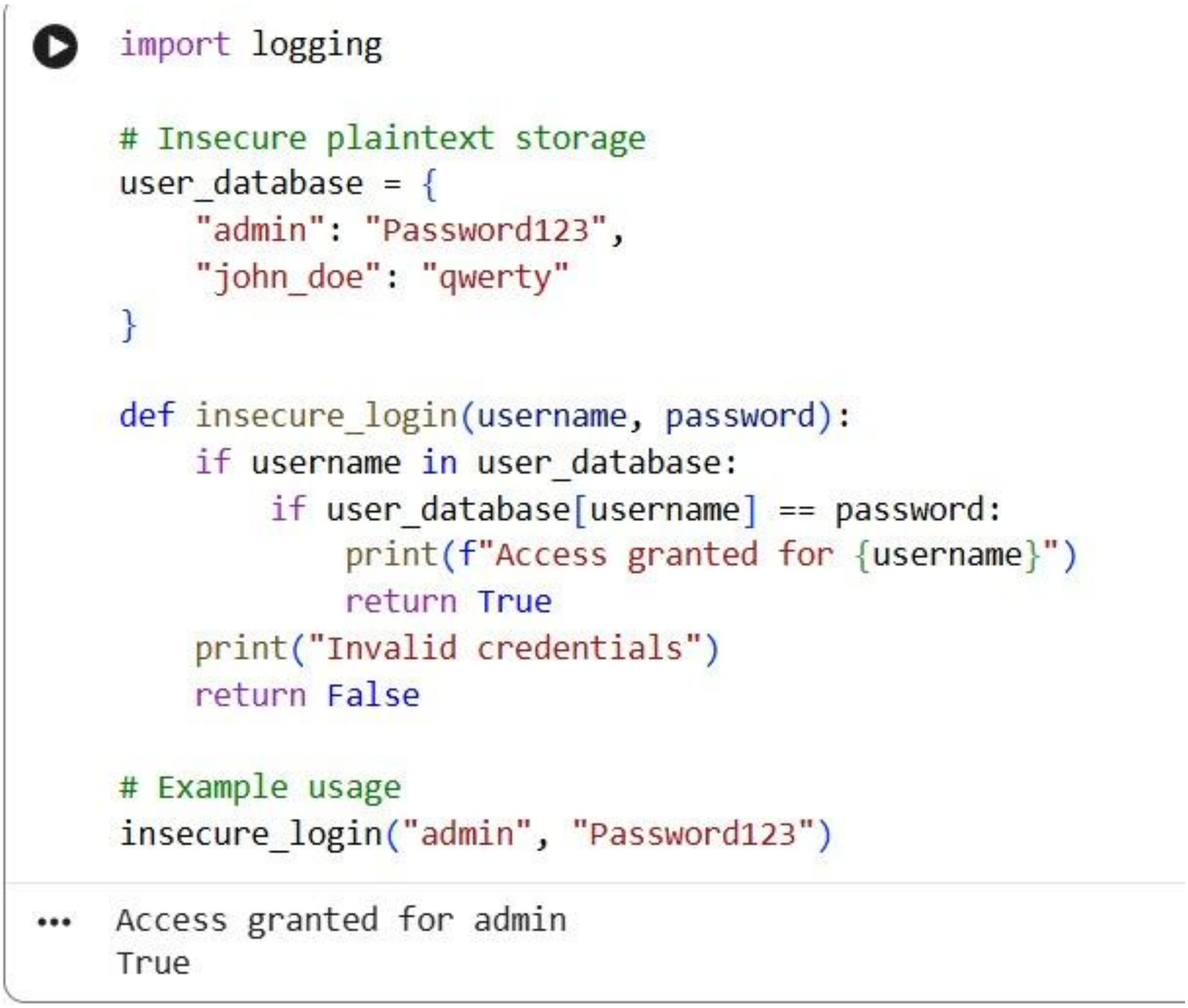
Task: Use an AI tool to generate a Python-based login system.
Analyze: Check whether the AI uses secure password handling practices.

Prompt:

Generate a Python-based login system. Analyze whether it uses secure password handling by checking for plain-text password storage, weak validation, or missing hashing. Identify security flaws, then provide a revised version using password hashing and basic input validation.

1. Insecure Login System (Initial Version)

This script demonstrates a common but high-risk approach where sensitive credentials are stored and compared in cleartext.



```
import logging

# Insecure plaintext storage
user_database = {
    "admin": "Password123",
    "john_doe": "qwerty"
}

def insecure_login(username, password):
    if username in user_database:
        if user_database[username] == password:
            print(f"Access granted for {username}")
            return True
    print("Invalid credentials")
    return False

# Example usage
insecure_login("admin", "Password123")
```

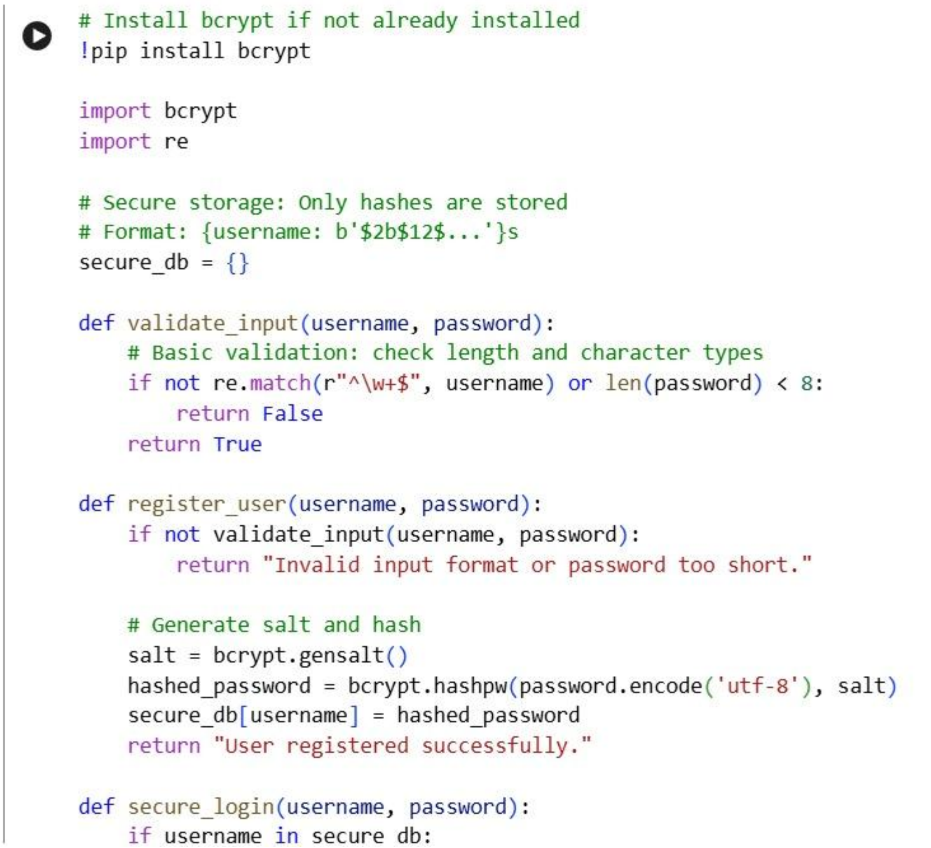
```
Access granted for admin
True
```

2. Security Analysis & Flaw Identification

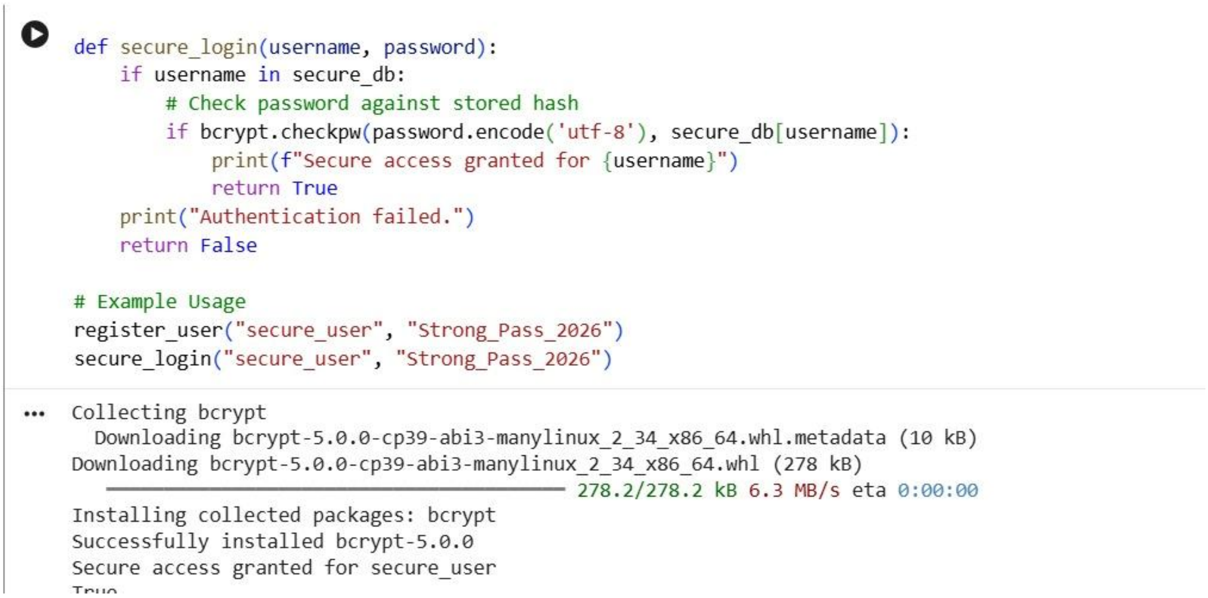
The implementation exhibits major security weaknesses. Passwords are stored in plain text, meaning any data breach would fully expose user accounts. The absence of input validation allows empty or malformed inputs, increasing injection risk. Additionally, passwords are not hashed, eliminating secrecy and exposing credentials even to administrators. Storing raw passwords also violates data minimization principles, as authentication requires verification—not retention—of actual passwords.

3. Revised Secure Login System

This version implements Privacy by Design using the bcrypt library for salted password hashing and standardized input validation.



```
# Install bcrypt if not already installed
!pip install bcrypt

import bcrypt
import re

# Secure storage: Only hashes are stored
# Format: {username: b'$2b$12$...'}s
secure_db = {}

def validate_input(username, password):
    # Basic validation: check length and character types
    if not re.match(r"^\w+$", username) or len(password) < 8:
        return False
    return True

def register_user(username, password):
    if not validate_input(username, password):
        return "Invalid input format or password too short."

    # Generate salt and hash
    salt = bcrypt.gensalt()
    hashed_password = bcrypt.hashpw(password.encode('utf-8'), salt)
    secure_db[username] = hashed_password
    return "User registered successfully."

def secure_login(username, password):
    if username in secure_db:
```



```
def secure_login(username, password):
    if username in secure_db:
        # Check password against stored hash
        if bcrypt.checkpw(password.encode('utf-8'), secure_db[username]):
            print(f"Secure access granted for {username}")
            return True
    print("Authentication failed.")
    return False

# Example Usage
register_user("secure_user", "Strong_Pass_2026")
secure_login("secure_user", "Strong_Pass_2026")
```

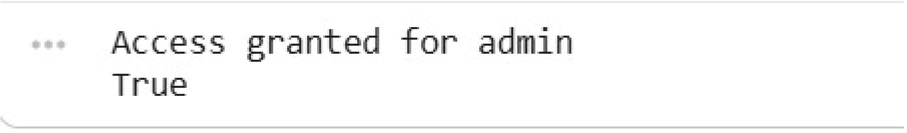
```
Collecting bcrypt
  Downloading bcrypt-5.0.0-cp39-abi3-manylinux_2_34_x86_64.whl.metadata (10 kB)
Downloading bcrypt-5.0.0-cp39-abi3-manylinux_2_34_x86_64.whl (278 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 278.2/278.2 kB 6.3 MB/s eta 0:00:00
Installing collected packages: bcrypt
Successfully installed bcrypt-5.0.0
Secure access granted for secure_user
True
```

4. Privacy-Aware Principles

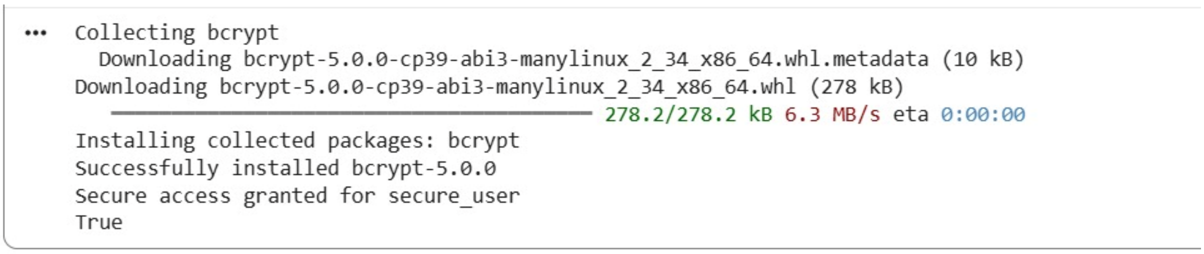
A secure authentication system should follow three key practices: data minimization, by storing only password hashes and never plain passwords; salted hashing, using unique salts per user to prevent rainbow table attacks; and secure storage, ensuring hashed credentials are encrypted at rest and protected by strict access controls.

Expected Output:

- Identification of security flaws (plain-text passwords, weak validation).
- Revised version using password hashing and input validation.
- Short note on best practices for secure authentication.



```
Access granted for admin
True
```



```
Collecting bcrypt
  Downloading bcrypt-5.0.0-cp39-abi3-manylinux_2_34_x86_64.whl.metadata (10 kB)
Downloading bcrypt-5.0.0-cp39-abi3-manylinux_2_34_x86_64.whl (278 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 278.2/278.2 kB 6.3 MB/s eta 0:00:00
Installing collected packages: bcrypt
Successfully installed bcrypt-5.0.0
Secure access granted for secure_user
True
```

Task Description #5 (Privacy in Data Logging)

Task: Use an AI tool to generate a Python script that logs user activity (username, IP address, timestamp).

Analyze: Examine whether sensitive data is logged unnecessarily or insecurely.

Prompt:

Role: Act as a Cybersecurity and Data Privacy Expert.

Task Part 1 (Code Generation):

Write a Python script using the logging library to track user activity. The script should capture the username, IP address, and timestamp whenever a user accesses a specific function or endpoint.

Task Part 2 (Privacy Analysis):

Analyze the script you just wrote for privacy risks. Specifically, evaluate:

Whether Personally Identifiable Information (PII) is being logged unnecessarily.

The security of the log storage (e.g., plaintext vs. encrypted).

Compliance risks regarding regulations like GDPR or CCPA.

Task Part 3 (Improved Version):

Provide an optimized version of the script that implements "Privacy by Design." This version should:

Apply minimalism (only log what is essential).

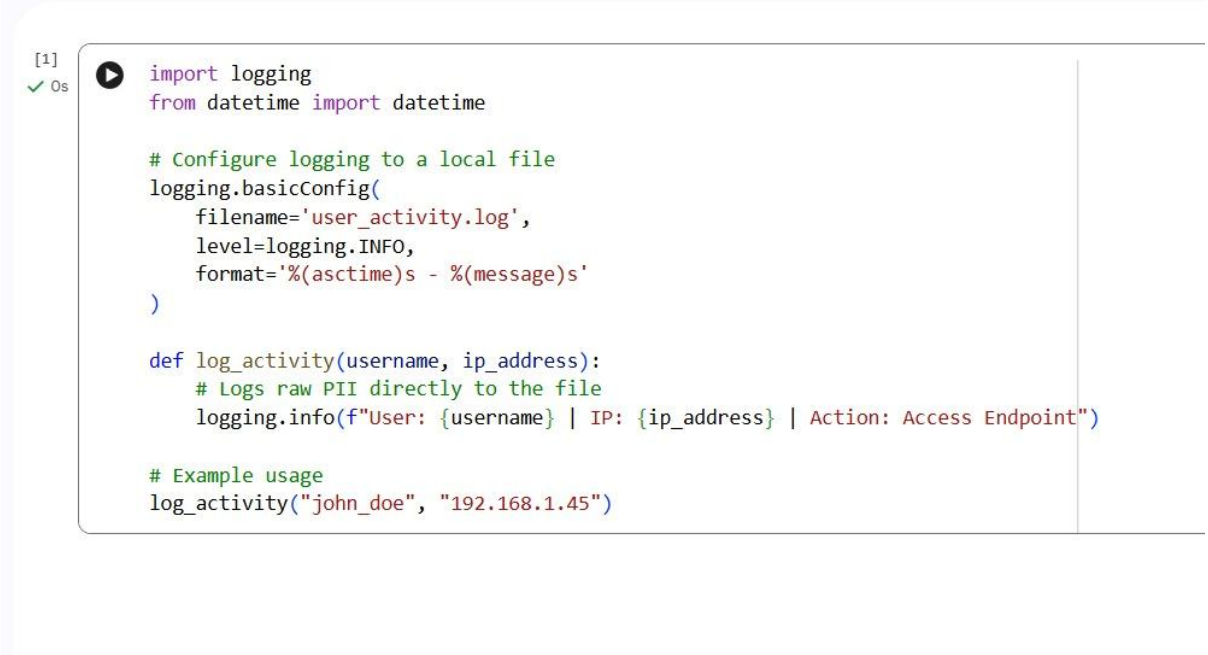
Use anonymization or masking (e.g., hashing usernames or masking IP octets).

Ensure sensitive data is not stored in cleartext.

Task Part 4 (Education):

Explain the core principles of Privacy-Aware Logging, focusing on Data Minimization, Purpose Limitation, and Secure Storage.

1: Initial Python Script (Standard Logging)



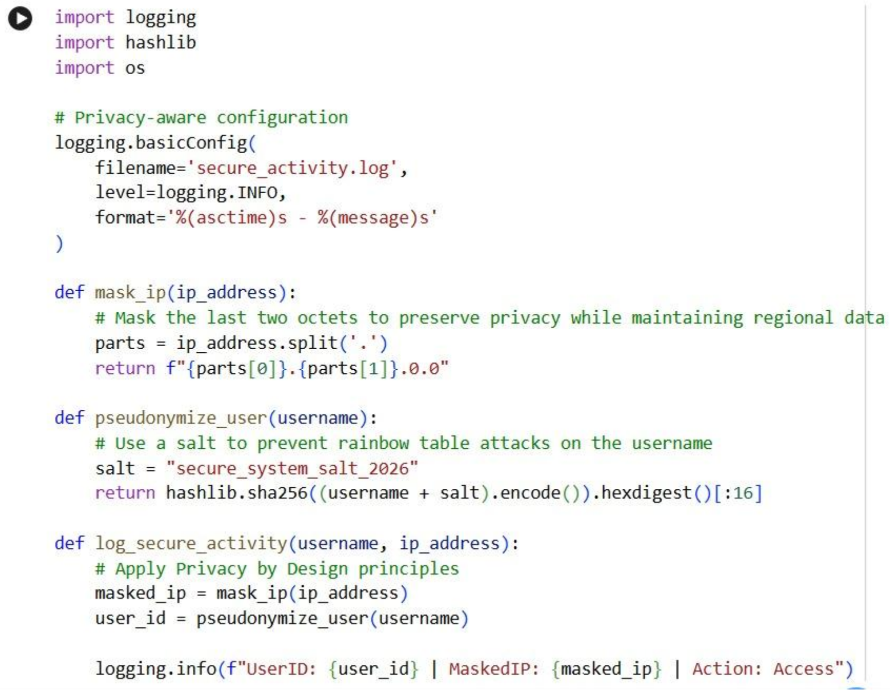
```
import logging
from datetime import datetime

# Configure logging to a local file
logging.basicConfig(
    filename='user_activity.log',
    level=logging.INFO,
    format='%(asctime)s - %(message)s'
)

def log_activity(username, ip_address):
    # Logs raw PII directly to the file
    logging.info(f"User: {username} | IP: {ip_address} | Action: Access Endpoint")

# Example usage
log_activity("john_doe", "192.168.1.45")
```

2: Privacy Analysis & Risk Assessment

The script introduces privacy and security risks by logging raw usernames and full IP addresses, resulting in unnecessary exposure of PII. Storing this data in plaintext increases the impact of a potential breach. Additionally, this practice may violate GDPR, which classifies IP addresses as personal data, and CCPA, which increases compliance burden for data access and deletion requests.

3: Improved Version (Privacy by Design)

This version implements Data Minimization and Pseudonymization using SHA-256 hashing for usernames and octet masking for IP addresses



```python
import logging
import hashlib
import os

# Privacy-aware configuration
logging.basicConfig(
    filename='secure_activity.log',
    level=logging.INFO,
    format='%(asctime)s - %(message)s'
)

def mask_ip(ip_address):
    # Mask the last two octets to preserve privacy while maintaining regional data
    parts = ip_address.split('.')
    return f"{parts[0]}.{parts[1]}.0.0"

def pseudonymize_user(username):
    # Use a salt to prevent rainbow table attacks on the username
    salt = "secure_system_salt_2026"
    return hashlib.sha256((username + salt).encode()).hexdigest()[:16]

def log_secure_activity(username, ip_address):
    # Apply Privacy by Design principles
    masked_ip = mask_ip(ip_address)
    user_id = pseudonymize_user(username)

    logging.info(f"UserID: {user_id} | MaskedIP: {masked_ip} | Action: Access")
```

4: Principles of Privacy-Aware Logging

To maintain compliance and security, developers should follow three core principles. Data minimization requires collecting only what is strictly necessary—avoid logging real identities and use hashes or tokens when differentiation is needed. Purpose limitation ensures logs are used solely for their original intent, such as security or debugging; repurposing them without consent can trigger regulatory penalties. Secure storage and retention demands encrypting sensitive logs and enforcing automatic deletion policies (e.g., 30–90 days) to reduce breach impact.