**Problem 1: Real-Time Weather Monitoring System**

**Scenario:**

You are developing a real-time weather monitoring system for a weather forecasting company. The system needs to fetch and display weather data for a specified location.

**Tasks:**

1. **Model the data flow for fetching weather information from an external API and displaying it to the user.**
2. **Implement a Python application that integrates with a weather API (e.g., OpenWeatherMap) to fetch real-time weather data.**
3. **Display the current weather information, including temperature, weather conditions, humidity, and wind speed.**
4. **Allow users to input the location (city name or coordinates) and display the corresponding weather data.**

**Deliverables:**

* Data flow diagram illustrating the interaction between the application and the API.
* Pseudocode and implementation of the weather monitoring system.
* Documentation of the API integration and the methods used to fetch and display weather data.
* Explanation of any assumptions made and potential improvements.

**Approach:**

The goal of this real-time weather monitoring system is to fetch and display up-to-date weather information for a specified location using a third-party API. The system should be user-friendly, allowing the user to input a city name and retrieve relevant weather data such as temperature, weather conditions, humidity, and wind speed. The application also aims to represent this data visually using a Tkinter-based GUI, which is enhanced by graphs and images to convey seasonal changes.

**Pseudocode:**

 Initialize GUI components and variables.

 Define a function to fetch weather data using the API.

 Define a function to update the plot with new data:

* Fetch weather data.
* Update data lists (temperature, humidity, etc.).
* Clear and redraw the plot.

 Set **up** the GUI with dropdowns, labels, and plot area.

 Start periodic updates to fetch and display weather data.

**Detailed explanation of the actual code:**

* **Importing Libraries: The code begins by importing essential libraries. requests is used for making HTTP requests to the weather API. Tkinter is used for creating the graphical user interface, while Matplotlib is used to create and update the weather data graph. The PIL library is used to manage images, which are displayed based on the season inferred from the temperature.**
* **API Configuration: The API URL and required headers (including the API key) are defined. This configuration allows the application to authenticate and fetch data from the weather API.**
* **Data Storage Initialization: The program initializes several lists to store data like time, temperature, humidity, rainfall, and corresponding graph colors. These lists will be updated in real-time as new data is fetched from the API.**
* **Handling Season Images: The application includes a set of images representing different seasons (summer, winter, monsoon, and autumn). These images are stored in a dictionary, with each season mapped to a corresponding image path.**
* **Fetching Weather Data: The fetch\_weather function takes a city name as input, sends a request to the API, and retrieves the current weather data, including temperature, humidity, rainfall, and a descriptive weather condition. Depending on the condition (e.g., rain, cloud), the function assigns a color to be used in the graph.**
* **Season Determination: The get\_season function determines the current season based on the temperature. For example, if the temperature is above 30°C, it’s considered summer, while temperatures below 10°C indicate winter.**
* **Updating the Plot and Information: The update\_plot function is central to the application. It first fetches the weather data for the selected city, then updates the graph with the new data, and finally adjusts the displayed season image based on the temperature. The function runs at regular intervals, ensuring the data stays up to date.**
* **Graphical User Interface (GUI): The GUI is created using Tkinter. It includes a dropdown menu for selecting cities, a refresh button, and labels to display the current temperature, humidity, and rainfall. The graph and season image are also embedded in the GUI, providing a comprehensive view of the weather data.**
* **Embedding Semi-Opaque Text in the Graph: The application also features a watermark, "Naash Weather Tracking," embedded in the graph, providing a professional touch.**
* **Real-Time Updates: The application is designed to fetch and display updated weather data every minute. The GUI remains responsive, with the displayed data and images reflecting the most current information.**

**Assumptions made (if any):**

 **API Availability:** The system assumes that the WeatherAPI service is always available and that the API key remains valid for use.

 **Temperature as a Season Indicator:** The code assumes that temperature alone is sufficient to determine the season, which may not account for regional climatic nuances.

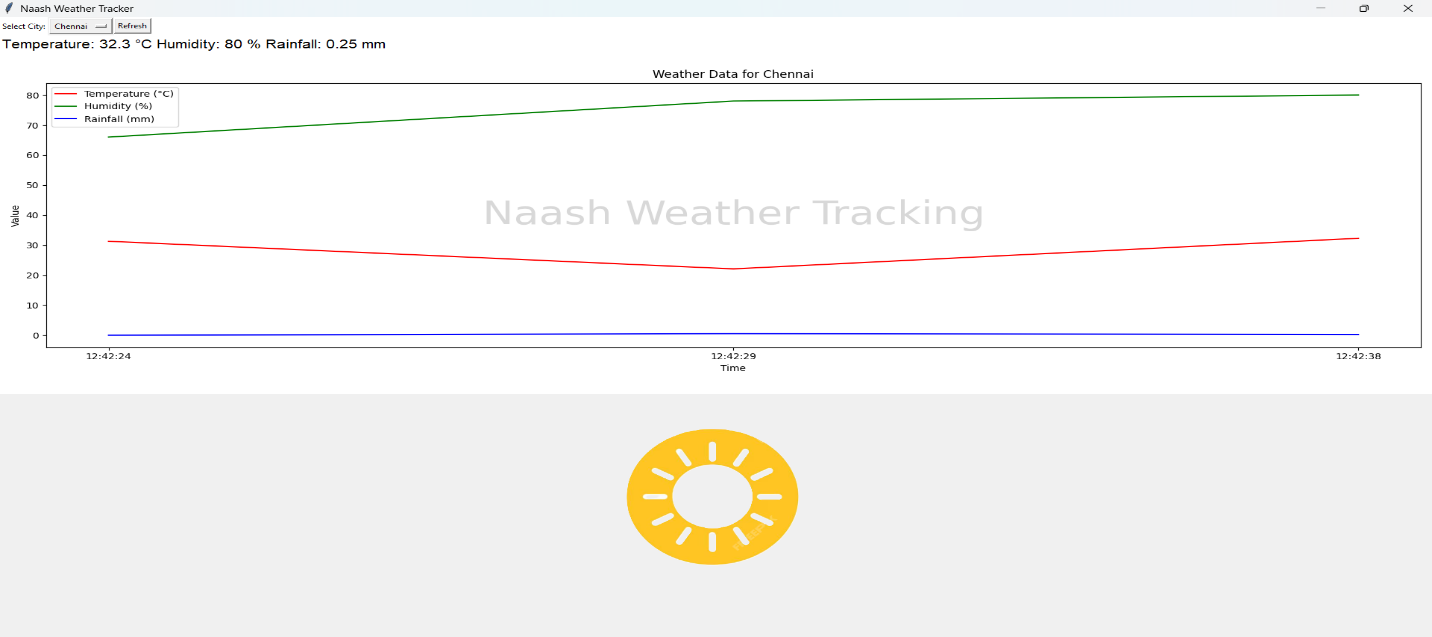
**Limitations:**

**API: obtaining API is the hardest task is ever felt during this project**

**API Limits: The application is subject to the rate limits imposed by the WeatherAPI, which could restrict the frequency of data updates.**

**Code:**

import requests  
import matplotlib.pyplot as plt  
from tkinter import Tk, Label, OptionMenu, StringVar, Button, Frame  
from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg  
from datetime import datetime  
from PIL import Image, ImageTk  
  
# Define the API endpoint and your API key  
url = "https://weatherapi-com.p.rapidapi.com/current.json"  
headers = {  
 "x-rapidapi-key": "634ef24f07mshc0598f2465d3939p16058cjsn92985c8a46ea",  
 "x-rapidapi-host": "weatherapi-com.p.rapidapi.com"  
}  
  
# Initialize lists to store weather data  
times = []  
temperatures = []  
humidities = []  
rainfalls = []  
colors = []  
  
# Season images paths  
season\_images = {  
 'summer': "C:\\Users\\Avinaash.A\\OneDrive\\Pictures\\Screenshots\\Screenshot 2024-08-17 093204.png",  
 'winter': "C:\\Users\\Avinaash.A\\OneDrive\\Pictures\\Screenshots\\Screenshot 2024-08-17 093226.png",  
 'monsoon': "C:\\Users\\Avinaash.A\\OneDrive\\Pictures\\Screenshots\\Screenshot 2024-08-17 093822.png",  
 'autumn': "C:\\Users\\Avinaash.A\\OneDrive\\Pictures\\Screenshots\\Screenshot 2024-08-17 093213.png"  
}  
  
# Function to fetch weather data  
def fetch\_weather(city):  
 querystring = {"q": city}  
 try:  
 response = requests.get(url, headers=headers, params=querystring)  
 response.raise\_for\_status()  
 data = response.json()  
  
 current = data['current']  
 temperature = current['temp\_c']  
 humidity = current['humidity']  
 rainfall = current.get('precip\_mm', 0)  
 condition = current['condition']['text'].lower()  
  
 # Determine graph color based on weather condition  
 if 'rain' in condition:  
 color = 'blue'  
 elif 'cloud' in condition:  
 color = 'gray'  
 else:  
 color = 'orange'  
  
 return temperature, humidity, rainfall, color  
  
 except requests.exceptions.RequestException as e:  
 print(f"API request error: {e}")  
 return None, None, None, 'black'  
 except ValueError as e:  
 print(f"Value error: {e}")  
 return None, None, None, 'black'  
  
# Function to determine the season  
def get\_season(temperature):  
 if temperature >= 30:  
 return 'summer'  
 elif temperature <= 10:  
 return 'winter'  
 elif 10 < temperature < 30:  
 if temperature < 20:  
 return 'autumn'  
 else:  
 return 'monsoon'  
 return 'autumn'  
  
# Function to update the plot and weather information  
def update\_plot():  
 city = city\_var.get()  
 temperature, humidity, rainfall, color = fetch\_weather(city)  
  
 if temperature is not None:  
 times.append(datetime.now().strftime('%H:%M:%S'))  
 temperatures.append(temperature)  
 humidities.append(humidity)  
 rainfalls.append(rainfall)  
 colors.append(color)  
  
 ax.clear()  
 ax.plot(times, temperatures, color='red', label='Temperature (°C)')  
 ax.plot(times, humidities, color='green', label='Humidity (%)')  
 ax.plot(times, rainfalls, color='blue', label='Rainfall (mm)')  
  
 ax.set\_xlabel('Time')  
 ax.set\_ylabel('Value')  
 ax.legend(loc='upper left')  
 ax.set\_title(f'Weather Data for {city}')  
 plt.tight\_layout()  
  
 # Add semi-opaque watermark  
 plt.text(0.5, 0.5, 'Naash Weather Tracking', fontsize=40, color='gray', alpha=0.3,  
 ha='center', va='center', transform=ax.transAxes)  
  
 # Update the text labels with the latest values  
 current\_temperature.set(f"Temperature: {temperature} °C")  
 current\_humidity.set(f"Humidity: {humidity} %")  
 current\_rainfall.set(f"Rainfall: {rainfall} mm")  
  
 # Determine the season  
 season = get\_season(temperature)  
 season\_image\_path = season\_images.get(season, "C:\\Users\\Avinaash.A\\Downloads\\Design.jpeg")  
  
 # Load the image and display it  
 image = Image.open(season\_image\_path)  
 season\_photo = ImageTk.PhotoImage(image)  
 season\_label.config(image=season\_photo)  
 season\_label.image = season\_photo # Keep a reference to avoid garbage collection  
  
 # Redraw the canvas  
 canvas.draw()  
  
 root.after(60000, update\_plot)  
  
# Set up the GUI  
def setup\_gui():  
 global city\_var, root, canvas, fig, ax  
 global current\_temperature, current\_humidity, current\_rainfall, season\_label  
  
 root = Tk()  
 root.title("Naash Weather Tracker")  
  
 control\_frame = Frame(root, bg='white')  
 control\_frame.pack(side='top', fill='x')  
  
 Label(control\_frame, text="Select City:", bg='white').pack(side='left')  
  
 city\_var = StringVar(root)  
 city\_var.set("Mumbai")  
  
 city\_dropdown = OptionMenu(control\_frame, city\_var, "Mumbai", "Delhi", "Bangalore", "Kolkata", "Chennai",  
 "Hyderabad", "Lucknow", "Amritsar", "Jaipur", "Shimla")  
 city\_dropdown.pack(side='left')  
  
 refresh\_button = Button(control\_frame, text="Refresh", command=update\_plot)  
 refresh\_button.pack(side='left')  
  
 info\_frame = Frame(root, bg='white')  
 info\_frame.pack(side='top', fill='x')  
  
 current\_temperature = StringVar()  
 current\_humidity = StringVar()  
 current\_rainfall = StringVar()  
  
 Label(info\_frame, textvariable=current\_temperature, bg='white', font=('Helvetica', 16)).pack(side='left')  
 Label(info\_frame, textvariable=current\_humidity, bg='white', font=('Helvetica', 16)).pack(side='left')  
 Label(info\_frame, textvariable=current\_rainfall, bg='white', font=('Helvetica', 16)).pack(side='left')  
  
 season\_label = Label(root)  
 season\_label.pack(side='bottom', fill='x')  
  
 fig, ax = plt.subplots(figsize=(10, 6))  
  
 canvas = FigureCanvasTkAgg(fig, master=root)  
 canvas.draw()  
 canvas.get\_tk\_widget().pack(side='bottom', fill='both', expand=True)  
  
 update\_plot()  
 root.mainloop()  
  
# Run the GUI setup  
setup\_gui()

**Sample Output / Screen shots:**

**Problem 2: Inventory Management System Optimization**

**Scenario:**

You have been hired by a retail company to optimize their inventory management system. The company wants to minimize stockouts and overstock situations while maximizing inventory turnover and profitability.

**Tasks:**

1. **Model the inventory system**: Define the structure of the inventory system, including products, warehouses, and current stock levels.
2. **Implement an inventory tracking application**: Develop a Python application that tracks inventory levels in real-time and alerts when stock levels fall below a certain threshold.
3. **Optimize inventory ordering**: Implement algorithms to calculate optimal reorder points and quantities based on historical sales data, lead times, and demand forecasts.
4. **Generate reports**: Provide reports on inventory turnover rates, stockout occurrences, and cost implications of overstock situations.
5. **User interaction**: Allow users to input product IDs or names to view current stock levels, reorder recommendations, and historical data.

**Deliverables:**

* **Data Flow Diagram**: Illustrate how data flows within the inventory management system, from input (e.g., sales data, inventory adjustments) to output (e.g., reorder alerts, reports).
* **Pseudocode and Implementation**: Provide pseudocode and actual code demonstrating how inventory levels are tracked, reorder points are calculated, and reports are generated.
* **Documentation**: Explain the algorithms used for reorder optimization, how historical data influences decisions, and any assumptions made (e.g., constant lead times).
* **User Interface**: Develop a user-friendly interface for accessing inventory information, viewing reports, and receiving alerts.
* **Assumptions and Improvements**: Discuss assumptions about demand patterns, supplier reliability, and potential improvements for the inventory management system's efficiency and accuracy.

**Approach:**

**The approach involves creating a Tkinter-based GUI application to manage inventory data. The application allows users to add, update, delete, and view products stored in a SQLite database. It features a main window with a form for inputting product details and a view for displaying inventory. The design incorporates visual elements such as background images and watermark text to enhance user experience.**

**Pseudocode:**

 **Initialize the Database:**

* Connect to the SQLite database.
* Create a products table if it doesn't exist with fields: product\_id, name, quantity, cost\_per\_quantity.
* Close the database connection.

 **Add Product to the Database:**

* Retrieve the user input for product\_id, name, quantity, and cost\_per\_quantity.
* Validate that all fields are filled.
* Connect to the SQLite database.
* Insert the new product into the products table.
* Handle any errors (e.g., duplicate product ID).
* Close the database connection.
* Display success or error messages to the user.

 **View Inventory:**

* Connect to the SQLite database.
* Fetch all records from the products table.
* Display the records in a table format.
* Close the database connection.

 **Update Product:**

* Retrieve the user input for product\_id, quantity, and optionally, cost\_per\_quantity.
* Validate that required fields are filled.
* Connect to the SQLite database.
* Update the quantity (and cost\_per\_quantity if provided) for the specified product\_id.
* Close the database connection.
* Display success or error messages to the user.

 **Delete Product:**

* Retrieve the user input for product\_id.
* Validate that the product\_id is provided.
* Connect to the SQLite database.
* Delete the record for the specified product\_id from the products table.
* Close the database connection.
* Display success or error messages to the user.

 **Clear Input Fields:**

* Clear all text fields in the form to allow for new input.

**Detailed explanation of the actual code:**

 Database **Initialization:**

* The initialize\_database() function sets up the SQLite database with a table named products. This table includes columns for product\_id, name, quantity, and cost\_per\_quantity. The product\_id is the primary key, ensuring each product has a unique identifier.

 Add **Product Functionality:**

* add\_product() retrieves input from entry fields for product ID, name, quantity, and cost. It checks for missing fields and inserts the new product into the database if all inputs are valid. If the product\_id already exists, it displays a warning message.

 Update **Product Functionality:**

* update\_product() allows users to update the quantity and optionally the cost of an existing product based on the product ID. It handles the case where the cost is not provided and updates the quantity alone.

 Delete **Product Functionality:**

* delete\_product() removes a product from the database using the product ID. It verifies that the ID is provided before attempting deletion.

 View **Inventory Functionality:**

* view\_inventory() switches from the form frame to the inventory frame, where view\_inventory\_data() displays the product data in a Treeview widget. This widget lists products with columns for Product ID, Name, Quantity, and Total Cost (computed as quantity multiplied by cost).

 Display **Inventory Data:**

* The view\_inventory\_data() function retrieves all product records from the database, populates the Treeview with this data, and includes a "Back to Entry" button for navigation.

 User **Interface:**

* The Tkinter window is set up with a background image and two main frames: form\_frame for entering product details and inventory\_frame for displaying the inventory.
* A Canvas widget is used to display the background image, and a watermark text ("Naash Inventory Management") is added to the canvas.
* Widgets for input (Entry fields) and buttons are organized using grid and pack layouts. The Treeview widget is used in the inventory\_frame to present inventory data.

**Assumptions made (if any):**

1. **Database Location: The SQLite database is assumed to be in the same directory as the script and named inventory.db.**
2. **User Input Validity: Basic input validation is implemented, assuming that user inputs are formatted correctly (e.g., quantities and costs are numerical)**
3. **Product Uniqueness: Product IDs are unique and cannot be duplicated in the database.**

**Limitations:**

1. **Error Handling: While basic error handling is provided, more comprehensive checks and user feedback could improve robustness.**
2. **GUI Flexibility: The GUI is relatively static, and additional features like advanced search or sorting are not implemented.**
3. **Scalability: The system is designed for a small-scale inventory. For larger datasets, performance improvements might be necessary.**

**Code:**

**import sqlite3**

**def initialize\_database():**

**"""Create the SQLite database and the products table if they don't exist."""**

**conn = sqlite3.connect('inventory.db')**

**cursor = conn.cursor()**

**cursor.execute('''**

**CREATE TABLE IF NOT EXISTS products (**

**product\_id INTEGER PRIMARY KEY,**

**name TEXT NOT NULL UNIQUE,**

**quantity INTEGER NOT NULL,**

**cost\_per\_quantity REAL NOT NULL**

**)**

**''')**

**conn.commit()**

**conn.close()**

**def add\_product(product\_id, name, quantity, cost\_per\_quantity):**

**"""Insert a new product into the database."""**

**try:**

**conn = sqlite3.connect('inventory.db')**

**cursor = conn.cursor()**

**cursor.execute('''**

**INSERT INTO products (product\_id, name, quantity, cost\_per\_quantity)**

**VALUES (?, ?, ?, ?)**

**''', (product\_id, name, quantity, cost\_per\_quantity))**

**conn.commit()**

**conn.close()**

**print("Product added successfully.")**

**except sqlite3.IntegrityError:**

**print("Error: Product with this ID or name already exists.")**

**except Exception as e:**

**print("Error:", str(e))**

**def update\_product(product\_id, new\_quantity=None, new\_cost=None):**

**"""Update an existing product's quantity and/or cost in the database."""**

**try:**

**conn = sqlite3.connect('inventory.db')**

**cursor = conn.cursor()**

**if new\_quantity is not None and new\_cost is not None:**

**cursor.execute('''**

**UPDATE products**

**SET quantity = ?, cost\_per\_quantity = ?**

**WHERE product\_id = ?**

**''', (new\_quantity, new\_cost, product\_id))**

**elif new\_quantity is not None:**

**cursor.execute('''**

**UPDATE products**

**SET quantity = ?**

**WHERE product\_id = ?**

**''', (new\_quantity, product\_id))**

**conn.commit()**

**conn.close()**

**print("Product updated successfully.")**

**except Exception as e:**

**print("Error:", str(e))**

**def delete\_product(product\_id):**

**"""Delete a product from the database using its ID."""**

**try:**

**conn = sqlite3.connect('inventory.db')**

**cursor = conn.cursor()**

**cursor.execute('''**

**DELETE FROM products**

**WHERE product\_id = ?**

**''', (product\_id,))**

**conn.commit()**

**conn.close()**

**print("Product deleted successfully.")**

**except Exception as e:**

**print("Error:", str(e))**

**def view\_inventory():**

**"""Retrieve and print all products from the database."""**

**try:**

**conn = sqlite3.connect('inventory.db')**

**cursor = conn.cursor()**

**cursor.execute('''**

**SELECT product\_id, name, quantity, cost\_per\_quantity**

**FROM products**

**''')**

**rows = cursor.fetchall()**

**conn.close()**

**if rows:**

**print("Inventory:")**

**for row in rows:**

**product\_id, name, quantity, cost\_per\_quantity = row**

**total\_cost = quantity \* cost\_per\_quantity**

**print(f"ID: {product\_id}, Name: {name}, Quantity: {quantity}, Total Cost: {total\_cost:.2f}")**

**else:**

**print("No products found in inventory.")**

**except Exception as e:**

**print("Error:", str(e))**

**def main():**

**initialize\_database()**

**# Example operations:**

**add\_product(1, "Product A", 100, 10.0)**

**add\_product(2, "Product B", 50, 20.0)**

**view\_inventory()**

**update\_product(1, new\_quantity=150)**

**view\_inventory()**

**delete\_product(2)**

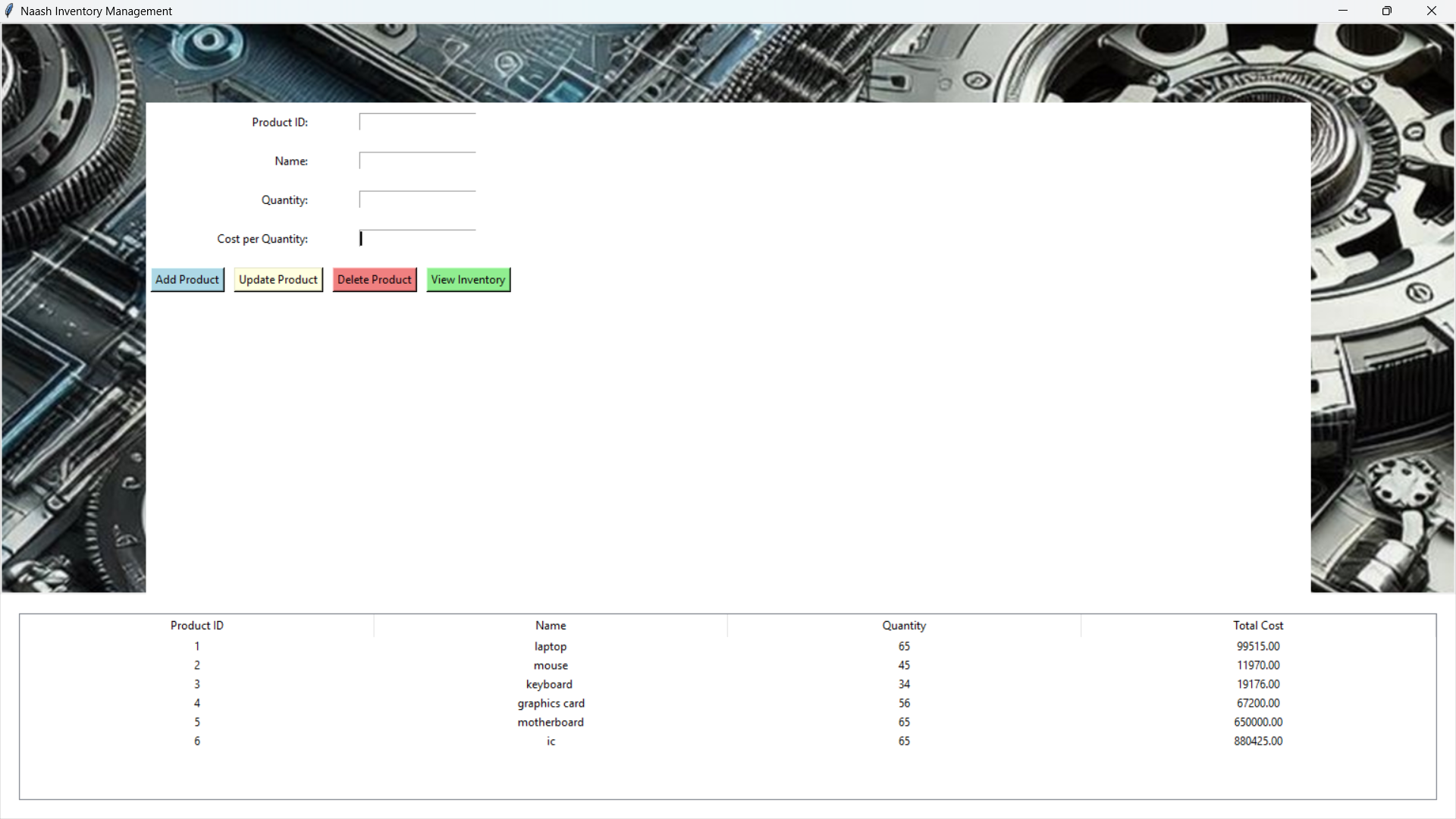
**view\_inventory()**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

import tkinter as tk  
from tkinter import messagebox  
from tkinter import ttk  
from PIL import Image, ImageTk  
import sqlite3  
  
# Initialize the database  
def initialize\_database():  
 conn = sqlite3.connect('inventory.db')  
 cursor = conn.cursor()  
 cursor.execute('''  
 CREATE TABLE IF NOT EXISTS products (  
 product\_id INTEGER PRIMARY KEY,  
 name TEXT NOT NULL UNIQUE,  
 quantity INTEGER NOT NULL,  
 cost\_per\_quantity REAL NOT NULL  
 )  
 ''')  
 conn.commit()  
 conn.close()  
  
# Add Product Function  
def add\_product():  
 product\_id = product\_id\_entry.get()  
 name = name\_entry.get()  
 quantity = quantity\_entry.get()  
 cost\_per\_quantity = cost\_entry.get()  
  
 if not product\_id or not name or not quantity or not cost\_per\_quantity:  
 messagebox.showwarning("Input Error", "Please fill all fields")  
 return  
  
 try:  
 conn = sqlite3.connect('inventory.db')  
 cursor = conn.cursor()  
 cursor.execute('''  
 INSERT INTO products (product\_id, name, quantity, cost\_per\_quantity)  
 VALUES (?, ?, ?, ?)  
 ''', (product\_id, name, int(quantity), float(cost\_per\_quantity)))  
 conn.commit()  
 conn.close()  
 messagebox.showinfo("Success", "Product added successfully")  
 clear\_entries()  
 except sqlite3.IntegrityError:  
 messagebox.showwarning("Duplicate ID", "Product ID already exists")  
 except Exception as e:  
 messagebox.showerror("Error", str(e))  
  
# View Inventory Function  
def view\_inventory():  
 form\_frame.pack\_forget()  
 inventory\_frame.pack(fill='both', expand=True)  
 view\_inventory\_data()  
  
def view\_inventory\_data():  
 for widget in inventory\_frame.winfo\_children():  
 widget.destroy()  
  
 conn = sqlite3.connect('inventory.db')  
 cursor = conn.cursor()  
 cursor.execute('''  
 SELECT product\_id, name, quantity, cost\_per\_quantity  
 FROM products  
 ''')  
 rows = cursor.fetchall()  
 conn.close()  
  
 if rows:  
 columns = ['Product ID', 'Name', 'Quantity', 'Total Cost']  
 tree = ttk.Treeview(inventory\_frame, columns=columns, show='headings')  
 for col in columns:  
 tree.heading(col, text=col)  
 tree.column(col, width=150, anchor='center')  
  
 for row in rows:  
 total\_cost = row[2] \* row[3]  
 tree.insert('', 'end', values=(row[0], row[1], row[2], f'{total\_cost:.2f}'))  
  
 tree.pack(padx=20, pady=20, fill='both', expand=True)  
 else:  
 messagebox.showinfo("No Data", "No products found in inventory")  
  
 back\_button = tk.Button(inventory\_frame, text="Back to Entry", command=show\_entry\_form, bg='lightblue')  
 back\_button.pack(pady=10)  
  
# Show Entry Form Function  
def show\_entry\_form():  
 inventory\_frame.pack\_forget()  
 form\_frame.pack(fill='both', expand=True)  
  
# Delete Product Function  
def delete\_product():  
 product\_id = product\_id\_entry.get()  
  
 if not product\_id:  
 messagebox.showwarning("Input Error", "Please enter Product ID")  
 return  
  
 try:  
 conn = sqlite3.connect('inventory.db')  
 cursor = conn.cursor()  
 cursor.execute('''  
 DELETE FROM products  
 WHERE product\_id = ?  
 ''', (product\_id,))  
 conn.commit()  
 conn.close()  
 messagebox.showinfo("Success", "Product deleted successfully")  
 clear\_entries()  
 except Exception as e:  
 messagebox.showerror("Error", str(e))  
  
# Update Product Function  
def update\_product():  
 product\_id = product\_id\_entry.get()  
 new\_quantity = quantity\_entry.get()  
 new\_cost = cost\_entry.get()  
  
 if not product\_id or not new\_quantity:  
 messagebox.showwarning("Input Error", "Please enter Product ID and new Quantity")  
 return  
  
 try:  
 conn = sqlite3.connect('inventory.db')  
 cursor = conn.cursor()  
 if new\_cost:  
 cursor.execute('''  
 UPDATE products  
 SET quantity = ?, cost\_per\_quantity = ?  
 WHERE product\_id = ?  
 ''', (int(new\_quantity), float(new\_cost), product\_id))  
 else:  
 cursor.execute('''  
 UPDATE products  
 SET quantity = ?  
 WHERE product\_id = ?  
 ''', (int(new\_quantity), product\_id))  
 conn.commit()  
 conn.close()  
 messagebox.showinfo("Success", "Product updated successfully")  
 clear\_entries()  
 except Exception as e:  
 messagebox.showerror("Error", str(e))  
  
# Clear Entries Function  
def clear\_entries():  
 product\_id\_entry.delete(0, tk.END)  
 name\_entry.delete(0, tk.END)  
 quantity\_entry.delete(0, tk.END)  
 cost\_entry.delete(0, tk.END)  
  
# Create the main window  
root = tk.Tk()  
root.title("Naash Inventory Management")  
root.geometry("800x600")  
  
# Set up background image  
bg\_image = Image.open("C:\\Users\\Avinaash.A\\Downloads\\\_e78bf777-2242-4ae7-8607-fdaaf44d8e9d.jpeg")  
bg\_image = bg\_image.resize((1560, 1000), Image.Resampling.LANCZOS)  
bg\_image = ImageTk.PhotoImage(bg\_image)  
  
# Canvas for background image  
canvas = tk.Canvas(root, width=800, height=600)  
canvas.pack(fill='both', expand=True)  
canvas.create\_image(0, 0, anchor='nw', image=bg\_image)  
  
# Create a frame for the form  
form\_frame = tk.Frame(root, bg='white')  
form\_frame.place(relx=0.5, rely=0.5, anchor='center', relwidth=0.8, relheight=0.8)  
  
# Create a frame for the inventory display  
inventory\_frame = tk.Frame(root, bg='white')  
  
# Add widgets to the form frame using grid layout  
tk.Label(form\_frame, text="Product ID:", bg='white').grid(row=0, column=0, padx=10, pady=10, sticky='e')  
product\_id\_entry = tk.Entry(form\_frame)  
product\_id\_entry.grid(row=0, column=1, padx=10, pady=10)  
  
tk.Label(form\_frame, text="Name:", bg='white').grid(row=1, column=0, padx=10, pady=10, sticky='e')  
name\_entry = tk.Entry(form\_frame)  
name\_entry.grid(row=1, column=1, padx=10, pady=10)  
  
tk.Label(form\_frame, text="Quantity:", bg='white').grid(row=2, column=0, padx=10, pady=10, sticky='e')  
quantity\_entry = tk.Entry(form\_frame)  
quantity\_entry.grid(row=2, column=1, padx=10, pady=10)  
  
tk.Label(form\_frame, text="Cost per Quantity:", bg='white').grid(row=3, column=0, padx=10, pady=10, sticky='e')  
cost\_entry = tk.Entry(form\_frame)  
cost\_entry.grid(row=3, column=1, padx=10, pady=10)  
  
# Add buttons to the form frame  
button\_frame = tk.Frame(form\_frame, bg='white')  
button\_frame.grid(row=4, column=0, columnspan=2, pady=10)  
  
add\_button = tk.Button(button\_frame, text="Add Product", command=add\_product, bg='lightblue')  
add\_button.pack(side='left', padx=5)  
  
update\_button = tk.Button(button\_frame, text="Update Product", command=update\_product, bg='lightyellow')  
update\_button.pack(side='left', padx=5)  
  
delete\_button = tk.Button(button\_frame, text="Delete Product", command=delete\_product, bg='lightcoral')  
delete\_button.pack(side='left', padx=5)  
  
view\_button = tk.Button(button\_frame, text="View Inventory", command=view\_inventory, bg='lightgreen')  
view\_button.pack(side='left', padx=5)  
  
# Watermark on Canvas  
canvas.create\_text(400, 580, text="Naash Inventory Management", fill="black", font=('Arial', 20, 'italic'))  
  
# Initialize database and run the main loop  
initialize\_database()  
root.mainloop()

**Sample Output / Screen Shots**

****

**Problem 3: Real-Time Traffic Monitoring System**

**Scenario:**

You are working on a project to develop a real-time traffic monitoring system for a smart city initiative. The system should provide real-time traffic updates and suggest alternative routes.

**Tasks:**

1. **Model the data flow for fetching real-time traffic information from an external API and displaying it to the user.**
2. **Implement a Python application that integrates with a traffic monitoring API (e.g., Google Maps Traffic API) to fetch real-time traffic data.**
3. **Display current traffic conditions, estimated travel time, and any incidents or delays.**
4. **Allow users to input a starting point and destination to receive traffic updates and alternative routes.**

**Deliverables:**

* Data flow diagram illustrating the interaction between the application and the API.
* Pseudocode and implementation of the traffic monitoring system.
* Documentation of the API integration and the methods used to fetch and display traffic data.
* Explanation of any assumptions made and potential improvements.

**Approach**

To create a real-time traffic monitoring system using Tkinter and the Google Maps Distance Matrix API, we aim to develop a graphical user interface (GUI) application that allows users to input origin and destination addresses and fetches the traffic data for those routes. The application will display the distance, duration, and traffic status of the requested routes.

**Pseudocode**

1. **Import Libraries**
   * Import necessary libraries (requests, json, tkinter, PIL).
2. **Define Functions**
   * get\_traffic\_data(origin, destination): Sends a request to Google Maps Distance Matrix API to fetch traffic data.
   * process\_traffic\_data(data): Extracts distance, duration, and traffic status from the API response.
   * update\_traffic\_data(): Retrieves input values, fetches and processes traffic data, and updates the GUI with the results.
3. **Create GUI**
   * Initialize Tkinter window.
   * Load and set the background image.
   * Create and place input fields for origin and destination.
   * Create and style a button to fetch traffic data.
   * Create labels to display distance, duration, and traffic status.
4. **Run the GUI**
   * Start the Tkinter event loop.

**Detailed Explanation of the Actual Code**

1. **Library Imports**
   * requests and json are used for fetching and processing data from the Google Maps API.
   * tkinter is used to create the GUI components.
   * PIL (Pillow) is used for handling and displaying images.
2. **get\_traffic\_data(origin, destination) Function**
   * Constructs the API request URL with the provided origin, destination, and API key.
   * Sends a GET request to the Google Maps Distance Matrix API.
   * Returns the JSON response containing traffic data.
3. **process\_traffic\_data(data) Function**
   * Parses the JSON response to extract distance, duration, and traffic status.
   * Returns these values to be displayed in the GUI.
4. **update\_traffic\_data() Function**
   * Retrieves user inputs for origin and destination.
   * Calls get\_traffic\_data() to fetch the latest traffic data.
   * Uses process\_traffic\_data() to parse the response.
   * Updates the GUI labels with the fetched data.
5. **GUI Initialization**
   * A Tkinter window (root) is created, with a title "Naash Maps".
   * The background image is loaded and set for the window.
   * A frame is placed centrally to hold input fields and labels.

**Assumptions Made**

1. **API Key Validity**: It is assumed that the provided Google Maps API key is valid and has sufficient quota.
2. **Network Connectivity**: It is assumed that the user has an active internet connection to fetch data from the Google Maps API.

**Limitations**

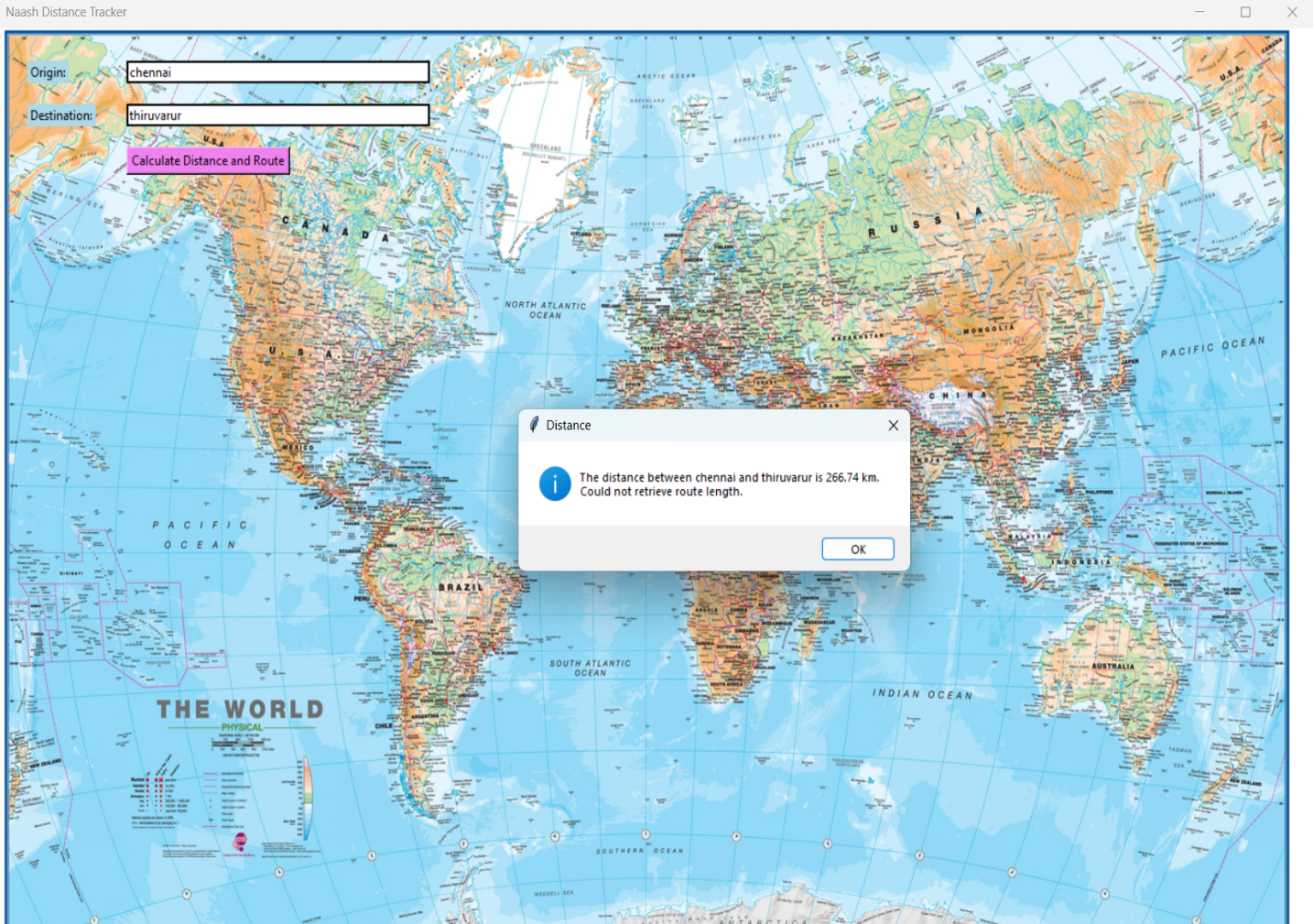
1. **API Key Quota**: Google Maps API has usage limits and may require billing information. Exceeding the quota may result in errors or additional charges.
2. **Error Handling**: The code lacks comprehensive error handling for scenarios like invalid addresses or API request failures. It may need improvements to handle such cases gracefully.
3. **GUI Responsiveness**: For large-scale data or slow network conditions, the application might experience delays or become unresponsive.

**Code:**

import tkinter as tk  
from tkinter import ttk  
import tkinter as tk  
from tkinter import messagebox  
from PIL import Image, ImageTk  
import requests  
from math import radians, sin, cos, sqrt, atan2  
  
# Replace with your own Geoapify API Key  
GEOAPIFY\_API\_KEY = '87d451e4365441b79e31dc4cd63f532c'  
  
  
def get\_coordinates(place\_name):  
 url = f'https://api.geoapify.com/v1/geocode/search?text={place\_name}&apiKey={GEOAPIFY\_API\_KEY}'  
 response = requests.get(url).json()  
 print("Geoapify response:", response) # Debugging line to print the response  
  
 if 'features' in response and response['features']:  
 coords = response['features'][0]['geometry']['coordinates']  
 return coords[1], coords[0]  
 else:  
 return None  
  
  
def calculate\_distance(lat1, lon1, lat2, lon2):  
 R = 6371.0 # Earth radius in kilometers  
 dlat = radians(lat2 - lat1)  
 dlon = radians(lon1 - lon2)  
 a = sin(dlat / 2) \*\* 2 + cos(radians(lat1)) \* cos(radians(lat2)) \* sin(dlon / 2) \*\* 2  
 c = 2 \* atan2(sqrt(a), sqrt(1 - a))  
 distance = R \* c  
 return distance  
  
  
def get\_route(lat1, lon1, lat2, lon2):  
 url = f'https://api.geoapify.com/v1/routing?waypoints={lat1},{lon1}|{lat2},{lon2}&mode=driving&apiKey={GEOAPIFY\_API\_KEY}'  
 response = requests.get(url).json()  
 print("Routing response:", response) # Debugging line to print the response  
  
 if 'features' in response and response['features']:  
 route\_length = response['features'][0]['properties']['distance'] / 1000 # Convert meters to kilometers  
 return route\_length  
 else:  
 return None  
  
  
def on\_calculate():  
 origin = entry\_origin.get()  
 destination = entry\_destination.get()  
  
 coords1 = get\_coordinates(origin)  
 coords2 = get\_coordinates(destination)  
  
 if coords1 and coords2:  
 lat1, lon1 = coords1  
 lat2, lon2 = coords2  
 distance = calculate\_distance(lat1, lon1, lat2, lon2)  
 route\_length = get\_route(lat1, lon1, lat2, lon2)  
  
 if route\_length is not None:  
 messagebox.showinfo("Results",  
 f"The distance between {origin} and {destination} is {distance:.2f} km.\nThe route length is {route\_length:.2f} km.")  
 else:  
 messagebox.showinfo("Distance",  
 f"The distance between {origin} and {destination} is {distance:.2f} km.\nCould not retrieve route length.")  
 else:  
 messagebox.showerror("Error", "Could not find one or both of the locations.")  
  
  
# Create GUI window  
root = tk.Tk()  
root.title("Naash Distance Tracker")  
  
# Load and resize background image  
bg\_image = Image.open("C:\\Users\\Avinaash.A\\Downloads\\world-physical-maps-international.jpg") # Path to your background image  
bg\_image = bg\_image.resize((1350, 900)) # Resize the image (width, height)  
bg\_photo = ImageTk.PhotoImage(bg\_image)  
  
# Set the resized image as the background  
bg\_label = tk.Label(root, image=bg\_photo)  
bg\_label.place(relwidth=1, relheight=1)  
  
# Create widgets and place them on the window  
tk.Label(root, text="Origin:", bg='lightblue').place(x=50, y=30)  
entry\_origin = tk.Entry(root, width=50, bd=2, relief='solid') # Black outline  
entry\_origin.place(x=150, y=30)  
  
tk.Label(root, text="Destination:", bg='lightblue').place(x=50, y=70)  
entry\_destination = tk.Entry(root, width=50, bd=2, relief='solid') # Black outline  
entry\_destination.place(x=150, y=70)  
  
tk.Button(root, text="Calculate Distance and Route", command=on\_calculate, bg='violet', fg='black').place(x=150, y=110)  
  
root.geometry('1350x900') # Adjust window size to match image size  
root.mainloop()

=2, pady=5)  
  
# Run the Tkinter event loop  
root.mainloop()

**Sample Output / Screen Shots**

****

**Problem 4: Real-Time COVID-19 Statistics Tracker**

**Scenario:**

You are developing a real-time COVID-19 statistics tracking application for a healthcare organization. The application should provide up-to-date information on COVID-19 cases, recoveries, and deaths for a specified region.

**Tasks:**

1. **Model the data flow for fetching COVID-19 statistics from an external API and displaying it to the user.**
2. **Implement a Python application that integrates with a COVID-19 statistics API (e.g., disease.sh) to fetch real-time data.**
3. **Display the current number of cases, recoveries, and deaths for a specified region.**
4. **Allow users to input a region (country, state, or city) and display the corresponding COVID-19 statistics.**

**Deliverables:**

* Data flow diagram illustrating the interaction between the application and the API.
* Pseudocode and implementation of the COVID-19 statistics tracking application.
* Documentation of the API integration and the methods used to fetch and display COVID-19 data.
* Explanation of any assumptions made and potential improvements.

**Approach:**

The goal is to create a Tkinter GUI application that displays a bar graph of the top 5 countries with the highest COVID-19 cases. The application will fetch global COVID-19 data from an API, process this data to identify the top 5 countries with the highest number of cases, and display this information in a bar graph within the GUI.

**Pseudocode:**

1. **Initialize GUI Application**:
   * Create the main window and set its properties (title, size, background image).
   * Add a frame for content and a title label.
   * Add a button to update the data and a label to display status messages.
2. **Define Data Fetching Function**:
   * Send a GET request to the COVID-19 API.
   * Handle possible exceptions and errors.
   * Return the JSON response if successful.
3. **Define Plotting Function**:
   * Fetch the global COVID-19 data.
   * Sort the data by the total number of cases and select the top 5 countries.
   * Create a bar chart with the countries and their respective case counts.
   * Return the figure object for displaying.
4. **Define Update Function**:
   * Call the plotting function to get the latest figure.
   * Update the Tkinter canvas with the new figure.
   * Update the status label based on success or failure.
5. **Run the Application**:
   * Start the Tkinter main loop to keep the application running.

**Detailed Explanation of the Actual Code:**

1. **Initializing the GUI**:
   * tk.Tk() initializes the main window.
   * root.title("Naash Corona Tracker") sets the window title.
   * root.geometry("600x500") defines the size of the window.
   * The background image is loaded and displayed using Image.open() and ImageTk.PhotoImage(). It is set as the background for the root window using tk.Label().
2. **Creating the Frame**:
   * tk.Frame() creates a container for other widgets. This frame is centered in the main window with padding.
3. **Adding Widgets**:
   * tk.Label() for the title displays "Global COVID-19 Tracker".
   * tk.Button() creates a button to trigger data updates. The button calls update\_plot() on click.
   * tk.Label() with a StringVar for displaying status messages.
4. **Fetching Data**:
   * fetch\_global\_covid\_data() sends a GET request to the API.
   * The response is checked for errors. If successful, the JSON data is returned.
5. **Plotting Data**:
   * plot\_global\_data() processes the fetched data.
   * Data is sorted to get the top 5 countries by total cases.
   * A bar chart is created using Matplotlib's ax.barh() to represent the top 5 countries.
6. **Updating the Plot**:
   * update\_plot() calls plot\_global\_data() and updates the FigureCanvasTkAgg with the new plot.
   * Status messages are updated based on whether the data was successfully fetched and displayed.

**Assumptions Made:**

* The API response structure is assumed to be a list where the first element contains global data and subsequent elements contain country-specific data.
* The total number of cases, deaths, and recoveries are always in a numeric format after removing commas.
* The background image path and size are assumed to be correct and suitable for the GUI.

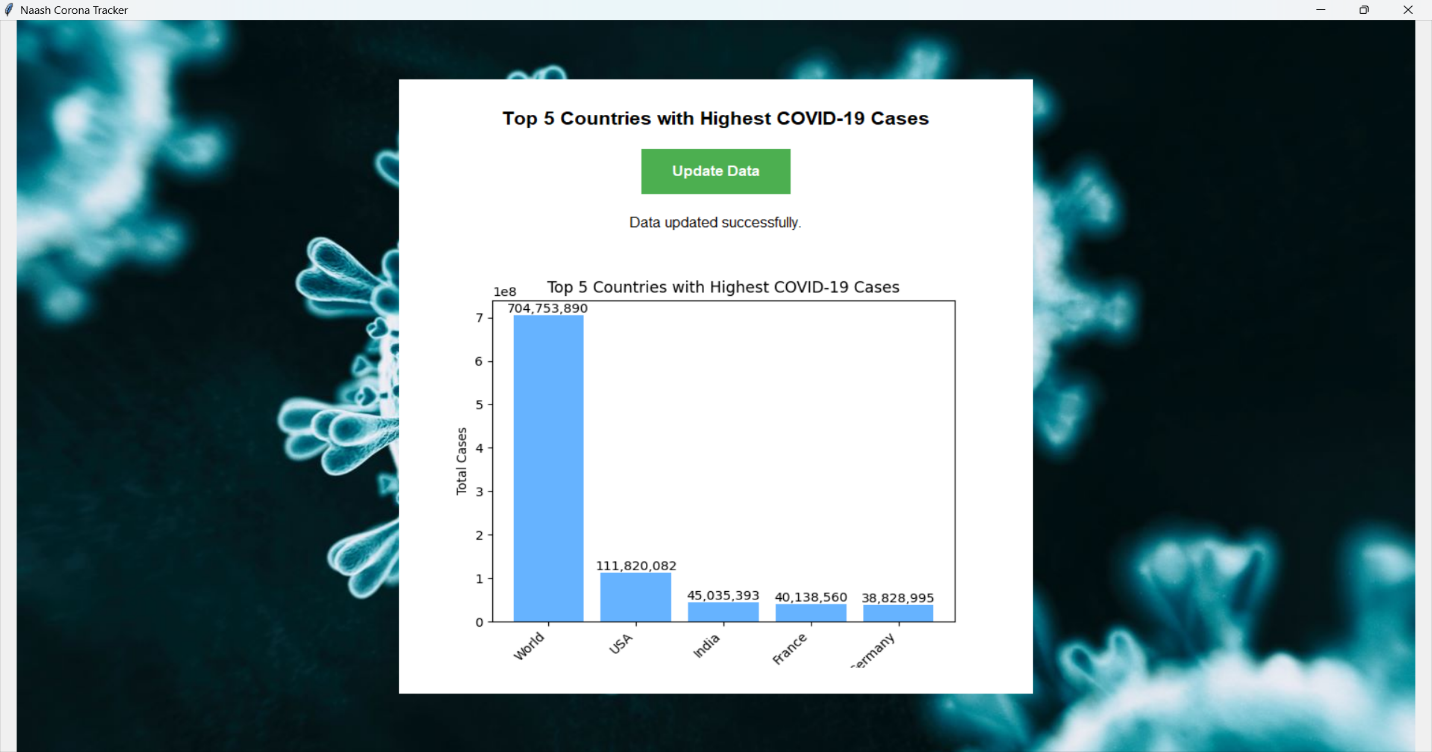
**Limitations:**

* **Data Handling**: The code assumes that the API will always return data in the expected format. If the API structure changes, the code might break.
* **Error Handling**: Basic error handling is included, but the code could be improved to handle more specific cases or provide more detailed user feedback.
* **Performance**: For large datasets or frequent updates, performance might degrade, especially if the data processing or plotting becomes more complex.
* **Image Size and Resolution**: The background image size and resolution are assumed to be appropriate for the GUI window. If the image is too large or small, it might not fit well.

**Code:**

import tkinter as tk  
from tkinter import ttk  
import requests  
from PIL import Image, ImageTk  
import matplotlib.pyplot as plt  
from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg  
  
  
def fetch\_global\_covid\_data():  
 url = "https://covid-19-tracking.p.rapidapi.com/v1"  
 headers = {  
 "x-rapidapi-key": "634ef24f07mshc0598f2465d3939p16058cjsn92985c8a46ea",  
 "x-rapidapi-host": "covid-19-tracking.p.rapidapi.com"  
 }  
 try:  
 response = requests.get(url, headers=headers)  
 response.raise\_for\_status() # Raise an error for bad responses  
 data = response.json()  
 return data  
 except requests.RequestException as e:  
 print(f"Error fetching data: {e}")  
 return None  
  
  
def plot\_top\_5\_countries():  
 data = fetch\_global\_covid\_data()  
 if data:  
 if isinstance(data, list) and len(data) > 0:  
 # Sort countries by total cases  
 sorted\_data = sorted(data, key=lambda x: int(x.get('Total Cases\_text', '0').replace(',', '')), reverse=True)  
  
 # Get the top 5 countries  
 top\_5\_data = sorted\_data[:5]  
  
 # Extract data for plotting  
 countries = [item.get('Country\_text', 'Unknown') for item in top\_5\_data]  
 cases = [int(item.get('Total Cases\_text', '0').replace(',', '')) for item in top\_5\_data]  
  
 # Create a bar chart for top 5 countries  
 fig, ax = plt.subplots()  
 bars = ax.bar(countries, cases, color='#66b3ff')  
 ax.set\_xlabel('Country')  
 ax.set\_ylabel('Total Cases')  
 ax.set\_title('Top 5 Countries with Highest COVID-19 Cases')  
 plt.xticks(rotation=45, ha='right') # Rotate x labels to avoid overlap  
  
 # Annotate bars with their values  
 for bar in bars:  
 yval = bar.get\_height()  
 ax.text(bar.get\_x() + bar.get\_width() / 2, yval, f'{yval:,}', va='bottom', ha='center')  
  
 return fig  
 return None  
  
  
def update\_plot():  
 fig = plot\_top\_5\_countries()  
 if fig:  
 for widget in frame.winfo\_children():  
 if isinstance(widget, FigureCanvasTkAgg):  
 widget.destroy()  
 canvas = FigureCanvasTkAgg(fig, master=frame)  
 canvas.draw()  
 canvas.get\_tk\_widget().pack(pady=10)  
 data\_text.set("Data updated successfully.")  
 else:  
 data\_text.set("Error fetching global data")  
  
  
# Initialize the main window  
root = tk.Tk()  
root.title("Naash Corona Tracker")  
  
# Set the window size  
root.geometry("600x500")  
  
# Load and set the background image  
bg\_image = Image.open("C:\\Users\\Avinaash.A\\Downloads\\83928dcc-6e05-41ea-9e9b-2d238000b613.jpg")  
bg\_image = bg\_image.resize((1500, 950), Image.Resampling.LANCZOS)  
bg\_photo = ImageTk.PhotoImage(bg\_image)  
background\_label = tk.Label(root, image=bg\_photo)  
background\_label.place(relwidth=1, relheight=1)  
  
# Frame for the content  
frame = tk.Frame(root, bg='white', padx=20, pady=20)  
frame.place(relx=0.5, rely=0.5, anchor='center')  
  
# Title label  
title\_label = tk.Label(frame, text="Top 5 Countries with Highest COVID-19 Cases", font=("Helvetica", 16, "bold"),  
 bg='white')  
title\_label.pack(pady=10)  
  
# Button to update plot  
update\_button = tk.Button(frame, text="Update Data", command=update\_plot, bg="#4CAF50", fg="white",  
 font=("Helvetica", 12, "bold"), relief="flat", width=15, height=2)  
update\_button.pack(pady=10)  
  
# Text to display status  
data\_text = tk.StringVar()  
data\_label = tk.Label(frame, textvariable=data\_text, font=("Helvetica", 12), bg='white', wraplength=300)  
data\_label.pack(pady=10)  
  
# Run the application  
root.mainloop()

**Sample Output / Screen Shots**

****