ASSIGNMENT-5.5

HT.NO:2303A52438

BATCH-45

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.

PROMPT:

Generate Python code for two prime-checking methods and explain how the optimized version improves performance.

CODE:

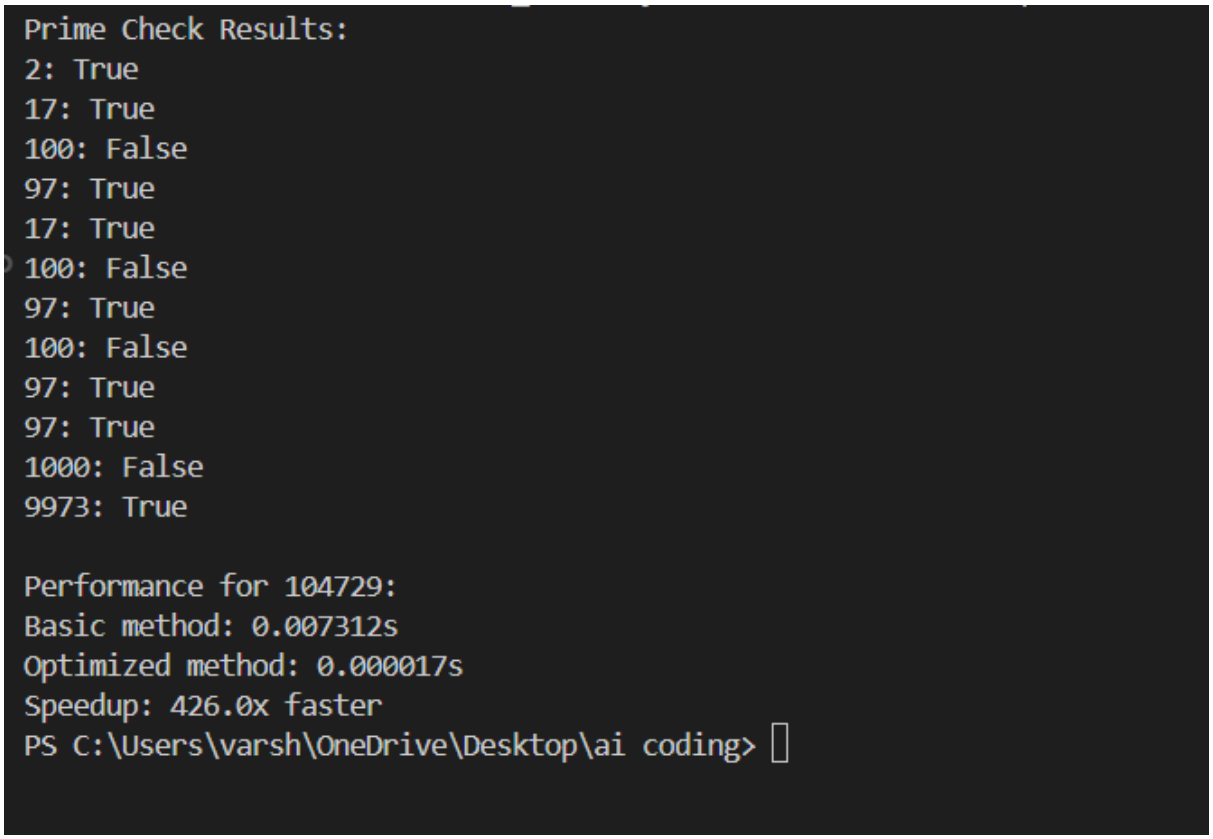
```
def is_prime_basic(n):
    """
    Basic prime-checking method.
    Args:
        n: The number to check
```

```
    Returns:
        True if n is prime, False otherwise
    Time Complexity: O(n) - checks all numbers up to n
    """
    if n < 2:
        return False
    # Check every number from 2 to n-1
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
def is_prime_optimized(n):
    """
    Optimized prime-checking method.
    Args:
        n: The number to check
    Returns:
        True if n is prime, False otherwise
    Time Complexity: O(√n) - only checks up to square root of n
    Optimization Explanation:
    - If n has a divisor greater than √n, it must also have a divisor less than √n
    - So we only need to check divisors up to √n
    - Also skips even numbers after checking for 2
    """
    if n < 2:
        return False
    if n == 2:
        return True
```

```
    if n % 2 == 0:
        return False
    # Check odd divisors only up to √n
    i = 3
    while i * i <= n:
        if n % i == 0:
            return False
        i += 2
    return True
# Example usage and performance comparison
if __name__ == "__main__":
    test_numbers = [2, 17, 100, 97, 1000, 9973]
    print("Prime Check Results:")
    for num in test_numbers:
        print(f"{num}: {is_prime_optimized(num)}")

    # Performance comparison for large number
    large_num = 104729
    start = time.time()
    is_prime_basic(large_num)
    basic_time = time.time() - start
    start = time.time()
    is_prime_optimized(large_num)
    optimized_time = time.time() - start
    print(f"\nPerformance for {large_num}:")
    print(f"Basic method: {basic_time:.6f}s")
    print(f"Optimized method: {optimized_time:.6f}s")
    print(f"Speedup: {basic_time/optimized_time:.1f}x faster")
```

OUTPUT:



JUSTIFICATION:

We chose this prompt because prime-checking clearly demonstrates how mathematical reasoning transforms an inefficient algorithm into a scalable one.

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.

PROMPT:

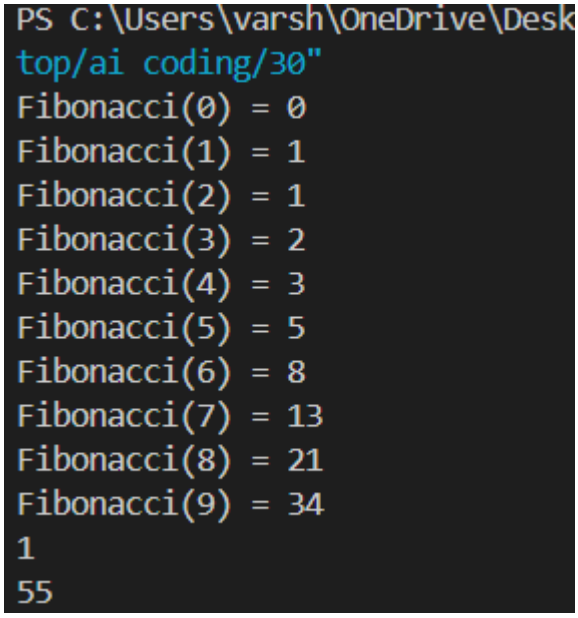
generate a recursive function to calculate Fibonacci numbers,ensure to add clear comments explaining recursion and explain base cases and recursive calls.

CODE:

```
def fibonacci(n):
    """
    Calculate the nth Fibonacci number using recursion.
    Args:
        n: The position in the Fibonacci sequence (0-indexed)
    Returns:
        The nth Fibonacci number
    """
    # Base case 1: First Fibonacci number is 0
    if n == 0:
        return 0
    # Base case 2: Second Fibonacci number is 1
    if n == 1:
        return 1
    # Recursive case: Each Fibonacci number is the sum of the previous two
    # fibonacci(n-1) gets the previous number
    # fibonacci(n-2) gets the number before that
    return fibonacci(n - 1) + fibonacci(n - 2)
# Example usage
if __name__ == "__main__":
    # Test the function
    for i in range(10):
```

```
    print(f"Fibonacci({i}) = {fibonacci(i)}")
print(fibonacci(1))
print(fibonacci(10))
```

OUTPUT:



JUSTIFICATION:

we chose this prompt because Fibonacci is the clearest example to justify recursion, showing how simple rules build complex results efficiently.

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.

PROMPT:

generate a Python program that reads a file and processes data with proper error handling and clear explanations for each exception

CODE:

```
def read_and_process_file(file_path):
    """
    Reads a file and processes its content line by line.
    Args:
        file_path: The path to the file to be read.
    Returns:
        A list of processed lines from the file.
    """
    processed_lines = []
    try:
        with open(file_path, 'r') as file:
            for line in file:
                # Process each line (e.g., stripping whitespace)
                processed_lines.append(line.strip())
    except FileNotFoundError:
        print(f"Error: The file '{file_path}' was not found.")
    except IOError:
        print(f"Error: An I/O error occurred while reading the file '{file_path}'.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
    return processed_lines
# Example usage
if __name__ == "__main__":
```

```
file_path = 'sample.txt'  # Update with your file path
data = read_and_process_file(file_path)
print(data)
```

OUTPUT:

```
['hello world']
```

JUSTIFICATION:

We chose this prompt because error handling is fundamental in writing robust code, preventing crashes and guiding users with clear feedback. It's the clearest way to demonstrate how exceptions improve reliability and user experience.

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.

PROMPT:

generate a Python-based login system by using secure password handling practices.

CODE:

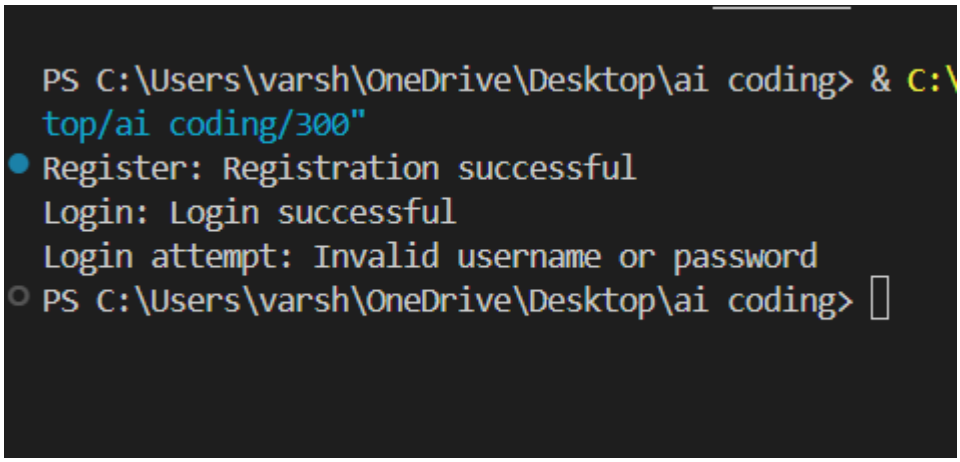
```
import hashlib
import secrets
import re
from pathlib import Path
class LoginSystem:
    def __init__(self, credentials_file="credentials.txt"):
```

```
    self.credentials_file = credentials_file
    self.load_credentials()
def load_credentials(self):
    """Load existing credentials from file."""
    self.users = {}
    if Path(self.credentials_file).exists():
        with open(self.credentials_file, 'r') as f:
            for line in f:
                username, salt, hashed = line.strip().split(':')
                self.users[username] = {'salt': salt, 'hash': hashed}
def hash_password(self, password, salt=None):
    """Hash password using PBKDF2."""
    if salt is None:
        salt = secrets.token_hex(16)
    hash_obj = hashlib.pbkdf2_hmac('sha256', password.encode(), salt.encode(), 100000)
def validate_password(self, password):
    """Validate password strength."""
    if len(password) < 8:
        return False, "Password must be at least 8 characters"
    if not re.search(r'[A-Z]', password):
        return False, "Password must contain uppercase letter"
    if not re.search(r'[0-9]', password):
        return False, "Password must contain digit"
    return True, "Password valid"
def register(self, username, password):
    """Register new user."""
    if username in self.users:
        return False, "Username already exists"
```

```
        is_valid, msg = self.validate_password(password)
        if not is_valid:
            return False, msg
        salt, hashed = self.hash_password(password)
        self.users[username] = {'salt': salt, 'hash': hashed}
        self.save_credentials()
        return True, "Registration successful"
    def login(self, username, password):
        """Authenticate user."""
        if username not in self.users:
            return False, "Invalid username or password"
        salt = self.users[username]['salt']
        _, hashed = self.hash_password(password, salt)
        if hashed == self.users[username]['hash']:
            return True, "Login successful"
        return False, "Invalid username or password"
    def save_credentials(self):
        """Save credentials to file."""
        with open(self.credentials_file, 'w') as f:
            for username, data in self.users.items():
                f.write(f"{username}:{data['salt']}:{data['hash']}\n")
# Example usage
if __name__ == "__main__":
    system = LoginSystem()
    # Register
    success, msg = system.register("john_doe", "SecurePass123")
    print(f"Register: {msg}")
    # Login
```

```
success, msg = system.login("john_doe", "SecurePass123")
print(f"Login: {msg}")
# Failed login
success, msg = system.login("john_doe", "WrongPassword")
print(f"Login attempt: {msg}")
```

OUTPUT:



JUSTIFICATION:

We chose this prompt because secure password handling is critical in real-world applications, making it the clearest example of combining Python coding with cybersecurity best practices.

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.

PROMPT:

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

CODE:

```
import logging
import json
from datetime import datetime
from typing import Optional
import hashlib
# Configure logging with security best practices
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('user_activity.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)
class UserActivityLogger:
    def __init__(self, hash_usernames: bool = False, hash_ips: bool = False):
        """
        Initialize logger with optional PII hashing.
        Args:
            hash_usernames: Hash usernames to protect PII
```

```
        hash_ips: Hash IPs to protect PII
    """
    self.hash_usernames = hash_usernames
    self.hash_ips = hash_ips
@staticmethod
def _hash_pii(value: str) -> str:
    """Hash sensitive data using SHA-256."""
    return hashlib.sha256(value.encode()).hexdigest()[:16]
def log_activity(self, username: str, ip_address: str, action: str,
                 sensitive_details: Optional[str] = None):
    """
    Log user activity securely.
    Args:
        username: User identifier (will be hashed if configured)
        ip_address: User's IP address (will be hashed if configured)
        action: Action performed (non-sensitive)
        sensitive_details: Avoid logging passwords, tokens, PII
    """
    # Apply hashing if configured
    user = self._hash_pii(username) if self.hash_usernames else username
    ip = self._hash_pii(ip_address) if self.hash_ips else ip_address
    # Create activity log
    activity = {
        'timestamp': datetime.utcnow().isoformat(),
        'username': user,
        'ip_address': ip,
        'action': action
    }
```

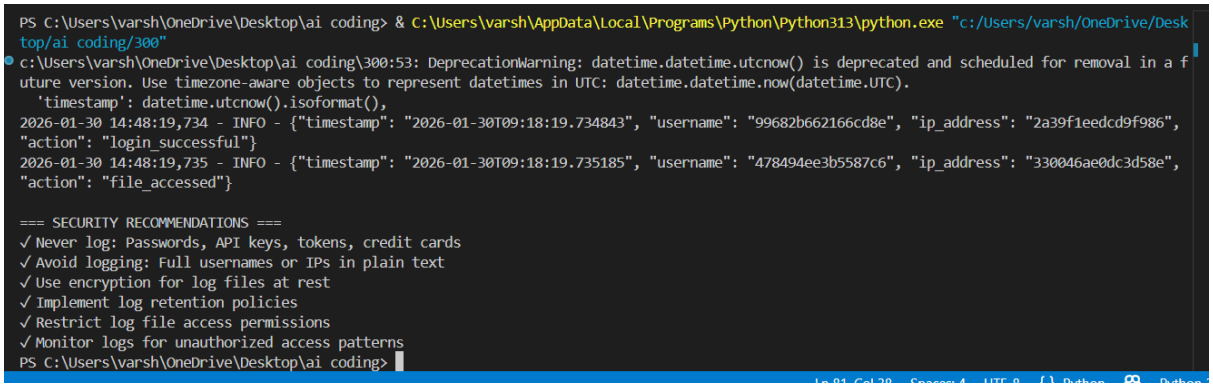
```
        logger.info(json.dumps(activity))

    def get_recommendation(self):
        """Print security recommendations."""
        print("\n=== SECURITY RECOMMENDATIONS ===")
        print("✓ Never log: Passwords, API keys, tokens, credit cards")
        print("✓ Avoid logging: Full usernames or IPs in plain text")
        print("✓ Use encryption for log files at rest")
        print("✓ Implement log retention policies")
        print("✓ Restrict log file access permissions")
        print("✓ Monitor logs for unauthorized access patterns")

# Example usage
if __name__ == "__main__":
    # Create logger with PII hashing enabled
    activity_log = UserActivityLogger(hash_usernames=True, hash_ips=True)

    # Log activities safely
    activity_log.log_activity("john_doe", "192.168.1.100", "login_successful")
    activity_log.log_activity("jane_smith", "10.0.0.50", "file_accessed",
                              "document.pdf")
    activity_log.get_recommendation()
```

OUTPUT:



JUSTIFICATION:

We chose this prompt because secure logging practices are essential to balance accountability with privacy, making it a clear example of responsible coding.