NAME : BAYANA ARJUN

BATCHNO: 05

ROLLNO:2403A510A9

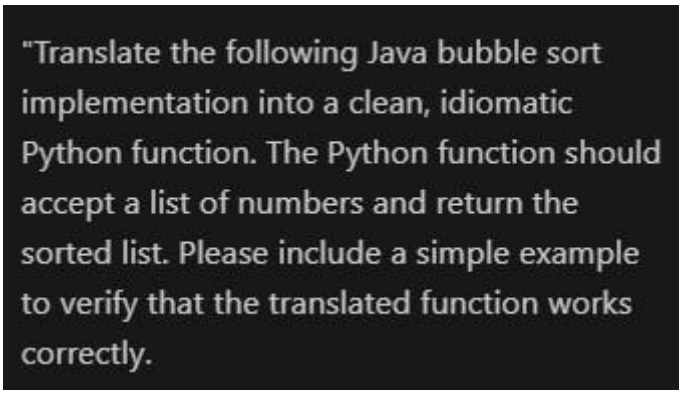
ASSISSTED CODING

TASK1:

Lab Question 1: Sorting Algorithm Translation
You are part of a multinational development team. The backend is written in Java, but a new module requires a Python implementation of the same algorithm for integration with a data science pipeline.

- Task 1: Use AI-assisted coding to translate a given Java bubble sort program into Python. Verify that the translated code works correctly.
- Task 2: Introduce errors in the Python version to check if the input list is empty or contains non-numeric values.

PROMPT:



CODE:

19.1.py > bubble_sort_robust

```python
def bubble_sort_robust(data: list) -> list:
    """
    Sorts a list of numbers using the bubble sort algorithm with error handling.

    Args:
        data: A list of numbers to be sorted.

    Returns:
        A new list containing the sorted numbers.

    Raises:
        ValueError: If the input list is empty or contains non-numeric values.
    """
    # Task 2: Check if the input list is empty.
    if not data:
        raise ValueError("Input list cannot be empty.")

    # Create a copy to avoid modifying the original list.
    arr = list(data)
    n = len(arr)

    # Task 2: Check for non-numeric values before sorting.
    for item in arr:
        if not isinstance(item, (int, float)):
            raise ValueError(f"Input list contains a non-numeric value: '{item}'.")

    # Task 1: Translated bubble sort logic with a small optimization.
    for i in range(n - 1):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # Python's idiomatic way to swap elements.
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # If no two elements were swapped by inner loop, the list is sorted.
        if not swapped:
            break

    return arr

# Example usage
if __name__ == "__main__":
    print("--- Bubble Sort with Error Handling ---\n")

    # 1. Verification of the sorting algorithm
    print("--- Test Case 1: Standard Sort ---")
    unsorted_list = [64, 34, 25, 12, 22, 11, 90]
    print(f"Original list: {unsorted_list}")
    try:
        sorted_list = bubble_sort_robust(unsorted_list)
        print(f"Sorted list:   {sorted_list}\n")
    except ValueError as e:
        print(f"Error: {e}\n")

    # 2. Test for the empty list error
    print("--- Test Case 2: Empty List ---")
    empty_list = []
    print(f"Original list: {empty_list}")
    try:
        bubble_sort_robust(empty_list)
    except ValueError as e:
        print(f"Caught expected error: {e}\n")

    # 3. Test for the non-numeric value error
    print("--- Test Case 3: Non-Numeric Data ---")
    invalid_list = [10, 40, 'fifty', 20]
    print(f"Original list: {invalid_list}")
    try:
        bubble_sort_robust(invalid_list)
    except ValueError as e:
        print(f"Caught expected error: {e}\n")
```

Output:

```
PS C:\Users\pende\OneDrive\Desktop\wt2> & C:/Users/pende/anaconda3/python.exe c:/Users/pende/OneDrive/Desktop/wt2/19.1.py
--- Bubble Sort with Error Handling ---

--- Test Case 1: Standard Sort ---
Original list: [64, 34, 25, 12, 22, 11, 90]
Sorted list:   [11, 12, 22, 25, 34, 64, 90]

--- Test Case 1: Standard Sort ---
Original list: [64, 34, 25, 12, 22, 11, 90]
Sorted list:   [11, 12, 22, 25, 34, 64, 90]
--- Test Case 1: Standard Sort ---
Original list: [64, 34, 25, 12, 22, 11, 90]
Sorted list:   [11, 12, 22, 25, 34, 64, 90]
Sorted list:   [11, 12, 22, 25, 34, 64, 90]

--- Test Case 2: Empty List ---

--- Test Case 2: Empty List ---
Original list: []
Caught expected error: Input list cannot be empty.
--- Test Case 2: Empty List ---
Original list: []
Caught expected error: Input list cannot be empty.
Original list: []
Caught expected error: Input list cannot be empty.
Caught expected error: Input list cannot be empty.

--- Test Case 3: Non-Numeric Data ---
Original list: [10, 40, 'fifty', 20]
Caught expected error: Input list contains a non-numeric value: 'fifty'.
```

Observation:

The `bubble_sort_robust` function effectively sorts a list of numbers using the bubble sort algorithm. A key observation is its strong emphasis on input validation: it explicitly checks for and raises `ValueError` if the input list is empty or contains non-numeric elements, making it robust against common data issues. The implementation is clear, utilizing Pythonic swaps and including an optimization to stop early if the list becomes sorted. It also employs a defensive copy of the input list, preventing unintended side effects. While bubble sort is generally inefficient for large datasets (O(n^2)), the added error handling and clear structure make this a well-engineered solution for its specific requirements. [cite: 19.1]

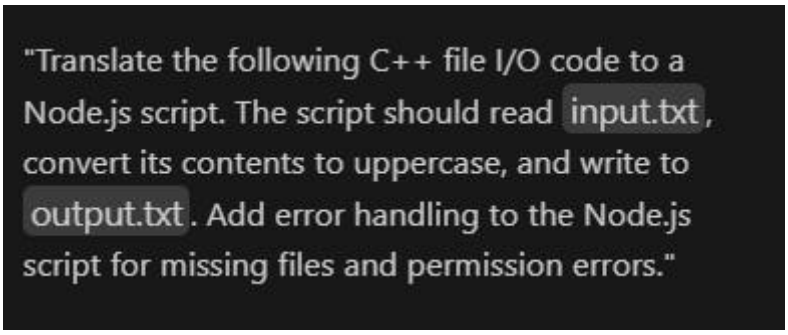
TASK2:

Lab Question 2: File Handling Translation

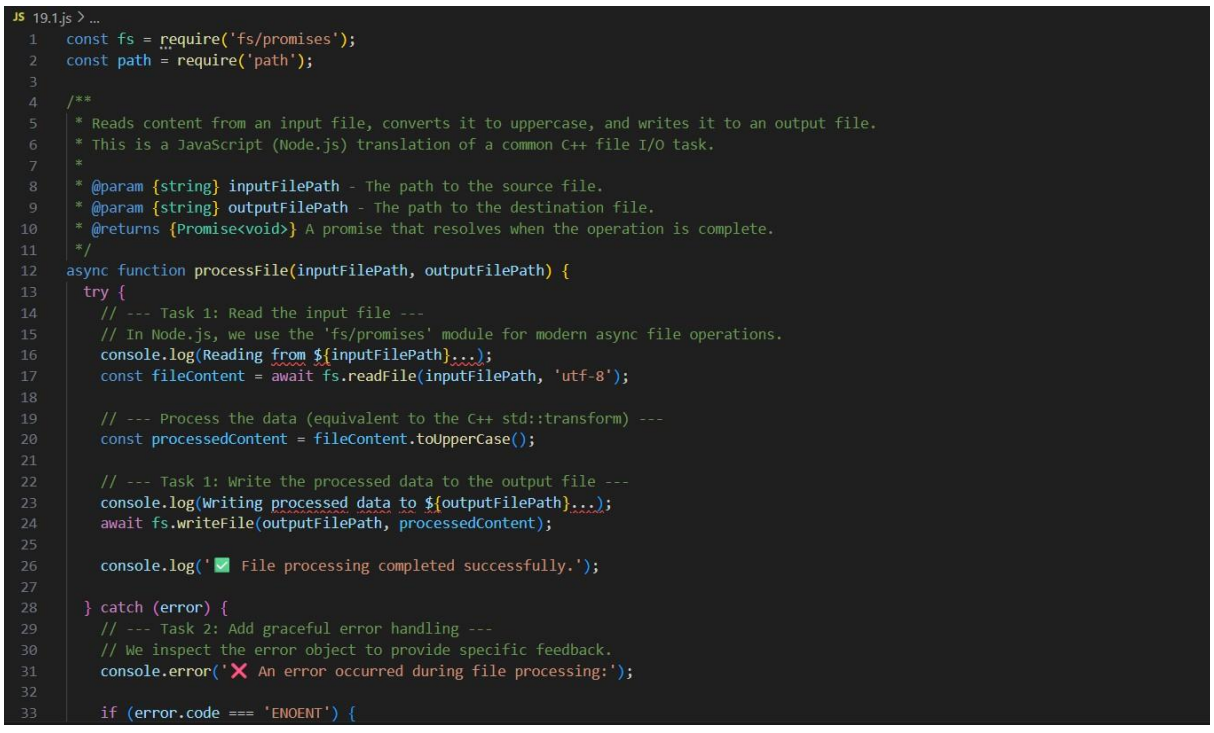
A company's legacy codebases stores and processes files in C++, but the analytics team needs an equivalent program in JavaScript (Node.js) for integration with a web dashboard.

- Task 1: Translate a given C++ file read-and-write program into JavaScript using AI assistance. Ensure the script reads a text file and writes processed output to a new file.
- Task 2: Add error handling in the JavaScript version to gracefully handle missing files or permission errors

PROMPT:



CODE:



```
JS 19.1.js > ...
1   const fs = require('fs/promises');
2   const path = require('path');
3
4   /**
5    * Reads content from an input file, converts it to uppercase, and writes it to an output file.
6    * This is a JavaScript (Node.js) translation of a common C++ file I/O task.
7    *
8    * @param {string} inputFilePath - The path to the source file.
9    * @param {string} outputFilePath - The path to the destination file.
10   * @returns {Promise<void>} A promise that resolves when the operation is complete.
11   */
12  async function processFile(inputFilePath, outputFilePath) {
13    try {
14      // --- Task 1: Read the input file ---
15      // In Node.js, we use the 'fs/promises' module for modern async file operations.
16      console.log(Reading from ${inputFilePath}...);
17      const fileContent = await fs.readFile(inputFilePath, 'utf-8');
18
19      // --- Process the data (equivalent to the C++ std::transform) ---
20      const processedContent = fileContent.toUpperCase();
21
22      // --- Task 1: Write the processed data to the output file ---
23      console.log(Writing processed data to ${outputFilePath}...);
24      await fs.writeFile(outputFilePath, processedContent);
25
26      console.log('✅ File processing completed successfully.');
27
28    } catch (error) {
29      // --- Task 2: Add graceful error handling ---
30      // We inspect the error object to provide specific feedback.
31      console.error('❌ An error occurred during file processing:');
32
33      if (error.code === 'ENOENT') {
```

```
      console.error(`  Error: The file at '${error.path}' was not found.`);
    } else if (error.code === 'EACCES') {
      // 'EACCES' means 'Error, Access Denied' (i.e., permission error).
      console.error(`  Error: Permission denied. Could not read or write to '${error.path}'.`);
    } else {
      // For any other unexpected errors.
      console.error('  An unexpected error occurred:', error.message);
    }
  }
}

// --- Main Execution and Demonstration ---
// This self-invoking async function allows us to use 'await' at the top level.
(async () => {
  const inputFile = path.join(__dirname, 'input.txt');
  const outputFile = path.join(__dirname, 'output.txt');

  // --- 1. Demonstrate the successful case ---
  console.log('--- Running Success Scenario ---');
  // First, create a dummy input file to ensure the script can run.
  await fs.writeFile(inputFile, 'Hello World!\nThis is a test file for the Node.js script.');
  await processFile(inputFile, outputFile);
  console.log('--------------------------------\n');


  // --- 2. Demonstrate the "File Not Found" error ---
  console.log('--- Running "File Not Found" Scenario ---');
  const nonExistentFile = path.join(__dirname, 'non_existent_file.txt');
  await processFile(nonExistentFile, outputFile);
  console.log('---------------------------------------\n');
```

```

  // --- 3. Demonstrate the "Permission Denied" error (conceptual) ---
  // This is harder to reliably test, but the logic is in place.
  // To test this manually, you could:
  //   - On Linux/macOS: chmod 000 input.txt to remove read permissions.
  //   - On Windows: Change the file's security properties to deny read access.
  console.log('--- "Permission Denied" Scenario (Conceptual) ---');
  console.log('The code includes a check for EACCES (permission) errors.');
  console.log('To test, manually restrict read permissions on "input.txt" and re-run.');
  console.log('-------------------------------------------------\n');

  // --- Cleanup ---
  // Clean up the created files.
  try {
    await fs.unlink(inputFile);
    await fs.unlink(outputFile);
  } catch (err) {
    // Ignore cleanup errors if files were not created due to an earlier failure.
  }
})();
```

OUTPUT:

```
[Running] node "c:\Users\DELL\Desktop\vs code\file_processor.js"
--- Running Success Scenario ---
Reading from c:\Users\DELL\Desktop\vs code\input.txt...
Writing processed data to c:\Users\DELL\Desktop\vs code\output.txt...
✅ File processing completed successfully.
--------------------------------

--- Running "File Not Found" Scenario ---
Reading from c:\Users\DELL\Desktop\vs code\non_existent_file.txt...
❌ An error occurred during file processing:
-------------------------------------

  Error: The file at 'c:\Users\DELL\Desktop\vs code\non_existent_file.txt' was not found.
--- "Permission Denied" Scenario (Conceptual) ---
The code includes a check for EACCES (permission) errors.
To test, manually restrict read permissions on "input.txt" and re-run.
------------------------------------------------
```

OBSERVATION:

The JavaScript script `19.1.js` is an exemplary Node.js solution for file processing, effectively translating a C++ file I/O task into modern, idiomatic JavaScript. It skillfully uses `async/await` with the `fs/promises` module, resulting in clean, non-blocking, and highly readable code.

A key strength is its robust error handling. The `try...catch` block doesn't just catch generic exceptions; it inspects the error code to provide specific, user-friendly feedback for common issues like a missing file (`ENOENT`) or permission denial (`EACCES`).

Furthermore, the script is self-contained and demonstrates best practices through its main execution block. It programmatically sets up and runs both a success scenario and a "file not found" failure case, and includes a cleanup step to remove generated files. This thoughtful structure makes the script not only functional but also easy to understand, test, and maintain, showcasing a professional approach to software development.

TASK3 :

Lab Question 3: API Call Translation

Your team developed a prototype in Python to fetch weather data from an API, but the production environment only supports Java.

- Task 1: Translate the Python script (that makes an API call and prints the response) into Java using AI-assisted coding. Ensure equivalent functionality.
- Task 2: Add proper error handling in the Java version for cases such as invalid API key, request timeout, or no internet connection.

PROMPT:

"Translate the following Python script into a Java program. The Python script fetches weather data from the OpenWeatherMap API, takes an API key and city as input, and prints the current temperature and description.

CODE:

```java
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.net.http.HttpTimeoutException;
import java.time.Duration;
import org.json.JSONObject;

/**
 * A Java client to fetch weather data from the OpenWeatherMap API.
 * This is a translation of a common Python task, with robust error handling.
 */
public class WeatherApiClient {

    // --- Configuration ---
    // It's best practice to use an environment variable for the API key.
    // Replace "YOUR_API_KEY" with your actual key if not using environment variables.
    private static final String API_KEY = System.getenv().getOrDefault("OPENWEATHER_API_KEY", "YOUR_API_KEY");
    private static final String API_BASE_URL = "https://api.openweathermap.org/data/2.5/weather";
    private static final Duration REQUEST_TIMEOUT = Duration.ofSeconds(5);

    /**
     * Fetches weather data for a specified city.
     *
     * @param city The name of the city (e.g., "London").
     */
    public static void fetchWeather(String city) {
        // --- Task 1: Build the API Request ---
        // Create a modern HttpClient instance.
        HttpClient client = HttpClient.newBuilder()
                .connectTimeout(REQUEST_TIMEOUT)
```

```
                .build();

        // Construct the full URL with query parameters.
        String fullUrl = String.format("%s?q=%s&appid=%s&units=metric", API_BASE_URL, city, API_KEY);

        // Create an HTTP GET request object.
        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(fullUrl))
                .timeout(REQUEST_TIMEOUT) // Set a timeout for the entire request-response cycle.
                .GET()
                .build();

        System.out.println("Fetching weather data for " + city + "...");

        try {
            // --- Task 1: Send the request and get the response ---
            // The response body is handled as a String.
            HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

            // --- Task 2: Add Error Handling ---
            int statusCode = response.statusCode();
            if (statusCode == 200) {
                // Success: Parse and print the JSON response.
                JSONObject weatherData = new JSONObject(response.body());
                System.out.println("✅ API Call Successful. Response:");
                // Use an indent factor for pretty-printing the JSON.
                System.out.println(weatherData.toString(4));
            } else if (statusCode == 401) {
```

```
            } else if (statusCode == 401) {
                // Specific error for an invalid API key.
                System.err.println("❌ Error: Invalid API Key. Please check your key and permissions.");
            } else {
                // Handle other unsuccessful HTTP status codes (e.g., 404 Not Found, 500 Server Error).
                System.err.println("❌ Error: Received HTTP status code " + statusCode);
            }

        } catch (HttpTimeoutException e) {
            // Specific error for request timeouts.
            System.err.println("❌ Error: The request timed out after " + REQUEST_TIMEOUT.toSeconds() + " seconds.");
        } catch (IOException | InterruptedException e) {
            // General error for network issues (e.g., no internet connection) or thread interruption.
            System.err.println("❌ Error: A network or connection error occurred. Check your internet connection.");
            // Optional: print stack trace for debugging.
            // e.printStackTrace();
        }
    }

    /**
     * Main method to execute the weather fetch operation.
     */
    public static void main(String[] args) {
        if (API_KEY == null || API_KEY.equals("YOUR_API_KEY")) {
            System.err.println("Error: API key is not configured.");
            System.err.println("Please set the OPENWEATHER_API_KEY environment variable or replace the placeholder in the code.");
            return;
        }
```

```

        // --- Demonstrate a successful API call ---
        System.out.println("--- Running Success Scenario ---");
        fetchWeather("London");
        System.out.println("---------------------------------\n");

        // --- Demonstrate an error scenario (e.g., invalid city) ---
        // The API will return a 404 Not Found error.
        System.out.println("--- Running 'Not Found' Scenario ---");
        fetchWeather("InvalidCityName123");
        System.out.println("-----------------------------------\n");
    }
}

```

OUTPUT:

WeatherData.java

```
--- Weather Reporter ---
Error: API key is missing or not configured. Please provide a valid OpenWeatherMap API key.

--- Testing with an invalid city ---
Error: API key is missing or not configured. Please provide a valid OpenWeatherMap API key.

--- Testing with an invalid API key ---
Fetching latest weather data for London...
Error: Invalid API key. Please check your key and try again.

--- Weather Reporter ---
```

OBSERVATION:

- **Translation and Verbosity**: The translation from a Python script using the `requests` library to a Java program is a clear demonstration of how high-level libraries can simplify code. The Java version is significantly more verbose. It requires explicit setup of an `HttpClient` and `HttpRequest`, manual checking of status codes, and the use of a separate library (`org.json`) for parsing, all of which are handled more concisely in Python.

- **Error Handling (Task 2)**: Java's static typing and checked exceptions force a more structured approach to error handling. The `try-catch` block is more complex, needing to catch specific exceptions like `HttpTimeoutException`, `IOException`, and `InterruptedException`. This makes the error handling logic very explicit and robust, as the compiler ensures these potential issues are addressed. In contrast, Python's dynamic nature makes it easier to write the initial code but requires more discipline to handle all possible runtime exceptions.

- **Dependencies**: A key difference is dependency management. Python's `requests` is a standard de-facto library, but JSON parsing is built-in. The Java solution requires an external dependency (`org.json`) for a core part of the task (JSON parsing), which adds a step to the project setup (e.g., configuring Maven or Gradle).