

Name: Achi Vaishnavi

Hall ticket:2303A51554

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-55.py > ...
1 #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
2 # Naive Approach to Check for Prime Number
3 def is_prime_naive(n):
4     if n <= 1:
5         return False
6     for i in range(2, n):
7         if n % i == 0:
8             return False
9     return True
10 # Optimized Approach to Check for Prime Number
11 def is_prime_optimized(n):
12     if n <= 1:
13         return False
14     if n <= 3:
15         return True
16     if n % 2 == 0 or n % 3 == 0:
17         return False
18     i = 5
19     while i * i <= n:
20         if n % i == 0 or n % (i + 2) == 0:
21             return False
22         i += 6
23     return True
24 # Explanation of Optimization:
25 # The naive approach checks for factors from 2 to n-1, which results in O(n) time complexity.
26 # The optimized approach reduces the number of checks significantly:
27 # 1. It first eliminates numbers less than or equal to 1, and directly returns
28 #    True for 2 and 3.
29 # 2. It eliminates even numbers and multiples of 3 right away.
30 # 3. It then checks for factors only up to the square root of n, since if n is divisible
31 #    by any number greater than its square root, it must also be divisible by a number
32 #    smaller than its square root.
33 # 4. It checks only numbers of the form 6k ± 1, as all prime numbers greater than 3
34 #    can be expressed in this form, further reducing the number of checks.
35 # This results in an average time complexity of O(√n), making it much more efficient for larger numbers.
36 # Example Usage
37 number = 29
38 print(f"Naive: Is {number} prime? {is_prime_naive(number)}")
39 print(f"Optimized: Is {number} prime? {is_prime_optimized(number)}")
40
```

Output:

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

Justification :

This task explains the importance of algorithm efficiency in AI-generated code. The basic method wastes time through unnecessary checks, while the optimized method uses a mathematical rule to reduce computations. This comparison shows that AI-generated code should always be reviewed and optimized. Efficient algorithms improve performance and reduce system load.

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 :

```
42 #generate a recursive function to calculate Fibonacci numbers.add clear comments explaining recursion.explain base cases and recursive calls.
43 # Recursive Function to Calculate Fibonacci Numbers
44 def fibonacci(n):
45     # Base Case: If n is 0, return 0
46     if n == 0:
47         return 0
48     # Base Case: If n is 1, return 1
49     elif n == 1:
50         return 1
51     # Recursive Case: For n greater than 1, return the sum of the two preceding Fibonacci numbers
52     else:
53         return fibonacci(n - 1) + fibonacci(n - 2)
54 # Explanation of Recursion:
55 # 1. Base Cases:
56 # - The function has two base cases: when n is 0 and when n is 1.
57 # - These base cases are essential because they provide the stopping condition for the recursion.
58 # - Without base cases, the function would call itself indefinitely, leading to a stack overflow.
59 # 2. Recursive Calls:
60 # - For any n greater than 1, the function calls itself twice: once with (n - 1) and once with (n - 2).
61 # - This means that to compute fibonacci(n), the function needs to compute fibonacci(n - 1) and fibonacci(n - 2).
62 # - Each of these calls will further break down into smaller calls until they reach the base cases.
63 # Example Usage
64 number = 6
65 print(f'Fibonacci of {number} is {fibonacci(number)}')
66
```

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 break a larger problem into smaller, manageable subproblems. Adding comments improves readability and minimizes 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:

```

68 #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
69 # Python Program to Read a File and Process Data with Error Handling
70 def read_and_process_file(file_path):
71     try:
72         # Attempt to open the file
73         with open(file_path, 'r') as file:
74             data = file.readlines()
75             # Process the data (for example, print each line)
76             for line in data:
77                 print(line.strip())
78     except FileNotFoundError:
79         # This exception is raised when the specified file does not exist
80         print(f"Error: The file '{file_path}' was not found. Please check the file path and try again.")
81     except PermissionError:
82         # This exception is raised when there are insufficient permissions to read the file
83         print(f"Error: You do not have permission to read the file '{file_path}'. Please check your permissions.")
84     except IsADirectoryError:
85         # This exception is raised when a directory is provided instead of a file
86         print(f"Error: The path '{file_path}' is a directory, not a file. Please provide a valid file path.")
87     except Exception as e:
88         # Catch-all for any other exceptions that may occur
89         print(f"An unexpected error occurred: {e}")
90 # Example Usage
91 file_path = 'example.txt' # Replace with your file path
92 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 through proper exception handling. Clear and meaningful error messages help users understand problems instead of causing the program to crash. Ethical AI coding requires anticipating possible failures and handling them responsibly. Effective error handling significantly improves software reliability and builds 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:

```
94 #generate a Python-based login system.Check whether the AI uses secure password handling practices.Identification of security flaws (plain-text passwords, weak
95 import hashlib
96 import re
97 # Simple Login System with Secure Password Handling
98 class LoginSystem:
99     def __init__(self):
100         # In a real application, user data would be stored in a database
101         self.users = {}
102
103     def hash_password(self, password):
104         # Hash the password using SHA-256 for secure storage
105         return hashlib.sha256(password.encode()).hexdigest()
106
107     def validate_password(self, password):
108         # Validate password strength: at least 8 characters, including letters and numbers
109         if len(password) < 8:
110             return False
111         if not re.search(r"[a-zA-Z]", password):
112             return False
113         if not re.search(r"[0-9]", password):
114             return False
115         return True
116
117     def register(self, username, password):
118         if username in self.users:
119             print("Error: Username already exists.")
120             return
121         if not self.validate_password(password):
122             print("Error: Password must be at least 8 characters long and include both letters and numbers.")
123             return
124         hashed_password = self.hash_password(password)
125         self.users[username] = hashed_password
126         print("User registered successfully.")
127
128     def login(self, username, password):
129         if username not in self.users:
130             print("Error: Username does not exist.")
131             return
132         hashed_password = self.hash_password(password)
133         if self.users[username] == hashed_password:
134             print("Login successful.")
135         else:
136             print("Error: Incorrect password.")
137
138 # Example Usage
139 login_system = LoginSystem()
140 login_system.register("user1", "Password123")
141 login_system.login("user1", "Password123")
```

```
119         if username in self.users:
120             print("Error: Username already exists.")
121             return
122         if not self.validate_password(password):
123             print("Error: Password must be at least 8 characters long and include both letters and numbers.")
124             return
125         hashed_password = self.hash_password(password)
126         self.users[username] = hashed_password
127         print("User registered successfully.")
128
129     def login(self, username, password):
130         if username not in self.users:
131             print("Error: Username does not exist.")
132             return
133         hashed_password = self.hash_password(password)
134         if self.users[username] == hashed_password:
135             print("Login successful.")
136         else:
137             print("Error: Incorrect password.")
138
139 # Example Usage
140 login_system = LoginSystem()
141 login_system.register("user1", "Password123")
142 login_system.login("user1", "Password123")
143
144 # Best Practices for Secure Authentication:
145 # 1. Always hash passwords before storing them to prevent plain-text password storage.
146 # 2. Use strong password policies to ensure users create secure passwords.
147 # 3. Implement account lockout mechanisms to prevent brute-force attacks.
148 # 4. Use secure protocols (like HTTPS) to protect data in transit.
```

Output:

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

Justification:

This task highlights the importance of performing security reviews on AI-generated login systems. Storing passwords in plain text makes user credentials vulnerable in the event of a

data breach. Password hashing protects sensitive information even if the system is compromised. Developers should never rely entirely on AI-generated code without thoroughly 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 masked logging.Explanation of privacy-aware logging principles.

Code:

```
148 #generate a Python script that logs user activity (username, IP address, timestamp).Examine whether sensitive data is logged unnecessarily or insecurely.Identi
149 import logging
150
151 logging.basicConfig(
152     level=logging.INFO,
153     format='%(asctime)s - %(message)s'
154 )
155
156 def log_user_activity(username, ip_address):
157     anonymized_ip = '.'.join(ip_address.split('.')[:-1] + ['xxx'])
158     logging.info(f'User: {username}, IP: {anonymized_ip}')
159
160 log_user_activity('user1', '192.168.1.100')
161
162 # Explanation of Privacy-Aware Logging Principles:
163 # 1. Minimize Data Collection: Only log information that is necessary for the intended purpose
164 #    (e.g., username and anonymized IP address) to reduce privacy risks.
165 # 2. Anonymization: Mask or anonymize sensitive data (like IP addresses) to protect user privacy.
166 # 3. Secure Storage: Ensure that log files are stored securely with appropriate access controls
167 #    to prevent unauthorized access.
168 # 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 potential for data misuse. Masking sensitive information reduces exposure and strengthens user privacy. Ethical logging practices comply with privacy laws and respect user rights. Collecting and storing only necessary data helps prevent misuse and improves overall system trust.