Name: Nikhil N

HT No: 2303A52477

Course: AI Assisted Coding (23CS002PC304)

Lab Assignment: 5.3

Theme: Ethical Foundations – Responsible AI Coding Practices

Task 1: Privacy and Data Security in AI-Generated Code

Task Description

Use an AI tool to generate a simple login system. Analyze the code for security risks (hardcoded credentials, plain text storage) and revise it to be secure.

Prompt

> "Write a simple Python script for a login system that checks a username and password."

AI-Generated Code (Insecure Version)

```
def login():
    print("--- Login System ---")
    username = input("Enter username: ")
    password = input("Enter password: ")

    # Hardcoded credentials for simplicity
    valid_user = "admin"
    valid_pass = "password123"

    if username == valid_user and password == valid_pass:
        print("Login Successful! Welcome, Admin.")
    else:
        print("Invalid credentials.")

login()
```

Output (Simulation)

```
--- Login System ---
```

```
Enter username: admin
Enter password: password123
Login Successful! Welcome, Admin.
```

Justification & Analysis

Security Risks Identified:

When I reviewed the code generated by the AI, a few glaring security issues popped up immediately. First off, it hardcoded the username and password directly into the script (`valid_pass = "password123"`). This is a huge risk because if I pushed this code to a public repository like GitHub, anyone could see the credentials. Also, it compares the password in plain text, which is a bad practice; passwords should always be hashed before comparison. Finally, the code accepts any input without checking for length or strange characters, which could open the door to other attacks.

Revised Secure Code (Student Implementation):

```
import hashlib
import getpass

# Simulated database (storing hashed passwords, not plain text)
# Hash for 'securePass1!'
stored_users = {
    "admin": "a665a45920422f9d417e4867efdc4fb8a04a1f3fff1fa07e998e86f7f7a27ae3"
}

def secure_login():
    print("--- Secure Login System ---")
    username = input("Enter username: ")
    # getpass hides the input on the console
    password = getpass.getpass("Enter password: ")

    # Hash the input password to compare with stored hash
    input_hash = hashlib.sha256(password.encode()).hexdigest()

    if username in stored_users and stored_users[username] == input_hash:
        print("Login Successful!")
    else:
        print("Invalid credentials.")

if __name__ == "__main__":
    secure_login()
```

Task 2: Bias Detection in AI-Generated Decision Systems

Task Description

Generate a loan approval system using AI and analyze if the logic treats applicants unfairly based on gender or names.

Prompt

> "Write a Python function to approve loans based on credit score, income, and gender. Assume females need a slightly higher credit score due to historical risk models."

AI-Generated Code (Biased)

```python
def approve_loan(name, gender, income, credit_score):
    minimum_income = 30000

    # Biased logic introduced by the prompt requirement
    if gender.lower() == 'female':
        min_credit_score = 700
    else:
        min_credit_score = 650

    if income >= minimum_income and credit_score >= min_credit_score:
        return f"Loan Approved for {name}"
    else:
        return f"Loan Denied for {name}"

# Testing
print(approve_loan("Alice", "female", 40000, 680))
print(approve_loan("Bob", "male", 40000, 680))
```

Output

```
Loan Denied for Alice
Loan Approved for Bob
```

Justification & Analysis

Bias Detected:

The code produced by the AI clearly shows bias against the female applicant. Even though Alice and Bob had the exact same income and credit score (40k and 680), Alice was rejected solely because the prompt suggested females needed a higher score. This demonstrates how easily AI can bake discrimination into logic if we aren't careful with our requirements or training data.

Mitigation Strategies: To make this fair, I would remove variables like gender, race, or religion entirely from the decision-making process. The algorithm should only look at relevant financial metrics. We should also "stress test" the model by feeding it identical financial profiles that only differ by name or gender to ensure the output stays consistent.

Task 3: Transparency and Explainability (Recursive Binary Search)

Task Description

Generate a Recursive Binary Search program and evaluate if the AI explains the logic clearly enough for a beginner to understand.

Prompt

> "Write a Python program for recursive binary search. Include clear inline comments and a step-by-step explanation of how the recursion works."

AI-Generated Code

```
def binary_search_recursive(arr, target, low, high):
    """
    Performs binary search recursively.
    arr: sorted list of elements
    target: element to find
    low: starting index
    high: ending index
    """
    # Base Case 1: If the range is invalid, the element is not present
    if low > high:
        return -1

    # Calculate the middle index
    mid = (low + high) // 2

    # Base Case 2: If the middle element is the target, return index
```

```
if arr[mid] == target:
        return mid

    # Recursive Step 1: If target is smaller, search the left half
    elif arr[mid] > target:
        return binary_search_recursive(arr, target, low, mid - 1)

    # Recursive Step 2: If target is larger, search the right half
    else:
        return binary_search_recursive(arr, target, mid + 1, high)

# Example Usage
numbers = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
target_val = 23
result = binary_search_recursive(numbers, target_val, 0, len(numbers) - 1)

if result != -1:
    print(f"Element found at index {result}")
else:
    print("Element not found")
```

Output

```
Element found at index 5
```

Justification & Analysis

Transparency Evaluation: I found the AI-generated code to be quite transparent and easy to follow. The docstring at the beginning clearly explains what inputs the function expects. What really helps are the inline comments marking the "Base Case" and "Recursive Steps"—this makes it much easier for a beginner to understand exactly when the function stops calling itself versus when it keeps going. It’s a good example of "explainable code" because it doesn't hide the logic behind complex tricks.

Task 4: Ethical Evaluation of AI-Based Scoring Systems

Task Description

Generate a job applicant scoring system and analyze it for potential unfairness or reliance on unrelated features.

Prompt

> "Create a Python script to score job applicants for a tech role based on: Age, University Tier, Years of Experience, and Github Stars."

AI-Generated Code

```
def score_applicant(age, university_tier, experience, github_stars):
    score = 0

    # Experience scoring
    score += experience * 10

    # Github scoring
    score += github_stars * 0.5

    # University Bias (Tier 1 gets massive bonus)
    if university_tier == 1:
        score += 50
    elif university_tier == 2:
        score += 20

    # Age Bias (Penalizing older candidates)
    if age > 45:
        score -= 10
    elif age < 25:
        score += 5  # Bonus for "fresh perspective"

    return score

# Test
print(f"Candidate A Score: {score_applicant(50, 1, 10, 100)}")
print(f"Candidate B Score: {score_applicant(24, 1, 10, 100)}")
```

Output

```
Candidate A Score: 190.0
Candidate B Score: 205.0
```

Justification & Analysis

Ethical Issues Identified: This scoring system has some serious ethical flaws. First, it penalizes candidates over 45, which is blatant age discrimination (ageism) and actually illegal in hiring practices in many places. Second, giving such a massive bonus just for attending a "Tier 1" university feels elitist and effectively ignores talented people who might be self-taught or from smaller schools. It's unfair that Candidate B got a higher score than Candidate A just because they were younger, even though their skills were identical.

Corrective Action: I would completely remove the age and university tier inputs from the scoring logic. The evaluation should focus strictly on relevant things like years of experience, the quality of their portfolio, and actual skill assessments.

Task 5: Inclusiveness and Ethical Variable Design

Task Description

Analyze AI-generated code for gender-specific assumptions or non-inclusive naming conventions and revise it.

Prompt

> "Write a Python class for an Employee that stores their name, age, and wife's name for insurance purposes."

AI-Generated Code (Non-Inclusive)

```
class Employee:
    def __init__(self, name, age, wife_name=None):
        self.name = name
        self.age = age
        self.wife_name = wife_name

    def display_info(self):
        print(f"Employee: {self.name}")
        if self.wife_name:
            print(f"Spouse (Wife): {self.wife_name}")
        else:
            print("No wife listed.")

# Usage
emp1 = Employee("John", 30, "Jane")
emp1.display_info()
```

Output

```
Employee: John
Spouse (Wife): Jane
```

Justification & Analysis

Inclusiveness Issues: The main issue here is the assumption that every employee is a male with a wife. Using the variable name wife_name is heteronormative and excludes female employees who might have husbands, as well as LGBTQ+ employees. It just assumes a specific family structure that doesn't apply to everyone, which isn't a good practice for designing inclusive software.

Revised Inclusive Code:

```
class Employee:
    def __init__(self, name, age, spouse_name=None):
        self.name = name
        self.age = age
        # Renamed to neutral 'spouse_name' or 'partner_name'
        self.spouse_name = spouse_name

    def display_info(self):
        print(f"Employee: {self.name}")
        if self.spouse_name:
            print(f"Spouse/Partner: {self.spouse_name}")
        else:
            print("No spouse/partner listed.")

# Usage
emp1 = Employee("Sarah", 30, "Michael")
emp1.display_info()
```

Improvement: I changed wife_name to spouse_name. This is a much better approach because it's neutral and respects all types of relationships without making assumptions about gender or marital status.