

# **School of Computer Science and Artificial Intelligence**

---

## **Lab Assignment -5.5**

---

**Program** :B. Tech (CSE)  
**Specialization** :AIML  
**Course Title** : AI ASSISTED CODING  
**Course Code** : 23CS002PC304  
**Semester** : VI  
**Academic Session** : 2025-2026  
**Name of Student** : PASHIKANTI KARTHISHA  
**Enrollment No.** : 2303A52099  
**Batch No.** : 33

### **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.”

#### **Expected Output:**

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

#### **Naive approach(basic)**

The screenshot shows the AIAC IDE interface. On the left, the file tree displays various Python files under the 'AIAC' project. The main editor window shows the code for 'primechecking\_naiveapproach.py'. The right side features a 'Prime checking methods and complexity' panel with the following content:

```

Prime checking methods and complexity
4m - +135 - Auto

Write Python code for prime checking using:
1) Naive method (2 to n)
2) Optimized method (2 to sqrt(n), skipping evens)

• Checks only up to  $\sqrt{n}$ 
• Skips even numbers (except 2)
• Time complexity:  $O(\sqrt{n})$ 
• For n=1000, performs ~15 checks (only odd numbers from 3 to 31)

Why the optimized method is faster
1. Only checks up to  $\sqrt{n}$ : if  $n = a \times b$  and both a and b are >  $\sqrt{n}$ , then  $a \times b > n$ , which is impossible. So at least one factor must be  $\leq \sqrt{n}$ .
2. Skips evens: after checking 2, all other even numbers are composite, roughly halving the checks.

For n=1000, the optimized method is about 66x faster (~15 checks vs ~998).
The file includes both functions, test cases, and a time complexity explanation. Both methods produce the same results, as shown in the test output.

```

The terminal window below shows the execution of the code and its output.

## Time Complexity (Naive)

- Worst case (when n is prime): loop runs **n-2 times**
- Time Complexity:  $O(n)$**

## Optimized approach

The screenshot shows the AIAC IDE interface. The file tree on the left shows the 'AIAC' project with various files. The main editor window displays the code for 'primechecking\_OptimizedApproach.py'. The right side contains the same 'Prime checking methods and complexity' panel as the previous screenshot, with the following content:

```

Prime checking methods and complexity
6m - +135 - Auto

Write Python code for prime checking using:
1) Naive method (2 to n)
2) Optimized method (2 to sqrt(n), skipping evens)

• Checks only up to  $\sqrt{n}$ 
• Skips even numbers (except 2)
• Time complexity:  $O(\sqrt{n})$ 
• For n=1000, performs ~15 checks (only odd numbers from 3 to 31)

Why the optimized method is faster
1. Only checks up to  $\sqrt{n}$ : if  $n = a \times b$  and both a and b are >  $\sqrt{n}$ , then  $a \times b > n$ , which is impossible. So at least one factor must be  $\leq \sqrt{n}$ .
2. Skips evens: after checking 2, all other even numbers are composite, roughly halving the checks.

For n=1000, the optimized method is about 66x faster (~15 checks vs ~998).
The file includes both functions, test cases, and a time complexity explanation. Both methods produce the same results, as shown in the test output.

```

The terminal window at the bottom shows the execution of the optimized code and its output.

### Time Complexity (Optimized)

- Loop runs only up to  $\sqrt{n}$
- **Time Complexity:  $O(\sqrt{n})$**

### Comparison (Efficiency Improvement)

Method	Checks up to	Time Complexity	Speed
Naive	$n-1$	$O(n)$	Slow for large n
Optimized	$\sqrt{n}$	$O(\sqrt{n})$	Much faster

Because if  $n = a \times b$ , then **at least one factor must be  $\leq \sqrt{n}$** , so checking beyond  $\sqrt{n}$  is unnecessary → fewer iterations → faster performance.

---

### 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.

**Expected Output:**

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

The screenshot shows the AIAC (AI Assistant for Code) application. On the left, the file tree displays various Python files under the 'AIAC' folder. The main area shows two tabs: 'RecursiveFibonacciProgramWellCommented.py' and 'primechecker\_OptimizedApproach.py'. The 'RecursiveFibonacciProgramWellCommented.py' tab contains the following code:

```

1 def fib(n):
2     if n == 0:
3         return 0
4
5     # Base Case 2
6     if n == 1:
7         return 1
8
9     # Recursive call
10    return fib(n - 1) + fib(n - 2)
11
12 n = int(input("Enter n: "))
13 result = fib(n)
14
15 if result is not None:
16     print("Fibonacci number is:", result)
17
18
19
20
21
22
23
24

```

The terminal below shows the execution of the program:

```

PS D:\3-2\AIAC> & c:/Users/pashi/AppData/Local/Microsoft/WindowsApps/python3.11.exe d:/3-2/AIAC/RecursiveFibonacciProgramWellCommented.py
Enter n: 2
Fibonacci number is: 1
PS D:\3-2\AIAC> & c:/Users/pashi/AppData/Local/Microsoft/WindowsApps/python3.11.exe d:/3-2/AIAC/RecursiveFibonacciProgramWellCommented.py
Enter n: 3
Fibonacci number is: 2
PS D:\3-2\AIAC>

```

The right side of the interface features an AI panel titled 'Prime checking methods and complexity' which provides a comparison between a naive method and an optimized method for prime checking.

## How Recursion Works (Fibonacci)

Recursion means **a function calls itself** to solve a problem by breaking it into **smaller subproblems**.

In Fibonacci:

$$fib(n) = fib(n - 1) + fib(n - 2)$$

So the function keeps calling itself until it reaches a **base case**.

### 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.”

**Expected Output:**

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

The screenshot shows the AIAC (AI Assistant for Code) interface. On the left, there's a file tree with various Python files like `fibonacci.py`, `prime\_checker.py`, etc. The main area displays a Python script for handling file operations:

```

1  # Transparency in Error Handling.py > read_and_process_file
2  def read_and_process_file(filename):
3      # Raises when file does not exist
4      print("Error: File not found. Please check the file name or path.")
5
6  except FileNotFoundError:
7      # Raised when file exists but access is denied
8      print("Error: Permission denied. You do not have access to this file.")
9
10 except UnicodeDecodeError:
11     # Raised when file encoding cannot be decoded
12     print("Error: Unable to decode file content. Encoding issue.")
13
14 except Exception as e:
15     # Handles any other unexpected errors
16     print("Unexpected error occurred:", e)
17
18 else:
19     # Executes only if no exception occurs
20     print("File processed successfully.")
21
22 finally:
23     # Always executes (cleanup or final message)
24     print("Program execution completed.")
25

```

Below the code editor is a terminal window showing command-line interactions with the script. The right side of the interface has a sidebar for generating Python programs, managing agents, and viewing recent activity.

### Validation: Explanation vs Runtime Behavior

Scenario	Runtime Result	Explanation Matches?
File not found	FileNotFoundException message	<input checked="" type="checkbox"/> Yes
Permission denied	PermissionError message	<input checked="" type="checkbox"/> Yes
Encoding issue	UnicodeDecodeError message	<input checked="" type="checkbox"/> Yes
No error	File content + success message	<input checked="" type="checkbox"/> Yes
Any other error	Generic exception message	<input checked="" type="checkbox"/> Yes

### 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.

#### 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.

The screenshot shows the AIAC application interface. On the left, there's a file tree with various Python files under 'AIAC'. The main area displays two tabs of Python code: 'Transparency in Error Handling.py' and 'Revised Secure Version (Using Hashing & Validation).py'. The right side features a 'Review Next File' button, a 'Tasks' section with a numbered list of security-related items, and a 'Agents' sidebar with recent activity. At the bottom, a terminal window shows the command '11.exe "d:/3-2/AIAC/Revised Secure Version (Using Hashing & Validation).py"' being run, with the output 'Enter username: abc' and 'Enter password: bdgudy'.

## Why This Version Is Secure

### Improvements Made

- Password hashing (SHA-256)**  
→ Stored passwords are not readable.
- Input validation**  
→ Prevents empty and weak inputs.
- No plain-text comparison**  
→ Compares hashed values only.
- Better resistance to attacks**  
→ Protects user credentials.

### Best Practices for Secure Authentication (Short Note)

- Always hash passwords (use bcrypt, argon2, or sha256).
- Never store passwords in plain text.
- Validate inputs to prevent weak credentials.
- Use salting for extra security.
- Avoid hardcoding credentials.
- Limit login attempts to prevent brute-force attacks.

### 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.

### Expected Output:

- Identified privacy risks in logging.
- Improved version with minimal, anonymized, or masked logging.
- Explanation of privacy-aware logging principles.

The screenshot shows the AIAC (AI Assistant for Code) IDE interface. On the left, the file tree displays various Python files under the 'AINC' folder, including 'activity.log', 'calculator.py', 'coding\_queries\_dataset...', 'customer\_support\_e...', 'evenodd.py', 'Fact.py', 'fewshot\_armstrong.py', 'fewshot evenodd.py', 'Improved Privacy Aw...', 'oneshot\_factory.py', 'prime\_checker.py', 'primechecking\_naive...', 'primechecking\_Optim...', 'primesnumber.py', 'prompting\_technique...', 'prompting\_technique...', 'prompting\_technique...', 'reversedfibonacci.py', 'Revised Secure Versio...', 'shape\_area.py', 'simple\_marks\_sorting...', 'simple\_prime\_checker...', 'social\_media\_post\_cla...', 'social\_media\_posts\_d...', 'Transparency in Error...', 'travel\_queries\_dataset...', 'travel\_query\_classifier...', 'unit\_conversion.py', 'zero\_shotpalindrome.py', 'zeroshot perfectnum...'. The main editor window shows a Python script titled 'Improved Privacy-Aware Logging (Masked & Minimal).py'. The code defines a function 'log\_user\_activity' that takes 'username' and 'ip\_address' as parameters. It first masks the IP address by taking the last octet and returning a string where the last octet is replaced by 'xxx'. Then it hashes the username using SHA-256 and returns the hashed value. A tooltip on the right provides instructions: 'Generate a Python script that logs user activity including username, IP address, and timestamp.' Below the code, a terminal window shows the command 'PS D:\3-2\AIAC> & C:/Users/pashi/AppData/Local/Microsoft/WindowsApps/python3.11.exe "d:/3-2/AIAC/Improved Privacy-Aware Logging (Masked & Minimal).py"'. The terminal also prompts for 'Enter IP address:' and 'User activity logged securely.' The bottom status bar indicates 'Cursor Tab - Ln 23 Col 16 - Spaces: 4 - UFT-8 - Python - 3.11.9 64-bit (Microsoft Store)'.

```
import logging
from datetime import datetime
import hashlib

logging.basicConfig(
    filename="activity.log",
    level=logging.INFO,
    format=f"%(asctime)s - %(message)s"
)

# Mask IP address (last octet hidden)
def mask_ip(ip):
    parts = ip.split(".")
    if len(parts) == 4:
        return f"{parts[0]}.{parts[1]}.{parts[2]}.xxx"
    return "Invalid IP"

# Hash username for anonymity
def anonymize_username(username):
    return hashlib.sha256(username.encode()).hexdigest()[:10]

def log_user_activity(username, ip_address):
    masked_ip = mask_ip(ip_address)
    anon_user = anonymize_username(username)

    print(f"Logged activity for user {anon_user} at {masked_ip} on {datetime.now()}")
```

### Improvements Made (Privacy Protection)

Issue	Insecure Version	Improved Version
Username	Plain text	Hashed (anonymous)
IP Address	Full IP	Masked IP
Data Minimization	✗ No	✓ Yes
Privacy Compliance	✗ Weak	✓ Strong

### Part 3: Privacy-Aware Logging Principles (Explanation)

#### Key Principles Followed

##### 1. Data Minimization

- Log only what is necessary for debugging or auditing.

##### 2. Anonymization

- Hash usernames to avoid identity exposure.

### **3. Masking Sensitive Fields**

- Hide part of IP addresses.

### **4. Purpose Limitation**

- Logs should serve a specific purpose (monitoring, security).

### **5. Security of Logs**

- Logs should not expose personal data if compromised.

#### **Validation: Explanation Matches Runtime Behavior**

- User input is not logged directly
- Logs contain masked IP and hashed username
- Even if activity.log is accessed, real identity cannot be reconstructed
- Privacy risks reduced
- Behavior matches explanation