Name: Shryitha Suddala

Hall ticket:2303A51637

Batch:29

Assessment: 5.5

Task 1: Transparency in Algorithm Optimization

Prompt:

Generate Python code for two prime-checking methods and explain how the optimized version improves performance. Give two codes one for naive approach and optimized approach

Code:

AI-5.5.py > ...

```python
#Generate Python code for two prime-checking methods and explain how the optimized version improves performance.give two codes one for naive approach and optim
# Naive Approach to Check for Prime Number
def is_prime_naive(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
# Optimized Approach to Check for Prime Number
def is_prime_optimized(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
# Explanation of Optimization:
# The naive approach checks for factors from 2 to n-1, which results in O(n) time complexity.
# The optimized approach reduces the number of checks significantly:
# 1. It first eliminates numbers less than or equal to 1, and directly returns
#    True for 2 and 3.
# 2. It eliminates even numbers and multiples of 3 right away.
# 3. It then checks for factors only up to the square root of n, since if n is divisible
#    by any number greater than its square root, it must also be divisible by a number
#    smaller than its square root.
# 4. It checks only numbers of the form 6k ± 1, as all prime numbers greater than 3
#    can be expressed in this form, further reducing the number of checks.
# This results in an average time complexity of O(√n), making it much more efficient for larger numbers.
# Example Usage
number = 29
print(f"Naive: Is {number} prime? {is_prime_naive(number)}")
print(f"Optimized: Is {number} prime? {is_prime_optimized(number)}")
```

Output:

```
Naive: Is 29 prime? True
Optimized: Is 29 prime? True
PS D:\AI>
```

Justification :

This task highlights the importance of understanding algorithm efficiency when using AI-generated code. The basic approach performs many unnecessary checks, leading to increased execution time. In contrast, the optimized method minimizes computations by applying a mathematical rule. By comparing both approaches, we learn that AI-generated

code should be carefully analyzed and optimized before use. Efficient algorithms help reduce system load and significantly improve overall performance.

Task 2: Transparency in Recursive Algorithms

Prompt :

Generate a recursive function to calculate Fibonacci numbers.add clear comments explaining recursion. Explain base cases and recursive calls.

Code :

```python
#generate a recursive function to calculate Fibonacci numbers.add clear comments explaining recursion.explain base cases and recursive calls.
# Recursive Function to Calculate Fibonacci Numbers
def fibonacci(n):
    # Base Case: If n is 0, return 0
    if n == 0:
        return 0
    # Base Case: If n is 1, return 1
    elif n == 1:
        return 1
    # Recursive Case: For n greater than 1, return the sum of the two preceding Fibonacci numbers
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
# Explanation of Recursion:
# 1. Base Cases:
#    - The function has two base cases: when n is 0 and when n is 1.
#    - These base cases are essential because they provide the stopping condition for the recursion.
#    - Without base cases, the function would call itself indefinitely, leading to a stack overflow.
# 2. Recursive Calls:
#    - For any n greater than 1, the function calls itself twice: once with (n - 1) and once with (n - 2).
#    - This means that to compute fibonacci(n), the function needs to compute fibonacci(n - 1) and fibonacci(n - 2).
#    - Each of these calls will further break down into smaller calls until they reach the base cases.
# Example Usage
number = 6
print(f"Fibonacci of {number} is {fibonacci(number)}")

```

Output:

```
Fibonacci of 6 is 8
PS D:\AI>
```

Justification :

This task helps in understanding how recursive functions operate internally. Base cases clearly define when recursion should terminate. Recursive calls divide a larger problem into smaller, manageable subproblems. Adding comments enhances readability and reduces the chances of logical errors. Clear and well-structured recursion makes AI-generated code easier to maintain and more reliable.

Task 3: Transparency in Error Handling

Prompt:

Generate a Python program that reads a file and processes data. Generate code with proper error handling and clear explanations for each exception.Code with meaningful exception handling. Clear comments explaining each error scenario. Validation that explanations align with runtime behaviour.

Code:

```python
#generate a Python program that reads a file and processes data.Generate code with proper error handling and clear explanations for each exception.Code with me
# Python Program to Read a File and Process Data with Error Handling
def read_and_process_file(file_path):
    try:
        # Attempt to open the file
        with open(file_path, 'r') as file:
            data = file.readlines()
            # Process the data (for example, print each line)
            for line in data:
                print(line.strip())
    except FileNotFoundError:
        # This exception is raised when the specified file does not exist
        print(f"Error: The file '{file_path}' was not found. Please check the file path and try again.")
    except PermissionError:
        # This exception is raised when there are insufficient permissions to read the file
        print(f"Error: You do not have permission to read the file '{file_path}'. Please check your permissions.")
    except IsADirectoryError:
        # This exception is raised when a directory is provided instead of a file
        print(f"Error: The path '{file_path}' is a directory, not a file. Please provide a valid file path.")
    except Exception as e:
        # Catch-all for any other exceptions that may occur
        print(f"An unexpected error occurred: {e}")
# Example Usage
file_path = 'example.txt'  # Replace with your file path
read_and_process_file(file_path)
```

Output:

```
Error: The file 'example.txt' was not found. Please check the file path and try again.
PS D:\AI>
```

Justification :

This task focuses on developing robust and user-friendly programs. The program safely handles common file-related errors using proper exception handling. Clear and meaningful error messages help users understand issues instead of causing the program to crash. Ethical AI coding involves anticipating possible failures and handling them responsibly. Effective error handling significantly improves software reliability and user trust.

Task 4: Security in User Authentication

Prompt:

Generate a Python-based login system. Check whether the AI uses secure password handling practices. 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.

Code:

```python
#generate a Python-based login system.Check whether the AI uses secure password handling practices.Identification of security flaws (plain-text passwords, weak
import hashlib
import re
# Simple Login System with Secure Password Handling
class LoginSystem:
    def __init__(self):
        # In a real application, user data would be stored in a database
        self.users = {}

    def hash_password(self, password):
        # Hash the password using SHA-256 for secure storage
        return hashlib.sha256(password.encode()).hexdigest()

    def validate_password(self, password):
        # Validate password strength: at least 8 characters, including letters and numbers
        if len(password) < 8:
            return False
        if not re.search(r"[a-zA-Z]", password):
            return False
        if not re.search(r"[0-9]", password):
            return False
        return True

    def register(self, username, password):
        if username in self.users:
            print("Error: Username already exists.")
            return
        if not self.validate_password(password):
            print("Error: Password must be at least 8 characters long and include both letters and numbers.")
            return
        hashed_password = self.hash_password(password)
        self.users[username] = hashed_password
        print("User registered successfully.")

    def login(self, username, password):
        if username not in self.users:
            print("Error: Username does not exist.")
            return
        hashed_password = self.hash_password(password)
        if self.users[username] == hashed_password:
            print("Login successful.")
        else:
            print("Error: Incorrect password.")
# Example Usage
login_system = LoginSystem()
login_system.register("user1", "Password123")
login_system.login("user1", "Password123")
```

```python
            print("Error: Username already exists.")
            return
        if not self.validate_password(password):
            print("Error: Password must be at least 8 characters long and include both letters and numbers.")
            return
        hashed_password = self.hash_password(password)
        self.users[username] = hashed_password
        print("User registered successfully.")

    def login(self, username, password):
        if username not in self.users:
            print("Error: Username does not exist.")
            return
        hashed_password = self.hash_password(password)
        if self.users[username] == hashed_password:
            print("Login successful.")
        else:
            print("Error: Incorrect password.")
# Example Usage
login_system = LoginSystem()
login_system.register("user1", "Password123")
login_system.login("user1", "Password123")
# Best Practices for Secure Authentication:
# 1. Always hash passwords before storing them to prevent plain-text password storage.
# 2. Use strong password policies to ensure users create secure passwords.
# 3. Implement account lockout mechanisms to prevent brute-force attacks.
# 4. Use secure protocols (like HTTPS) to protect data in transit.
```

Output:

```
User registered successfully.
Login successful.
PS D:\AI>
```

Justification:

This task highlights the importance of performing security reviews on AI-generated login systems. Storing passwords in plain text makes them vulnerable if data is leaked. Password

hashing ensures that user credentials remain protected even if the system is compromised. Developers should never rely entirely on AI-generated code without validating its security aspects. Secure authentication mechanisms safeguard users and help build long-term trust.

Task 5: Privacy in Data Logging

Prompt:

Generate a Python script that logs user activity (username, IP address, timestamp).Examine whether sensitive data is logged unnecessarily or insecurely.Identified privacy risks in logging. Improved version with minimal, anonymized, or maskedlogging.Explanation of privacy-aware logging principles.

Code:

```python
#generate a Python script that logs user activity (username, IP address, timestamp).Examine whether sensitive data is logged unnecessarily or insecurely.Identi
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(message)s'
)

def log_user_activity(username, ip_address):
    anonymized_ip = '.'.join(ip_address.split('.')[:-1] + ['xxx'])
    logging.info(f'User: {username}, IP: {anonymized_ip}')

log_user_activity('user1', '192.168.1.100')
# Explanation of Privacy-Aware Logging Principles:
# 1. Minimize Data Collection: Only log information that is necessary for the intended purpose
#    (e.g., username and anonymized IP address) to reduce privacy risks.
# 2. Anonymization: Mask or anonymize sensitive data (like IP addresses) to protect user privacy.
# 3. Secure Storage: Ensure that log files are stored securely with appropriate access controls
#    to prevent unauthorized access.
# 4. Compliance: Adhere to data protection regulations and best practices regarding user data logging.

```

Output:

```
2026-01-30 10:24:05,795 - User: user1, IP: 192.168.1.xxx
PS D:\AI>
```

Justification:

This task highlights the importance of responsible data collection. Storing complete user details increases privacy risks and the chances of data misuse. Masking sensitive data reduces exposure and enhances user privacy. Ethical logging practices follow privacy laws and respect user rights. Collecting and storing only the necessary data helps prevent misuse and improves overall system trust.