Task-1:

Use an AI tool to generate a Python program that connects to a weather API.

Prompt:

Generate code to fetch weather data securely without exposing API keys in the code

> Generate code to fetch weather data securely without exposing API keysin the code

Code and Output:

```python
import os
import requests

API_KEY = os.getenv("WEATHER_API_KEY")
if not API_KEY:
    raise RuntimeError("Missing WEATHER_API_KEY environment variable")

CITY = "London"
URL = "https://api.openweathermap.org/data/2.5/weather"

params = {
    "q": CITY,
    "appid": API_KEY,
    "units": "metric"
}

response = requests.get(URL, params=params, timeout=10)
response.raise_for_status()

data = response.json()
print(f"{CITY}: {data['main']['temp']}°C")
```

Code:

Windows (PowerShell)

powershell Copy code

```powershell
setx WEATHER_API_KEY "your_api_key_here"
```

```python
import os
import requests

def get_weather(city: str) -> dict:
    api_key = os.getenv("WEATHER_API_KEY")
    if not api_key:
        raise RuntimeError("WEATHER_API_KEY not set")

    url = "https://api.openweathermap.org/data/2.5/weather"
    params = {
        "q": city,
        "appid": api_key,
        "units": "metric"
    }

    response = requests.get(url, params=params, timeout=10)
    response.raise_for_status()
    return response.json()


if __name__ == "__main__":
    data = get_weather("London")
    print(f"{data['name']}: {data['main']['temp']}°C")
```

Explanation:

1. The weather API key is set as an environment variable instead of being written in the Python code.
2. The program retrieves the key securely at runtime using os.getenv().
3. The code checks for a missing key and raises an error to prevent insecure execution.
4. The API request sends the key only in the request parameters, not in logs or files.
5. This method keeps the key safe while remaining compatible with local and production setups.

Task-2:

Task: Use an AI tool to generate a Python script that stores user data (name, email, password) in a file.
Analyse: Check if the AI stores sensitive data in plain text or without encryption

Prompt:

Generate a Python script that collects user details (name, email, and password) and stores them in a file. After generating the script, analyze whether the password and other sensitive data are stored in plain text or without encryption/hashing. Clearly state if the implementation is secure or insecure and explain why.

Code and Output:

python Copy code

```python
def collect_user_data():
    name = input("Enter your name: ")
    email = input("Enter your email: ")
    password = input("Enter your password: ")

    with open("users.txt", "a") as file:
        file.write(f"Name: {name}, Email: {email}, Password: {password}\n")

    print("User data saved successfully.")


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

] 6s

```python
def collect_user_data():
    name = input("Enter your name: ")
    email = input("Enter your email: ")
    password = input("Enter your password: ")

    with open("users.txt", "a") as file:
        file.write(f"Name: {name}, Email: {email}, Password: {password}\n")

    print("User data saved successfully.")


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

```
Enter your name: Manisha
Enter your email: manishaboddu537@gmail.com
Enter your password: Manisha
User data saved successfully.
```

Explanation:

1. The script collects the user’s name, email, and password and writes them directly to a file.
2. The password is stored in plain text, making it readable by anyone who accesses the file.
3. No encryption or hashing technique is applied to protect sensitive data.
4. This exposes users to serious security risks if the file is leaked or compromised.
5. Therefore, the implementation is insecure and does not follow data security best practices.

Task-3:

Use AI to generate an Armstrong number checking function
with comments and explanations.

Prompt:

Generate a Python function to check whether a number is an Armstrong number, with brief comments and a short explanation.

Code and Output:

```
def is_armstrong(number):
    digits = str(number)
    power = len(digits)
    total = 0

    for d in digits:
        total += int(d) ** power

    return total == number
```

```
User data saved successfully.
```

```
[2]
0s
def is_armstrong(number):
    digits = str(number)
    power = len(digits)
    total = 0

    for d in digits:
        total += int(d) ** power

    return total == number
```

Explanation:

1. def is _ Armstrong (number):
   Defines a function named is _ armstrong that takes a number as input.
2. digits = str(number)
   Converts the number into a string so each digit can be accessed individually.
3. power = len (digits)
   Counts how many digits are in the number.
4. total = 0
   Initialises a variable to store the sum of powered digits.
5. for d in digits:
   Iterates through each digit in the number.
6. total += int(d) ** power
   Converts the digit back to an integer, raises it to the power of the total number of digits, and adds it to the total.
7. return total == number
   Compares the calculated sum with the original number and returns True if they are equal, otherwise False.

Task-4:

Use AI to implement two sorting algorithms (e.g., Quick Sort and Bubble Sort.

Prompt:

Generate Python code for Quick Sort and Bubble Sort, and include comments explaining step-by-step how each works and where they differ."

Generate Python code for QuickSort and BubbleSort, and includecomments explaining step-by-step how each works and where they differ."Code for both algorithms.
• Transparent, comparative explanation of their logic and efficiency.

Code and Output:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr


def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quick_sort(left) + middle + quick_sort(right)
```

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr


def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quick_sort(left) + middle + quick_sort(right)
```

Explanation:

1. **Bubble Sort** works by repeatedly comparing and swapping adjacent elements until the list is sorted.
2. Its performance is inefficient for large datasets, with average and worst-case time complexity of **O(n²)**.
3. **Quick Sort** uses a divide-and-conquer approach by selecting a pivot and partitioning the list into smaller parts.
4. Quick Sort is much faster in practice, with an average time complexity of **O(n log n)**.
5. The main difference is that Bubble Sort uses simple swaps, while Quick Sort recursively divides the problem for better efficiency.

Task-5:

Prompt:

Create a Python-based product recommendation system that provides explainable recommendations, and evaluate whether the explanations are easy to understand.

Code and Output:

```
products = {
    "Laptop": ["electronics", "work", "portable"],
    "Headphones": ["electronics", "audio", "portable"],
    "Coffee Maker": ["kitchen", "home"],
    "Smartphone": ["electronics", "portable"],
    "Office Chair": ["work", "home"]
}

def recommend_products(user_preferences):
    recommendations = []

    for product, tags in products.items():
        common_tags = set(user_preferences).intersection(tags)
        if common_tags:
            recommendations.append((product, common_tags))

    return recommendations


user_preferences = ["electronics", "portable"]
```

```
results = recommend_products(user_preferences)

for product, reasons in results:
    print(f"Recommended: {product}")
    print(f"Reason: Matches your interest in {', '.join(reasons)}\n")
```

Explanation:

1. The system suggests items (such as products, movies, or courses) based on user preferences or behaviour.
2. Along with each recommendation, the system explains **why** it was suggested (e.g., similar past choices or interests).
3. Providing reasons increases **user trust and transparency** in the recommendations.

4. Explanations help users better understand and evaluate the suggested items.
5. This approach improves user satisfaction and engagement compared to recommendations without explanations.