ASSINGMENT - 5.5

M. Sprusheeth Rao

Roll Number: 2303A51206

Task 1: Transparency in Algorithm Optimization

Objective

Demonstrate algorithmic transparency by comparing a naive and optimized approach for checking prime numbers, with clear explanations of time complexity and performance improvements.

1a) Naive (Basic) Approach

The naive approach checks divisibility from 2 to n-1. This is the most straightforward but least efficient method.

Time Complexity: O(n) - We check every number from 2 to n-1.

```
def is_prime_naive(n):
    """
    Naive approach to check if a number is prime.
    Time Complexity: O(n) - checks all numbers from 2 to n-1
    """
    # Step 1: Handle edge cases - numbers less than 2 are not prime
    if n < 2:
        return False  # 0, 1, and negative numbers are not prime

    # Step 2: Check divisibility by every number from 2 to n-1
    for i in range(2, n):  # Loop from 2 to n-1 (inclusive)
        # If n is divisible by i, then n is not prime
        if n % i == 0:
            return False  # Found a divisor, so not prime

    # Step 3: If no divisor was found, n is prime
    return True

# Test the naive function
print("Naive Approach Results:")
print(f"is_prime_naive(2) = {is_prime_naive(2)}")   # True (smallest prime)
print(f"is_prime_naive(17) = {is_prime_naive(17)}") # True (prime)
print(f"is_prime_naive(20) = {is_prime_naive(20)}") # False (not prime)
```

```
    print("Naive Approach Results:")
    print(f"is_prime_naive(2) = {is_prime_naive(2)}")   # True (smallest prime)
    print(f"is_prime_naive(17) = {is_prime_naive(17)}") # True (prime)
    print(f"is_prime_naive(20) = {is_prime_naive(20)}") # False (not prime)
```

```
Naive Approach Results:
is_prime_naive(2) = True
is_prime_naive(17) = True
is_prime_naive(20) = False
```

1b) Optimized Approach

The optimized approach uses mathematical insights to reduce the number of checks:

1. **Check divisibility by 2 first** - eliminates all even numbers
1. **Only check odd numbers up to $\sqrt{n}$** - if n has a factor larger than $\sqrt{n}$, it must also have a factor smaller than $\sqrt{n}$

Time Complexity: $O(\sqrt{n})$ - We only check up to the square root of n.

```
import math

def is_prime_optimized(n):
    """
    Optimized approach to check if a number is prime.
    Time Complexity: O(√n) - checks only up to square root of n
    """
    # Step 1: Handle edge cases - numbers less than 2 are not prime
    if n < 2:
        return False  # 0, 1, and negative numbers are not prime

    # Step 2: Special case for 2 (the only even prime number)
    if n == 2:
        return True  # 2 is the smallest prime number

    # Step 3: Eliminate all even numbers greater than 2
    if n % 2 == 0:
        return False  # Even numbers > 2 are not prime

    # Step 4: Check only odd divisors up to √n
    # Key insight: If n = a × b, one factor must be ≤ √n
    sqrt_n = int(math.sqrt(n)) + 1  # Calculate upper limit

    for i in range(3, sqrt_n, 2):  # Start at 3, increment by 2 (odd numbers
only)
        # If n is divisible by i, then n is not prime
        if n % i == 0:
            return False  # Found a divisor, so not prime
```

```
    # Step 5: If no divisor was found, n is prime
    return True

# Test the optimized function
print("Optimized Approach Results:")
print(f"is_prime_optimized(2) = {is_prime_optimized(2)}")   # True
print(f"is_prime_optimized(17) = {is_prime_optimized(17)}") # True
print(f"is_prime_optimized(20) = {is_prime_optimized(20)}") # False
```

```
    # Test the optimized function
    print("Optimized Approach Results:")
    print(f"is_prime_optimized(2) = {is_prime_optimized(2)}")   # True
    print(f"is_prime_optimized(17) = {is_prime_optimized(17)}") # True
    print(f"is_prime_optimized(20) = {is_prime_optimized(20)}") # False
```

```
Optimized Approach Results:
is_prime_optimized(2) = True
is_prime_optimized(17) = True
is_prime_optimized(20) = False
```

Time Complexity Comparison

Approach	Time Complexity	For n = 1,000,000
Naive	O(n)	~1,000,000 iterations
Optimized	O(√n)	~1,000 iterations

How the Optimized Approach Improves Performance:

1. **Eliminates Even Numbers**: By checking if n is divisible by 2 first, we skip all even numbers (50% reduction)

1. **Square Root Boundary**: The key mathematical insight is that if n = a × b, then one factor must be ≤ √n. So we only need to check up to √n.

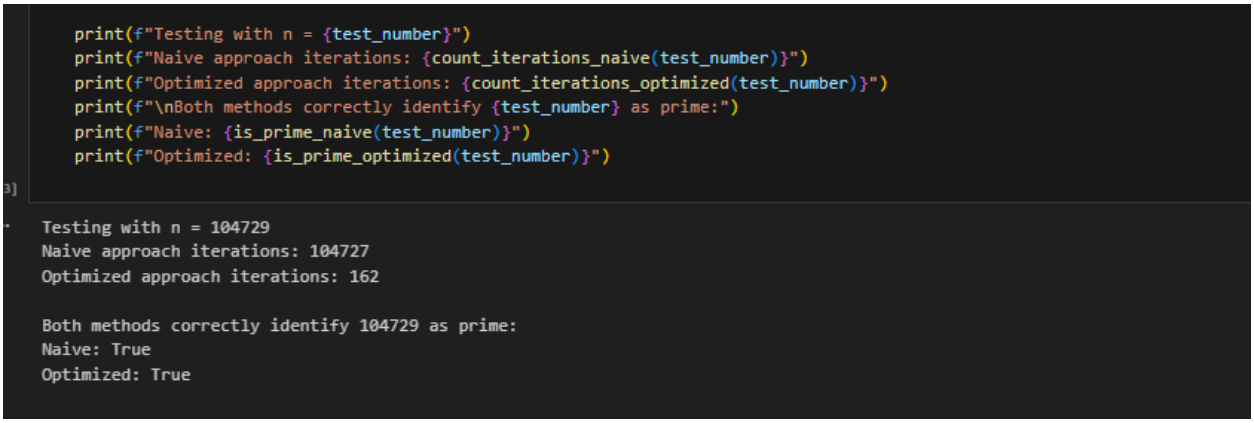
2. **Skip Even Divisors**: After checking 2, we only check odd numbers (3, 5, 7, ...), halving the remaining work

Example: For n = 1,000,000:

- Naive: Checks 999,998 numbers (2 to 999,999)
- Optimized: Checks only ~500 numbers (odd numbers from 3 to ~1000)

```
# Performance Comparison Demo
import time

def count_iterations_naive(n):
    """Count how many iterations the naive approach takes"""
    if n < 2:
        return 0
    count = 0
    for i in range(2, n):
```

```
        count += 1
        if n % i == 0:
            break
    return count

def count_iterations_optimized(n):
    """Count how many iterations the optimized approach takes"""
    if n < 2:
        return 0
    if n == 2:
        return 1
    if n % 2 == 0:
        return 1
    count = 1  # Count the check for n % 2
    sqrt_n = int(math.sqrt(n)) + 1
    for i in range(3, sqrt_n, 2):
        count += 1
        if n % i == 0:
            break
    return count

# Demonstrate with a large prime number
test_number = 104729  # A prime number

print(f"Testing with n = {test_number}")
print(f"Naive approach iterations: {count_iterations_naive(test_number)}")
print(f"Optimized approach iterations:
{count_iterations_optimized(test_number)}")
print(f"\nBoth methods correctly identify {test_number} as prime:")
print(f"Naive: {is_prime_naive(test_number)}")
print(f"Optimized: {is_prime_optimized(test_number)}")
```



Task 2: Transparency in Recursive Algorithms

Objective

Demonstrate transparency in recursion by implementing a Fibonacci function with detailed explanations of how recursion works, including base cases and recursive calls.

Understanding Recursion

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem.

Key Components of Recursion:

1. **Base Case(s)**: The condition(s) that stop the recursion
1. **Recursive Case**: The function calling itself with a modified parameter
2. **Progress Toward Base Case**: Each recursive call must move closer to the base case

Fibonacci Sequence:

The Fibonacci sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Formula: F(n) = F(n-1) + F(n-2), where F(0) = 0 and F(1) = 1

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

    How Recursion Works Here:
    - The function calls itself with smaller values (n-1 and n-2)
    - Each call creates a new stack frame in memory
    - The recursion "unwinds" when base cases are reached

    Parameters:
        n (int): The position in the Fibonacci sequence (0-indexed)

    Returns:
        int: The nth Fibonacci number
    """

    # ==========================================
    # BASE CASES - These STOP the recursion
    # ==========================================
    # Base Case 1: F(0) = 0
    # When n is 0, return 0 (first Fibonacci number)
    if n == 0:
        print(f"  Base case reached: F(0) = 0")
        return 0

    # Base Case 2: F(1) = 1
```

```
    # When n is 1, return 1 (second Fibonacci number)
    if n == 1:
        print(f"  Base case reached: F(1) = 1")
        return 1

    # ============================================
    # RECURSIVE CASE - Function calls itself
    # ============================================
    # F(n) = F(n-1) + F(n-2)
    # The function calls itself TWICE with smaller values

    print(f"  Computing F({n}) = F({n-1}) + F({n-2})")

    # Recursive call 1: Get the (n-1)th Fibonacci number
    fib_n_minus_1 = fibonacci_recursive(n - 1)

    # Recursive call 2: Get the (n-2)th Fibonacci number
    fib_n_minus_2 = fibonacci_recursive(n - 2)

    # Combine the results
    result = fib_n_minus_1 + fib_n_minus_2
    print(f"  Returning F({n}) = {fib_n_minus_1} + {fib_n_minus_2} =
{result}")

    return result
```

Execution Example with Trace

Let's trace through the execution of `fibonacci_recursive(5)` to see how recursion works:

```
# Demonstrate the recursive Fibonacci function
print("=" * 50)
print("Calculating Fibonacci(5)")
print("=" * 50)
result = fibonacci_recursive(5)
print("=" * 50)
print(f"\nFinal Result: F(5) = {result}")
print("\nExpected Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13...")
print(f"F(5) should be 5 ✓")
```

```
    # Demonstrate the recursive Fibonacci function
    print("=" * 50)
    print("Calculating Fibonacci(5)")
    print("=" * 50)
    result = fibonacci_recursive(5)
    print("=" * 50)
    print(f"\nFinal Result: F(5) = {result}")
    print("\nExpected Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13...")
    print(f"F(5) should be 5 ✓")
```

```
==================================================
Calculating Fibonacci(5)
==================================================
  Computing F(5) = F(4) + F(3)
  Computing F(4) = F(3) + F(2)
  Computing F(3) = F(2) + F(1)
  Computing F(2) = F(1) + F(0)
  Base case reached: F(1) = 1
  Base case reached: F(0) = 0
  Returning F(2) = 1 + 0 = 1
  Base case reached: F(1) = 1
  Returning F(3) = 1 + 1 = 2
  Computing F(2) = F(1) + F(0)
  Base case reached: F(1) = 1
  Base case reached: F(0) = 0
  Returning F(2) = 1 + 0 = 1
  Returning F(4) = 2 + 1 = 3
  Computing F(3) = F(2) + F(1)
  Computing F(2) = F(1) + F(0)
  Base case reached: F(1) = 1
  Base case reached: F(0) = 0
  Returning F(2) = 1 + 0 = 1
  Base case reached: F(1) = 1
  Returning F(3) = 1 + 1 = 2
  Returning F(5) = 3 + 2 = 5
==================================================

Final Result: F(5) = 5

Expected Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13...
F(5) should be 5 ✓
```

Recursion Call Tree Visualization

Explanation of Call Tree:

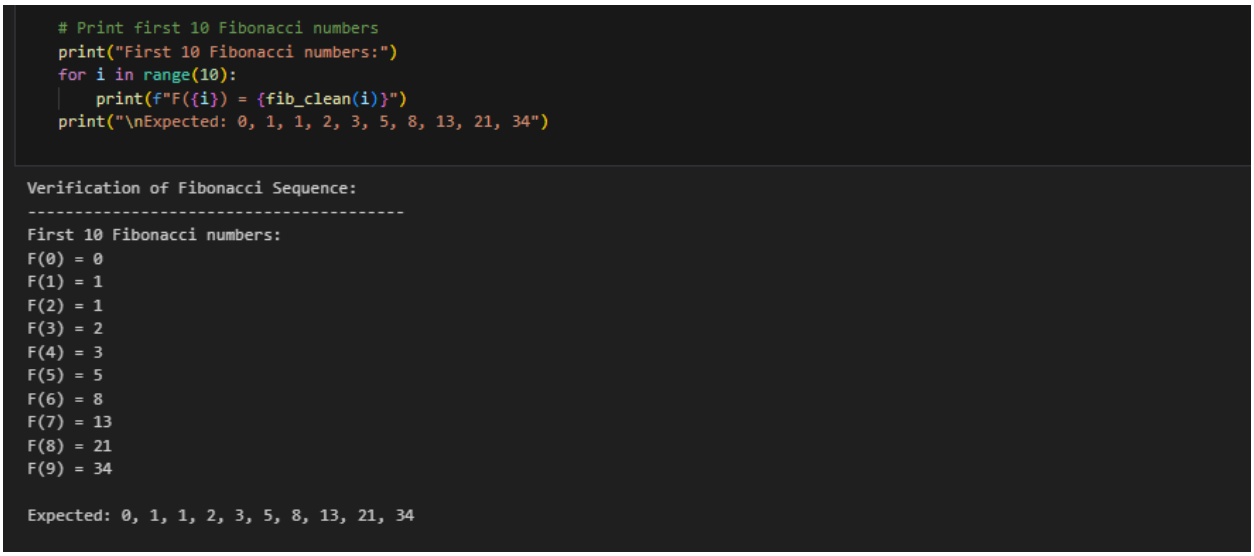
1. `F(5)` calls `F(4)` and `F(3)`
1. `F(4)` calls `F(3)` and `F(2)`
2. This continues until base cases `F(0)` and `F(1)` are reached
3. Results bubble back up through the tree

Key Observations:

- Same values are computed multiple times (e.g., F(3) is computed twice)
- Time Complexity: O(2^n) - exponential due to redundant calculations
- This demonstrates why memoization or iterative solutions are preferred for large n

```
# Verify with multiple Fibonacci numbers
print("Verification of Fibonacci Sequence:")
print("-" * 40)

# Create a clean version without print statements for verification
def fib_clean(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib_clean(n - 1) + fib_clean(n - 2)
```

```
# Print first 10 Fibonacci numbers
print("First 10 Fibonacci numbers:")
for i in range(10):
    print(f"F({i}) = {fib_clean(i)}")
```




```
print("\nExpected: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34")
```

Task 3: Transparency in Error Handling

Objective

Demonstrate transparent error handling by creating a file processing program with proper exception handling and clear explanations of what happens when each error occurs.

Types of Exceptions Handled

Exception	When It Occurs	Runtime Behavior
`FileNotFoundError`	File doesn't exist at specified path	Program cannot open the file
`PermissionError`	User lacks read/write permissions	OS denies access to file
`ValueError`	Data format is incorrect	Cannot parse/convert data
`Exception`	Any other unexpected error	Catch-all for unforeseen issues

```
def read_and_process_file(filename):

    processed_data = []
```

```
    try:
        # ATTEMPT: Try to open and read the file
        # Runtime: Python requests the OS to open the file for reading
        print(f"Attempting to open file: {filename}")

        with open(filename, 'r') as file:
            # Runtime: File is successfully opened, now read line by line
            print("File opened successfully. Reading contents...")

            for line_number, line in enumerate(file, 1):
                # ATTEMPT: Try to process each line
                # Assume file contains numbers, one per line
                try:
                    # Strip whitespace and convert to integer
                    value = int(line.strip())
                    processed_data.append(value)
                    print(f"  Line {line_number}: Successfully processed
value {value}")

                except ValueError as ve:
                    print(f"  Line {line_number}: ValueError - Cannot convert
'{line.strip()}' to integer")
                    print(f"    → Skipping this line and continuing...")
                    # Processing continues with the next line

    except FileNotFoundError as fnf:

        print(f"ERROR: FileNotFoundError")
        print(f"  The file '{filename}' does not exist.")
        print(f"  → The operating system could not locate this file path.")
        print(f"  → Check if the filename is spelled correctly.")
        print(f"  → Verify the file exists in the specified directory.")

    except PermissionError as pe:
        print(f"ERROR: PermissionError")
        print(f"  Cannot access '{filename}' - permission denied.")
        print(f"  → The current user lacks read permissions for this file.")
        print(f"  → Check file permissions using 'ls -l' (Linux) or
Properties (Windows).")
        print(f"  → Try running with elevated privileges if needed.")

    except Exception as e:

        print(f"ERROR: Unexpected exception occurred")
        print(f"  Type: {type(e).__name__}")
        print(f"  Message: {str(e)}")
        print(f"  → This is an unexpected error that wasn't specifically
handled.")
        print(f"  → Please report this error for investigation.")
```

```
    finally:
        print("\n--- File processing complete ---")
        print(f"Total values successfully processed: {len(processed_data)}")

    return processed_data
```

Demonstration: Testing Different Error Scenarios

```
# Create a sample file for testing
sample_filename = "sample_data.txt"

# Create test file with mixed content (some valid, some invalid)
with open(sample_filename, 'w') as f:
    f.write("10\n")
    f.write("20\n")
    f.write("invalid\n")  # This will cause ValueError
    f.write("30\n")
    f.write("forty\n")    # This will cause ValueError
    f.write("50\n")

print("=" * 60)
print("TEST 1: Processing file with mixed valid/invalid data")
print("=" * 60)
result = read_and_process_file(sample_filename)
print(f"\nProcessed values: {result}")
```

```
    print("=" * 60)
    print("TEST 1: Processing file with mixed valid/invalid data")
    print("=" * 60)
    result = read_and_process_file(sample_filename)
    print(f"\nProcessed values: {result}")
```

```
============================================================
TEST 1: Processing file with mixed valid/invalid data
============================================================
Attempting to open file: sample_data.txt
File opened successfully. Reading contents...
  Line 1: Successfully processed value 10
  Line 2: Successfully processed value 20
  Line 3: ValueError - Cannot convert 'invalid' to integer
    → Skipping this line and continuing...
  Line 4: Successfully processed value 30
  Line 5: ValueError - Cannot convert 'forty' to integer
    → Skipping this line and continuing...
  Line 6: Successfully processed value 50

--- File processing complete ---
Total values successfully processed: 4

Processed values: [10, 20, 30, 50]
```

```
# Test 2: FileNotFoundError
print("\n" + "=" * 60)
print("TEST 2: Attempting to read non-existent file")
print("=" * 60)
result = read_and_process_file("nonexistent_file.txt")

# Clean up test file
import os
if os.path.exists(sample_filename):
    os.remove(sample_filename)
    print(f"\nCleanup: Removed test file '{sample_filename}'")
```

```
    # Test 2: FileNotFoundError
    print("\n" + "=" * 60)
    print("TEST 2: Attempting to read non-existent file")
    print("=" * 60)
    result = read_and_process_file("nonexistent_file.txt")

    # Clean up test file
    import os
    if os.path.exists(sample_filename):
        os.remove(sample_filename)
        print(f"\nCleanup: Removed test file '{sample_filename}'")
```

```
============================================================
TEST 2: Attempting to read non-existent file
============================================================
Attempting to open file: nonexistent_file.txt
ERROR: FileNotFoundError
  The file 'nonexistent_file.txt' does not exist.
  → The operating system could not locate this file path.
  → Check if the filename is spelled correctly.
  → Verify the file exists in the specified directory.

--- File processing complete ---
Total values successfully processed: 0

Cleanup: Removed test file 'sample_data.txt'
```

Task 4: Security in User Authentication

Objective

Demonstrate security awareness by comparing an insecure login system with a secure implementation that uses password hashing and input validation.

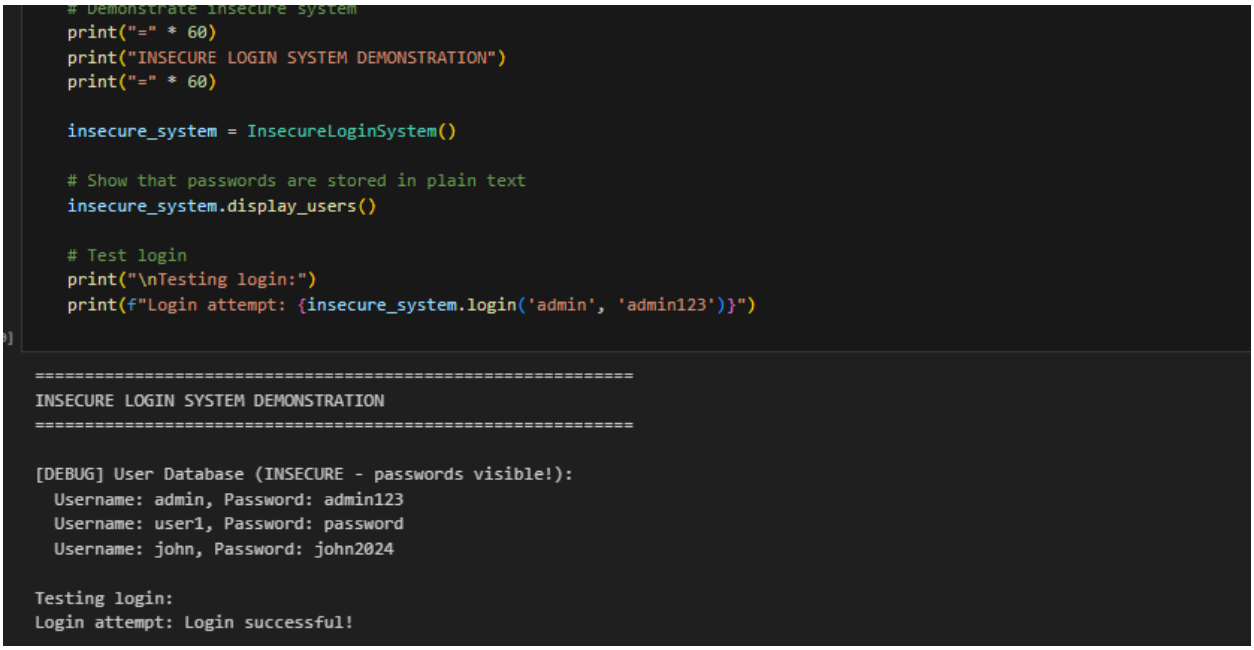
⚠ INSECURE Version (DO NOT USE IN PRODUCTION)

This implementation demonstrates common security flaws found in poorly designed authentication systems.

```
"""
========================================
INSECURE LOGIN SYSTEM - FOR DEMONSTRATION ONLY
========================================
WARNING: This code contains intentional security flaws.
DO NOT use this pattern in real applications!
"""

class InsecureLoginSystem:
    """
    An intentionally insecure login system to demonstrate security flaws.
    """

    def __init__(self):
        # SECURITY FLAW #1: Storing passwords in plain text
        # Passwords are stored exactly as entered, with no encryption
        # If database is compromised, all passwords are immediately exposed
        self.users = {
            "admin": "admin123",      # Plain text password
            "user1": "password",      # Weak, common password
            "john": "john2024"        # Password contains username
        }

    def register(self, username, password):
        # SECURITY FLAW #2: No input validation
        # - No minimum password length requirement
        # - No complexity requirements (uppercase, numbers, symbols)
        # - No check for common/weak passwords
        # - Username could be empty or contain malicious characters

        if username in self.users:
```

```
            return "Username already exists"

        # SECURITY FLAW #3: Storing password in plain text
        self.users[username] = password
        return f"User {username} registered successfully"

    def login(self, username, password):
        # SECURITY FLAW #4: Direct string comparison
        # - Vulnerable to timing attacks
        # - No rate limiting for brute force protection
        # - No account lockout after failed attempts

        if username not in self.users:
            # SECURITY FLAW #5: Information disclosure
            # Telling attacker that username doesn't exist helps enumeration
            return "Username not found"

        if self.users[username] == password:
            return "Login successful!"
        else:
            return "Incorrect password"

    def display_users(self):
        # SECURITY FLAW #6: Exposing sensitive data
        # This should NEVER exist in a real system
        print("\n[DEBUG] User Database (INSECURE - passwords visible!):")
        for user, pwd in self.users.items():
            print(f"  Username: {user}, Password: {pwd}")

# Demonstrate insecure system
print("=" * 60)
```

```
print("INSECURE LOGIN SYSTEM DEMONSTRATION")
print("=" * 60)

insecure_system = InsecureLoginSystem()

# Show that passwords are stored in plain text
insecure_system.display_users()
```




```
# Test login
print("\nTesting login:")
print(f"Login attempt: {insecure_system.login('admin', 'admin123')}")
```

Security Flaws Summary

Flaw	Description	Risk
Plain Text Passwords	Passwords stored without encryption	Database breach exposes all passwords
No Input Validation	No password strength requirements	Users create weak, guessable passwords
No Rate Limiting	Unlimited login attempts allowed	Vulnerable to brute force attacks
Information Disclosure	Reveals if username exists	Helps attackers enumerate valid usernames

Flaw	Description	Risk
No Account Lockout	No protection after failed attempts	Unlimited guessing possible
Direct Comparison	Simple string comparison	Vulnerable to timing attacks

☑ SECURE Version (Best Practices Implementation)

```
"""
========================================
SECURE LOGIN SYSTEM - BEST PRACTICES
========================================
This implementation demonstrates secure authentication practices.
"""

import hashlib
import secrets
import re
from datetime import datetime, timedelta

class SecureLoginSystem:
    """
    A secure login system implementing authentication best practices.
    """

    def __init__(self):
        # Users stored as: {username: {"password_hash": hash, "salt": salt,
"failed_attempts": 0, "lockout_until": None}}
        self.users = {}
        self.MAX_FAILED_ATTEMPTS = 5
        self.LOCKOUT_DURATION = timedelta(minutes=15)

        # Pre-register a demo user with secure password
        self._create_demo_user()

    def _create_demo_user(self):
        """Create a demo user for testing purposes."""
        salt = secrets.token_hex(32)  # Generate random salt
        password_hash = self._hash_password("SecureP@ss123!", salt)
        self.users["admin"] = {
            "password_hash": password_hash,
            "salt": salt,
            "failed_attempts": 0,
            "lockout_until": None
        }

    def _hash_password(self, password, salt):
        """
```

```
        SECURITY BEST PRACTICE: Hash password with salt using SHA-256.
        In production, use bcrypt, scrypt, or Argon2 for even better
security.
        """
        # Combine password and salt, then hash
        salted_password = password + salt
        return hashlib.sha256(salted_password.encode()).hexdigest()

    def _validate_password_strength(self, password):
        """
        SECURITY BEST PRACTICE: Enforce strong password requirements.
        Returns (is_valid, error_message)
        """
        errors = []

        # Minimum length check
        if len(password) < 8:
            errors.append("Password must be at least 8 characters long")

        # Uppercase letter check
        if not re.search(r'[A-Z]', password):
            errors.append("Password must contain at least one uppercase
letter")

        # Lowercase letter check
        if not re.search(r'[a-z]', password):
            errors.append("Password must contain at least one lowercase
letter")

        # Number check
        if not re.search(r'\d', password):
            errors.append("Password must contain at least one number")

        # Special character check
        if not re.search(r'[!@#$%^&*(),.?":{}|<>]', password):
            errors.append("Password must contain at least one special
character")

        if errors:
            return False, "; ".join(errors)
        return True, "Password meets all requirements"

    def _validate_username(self, username):
        """
        SECURITY BEST PRACTICE: Validate and sanitize username input.
        """
        if not username or len(username) < 3:
            return False, "Username must be at least 3 characters long"
```

```
        if len(username) > 50:
            return False, "Username must not exceed 50 characters"

        # Only allow alphanumeric and underscore
        if not re.match(r'^[a-zA-Z0-9_]+$', username):
            return False, "Username can only contain letters, numbers, and
underscores"

        return True, "Username is valid"

    def _is_account_locked(self, username):
        """Check if account is currently locked due to failed attempts."""
        if username not in self.users:
            return False

        user = self.users[username]
        if user["lockout_until"] and datetime.now() < user["lockout_until"]:
            remaining = (user["lockout_until"] - datetime.now()).seconds //
60
            return True, f"Account locked. Try again in {remaining} minutes."

        # Reset lockout if time has passed
        if user["lockout_until"] and datetime.now() >= user["lockout_until"]:
            user["failed_attempts"] = 0
            user["lockout_until"] = None

        return False, None

    def register(self, username, password):
        """
        Register a new user with validation and secure password storage.
        """
        # Validate username
        valid, msg = self._validate_username(username)
        if not valid:
            return f"Registration failed: {msg}"

        # Check if username exists (case-insensitive)
        if username.lower() in [u.lower() for u in self.users]:
            # SECURITY: Don't reveal if username exists - use generic message
            return "Registration failed: Please choose a different username"

        # Validate password strength
        valid, msg = self._validate_password_strength(password)
        if not valid:
            return f"Registration failed: {msg}"

        # Generate unique salt for this user
        salt = secrets.token_hex(32)
```

```
        # Hash the password with salt
        password_hash = self._hash_password(password, salt)

        # Store user with hashed password
        self.users[username] = {
            "password_hash": password_hash,
            "salt": salt,
            "failed_attempts": 0,
            "lockout_until": None
        }

        return f"User '{username}' registered successfully!"

    def login(self, username, password):
        """
        Authenticate user with secure password verification.
        """
        # Check for account lockout
        locked, msg = self._is_account_locked(username)
        if locked:
            return f"Login failed: {msg}"

        # SECURITY: Use constant-time comparison to prevent timing attacks
        # Also use generic error message to prevent username enumeration

        if username not in self.users:
            # SECURITY: Same message whether username exists or not
            return "Login failed: Invalid username or password"

        user = self.users[username]

        # Hash the provided password with the user's salt
        provided_hash = self._hash_password(password, user["salt"])

        # Compare hashes using constant-time comparison
        if secrets.compare_digest(provided_hash, user["password_hash"]):
            # Reset failed attempts on successful login
            user["failed_attempts"] = 0
            user["lockout_until"] = None
            return "Login successful! Welcome back."
        else:
            # Increment failed attempts
            user["failed_attempts"] += 1

            # Check if should lock account
            if user["failed_attempts"] >= self.MAX_FAILED_ATTEMPTS:
                user["lockout_until"] = datetime.now() +
self.LOCKOUT_DURATION
```

```
                return f"Login failed: Account locked due to too many failed
attempts"

            remaining = self.MAX_FAILED_ATTEMPTS - user["failed_attempts"]
            return f"Login failed: Invalid username or password ({remaining}
attempts remaining)"

    def display_users(self):
        """
        Display user information - passwords are NOT visible (only hashes).
        """
        print("\n[SECURE] User Database (passwords are hashed!):")
        for username, data in self.users.items():
            print(f"  Username: {username}")
            print(f"    Password Hash: {data['password_hash'][:20]}...
(truncated)")
            print(f"    Salt: {data['salt'][:20]}... (truncated)")
            print()

# Demonstrate secure system
print("=" * 60)
print("SECURE LOGIN SYSTEM DEMONSTRATION")
print("=" * 60)

secure_system = SecureLoginSystem()

# Show that passwords are hashed
secure_system.display_users()

# Test registration with weak password
print("\nTest 1: Registering with weak password")
result = secure_system.register("testuser", "weak")
print(f"Result: {result}")

# Test registration with strong password
print("\nTest 2: Registering with strong password")
result = secure_system.register("testuser", "Strong@Pass123")
print(f"Result: {result}")

# Test login
print("\nTest 3: Login with correct credentials")
result = secure_system.login("testuser", "Strong@Pass123")
print(f"Result: {result}")

# Test login with wrong password
print("\nTest 4: Login with wrong password")
result = secure_system.login("testuser", "wrongpassword")
print(f"Result: {result}")
```

```
# Show updated user database
secure_system.display_users()
```

```
    print(f"Result: {result}")

    # Show updated user database
    secure_system.display_users()

[11]

============================================================
SECURE LOGIN SYSTEM DEMONSTRATION
============================================================

[SECURE] User Database (passwords are hashed!):
  Username: admin
    Password Hash: 96998df49d5f47fdeada... (truncated)
    Salt: 863322eb8fa8da3b98a2... (truncated)


Test 1: Registering with weak password
Result: Registration failed: Password must be at least 8 characters long; Password must contain at least one uppercase letter; Password must contain at least one number; Password must contain at least one special character

Test 2: Registering with strong password
Result: User 'testuser' registered successfully!

Test 3: Login with correct credentials
Result: Login successful! Welcome back.

Test 4: Login with wrong password
Result: Login failed: Invalid username or password (4 attempts remaining)

[SECURE] User Database (passwords are hashed!):
  Username: admin
    Password Hash: 96998df49d5f47fdeada... (truncated)
    Salt: 863322eb8fa8da3b98a2... (truncated)

  Username: testuser
    Password Hash: 8376ae994d9e53a1e31e... (truncated)
    Salt: e2a23f441efb127b1da0... (truncated)
```

Best Practices for Secure User Authentication

Practice	Description
Password Hashing	Never store plain text passwords; use strong hashing algorithms (bcrypt, Argon2)
Salting	Add unique random salt to each password before hashing to prevent rainbow table attacks
Input Validation	Validate and sanitize all user inputs to prevent injection attacks
Password Strength	Enforce minimum length, complexity (uppercase, lowercase, numbers, symbols)
Rate Limiting	Limit login attempts to prevent brute force attacks
Account Lockout	Temporarily lock accounts after multiple failed attempts
Generic Error Messages	Don't reveal whether username or password is incorrect
Constant-Time Comparison	Use secure comparison functions to prevent timing attacks
HTTPS	Always transmit credentials over encrypted connections
Multi-Factor Authentication	Add additional verification layers for sensitive systems

Task 5: Privacy in Data Logging

Objective

Demonstrate privacy awareness by comparing an insecure logging system that exposes sensitive data with a privacy-aware version that anonymizes user information.

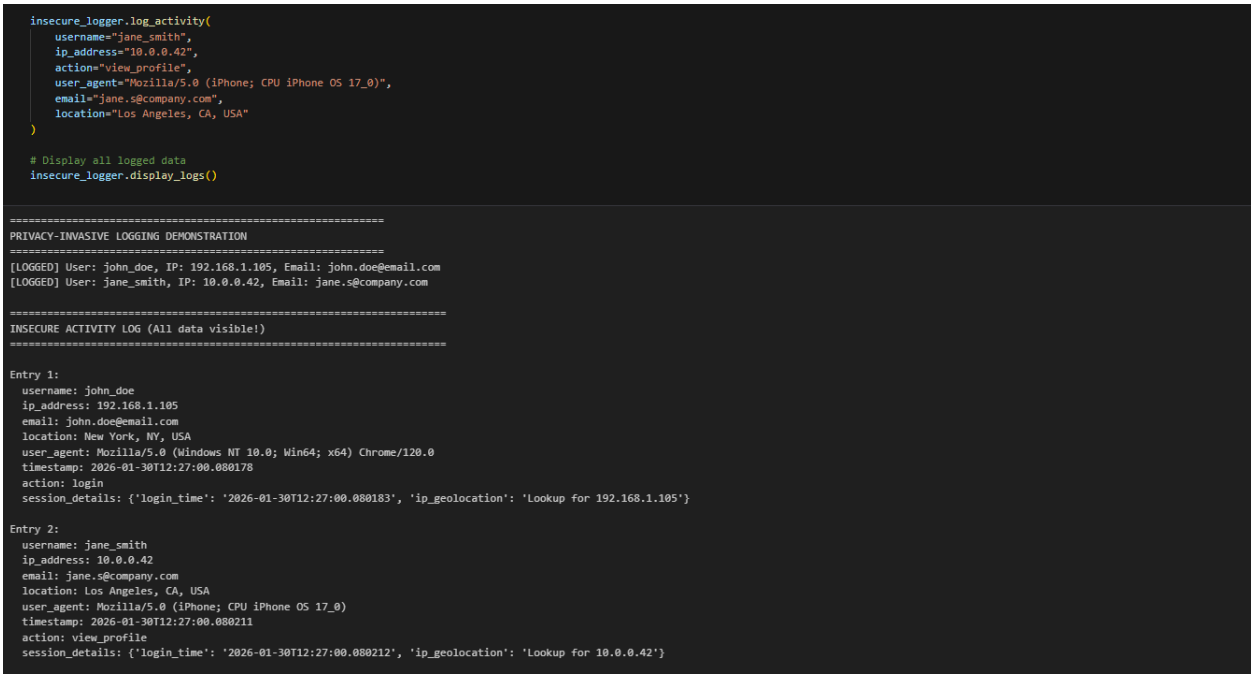
Privacy-Invasive Logging (POOR PRACTICE)

This logging system collects and stores more data than necessary, creating privacy risks.

```
"""
========================================
PRIVACY-INVASIVE LOGGING SYSTEM
========================================
WARNING: This demonstrates poor privacy practices.
DO NOT implement logging this way in production!
"""

from datetime import datetime
import json

class InsecureActivityLogger:
    """
    A logging system that collects excessive personal data.
    This demonstrates what NOT to do regarding user privacy.
    """

    def __init__(self, log_file="activity_log_insecure.json"):
        self.log_file = log_file
        self.logs = []

    def log_activity(self, username, ip_address, action, user_agent=None,
                     email=None, location=None):
        """
        PRIVACY RISK #1: Collects far more data than necessary
        - Full username (not anonymized)
        - Complete IP address (can identify user location)
        - Email address (PII - Personally Identifiable Information)
        - Precise location data
        - Full user agent string (can fingerprint user)
        """

        log_entry = {
            # PRIVACY RISK #2: Storing PII directly
            "username": username,           # Full username visible
            "ip_address": ip_address,       # Complete IP - can track user
            "email": email,                 # Email is sensitive PII
            "location": location,           # Precise location is sensitive
```

```
            # PRIVACY RISK #3: Excessive metadata
            "user_agent": user_agent,        # Can fingerprint browser/device
            "timestamp": datetime.now().isoformat(),  # Precise timing
            "action": action,

            # PRIVACY RISK #4: Storing derived data
            "session_details": {
                "login_time": datetime.now().isoformat(),
                "ip_geolocation": f"Lookup for {ip_address}",  # Even more
tracking
            }
        }

        self.logs.append(log_entry)

        # PRIVACY RISK #5: Logging sensitive data to console
        print(f"[LOGGED] User: {username}, IP: {ip_address}, Email: {email}")

    def display_logs(self):
        """Display all logged data - exposes everything."""
        print("\n" + "=" * 70)
        print("INSECURE ACTIVITY LOG (All data visible!)")
        print("=" * 70)
        for i, log in enumerate(self.logs, 1):
            print(f"\nEntry {i}:")
            for key, value in log.items():
                print(f"  {key}: {value}")

# Demonstrate insecure logging
print("=" * 60)
print("PRIVACY-INVASIVE LOGGING DEMONSTRATION")
print("=" * 60)

insecure_logger = InsecureActivityLogger()

# Log some user activities with full PII
insecure_logger.log_activity(
    username="john_doe",
    ip_address="192.168.1.105",
    action="login",
    user_agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/120.0",
    email="john.doe@email.com",
    location="New York, NY, USA"
)

insecure_logger.log_activity(
    username="jane_smith",
    ip_address="10.0.0.42",
```

```
    action="view_profile",
    user_agent="Mozilla/5.0 (iPhone; CPU iPhone OS 17_0)",
    email="jane.s@company.com",
    location="Los Angeles, CA, USA"
)

# Display all logged data
insecure_logger.display_logs()
```



Privacy Risks in the Insecure Logging System

Risk	Data Collected	Privacy Impact
PII Exposure	Full username, email	Direct identification of users
Location Tracking	Full IP address, geolocation	Physical location can be determined
Device Fingerprinting	Full user agent string	Users can be tracked across sessions
Excessive Collection	More data than needed	Violates data minimization principle
Console Logging	Sensitive data printed	Exposes data in logs/terminals
No Retention Policy	Data stored indefinitely	Increases breach impact over time

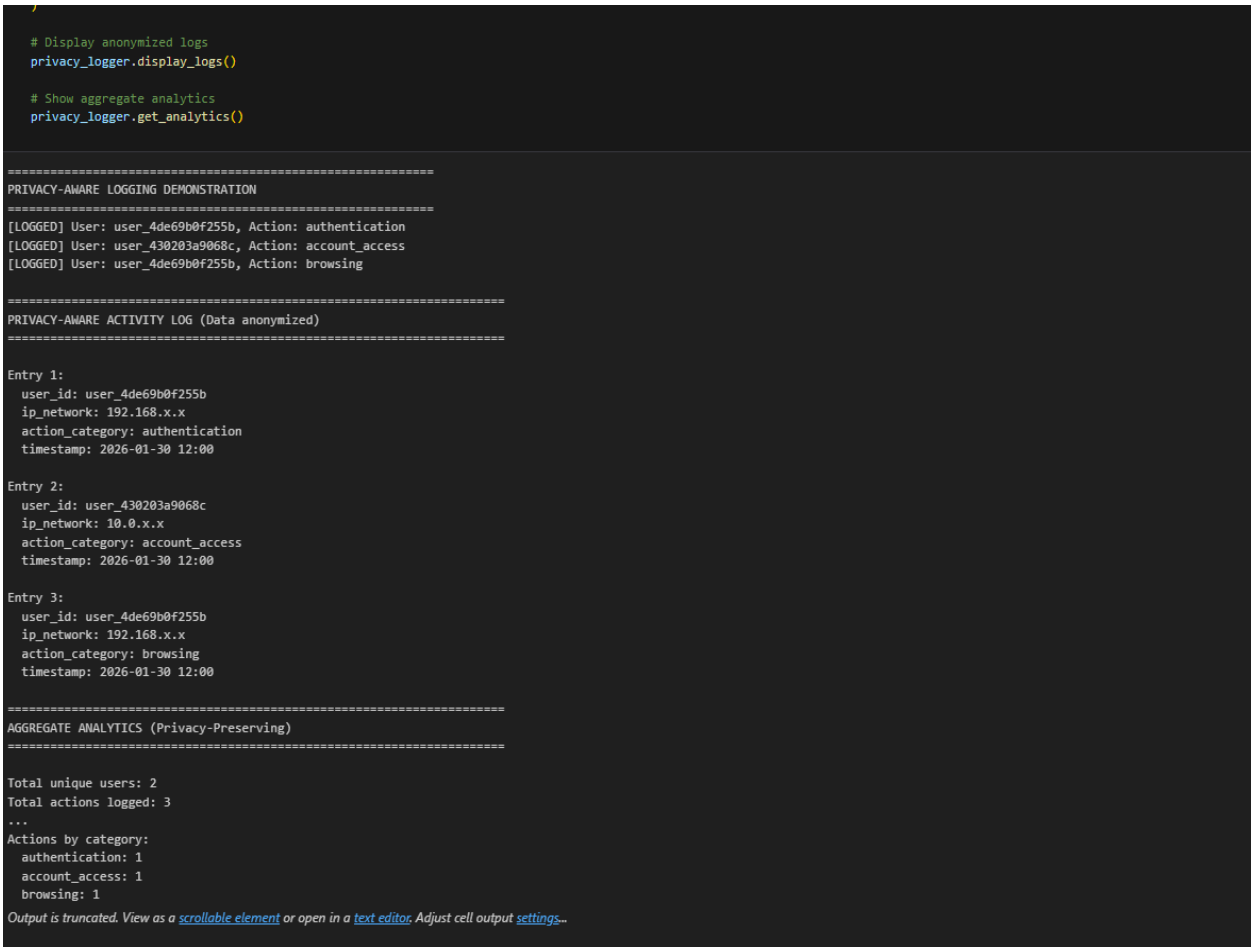
Privacy-Aware Logging (BEST PRACTICE)

```
"""
========================================
PRIVACY-AWARE LOGGING SYSTEM
========================================
This implementation demonstrates privacy-by-design principles.
"""

import hashlib
from datetime import datetime

class PrivacyAwareActivityLogger:
    """
    A logging system that respects user privacy through:
    - Data minimization (collect only what's necessary)
    - Anonymization (mask or hash identifying information)
    - Purpose limitation (log only for specific, legitimate purposes)
    """

    def __init__(self, log_file="activity_log_secure.json"):
        self.log_file = log_file
        self.logs = []

    def _anonymize_username(self, username):
        """
        PRIVACY PRACTICE #1: Anonymize usernames
        Use a one-way hash so we can track unique users without
        storing their actual identity.
        """
        # Create a salted hash of the username
        salt = "privacy_salt_2024"  # In production, use secure random salt
        anonymized = hashlib.sha256((username +
salt).encode()).hexdigest()[:12]
        return f"user_{anonymized}"

    def _mask_ip_address(self, ip_address):
        """
        PRIVACY PRACTICE #2: Mask IP addresses
        Keep only the network portion, remove host-identifying octets.
        Example: 192.168.1.105 → 192.168.x.x
        """
        parts = ip_address.split('.')
        if len(parts) == 4:
            # Mask the last two octets (host portion)
            return f"{parts[0]}.{parts[1]}.x.x"
        return "masked_ip"

    def _get_minimal_timestamp(self):
        """
```

```
        PRIVACY PRACTICE #3: Reduce timestamp precision
        Use only date and hour, not precise seconds/milliseconds.
        """
        now = datetime.now()
        return now.strftime("%Y-%m-%d %H:00")  # Round to the hour

    def _categorize_action(self, action):
        """
        PRIVACY PRACTICE #4: Categorize instead of logging exact actions
        This reduces the granularity of tracked behavior.
        """
        action_categories = {
            "login": "authentication",
            "logout": "authentication",
            "view_profile": "account_access",
            "update_profile": "account_access",
            "search": "browsing",
            "view_page": "browsing",
            "purchase": "transaction",
            "download": "content_access"
        }
        return action_categories.get(action, "general")

    def log_activity(self, username, ip_address, action):
        """
        Log user activity with privacy protections.

        Key differences from insecure version:
        - Username is anonymized (hashed)
        - IP address is masked
        - Timestamp precision is reduced
        - Only essential data is collected
        - No email, location, or user agent stored
        """

        log_entry = {
            # Anonymized/minimal data only
            "user_id": self._anonymize_username(username),  # Hashed, not
plain text
            "ip_network": self._mask_ip_address(ip_address),  # Only network
portion
            "action_category": self._categorize_action(action),  #
Categorized
            "timestamp": self._get_minimal_timestamp(),  # Reduced precision
        }

        self.logs.append(log_entry)

        # Privacy-aware console output (no PII)
```

```
        print(f"[LOGGED] User: {log_entry['user_id']}, Action:
{log_entry['action_category']}")

    def display_logs(self):
        """Display logs - all data is anonymized/masked."""
        print("\n" + "=" * 70)
        print("PRIVACY-AWARE ACTIVITY LOG (Data anonymized)")
        print("=" * 70)
        for i, log in enumerate(self.logs, 1):
            print(f"\nEntry {i}:")
            for key, value in log.items():
                print(f"  {key}: {value}")

    def get_analytics(self):
        """
        PRIVACY PRACTICE #5: Aggregate analytics instead of individual
tracking
        Provide useful insights without exposing individual user data.
        """
        print("\n" + "=" * 70)
        print("AGGREGATE ANALYTICS (Privacy-Preserving)")
        print("=" * 70)

        # Count actions by category
        action_counts = {}
        unique_users = set()

        for log in self.logs:
            action = log["action_category"]
            action_counts[action] = action_counts.get(action, 0) + 1
            unique_users.add(log["user_id"])

        print(f"\nTotal unique users: {len(unique_users)}")
        print(f"Total actions logged: {len(self.logs)}")
        print("\nActions by category:")
        for action, count in action_counts.items():
            print(f"  {action}: {count}")

# Demonstrate privacy-aware logging
print("=" * 60)
print("PRIVACY-AWARE LOGGING DEMONSTRATION")
print("=" * 60)

privacy_logger = PrivacyAwareActivityLogger()

# Log the same activities - notice the difference in what's stored
privacy_logger.log_activity(
    username="john_doe",
    ip_address="192.168.1.105",
```

```
    action="login"
)

privacy_logger.log_activity(
    username="jane_smith",
    ip_address="10.0.0.42",
    action="view_profile"
)

privacy_logger.log_activity(
    username="john_doe",
    ip_address="192.168.1.105",
    action="search"
)

# Display anonymized logs
privacy_logger.display_logs()

# Show aggregate analytics
privacy_logger.get_analytics()
```



Privacy-Aware Logging Principles

1. Data Minimization

Collect only the minimum data necessary for the intended purpose. If you don't need an email to track activity, don't collect it.

2. Anonymization & Pseudonymization

- **Anonymization**: Remove all identifying information (irreversible)
- **Pseudonymization**: Replace identifiers with pseudonyms (reversible with key)

3. Purpose Limitation

Data should only be collected for specific, legitimate purposes and not used beyond that scope.

4. IP Address Masking

Mask the last octets of IP addresses to prevent precise geolocation while retaining network-level information for analytics.

5. Aggregate Analytics

Provide insights through aggregate statistics rather than individual user tracking.

6. Data Retention Policies

- Define how long data is kept
- Automatically delete data after retention period
- Reduce precision over time

Why Privacy-Aware Logging Matters:

Reason	Explanation
Legal Compliance	GDPR, CCPA, and other regulations require privacy protection
User Trust	Users are more likely to use services that respect their privacy
Breach Impact	Less sensitive data means less damage if breach occurs
Ethical Responsibility	Respecting user privacy is the right thing to do
Business Risk	Privacy violations lead to fines, lawsuits, and reputational damage

Conclusion: Importance of AI Transparency, Security, and Privacy

Summary of Key Learnings

Transparency

Why it matters: Transparent algorithms build trust and enable verification.

- **Algorithm Optimization** (Task 1): Clear documentation of time complexity helps developers choose the right approach. The difference between O(n) and O(√n) can be critical for large-scale applications.
- **Recursive Algorithms** (Task 2): Understanding base cases and recursive calls is essential for debugging and predicting program behavior. Transparent code with proper comments makes maintenance easier.
- **Error Handling** (Task 3): Explicit error handling with clear messages helps users understand what went wrong and how to fix it. This reduces support burden and improves user experience.

Security

Why it matters: Security breaches can lead to financial loss, legal liability, and reputation damage.

- **User Authentication** (Task 4): The comparison between insecure (plain text passwords) and secure (hashed with salt) authentication demonstrates the critical importance of following security best practices.
- Key takeaways:
  - Never store passwords in plain text
  - Use strong hashing algorithms with unique salts
  - Implement rate limiting and account lockout
  - Validate all user inputs
  - Use generic error messages to prevent information disclosure

Privacy

Why it matters: Privacy is a fundamental right and a legal requirement in many jurisdictions.

- **Data Logging** (Task 5): The contrast between privacy-invasive and privacy-aware logging shows how to collect useful data while respecting user privacy.
- Key principles:
  - Data minimization: Collect only what you need
  - Anonymization: Mask or hash identifying information

- Purpose limitation: Use data only for stated purposes
- Aggregate analytics: Provide insights without tracking individuals

Real-World Applications

These principles apply across all software development:

Domain	Transparency	Security	Privacy
Healthcare	Explainable AI diagnoses	Protected health records	HIPAA compliance
Finance	Auditable algorithms	Secure transactions	Financial data protection
E-commerce	Clear pricing algorithms	Secure payment processing	Customer data minimization
Social Media	Transparent content moderation	Account security	User data privacy

Final Thoughts

Building trustworthy AI systems requires a commitment to:

1. **Transparency**: Make your code understandable and your algorithms explainable
1. **Security**: Protect systems and data from unauthorized access
2. **Privacy**: Respect user privacy and collect only necessary data

These are not optional features—they are fundamental requirements for responsible software development in the modern digital age.

- Roll Number: 2303A51206*