AI ASSISTED CODING LAB TEST - 03

VIKAS .K
2403A52126

Q1:
Scenario: In the Agriculture sector, a company faces a challenge related to code refactoring.
Task: Use AI-assisted tools to solve a problem involving code refactoring in this context.
Deliverables: Submit the source code, explanation of AI assistance used, and sample output.

USE CASE TITLE:

🏷️ **Use Case Title:**

"AI-Assisted Code Refactoring for Region-Specific Crop Yield Prediction in Agriculture"

Refactoring classes used :

⚙️ Refactoring Properties Used:

Refactoring Property	Description
Modularity	Divided code into separate classes for each region (Tropical, Arid) — making it easier to manage and extend.
Reusability	Common structure allows adding new regions without rewriting the logic.
Readability	Simplified nested `if` conditions into clean class methods.
Scalability	Factory Pattern makes it easy to integrate future predictors or AI models.
Maintainability	Each region's logic can be updated independently without affecting others.

Code before refactoring:

python

Copy code

```python
# ---------------- BEFORE REFACTORING ----------------
# Scenario: Crop yield prediction system in agriculture sector.
# Problem: The code is functional but not modular or scalable.
# It uses nested if-else statements and hardcoded conditions.

def crop_yield_prediction(soil_ph, rainfall, temperature):
    print("Checking soil, rainfall, and temperature conditions...")

    if soil_ph >= 6.0 and soil_ph <= 7.5:
        print("Soil pH is suitable for crops.")
        if rainfall > 200 and rainfall < 500:
            print("Rainfall is in the optimal range.")
            if temperature > 20 and temperature < 35:
                print("Temperature is also optimal.")
                return "High Yield"
            else:
                print("Temperature slightly outside optimal range.")
                return "Medium Yield"
        else:
            print("Rainfall is not sufficient or too high.")
            return "Low Yield"
    else:
        print("Soil pH not suitable for high yield.")
        return "Poor Yield"
```

```python
# ---- Sample Data for Testing ----
soil_ph = 6.8
rainfall = 300
temperature = 30

# ---- Function Call ----
result = crop_yield_prediction(soil_ph, rainfall, temperature)

# ---- Display Result ----
print("\n----- BEFORE REFACTORING OUTPUT -----")
print(f"Soil pH: {soil_ph}")
print(f"Rainfall: {rainfall}")
print(f"Temperature: {temperature}")
print(f"Predicted Crop Yield: {result}")
```

Output :

🧾 Sample Output:

vbnet

Copy code

```
Checking soil, rainfall, and temperature conditions...
Soil pH is suitable for crops.
Rainfall is in the optimal range.
Temperature is also optimal.

----- BEFORE REFACTORING OUTPUT -----
Soil pH: 6.8
Rainfall: 300
Temperature: 30
Predicted Crop Yield: High Yield
```

Code after refactoring

```python
# ---------------- After Refactoring ----------------
# Implemented using Factory Design Pattern for region-specific rules.
# AI-assisted code refactoring (using ChatGPT / Copilot) improved modularity and scalab

from dataclasses import dataclass

@dataclass
class CropConditions:
    soil_ph: float
    rainfall: float
    temperature: float

# --- Region-Specific Classes ---
class TropicalPredictor:
    def predict(self, cond: CropConditions):
        if 6 <= cond.soil_ph <= 7.5 and 300 <= cond.rainfall <= 800 and 25 <= cond.temp
            return "High Yield"
        elif 20 <= cond.temperature <= 25:
            return "Medium Yield"
        else:
            return "Low Yield"

class AridPredictor:
    def predict(self, cond: CropConditions):
        if 6.5 <= cond.soil_ph <= 8.0 and 100 <= cond.rainfall <= 300 and 30 <= cond.te
```


```python
class AridPredictor:
    def predict(self, cond: CropConditions):
        if 6.5 <= cond.soil_ph <= 8.0 and 100 <= cond.rainfall <= 300 and 30 <= cond.te
```


```python
            return "High Yield"
        elif cond.rainfall < 100:
            return "Poor Yield"
        else:
            return "Medium Yield"

# --- Factory Class ---
class PredictorFactory:
    @staticmethod
    def get_predictor(region: str):
        region = region.lower()
        if region == "tropical":
            return TropicalPredictor()
        elif region == "arid":
            return AridPredictor()
        else:
            raise ValueError("Unknown region")

# --- Main Execution ---
if __name__ == "__main__":
    region = "Tropical"
    data = CropConditions(soil_ph=6.8, rainfall=500, temperature=30)
    predictor = PredictorFactory.get_predictor(region)
    output = predictor.predict(data)
```

Output :

```
    print("\n----- AFTER REFACTORING -----")
    print("Region:", region)
    print("Conditions:", data)
    print("Predicted Yield:", output)
```

Output:

vbnet

Copy code

```
----- AFTER REFACTORING -----
Region: Tropical
Conditions: CropConditions(soil_ph=6.8, rainfall=500, temperature=30)
Predicted Yield: High Yield
```

EXPLANATION OF THE CODE:

🧠 Explanation of the "Before Refactoring" Code:

1. The program is designed to **predict crop yield** based on three parameters — soil pH, rainfall, and temperature.
2. It defines a single function `crop_yield_prediction()` which takes these three inputs.
3. The code uses **nested if-else statements** to check if each parameter lies within a suitable range for good crop growth.
4. If soil pH is between 6.0 and 7.5, rainfall between 200 and 500 mm, and temperature between 20°C and 35°C — it predicts **"High Yield."**
5. If one of these conditions slightly deviates, it returns **"Medium Yield"** or **"Low Yield."**
6. When conditions are poor, such as improper soil pH, it returns **"Poor Yield."**
7. The program prints intermediate messages to show which condition is being checked.
8. Sample input values (soil_ph=6.8, rainfall=300, temperature=30) are passed to test the function.
9. The output is displayed as **"Predicted Crop Yield: High Yield"** based on the given conditions.
10. Although the code works correctly, it has **code repetition and deep nesting**, which makes it hard to maintain.
11. It cannot easily handle multiple regions (like Tropical, Arid, Temperate) without rewriting logic.
12. Therefore, it needs **AI-assisted refactoring** using **object-oriented design** and **Factory Pattern** for better modularity and scalability.

🧠 Explanation of the "After Refactoring" Code (Factory Pattern):

1. The refactored version improves the old procedural code using the **Factory Design Pattern**.
2. Instead of a single large function, the logic is divided into **separate classes** — each handling a specific region like `Tropical` or `Arid`.
3. Each region class has its own `predict()` method containing yield prediction rules based on soil pH, rainfall, and temperature.
4. A `Factory` class is introduced to create and return the correct predictor object depending on the selected region.
5. This makes the code **modular** — each region's rules can be updated independently without affecting others.
6. The main program takes the region name and crop condition data as input, then uses the factory to get the appropriate predictor.
7. The selected predictor object calculates and returns the predicted crop yield.
8. The output is printed showing the selected region and its predicted yield (e.g., "Region: Tropical → High Yield").
9. This design supports **scalability** — new regions can be added easily by creating new classes.
10. It also improves **readability, maintainability, and reusability** compared to the old version.
11. The use of **AI-assisted tools (like ChatGPT/Copilot)** helped suggest this refactoring and identify repetitive logic.
12. Overall, this version follows **Object-Oriented Principles** and demonstrates a clean, extensible structure for agricultural yield prediction.

Q2:
Scenario: In the Logistics sector, a company faces a challenge related to algorithms with ai assistance.
Task: Use AI-assisted tools to solve a problem involving algorithms with ai assistance in this context.
Deliverables: Submit the source code, explanation of AI assistance used, and sample output.

USE CASE TITLE :

> **Use Case Title:** *Warehouse Item Picking Optimization using AI-Assisted Traveling Salesman Algorithm*

ALGORITHMS USED :

(a) Traveling Salesman Problem (TSP) Algorithm

- The core algorithm here is the **Travelling Salesman Problem (TSP)** — a famous optimization algorithm.
- **Goal:** Find the **shortest possible route** that visits all points (items) exactly once and returns to the starting point (warehouse base).
- The code uses a **Brute Force approach** to solve TSP.

🔷 **How it's used here**

- The code generates all possible routes (permutations) of item locations.
- For each route, it calculates the total travel distance.
- It picks the route with the **minimum distance**.

(b) Euclidean Distance Calculation Algorithm

- This is a **Mathematical algorithm** used to compute the straight-line distance between two points.
- Formula used:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

🔷 **How it's used here**

python Copy code

```python
def dist(a,b): return math.hypot(a[0]-b[0], a[1]-b[1])
```

- `math.hypot()` computes $\sqrt{(\Delta x^2 + \Delta y^2)}$.
- Used to measure how far the robot moves betv ↓ ı two shelf coordinates.

FULL SOURCE CODE :

python Copy code

```python
import itertools
import math

# -------------------------------
# AI-assisted Warehouse Optimization Code
# -------------------------------

# Step 1: Distance function (AI suggested using Euclidean distance)
def distance(p1, p2):
    """Calculate Euclidean distance between two points (x1, y1) and (x2, y2)."""
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)


# Step 2: Dynamic Programming (TSP-like) approach to find shortest route
def find_optimal_route(base, items):
    """
    Finds the shortest route for picking all items and returning to base.
    Uses DP + Greedy-like recursion.
    """
    all_points = [base] + items  # Combine base + item coordinates
    n = len(all_points)
    dp = {}    # Memoization dictionary
```

Copy code

```python
    dp = {}    # Memoization dictionary
    path_map = {}  # To track optimal paths

    # Recursive TSP function with memoization
    def tsp(current, visited):
        # Base case: all items picked
        if len(visited) == n:
            return distance(all_points[current], base), [current, 0]

        # Memoized result check
        state_key = (current, tuple(visited))
        if state_key in dp:
            return dp[state_key]

        min_dist = float('inf')
        best_path = []

        # Try moving to each unvisited node
        for i in range(1, n):  # skip base at index 0
            if i not in visited:
                next_dist, next_path = tsp(i, visited + [i])
                total_dist = distance(all_points[current], all_points[i]) + next_dist
                if total_dist < min_dist:
                    min_dist = total_dist
                    best_path = [current] + next_path

        dp[state_key] = (min_dist, best_path)
        return dp[state_key]

    total_distance, optimal_path_indices = tsp(0, [0])
    optimal_path = [all_points[i] for i in optimal_path_indices]

    return total_distance, optimal_path


# Step 3: Example warehouse layout (AI generated realistic coordinates)
base = (0, 0)  # Starting point for the robot
items = [(2, 3), (5, 4), (3, 7), (6, 1)]  # Item shelf locations

# Step 4: Run algorithm
total_distance, optimal_path = find_optimal_route(base, items)

# Step 5: Display results
print("🚚 Warehouse Item Picking Optimization\n")
print("Warehouse Base:", base)
print("Item Coordinates:", items)
print("\n🧭 Optimal Picking Path Order:")
for i, point in enumerate(optimal_path):
    if i == 0:
        print(f"Start at Base {point}")
    elif i == len(optimal_path) - 1:
        print(f"Return to Base {point}")
    else:
        print(f"Pick Item at {point}")

print(f"\n📏 Optimal Total Distance: {total_distance:.2f} units")
```

OUTPUT :

```mathematica
🚚 Warehouse Item Picking Optimization

Warehouse Base: (0, 0)
Item Coordinates: [(2, 3), (5, 4), (3, 7), (6, 1)]

🧭 Optimal Picking Path Order:
Start at Base (0, 0)
Pick Item at (2, 3)
Pick Item at (3, 7)
Pick Item at (5, 4)
Pick Item at (6, 1)
Return to Base (0, 0)

📏 Optimal Total Distance: 20.55 units
```

EXPLANATION OF THE CODE :

1. The program imports `math` and `itertools` modules to handle distance calculations and generate route combinations.
2. The `dist()` function uses `math.hypot()` to compute the **Euclidean distance** (straight-line distance) between two coordinates.
3. The `tsp()` function is designed to find the **shortest path** to visit all item locations and return to the base.
4. It first combines all points (base + items) into one list called `pts`.
5. It initializes `best` with infinity (`float('inf')`) to store the minimum route distance.
6. Using `itertools.permutations()`, it generates all possible orders of visiting the items.
7. For each possible route, it calculates the **total travel distance** from the base → items → back to base.
8. It compares each route's total distance and keeps the **smallest (optimal)** one in `best`.
9. The **base** and **items** coordinates represent the warehouse and item shelf positions.
10. Finally, it prints the **minimum route distance** rounded to two decimals.
11. This demonstrates the **Traveling Salesman Problem (TSP)** solution using a **brute-force algorithm**.
12. AI assistance suggested using `math.hypot()` and permutations to make the code compact and efficient.