2303A51449
Batch 03
Lab 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.”

CODE:

```
def is_prime_basic(n):
    """Basic method to check if a number is prime."""
    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 method to check if a number is prime."""
    if n < 2:
        return False
    if n == 2 or 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


# Example Usage
print(f"Is 17 prime (basic)? {is_prime_basic(17)}")
print(f"Is 17 prime (optimized)? {is_prime_optimized(17)}")
```

```
print(f"Is 25 prime (basic)? {is_prime_basic(25)}")
print(f"Is 25 prime (optimized)? {is_prime_optimized(25)}")

print(f"Is 97 prime (basic)? {is_prime_basic(97)}")
print(f"Is 97 prime (optimized)? {is_prime_optimized(97)}")

# You can also compare performance for larger numbers using the `time`
module
import time

num_to_check = 1000000007 # A large prime number

start_time = time.time()
result_basic = is_prime_basic(num_to_check)
end_time = time.time()
print(f"\nBasic method for {num_to_check}: {result_basic} (Time:
{end_time - start_time:.6f} seconds)")

start_time = time.time()
result_optimized = is_prime_optimized(num_to_check)
end_time = time.time()
print(f"Optimized method for {num_to_check}: {result_optimized} (Time:
{end_time - start_time:.6f} seconds)")
```

Output:

```
Is 17 prime (basic)? True
Is 17 prime (optimized)? True
Is 25 prime (basic)? False
Is 25 prime (optimized)? False
Is 97 prime (basic)? True
Is 97 prime (optimized)? True

Basic method for 1000000007: True (Time: 86.331080 seconds)
Optimized method for 1000000007: True (Time: 0.000974 seconds)
```

Time Complexity:

- **Basic Method (`is_prime_basic`):** Has a time complexity of **O(n)**. This means that as the input number n gets larger, the number of operations required by the basic method grows linearly with n. For example, if n is 1 million, it might perform close to 1 million checks.
- **Optimized Method (`is_prime_optimized`):** Has a time complexity of **O(sqrt(n))**. This is a significant improvement because the number of operations grows with the square root of n. For instance, if n is 1 million, `sqrt(n)` is 1,000, meaning it performs roughly 1,000 checks instead of 1 million.

Comparison Highlighting Efficiency Improvements:

As demonstrated in the execution of the code:

For `num_to_check = 1000000007` (a large prime number):

- **Basic Method:** Took approximately **86.33 seconds**.
- **Optimized Method:** Took only about **0.001 seconds**.

This dramatic difference in execution time clearly highlights the efficiency improvement. The optimized method leverages mathematical properties (like checking divisibility only up to the square root of n and the `6k ± 1` optimization) to drastically reduce the number of calculations needed, making it much faster for 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.

CODE:

```
# Example Usage
print(f"Is 17 prime (basic)? {is_prime_basic(17)}")
print(f"Is 17 prime (optimized)? {is_prime_optimized(17)}")

print(f"Is 25 prime (basic)? {is_prime_basic(25)}")
print(f"Is 25 prime (optimized)? {is_prime_optimized(25)}")

print(f"Is 97 prime (basic)? {is_prime_basic(97)}")
print(f"Is 97 prime (optimized)? {is_prime_optimized(97)}")

# You can also compare performance for larger numbers using the `time`
module
import time

num_to_check = 1000000007 # A large prime number

start_time = time.time()
result_basic = is_prime_basic(num_to_check)
end_time = time.time()
print(f"\nBasic method for {num_to_check}: {result_basic} (Time:
{end_time - start_time:.6f} seconds)")
```

```
start_time = time.time()
result_optimized = is_prime_optimized(num_to_check)
end_time = time.time()
print(f"Optimized method for {num_to_check}: {result_optimized} (Time:
{end_time - start_time:.6f} seconds)")
```

OUTPUT:

```
Is 17 prime (basic)? True
Is 17 prime (optimized)? True
Is 25 prime (basic)? False
Is 25 prime (optimized)? False
Is 97 prime (basic)? True
Is 97 prime (optimized)? True

Basic method for 1000000007: True (Time: 86.331080 seconds)
Optimized method for 1000000007: True (Time: 0.000974 seconds)
```

CODE:

```
def fibonacci_recursive(n):
    """Calculates the nth Fibonacci number using recursion.

    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...).

    Args:
        n: The index of the Fibonacci number to calculate (non-negative
integer).

    Returns:
        The nth Fibonacci number.
    """

    # Base Cases:
    # These are the conditions that stop the recursion. Without them,
    # the function would call itself indefinitely, leading to a stack
overflow.
    if n < 0:
        # Handle invalid input: Fibonacci numbers are typically defined
for non-negative integers.
        raise ValueError("Input must be a non-negative integer.")
```

```
    elif n == 0:
        # The 0th Fibonacci number is 0.
        return 0
    elif n == 1:
        # The 1st Fibonacci number is 1.
        return 1
    else:
        # Recursive Call:
        # This is where the function calls itself with smaller inputs.
        # To find the nth Fibonacci number, we sum the (n-1)th and
(n-2)th Fibonacci numbers.
        # The problem is broken down into simpler, identical
subproblems.
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)

# Example Usage:
print(f"Fibonacci(0): {fibonacci_recursive(0)}")     # Expected: 0
print(f"Fibonacci(1): {fibonacci_recursive(1)}")     # Expected: 1
print(f"Fibonacci(2): {fibonacci_recursive(2)}")     # Expected: 1 (0 +
1)
print(f"Fibonacci(3): {fibonacci_recursive(3)}")     # Expected: 2 (1 +
1)
print(f"Fibonacci(5): {fibonacci_recursive(5)}")     # Expected: 5 (2 +
3)
print(f"Fibonacci(10): {fibonacci_recursive(10)}") # Expected: 55


OUTPUT:
Fibonacci(0): 0
Fibonacci(1): 1
Fibonacci(2): 1
Fibonacci(3): 2
Fibonacci(5): 5
Fibonacci(10): 55
```

The well-commented recursive code I provided, along with the explanation of base cases and recursive calls, perfectly matches the actual execution results you observed.

Let's recap how recursion works in this context:

1. **Well-commented Recursive Code:** The `fibonacci_recursive(n)` function is designed with internal comments that delineate the purpose of the function, the base cases, and the recursive call logic.
2. **Clear Explanation of How Recursion Works:**

- **Base Cases (`n < 0`, `n == 0`, `n == 1`):** These are the stopping conditions. Without them, the function would call itself indefinitely. For `n=0` and `n=1`, the function returns fixed values directly, providing a known starting point.
- **Recursive Call (`else: return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)`):** This is the core of recursion. For any `n` greater than 1, the problem is broken down into two smaller, identical sub-problems (`fibonacci_recursive(n-1)` and `fibonacci_recursive(n-2)`). The function calls itself to solve these smaller problems, and their results are combined to solve the original problem. This continues until the base cases are reached.

3. **Verification that Explanation Matches Actual Execution:** The output you saw from executing the code (e.g., `Fibonacci(0): 0`, `Fibonacci(1): 1`, `Fibonacci(10): 55`) directly confirms that:
   - The base cases are correctly handled (e.g., `Fibonacci(0)` and `Fibonacci(1)`).
   - The recursive calls correctly compute the sequence by summing the previous two numbers (e.g., `Fibonacci(2)` is `Fibonacci(1) + Fibonacci(0)` which is `1 + 0 = 1`).

Every step in the execution aligns with the recursive logic and the explanation provided.

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.

```
def process_data(data, divisor=1, filename='data.txt'):
    """Demonstrates error handling for various scenarios.

    Args:
        data: A list of numbers to process.
        divisor: A number to divide elements by.
        filename: A file to attempt to open.
    """
    print(f"\n--- Attempting to process with data={data},
divisor={divisor}, filename='{filename}' ---")
```

```
    try:
        # Scenario 1: Type Error (if data is not iterable or numbers
are not numeric)
        total_sum = sum(data) # Might raise TypeError if 'data'
contains non-numeric types
        print(f"Sum of data: {total_sum}")

        # Scenario 2: ZeroDivisionError
        result = total_sum / divisor
        print(f"Result of division: {result}")

        # Scenario 3: FileNotFoundError
        with open(filename, 'r') as f:
            file_content = f.read()
            print(f"File content: {file_content[:50]}...") # Print
first 50 chars

    except TypeError as e:
        # Catches errors where operations are performed on incorrect
data types.
        print(f"Error: A TypeError occurred. Please ensure 'data'
contains only numbers. Details: {e}")
    except ZeroDivisionError:
        # Catches errors when attempting to divide by zero.
        print("Error: Cannot divide by zero. Please provide a non-zero
divisor.")
    except FileNotFoundError:
        # Catches errors when a specified file does not exist.
        print(f"Error: The file '{filename}' was not found. Please
check the path.")
    except ValueError as e:
        # Catches errors for invalid values, e.g., if sum() tried to
convert non-numeric string
        print(f"Error: A ValueError occurred. Invalid value detected.
Details: {e}")
    except Exception as e:
        # This is a general catch-all for any other unexpected errors.
        # It's good practice to catch more specific exceptions first.
        print(f"An unexpected error occurred: {e}")
    else:
        # The 'else' block executes ONLY if the 'try' block completes
without any exceptions.
```

```
        print("All operations in the 'try' block completed
successfully!")
    finally:
        # The 'finally' block always executes, regardless of whether an
exception occurred or not.
        # This is useful for cleanup operations like closing files or
releasing resources.
        print("Error handling attempt completed. Cleaning up (if
necessary)...")

# --- Test Cases ---

# Successful execution
process_data(data=[10, 20, 30], divisor=5)

# Demonstrate TypeError
process_data(data=[1, 'a', 3], divisor=2)

# Demonstrate ZeroDivisionError
process_data(data=[100, 200], divisor=0)

# Demonstrate FileNotFoundError
process_data(data=[1, 2, 3], divisor=1,
filename='non_existent_file.txt')

# Demonstrate a combination (e.g., FileNotFoundError without previous
errors)
process_data(data=[10, 20], divisor=2,
filename='another_missing_file.txt')

# Create a file for successful file read
with open('data.txt', 'w') as f:
    f.write('This is some sample content for the data file.')
process_data(data=[1, 2, 3], divisor=1, filename='data.txt')
```

OUTPUT:

```
Attempting to process with data=[10, 20, 30], divisor=5,
filename='data.txt' ---
Sum of data: 60
Result of division: 12.0
```

```
Error: The file 'data.txt' was not found. Please check the path.
Error handling attempt completed. Cleaning up (if necessary)...

--- Attempting to process with data=[1, 'a', 3], divisor=2,
filename='data.txt' ---
Error: A TypeError occurred. Please ensure 'data' contains only
numbers. Details: unsupported operand type(s) for +: 'int' and 'str'
Error handling attempt completed. Cleaning up (if necessary)...

--- Attempting to process with data=[100, 200], divisor=0,
filename='data.txt' ---
Sum of data: 300
Error: Cannot divide by zero. Please provide a non-zero divisor.
Error handling attempt completed. Cleaning up (if necessary)...

--- Attempting to process with data=[1, 2, 3], divisor=1,
filename='non_existent_file.txt' ---
Sum of data: 6
Result of division: 6.0
Error: The file 'non_existent_file.txt' was not found. Please check the
path.
Error handling attempt completed. Cleaning up (if necessary)...

--- Attempting to process with data=[10, 20], divisor=2,
filename='another_missing_file.txt' ---
Sum of data: 30
Result of division: 15.0
Error: The file 'another_missing_file.txt' was not found. Please check
the path.
Error handling attempt completed. Cleaning up (if necessary)...

--- Attempting to process with data=[1, 2, 3], divisor=1,
filename='data.txt' ---
Sum of data: 6
Result of division: 6.0
File content: This is some sample content for the data file....
All operations in the 'try' block completed successfully!
Error handling attempt completed. Cleaning up (if necessary)...

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

    # 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'}
```