2303A51203
Assignment:5.5

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

```python
# Basic primality test function
def is_prime_basic(num):
    if num <= 1:
        return False
    # Iterate from 2 up to num - 1
    for i in range(2, num):
        if num % i == 0:
            return False
    return True

# Optimized primality test function
def is_prime_optimized(num):
    if num <= 1:
        return False
    if num <= 3:
        return True # 2 and 3 are prime
    if num % 2 == 0 or num % 3 == 0:
        return False # multiples of 2 or 3 are not prime

    # Check for divisors from 5 up to sqrt(num)
    # with a step of 6 (all primes > 3 are of the form 6k ± 1)
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True

# Test cases for both methods
print("--- Basic Method ---")
print(f"Is 7 prime? {is_prime_basic(7)}")
print(f"Is 10 prime? {is_prime_basic(10)}")
print(f"Is 97 prime? {is_prime_basic(97)}")
print(f"Is 101 prime? {is_prime_basic(101)}")

print("\n--- Optimized Method ---")
print(f"Is 7 prime? {is_prime_optimized(7)}")
print(f"Is 10 prime? {is_prime_optimized(10)}")
print(f"Is 97 prime? {is_prime_optimized(97)}")
print(f"Is 101 prime? {is_prime_optimized(101)}")
print(f"Is 1000000007 prime? {is_prime_optimized(1000000007)}") # A large
prime number
```

```
--- Basic Method ---
Is 7 prime? True
Is 10 prime? False
Is 97 prime? True
Is 101 prime? True

--- Optimized Method ---
Is 7 prime? True
Is 10 prime? False
Is 97 prime? True
Is 101 prime? True
Is 1000000007 prime? 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.
Expected Output:
• Well-commented recursive code.
• Clear explanation of how recursion works.
• Verification that explanation matches actual execution.

```python
def fibonacci_recursive(n):
    # Base cases:
    # If n is 0, the Fibonacci number is 0.
    if n == 0:
        return 0
    # If n is 1, the Fibonacci number is 1.
    elif n == 1:
        return 1
    # Recursive call:
    # For n > 1, the Fibonacci number is the sum of the two preceding
ones.
    else:
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)

# Test cases
print(f"Fibonacci(0): {fibonacci_recursive(0)}")
print(f"Fibonacci(1): {fibonacci_recursive(1)}")
print(f"Fibonacci(2): {fibonacci_recursive(2)}")
print(f"Fibonacci(5): {fibonacci_recursive(5)}")
print(f"Fibonacci(10): {fibonacci_recursive(10)}")
```

Output:
```
Fibonacci(0): 0
Fibonacci(1): 1
Fibonacci(2): 1
Fibonacci(5): 5
Fibonacci(10): 55
```

### Explanation of the Recursive Fibonacci Function:

****Recursion** is a programming technique where a function calls itself directly or indirectly to solve a problem. It breaks a problem down into smaller, similar subproblems until it reaches a simple, solvable base case.**

In the `fibonacci_recursive(n)` function:

1. **Purpose:** This function calculates the nth Fibonacci number. The Fibonacci sequence starts with 0 and 1, and each subsequent number is the sum of the two preceding ones (e.g., 0, 1, 1, 2, 3, 5, 8, ...).

2. **Components of Recursion:**

    *   **Base Cases:** These are the conditions that stop the recursion. Without base cases, a recursive function would call itself infinitely, leading to a stack overflow error. In the Fibonacci function:
        *   `if n == 0: return 0` : The 0th Fibonacci number is defined as 0.
        *   `elif n == 1: return 1` : The 1st Fibonacci number is defined as 1.
        These two cases are the simplest forms of the problem and can be solved directly without further recursion.

    *   **Recursive Calls:** These are the parts where the function calls itself with smaller inputs. In the Fibonacci function:
        *   `else: return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)` : For any `n` greater than 1, the function defines the nth Fibonacci number as the sum of the (n-1)th and (n-2)th Fibonacci numbers. It calls itself with `n-1` and `n-2` to get these values. This progressively breaks down the problem until it reaches the base cases (0 or 1).

How it works:
When you call `fibonacci_recursive(5)`, for example, it will:
1. Call `fibonacci_recursive(4)` and `fibonacci_recursive(3)`.
2. `fibonacci_recursive(4)` will call `fibonacci_recursive(3)` and `fibonacci_recursive(2)`.
3. This continues until `n` becomes 0 or 1, at which point the base cases return a direct value, and the results are summed up the call stack.

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.

```python
def is_prime_safe(num):
    """
    Checks if a given number is prime with robust error handling.

    Args:
        num (int): The number to check for primality.

    Returns:
        bool: True if the number is prime, False otherwise.

    Raises:
        TypeError: If the input 'num' is not an integer.
        ValueError: If the input 'num' is a negative integer.
    """
    # Error handling for non-integer input
    if not isinstance(num, int):
        raise TypeError("Input must be an integer. Received type: " +
str(type(num).__name__))

    # Error handling for negative input
    if num < 0:
        raise ValueError("Input must be a non-negative integer for
primality check.")

    # Standard primality test logic
    if num <= 1:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

# Test cases with and without errors
print("--- is_prime_safe function ---")
print(f"Is 7 prime? {is_prime_safe(7)}")
print(f"Is 10 prime? {is_prime_safe(10)}")

print("\n--- Error Handling Demos for is_prime_safe ---")

# Demonstrating TypeError
try:
    is_prime_safe('abc')
except TypeError as e:
    print(f"Caught expected TypeError for 'abc': {e}")

try:
    is_prime_safe(5.5)
except TypeError as e:
    print(f"Caught expected TypeError for 5.5: {e}")
```

```
# Demonstrating ValueError
try:
    is_prime_safe(-5)
except ValueError as e:
    print(f"Caught expected ValueError for -5: {e}")
```

Output:
--- is_prime_safe function ---
Is 7 prime? True
Is 10 prime? False

--- Error Handling Demos for is_prime_safe ---
Caught expected TypeError for 'abc': Input must be an integer. Received type: str
Caught expected TypeError for 5.5: Input must be an integer. Received type: float
Caught expected ValueError for -5: Input must be a non-negative integer for primality check.

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.

```
# In-memory database for users (username: password)
# In a real application, this would be a secure database with hashed
passwords.
users_db = {}

def register_user(username, password):
    """
    Registers a new user in the in-memory database.
    Returns True if registration is successful, False if the username
already exists.
    """
    if username in users_db:
        print(f"Registration failed: Username '{username}' already
exists.")
        return False
```

```python
    # In a real system, the password would be hashed and salted before
storing.
    users_db[username] = password
    print(f"User '{username}' registered successfully.")
    return True

def login_user(username, password):
    """
    Authenticates a user.
    Returns True if login is successful, False otherwise.
    """
    stored_password = users_db.get(username)

    if stored_password and stored_password == password:
        print(f"Login successful for user '{username}'.")
        return True
    else:
        print(f"Login failed: Invalid username or password for
'{username}'.")
        return False


# --- Demonstration ---
print("--- User Registration ---")
register_user("alice", "password123")
register_user("bob", "securepwd")
register_user("alice", "anotherpassword") # Attempt to register existing
user

print("\n--- User Login ---")
login_user("alice", "password123") # Successful login
login_user("bob", "wrongpwd")     # Failed login (wrong password)
login_user("charlie", "anypwd")   # Failed login (non-existent user)
login_user("bob", "securepwd")     # Successful login

print("\n--- Current Users in DB (for demonstration purposes only) ---")
print(users_db)
```

output:
```
--- User Registration ---
User 'alice' registered successfully.
User 'bob' registered successfully.
Registration failed: Username 'alice' already exists.

--- User Login ---
Login successful for user 'alice'.
Login failed: Invalid username or password for 'bob'.
Login failed: Invalid username or password for 'charlie'.
Login successful for user 'bob'.

--- Current Users in DB (for demonstration purposes only) ---
{'alice': 'password123', 'bob': 'securepwd'}
```

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.

```
import datetime

# In-memory list to store activity logs
# In a real application, this would be written to a file, database, or a
dedicated logging service.
activity_logs = []

def log_user_activity(username, ip_address):
    """
    Logs user activity with username, IP address, and a timestamp.

    Args:
        username (str): The username of the active user.
        ip_address (str): The IP address from which the activity
originated.
    """
    timestamp = datetime.datetime.now()
    log_entry = {
        "timestamp": timestamp.strftime("%Y-%m-%d %H:%M:%S"),
        "username": username,
        "ip_address": ip_address
    }
    activity_logs.append(log_entry)
    print(f"Activity logged for {username} from {ip_address} at
{timestamp.strftime('%H:%M:%S')}")

# --- Demonstration ---
print("--- Logging User Activities ---")
log_user_activity("alice", "192.168.1.100")
log_user_activity("bob", "10.0.0.5")
log_user_activity("alice", "192.168.1.100") # Alice doing another activity

print("\n--- All Activity Logs ---")
for log in activity_logs:
    print(log)
```

Output :

```
--- Logging User Activities ---
Activity logged for alice from 192.168.1.100 at 06:33:41
Activity logged for bob from 10.0.0.5 at 06:33:41
Activity logged for alice from 192.168.1.100 at 06:33:41

--- All Activity Logs ---
{'timestamp': '2026-01-30 06:33:41', 'username': 'alice', 'ip_address': '192.168.1.100'}
{'timestamp': '2026-01-30 06:33:41', 'username': 'bob', 'ip_address': '10.0.0.5'}
{'timestamp': '2026-01-30 06:33:41', 'username': 'alice', 'ip_address': '192.168.1.100'}
```