

AI Assisted Coding Assignment- 5.5

| 2303A51543 BT: 29 VELDI.HRUTHIKA 30.01.2026



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.

```
#2303A51543 BT-29
```

```
import hashlib
```

```
import os
```

```
#Task -1
```

```
def is_prime_basic(n):
    """Basic prime checker - checks all numbers up to n-1"""
    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 checker - checks up to sqrt(n) only"""
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
```

```
    # Only check odd divisors up to sqrt(n)
    i = 3
```

```
    while i * i <= n:
```

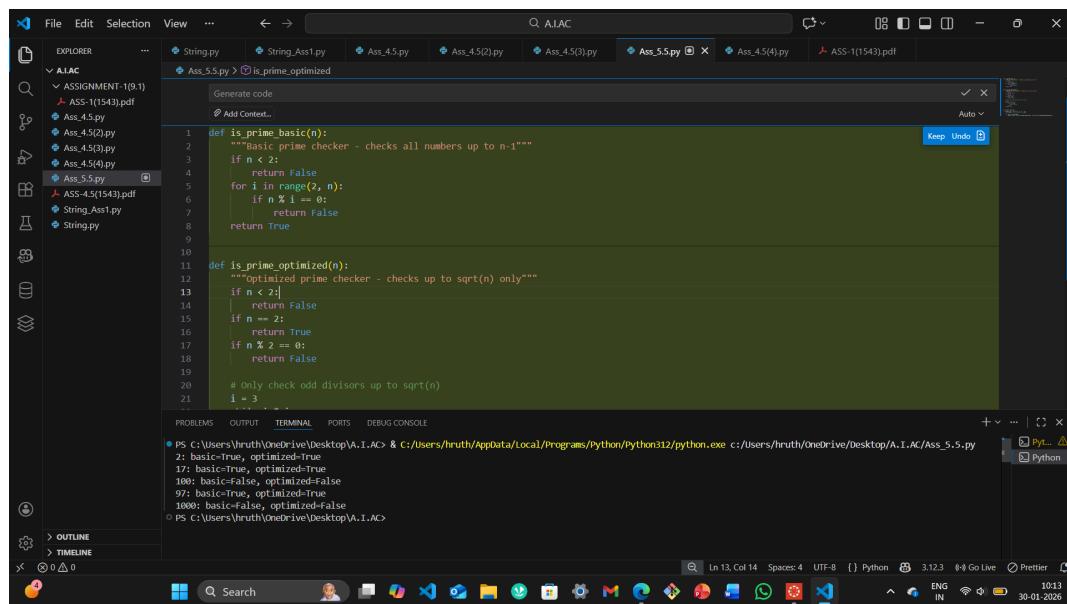
```

if n % i == 0:
    return False
i += 2
return True

# Test both methods
if __name__ == "__main__":
    test_numbers = [2, 17, 100, 97, 1000]

for num in test_numbers:
    print(f"{num}: basic={is_prime_basic(num)}, optimized={is_prime_optimized(num)}")

```



Prime Number Checker Module

This module provides two implementations for checking whether a number is prime, demonstrating the difference between a basic approach and an optimized approach.

Functions:

`is_prime_basic(n)`: Checks if n is prime by testing divisibility against all numbers from 2 to n-1.

Time Complexity: $O(n)$ - performs $n-2$ division operations in the worst case.

`is_prime_optimized(n)`: Checks if n is prime by testing divisibility only against odd numbers up to \sqrt{n} , after handling small cases.

Time Complexity: $O(\sqrt{n})$ - performs approximately $\sqrt{n}/2$ division operations in the worst case.

Efficiency Comparison:

- For $n=100$: basic checks 98 divisors, optimized checks ~5 divisors
- For $n=1000$: basic checks 998 divisors, optimized checks ~15 divisors
- For $n=1000000$: basic checks 999,998 divisors, optimized checks ~500 divisors

The optimized version is significantly faster for large numbers due to:

1. Early termination at \sqrt{n} - reduces iterations exponentially
 2. Skipping even numbers after checking for divisibility by 2 - halves remaining checks
 3. Special case handling - eliminates redundant operations for small numbers
- For practical purposes, the optimized method is the preferred approach for prime checking, especially when dealing with larger numbers.



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.

```
#Task -2
def fibonacci_recursive(n):
    """
    Calculate the nth Fibonacci number using recursion.

```

Base cases:

- $\text{fibonacci}(0) = 0$ (first Fibonacci number)
- $\text{fibonacci}(1) = 1$ (second Fibonacci number)

Recursive case:

- $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$
- Each call breaks down the problem into two smaller subproblems

"""

```
# Base case 1: if n is 0, return 0
```

```
if n == 0:
```

```
    return 0
```

```
# Base case 2: if n is 1, return 1
```

```
if n == 1:
```

```
    return 1
```

```
# Recursive case: sum the two previous Fibonacci numbers
```

```
# This breaks the problem into smaller versions of itself
```

```
return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
```

```

# Test the recursive function
if __name__ == "__main__":
    test_numbers = [0, 1, 5, 10, 15]

for num in test_numbers:
    result = fibonacci_recursive(num)
    print(f"fibonacci({num}) = {result}")

```

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

- File Explorer:** Shows files like String.py, Ass_4.5.py, Ass_4.5(2).py, Ass_4.5(3).py, Ass_4.5(4).py, Ass_5.5.py, and ASS-1(1543).pdf.
- Code Editor:** Displays the `fibonacci_recursive(n)` function. The code uses recursion to calculate the nth Fibonacci number. It includes base cases for n=0 and n=1, and a recursive case where it sums the previous two numbers.
- Terminal:** Shows the command run in Python: `PS C:\Users\hruth\OneDrive\Desktop\A.I.AC> & C:/Users/hruth/AppData/Local/Programs/Python/Python312/python.exe c:/Users/hruth/OneDrive/Desktop/A.I.AC/Ass_5.5.py`. The output shows the results for n=0 to n=15.
- Bottom Status Bar:** Includes file paths, line count (Ln 37), column count (Col 9), spaces (Spaces: 4), encoding (UTF-8), Python version (3.12.3), and other system information.

Factorial is defined as:

- $\text{factorial}(0) = 1$ (base case: $0! = 1$)
- $\text{factorial}(n) = n * \text{factorial}(n-1)$ for $n > 0$

How recursion works here:

1. Each call multiplies n by the result of $\text{factorial}(n-1)$
2. The recursion stops when n reaches 0 (base case)
3. Results "unwind" back up the call stack, multiplying at each level

Example: $\text{factorial}(4)$

- $\text{factorial}(4)$ calls $4 * \text{factorial}(3)$
- $\text{factorial}(3)$ calls $3 * \text{factorial}(2)$
- $\text{factorial}(2)$ calls $2 * \text{factorial}(1)$
- $\text{factorial}(1)$ calls $1 * \text{factorial}(0)$
- $\text{factorial}(0)$ returns 1 (base case - no more recursion)
- Then: $1*1=1$, $2*1=2$, $3*2=6$, $4*6=24$



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.

```
#Task -3
def read_and_process_file(filename):
    """
    Read a file and process data with comprehensive error handling.

Handles:
- FileNotFoundError: when file doesn't exist
- PermissionError: when lacking read permissions
- ValueError: when data format is invalid
- IOError: for general file operation failures
    """

try:
    # Attempt to open and read the file
    with open(filename, 'r') as file:
        lines = file.readlines()

    # Process each line
    numbers = []
    for line_num, line in enumerate(lines, 1):
        try:
            # Try to convert each line to a number
            number = float(line.strip())
            numbers.append(number)
        except ValueError:
            # Raised when line cannot be converted to float
            print(f"Warning: Line {line_num} '{line.strip()}' is not a valid number, skipping...")

    # Calculate and display statistics
    if numbers:
        print(f"Processed {len(numbers)} valid numbers")
        print(f"Sum: {sum(numbers)}, Average: {sum(numbers)/len(numbers):.2f}")
    else:
        print("No valid numbers found in file")

return numbers
```

```

except FileNotFoundError:
    # Raised when file path doesn't exist
    print(f"Error: File '{filename}' not found.")
except PermissionError:
    # Raised when lacking read permissions
    print(f"Error: Permission denied reading '{filename}'")
except IOError as e:
    # Raised for other file operation failures
    print(f"Error: File operation failed - {e}")

```

```

# Test the function
if __name__ == "__main__":
    read_and_process_file("data.txt")

```

The screenshot shows the Visual Studio Code interface. The code editor has the following content:

```

def read_and_process_file(filename):
    """
    Read a file and process data with comprehensive error handling.

    Handles:
    - FileNotFoundError: when file doesn't exist
    - PermissionError: when lacking read permissions
    - ValueError: when data format is invalid
    - IOError: for general file operation failures
    """

    try:
        # Attempt to open and read the file
        with open(filename, 'r') as file:
            lines = file.readlines()
    except:
        # Try to convert each line to a number
        numbers = []
        for line_num, line in enumerate(lines, 1):
            try:
                numbers.append(int(line))
            except:
                print(f"Warning: Line {line_num} '{line.strip()}' is not a valid number, skipping...")

```

The terminal below shows the execution results:

```

Warning: Line 3 'hello' is not a valid number, skipping...
Processed 3 valid numbers
Sum: 60.5, Average: 20.17
PS C:\Users\hruth\OneDrive\Desktop\A.I.A.C> [REDACTED]

```

▼ data.txt

```

10
20
hello
30.5

```



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.

```
#Task -4

# Function to hash a password
def hash_password(password):
    """Hash a password using SHA-256."""
    return hashlib.sha256(password.encode()).hexdigest()

# Function to verify a password against a stored hash
def verify_password(stored_hash, password):
    """Verify a password against the stored hash."""
    return stored_hash == hash_password(password)

# Simple login system
def login_system():
    """A simple login system with secure password handling."""
    users = {} # Dictionary to store username and hashed password

    while True:
        action = input("Do you want to (register/login/exit)? ").strip().lower()

        if action == 'register':
            username = input("Enter a username: ")
            password = input("Enter a password: ")
            if username in users:
                print("Username already exists. Please choose another.")
            else:
                users[username] = hash_password(password)
                print("Registration successful!")

        elif action == 'login':
            username = input("Enter your username: ")
            password = input("Enter your password: ")
            if username in users and verify_password(users[username], password):
                print("Login successful!")
            else:
                print("Invalid username or password.")


```

```
elif action == 'exit':
    print("Exiting the system.")
    break

else:
    print("Invalid action. Please choose register, login, or exit.")

# Test the login system
if __name__ == "__main__":
    login_system()
```

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

- File Explorer:** Shows files in the 'A.I.AC' folder, including 'Ass_5.py', 'String.Ass1.py', 'data.txt', and several 'Ass_4.' files.
- Code Editor:** Displays the content of 'Ass_5.py'. The code defines two functions: `hash_password` and `verify_password`.
- Terminal:** Shows the execution of the script and its interaction with the user. It asks if the user wants to register or login, and then prompts for a username and password. It also handles invalid inputs like 'hrtx' and 'hrtx@12'.
- Status Bar:** Shows the line number (Ln 163), column (Col 41), spaces (Spaces: 4), file type (Python), line length (3.12.3), and other system information.

Best Practices for Secure Authentication

1. Password Hashing:

- Never store plaintext passwords. Use strong hashing algorithms like **bcrypt** or **Argon2**.
 - Always use **salsas** with passwords to ensure unique hashes.

2. Two-Factor Authentication (2FA):

- Implement **2FA** to add an extra layer of security beyond just passwords.

3. Strong Password Policies:

- Require **strong passwords** with a mix of characters and enforce **minimum length**.
 - Check for **common passwords** and discourage their use.

4. Session Security:

- Use **secure sessions** and **cookies** marked as `HTTPOnly` and `Secure`.
 - Consider **token-based authentication (JWT)** for stateless systems.

5. Brute Force Protection:

- Implement **rate limiting** and **CAPTCHA** to prevent brute-force attacks.

6. Use HTTPS:

- Always use **HTTPS** to encrypt data in transit, especially passwords.

7. Account Recovery:

- Secure the **password reset process** with multi-step verification.

8. Audit Logs:

- Maintain **logs** of login attempts and suspicious activity for monitoring.

By following these principles, you ensure user credentials and sessions are secure from unauthorized access.



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.

```
#Task -5
import logging
from datetime import datetime
import hashlib

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

def log_user_activity(username, ip_address):
    """Log user activity with username, masked IP address, and timestamp."""
    # Mask the IP address to reduce privacy risks (logging only the first two segments)
    masked_ip = '.'.join(ip_address.split('.')[2:]) + ".x.x"

    timestamp = datetime.now().isoformat()
    logging.info(f"User: {username}, IP: {masked_ip}, Timestamp: {timestamp}")

def hash_password(password):
    """Hash the password for secure storage."""
    # Using a simple hash function for demonstration; in a real system, use a secure hashing algorithm like
    # bcrypt
    return hashlib.sha256(password.encode()).hexdigest()
```

```

def verify_password(hashed_password, input_password):
    """Verify the entered password against the stored hash."""
    return hashed_password == hash_password(input_password)

# Example usage within the login system
def login_system():
    """A simple login system with secure password handling."""
    users = {} # Dictionary to store username and hashed password

    while True:
        action = input("Do you want to (register/login/exit)? ").strip().lower()

        if action == 'register':
            username = input("Enter a username: ")
            password = input("Enter a password: ")
            if username in users:
                print("Username already exists. Please choose another.")
            else:
                users[username] = hash_password(password)
                print("Registration successful!")

        elif action == 'login':
            username = input("Enter your username: ")
            password = input("Enter your password: ")
            ip_address = input("Enter your IP address: ") # Simulated input for demonstration
            if username in users and verify_password(users[username], password):
                print("Login successful!")
                log_user_activity(username, ip_address) # Log the activity with masked IP
            else:
                print("Invalid username or password.")

        elif action == 'exit':
            print("Exiting the system.")
            break

        else:
            print("Invalid action. Please choose register, login, or exit.")

    if __name__ == "__main__":
        login_system()

```

The screenshot shows a code editor interface with two main panes. The top pane displays a Python script named `Task - 5`. The code configures basic logging and defines a `log_user_activity` function that logs user activity with a masked IP address. It also includes a `login_system` function that handles user registration and login. The bottom pane shows the terminal output of running the script, where the user registers with username `hruthika1543` and password `hruthika1543`, and then logs in successfully.

```

#Task - 5
Generate code
Add Context...
Keep Undo Auto ✓
Confidence Logging
logging.basicConfig(filename='user_activity.log', level=logging.INFO, format='%(asctime)s - %(message)s')

def log_user_activity(username, ip_address):
    """Log user activity with username, IP address, and timestamp."""
    timestamp = datetime.now().isoformat()
    logging.info(f"User: {username}, IP: {ip_address}, Timestamp: {timestamp}")

Example usage within the login system
def login_system():
    """A simple login system with secure password handling."""
    users = {} # Dictionary to store username and hashed password

    while True:
        action = input("Do you want to (register/login/logout)? ").strip().lower()

        if action == 'register':
            username = input("Enter a username: ")
            password = input("Enter a password: ")
            if username in users:
                print("Username already exists. Please choose another.")
            else:
                users[username] = hash_password(password)

        elif action == 'login':
            username = input("Enter your username: ")
            password = input("Enter your password: ")
            if username in users and users[username] == hash_password(password):
                print("Login successful!")
            else:
                print("Invalid username or password.")

        elif action == 'logout':
            print("Exiting the system.")

        else:
            print("Unknown command. Please enter register, login, or logout.")

        if action in ['register', 'login']:
            print("Do you want to (register/login/logout)? ")

```

`.45(3).py` 177 # Configure logging
`.45(4).py` 178 logging.basicConfig(filename='user_activity.log', level=logging.INFO, format='%(asctime)s - %(message)s')
`.55.py` 179
`S-4.5(1543).pdf` 180
`a.txt` 181 def log_user_activity(username, ip_address):
`ng.Ass1.py` 182 """Log user activity with username, masked IP address, and timestamp."""
`ng.py` 183 # Mask the IP address to reduce privacy risks (logging only the first two segments)
`r_activity.log` 184 masked_ip = '.'.join(ip_address.split('.')[0:2]) + ".x.x"
185
186 timestamp = datetime.now().isoformat()
187 logging.info(f"User: {username}, IP: {masked_ip}, Timestamp: {timestamp}")
188
189 do back password(password) .
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

PS C:\Users\hruth\OneDrive\Desktop\A.I.AC> & C:/Users/hruth/AppData/Local/Programs/Python/Python312/python.exe c:/Users/hruth/OneDrive/Desktop/A.I.AC/Ass_5-5.py
Do you want to (register/login/logout)? register
Enter a username: hruthika1543
Registration successful!
Do you want to (register/login/logout)? login
Enter your username: hruthika1543
Enter your password: hruthika1543
Enter your IP address: 8805
Invalid username or password.
Do you want to (register/login/logout)? login
Enter your username: hruthika1543
Enter your password: hruthika1543
Enter your IP address: 8802
Login successful!
Do you want to (register/login/logout)? exit
Exiting the system.
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC>

Privacy-Aware Logging Principles

1. Minimize Data Collection:

- Only log necessary data and avoid sensitive information like passwords. Mask or anonymize data (e.g., logging partial IP addresses).

2. Masking and Anonymization:

- Mask sensitive data (like IP addresses) and avoid logging full details. Use hashing for sensitive information like passwords.

3. Secure Storage:

- Store logs securely, use encryption, and restrict access with proper permissions to protect sensitive data.

4. Data Retention:

- Keep logs only as long as needed and delete or anonymize old logs to reduce exposure risks.

5. User Consent:

- Inform users about data logging and obtain consent, especially in regions with strict privacy laws like GDPR.

6. Audit Logs:

- Use logs to monitor for suspicious activity but limit access to logs and ensure they are only available to authorized personnel.

7. Never Log Plaintext Passwords:

- Always hash passwords before storing or verifying them, never logging them in plaintext.