PROJECT REPORT

1. Problem Statement:

The Recipe App code aims to create a graphical user interface (GUI) application for users to explore, save, and manage recipes. The application integrates with the Edamam API to fetch recipes based on user input and allows users to save their favorite recipes in a local SQLite database. While the code accomplishes its intended functionality, there are potential areas for improvement and refinement.

2. Distinguish Features &Source Code of Main Features:

Library Imports

```
import tkinter as tk
from tkinter import simpledialog, messagebox, Text
import requests
import sqlite3 as sql
from random import randint
import urllib.parse
```

The code begins by importing essential libraries such as Tkinter for GUI development, requests for HTTP requests, sqlite3 for database management, randint for generating random numbers, and urllib for URL parsing.All of these are to be used later on in the project.

Initialization and Configuration

```
FILENAME = "Recipe_database.db"
con = sql.connect(FILENAME)
C = con.cursor()
IDS = {-1}
APP_ID = "YOUR APP ID"
API_KEY = "YOUR API KEY"
URL = f'https://api.edamam.com/search?/app_id=${APP_ID}&app_key=${API_KEY}'
```

Variables like the database filename, connection object, API credentials (APP_ID and API_KEY), and the base URL for the Edamam API are initialized. The code also establishes a connection to the SQLite database and creates a cursor for executing SQL commands.

Database Table Creation

```
C.execute('''CREATE TABLE IF NOT EXISTS recipes (
                id INTEGER PRIMARY KEY,
                uri TEXT NOT NULL,
                label TEXT NOT NULL
            )''')
con.commit()
```

A table named 'Recipes_database' is created within the SQLite database to store information about recipes. The table includes columns for the recipe's ID, URI, and label.

GUI Setup

```
root = tk.Tk()
root.title("Recipe App")
root.configure(bg="navajowhite")
# Get the screen width and height
screen_width = root.winfo_screenwidth()
screen_height = root.winfo_screenheight()
root.geometry(f"{screen_width}x{screen_height}")

heading_label = tk.Label(root, text="Recipe App", bg="indianred", fg="White", font=("Felix Titling", 35))

# Pack the label to fill the entire top of the window horizontally
heading_label.pack(fill=tk.X)

result_text = Text(root, height=21, width=90, bg="darksalmon", font=("Calisto MT", 13))
result_text.pack(pady=1)
```

The code creates the main GUI window using Tkinter, setting its title and background color. Various GUI elements such as labels, buttons, and text widgets are defined with specific properties and styling.

Functionality Overview

The code allows users to find new recipes based on an ingredient you have available, view recipe details, save recipes to a database, search and display saved recipes, and open the link to the selected recipe. The interaction between the GUI and the backend functionality is structured through well-defined functions.

Function: `make_request(url)`

```
def make_request(url):
    response = requests.get(url)
    data = response.json()
    return data
```

This function sends an HTTP GET request to the specified URL using the `requests` library and returns the JSON response obtained from the request. It is a generic function used for making API requests.

Function: `get_url_q(keyword, _from=0, to=20)`

```
def get_url_q(key_word, _from=0, to=20):
    return URL + f'&q=${key_word}&to={to}&from={_from}'
```

Constructs and returns a URL for querying recipes with a specific keyword (`keyword`), the keyword is to be given by the users. The parameters `_from` and `to` define the range of results to retrieve. The constructed URL is used for searching recipes based on user input i.e. it takes a keyword and searches the specific API for that keyword and generates 20 recipes that could cater to your taste.

1. **`q` parameter:** This is commonly used in APIs to represent a query or search term. In the case of a recipe search, `q` is used to specify the keyword or ingredient for which the user wants to find recipes.
2. **`to` parameter:** This parameter defines the upper limit of the number of results to be returned by the API. It is often used to control pagination or limit the number of items in the response.
3. **`from` parameter:** This parameter specifies the starting point from which the results should be retrieved. It works in conjunction with the `to` parameter to determine the range of results to be returned.

Function: `get_url_r(Uri)`

```
def get_url_r(uri):
    return URL + f'&r={uri}'
```

Constructs and returns a URL for querying recipes using a specific URI. This function is utilized when making a request for a specific recipe identified by its URI.

Function: `display_recipe_labels(data, index)`

```
def display_recipe_labels(data, index):
    result_text.delete( index1: 1.0, tk.END)
    result_text.insert(tk.END,  chars: "Results:\n")
    for i, recipe in enumerate(data):
        index += 1
        result_text.insert(tk.END,  chars: f"   {index}) {recipe['recipe']['label']}\n")
    return index
```

This function is responsible for displaying recipe labels in the GUI. It takes the recipe data and the starting index as parameters, clears the existing content in the text widget, and inserts the formatted recipe labels with indices.

Function: `select_from_index(max_index)`

```
def select_from_index(max_index):
    select = -1
    while select is None or select <= 0 or select > max_index:
        select = simpledialog.askinteger( title: "Select Recipe",  prompt: f"Select Recipe # (1-{max_index}):", parent=root)
    return select - 1
```

Prompts the user to select a recipe from a given index range. It uses the `simpledialog` module to display a dialog box where the user can input the desired recipe number. It returns the selected index after validating the input.

Function: `filter_response(recipe)`

```
def filter_response(recipe):
    curr_recipe = {
        "ingredients_line": recipe["ingredientLines"],
        "label": recipe["label"],
        "url": recipe.get("url", ""),  # Check if "url" is present in the recipe
        "uri": recipe["uri"]
    }
    return curr_recipe
```

Filters relevant information from a given recipe response. It extracts details such as ingredients, label, URL, and URI, and stores them in a dictionary named `curr_recipe`. The function returns this dictionary.

Function: `display_recipe_dict(curr_recipe)`

```
def display_recipe_dict(curr_recipe):
    recipe_text = (f"\n==================================================\n{curr_recipe['label']}"
                   f":\n--------------------------------------------------\n")
    for line in curr_recipe["ingredients_line"]:
        recipe_text += f"    - {line}\n"
    recipe_text += f"\nDirections: {curr_recipe['url']}\n=================================================="
    result_text.delete( index1: 1.0, tk.END)
    result_text.insert(tk.END, recipe_text)
```

Displays the detailed information of a recipe in the GUI. It takes the `curr_recipe` dictionary as a parameter and formats the information, including the recipe label, ingredients, and URL, before displaying it in the text widget.

Function: `make_request_by_uri(id)`

```
def make_request_by_uri(id):
    C.execute( __sql: f"SELECT uri FROM recipes WHERE id = ?",  __parameters: (id,))
    uri = C.fetchall()[0][0]
    uri = urllib.parse.quote_plus(uri)
    url = get_url_r(uri)
    data = make_request(url)[0]
    return filter_response(data)
```

Retrieves the URI of a saved recipe from the database using the provided ID. It constructs a URL using `get_url_r(uri)` and makes an API request to get detailed information about the recipe. The response is then filtered using `filter_response` before being returned.

Function: `display_saved_recipe(id)`

```
def display_saved_recipe(id):
    global curr_recipe  # Use the global curr_recipe variable
    recipe = make_request_by_uri(id)
    curr_recipe = recipe  # Update the global curr_recipe
    display_recipe_dict(recipe)
```

Displays a saved recipe using its ID. It calls `make_request_by_uri` to obtain detailed information about the recipe and then updates the global variable `curr_recipe`. The recipe details are displayed in the GUI using `display_recipe_dict`.

Function: `save_recipe(curr_recipe)`

```
def save_recipe(curr_recipe):
    id = -1
    if id in IDS:
        id = randint( a: 100,  b: 999)
    C.execute( __sql: "INSERT into recipes (id, uri, label) values (?, ?, ?)",
              __parameters: (id, curr_recipe["uri"], curr_recipe["label"]))
    C.execute( __sql: "SELECT label FROM recipes WHERE label = ?",  __parameters: (curr_recipe["label"],))
    result = C.fetchall()
    saved_message = f"You are saving: '{result[0][0]}'\nSAVED"
    result_text.delete( index1: 1.0, tk.END)
    result_text.insert(tk.END, saved_message)
    if messagebox.askyesno( title: "Confirm Save",  message: "Confirm?"):
        con.commit()
```

Saves the current recipe to the SQLite database. It generates a random ID if the ID already exists in the set `IDS`. The function then inserts the recipe details (ID, URI, and label) into the 'recipes' table in the database.

Function: `search_my_recipes()`

```
def search_my_recipes():
    result_text.delete( index1: 1.0, tk.END)
    result_text.insert(tk.END,  chars: "Saved:\n----------------------------------------\n")
    C.execute("SELECT label, id FROM recipes")
    result = C.fetchall()
    i = 0
    for recipe in result:
        i += 1
        result_text.insert(tk.END,  chars: f"   {i}) {recipe[0]}\n")
    result_text.insert(tk.END,  chars: "----------------------------------------")
    select = ""
    while type(select) is not type(0):
        select = select_from_index(i)
    id = result[select][1]
    display_saved_recipe(id)
```

Displays saved recipes from the database in the GUI. It retrieves the labels and IDs of saved recipes, presents them to the user, and prompts the user to select a recipe. The selected recipe's details are then displayed using `display_saved_recipe`.

Function: `open_recipe_link()`

```
def open_recipe_link():
    global curr_recipe  # Use the global curr_recipe variable
    if curr_recipe is not None:
        url = curr_recipe.get("url", "")
        if url:
            import webbrowser
            webbrowser.open(url)
    else:
        messagebox.showinfo( title: "No Recipe",  message: "No recipe selected. Please select a recipe first.")
```

Opens the URL of the current recipe in a web browser. It uses the `webbrowser` module to open the URL stored in the `curr_recipe` dictionary. If no recipe is selected, it shows a message indicating that no recipe is selected.

Function: `query_recipes()`

```
def query_recipes():
    key_word = simpledialog.askstring( title: "Find New Recipe",  prompt: "Please enter an ingredient:")
    data = make_request(get_url_q(key_word))
    data = data['hits']
    index = display_recipe_labels(data,  index: 0)
    select = select_from_index(index)
    if select == 'm' and index == 14:
        _from = 20
        to = 40
        data2 = make_request(get_url_q(key_word, _from, to))
        data2 = data2['hits']
        index = display_recipe_labels(data2, index)
        data += data2
        select = -1
    select_recipe(data, index, select)
```

Prompts the user to enter an ingredient and queries recipes using the Edamam API. It constructs a URL using `get_url_q` and makes a request to retrieve recipe data. If the user chooses to load more results, a second API request is made, and the results are combined.

Function: `select_recipe(data, max_index, select)`

```
def select_recipe(data, max_index, select):
    global curr_recipe  # Use the global curr_recipe variable
    invalid = True
    while invalid:
        if select == -1:
            select = select_from_index(max_index)
        if select == 'm':
            display_recipe_labels(data,  index: 0)
            select = select_from_index(max_index)
        if select == 'q':
            return
        try:
            select = int(select)
            invalid = False
        except ValueError:
            invalid = True
            select = -1

    recipe_response = data[select]
    recipe = recipe_response["recipe"]
    curr_recipe = filter_response(recipe)
    display_recipe_dict(curr_recipe)
    if messagebox.askyesno( title: "Save Recipe",  message: "Would you like to save?"):
        save_recipe(curr_recipe)
```

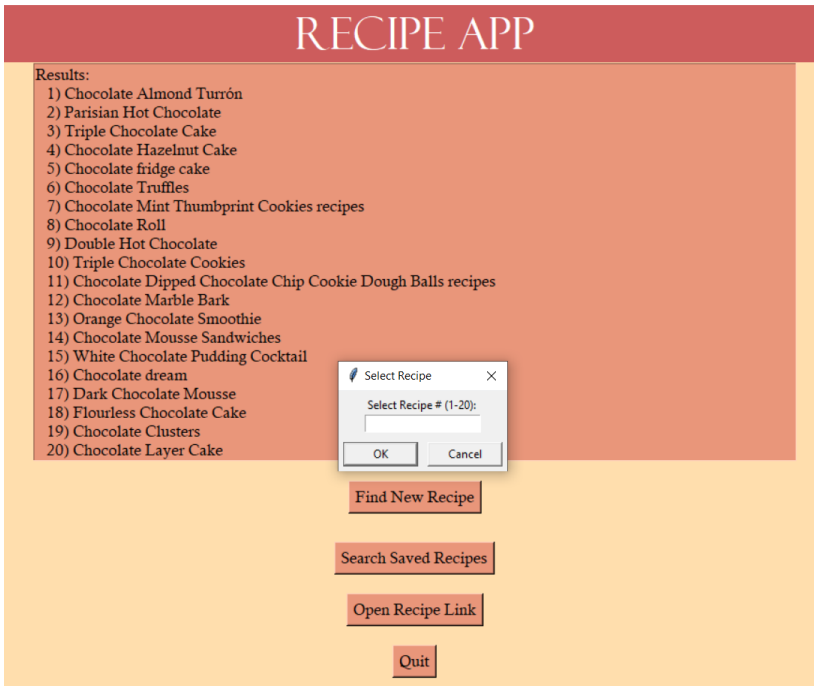
Allows the user to select a recipe from the displayed list. It handles user input, validates the selection, and retrieves detailed information about the chosen recipe. If the user decides to save the recipe, it calls the `save_recipe` function.

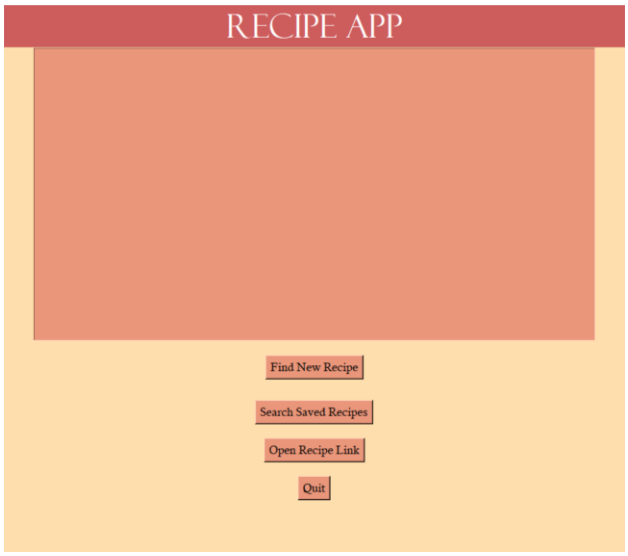
GUI Buttons and Main Loop

```
button_find_recipe=tk.Button(root,text="Find New Recipe",bg="darksalmon",font=("Calisto MT", 13),command=query_recipes)
button_find_recipe.pack(pady=20)

button_search_saved = tk.Button(root, text="Search Saved Recipes", bg="darksalmon", font=("Calisto MT", 13),
                                command=search_my_recipes)
button_search_saved.pack(pady=10)

button_open_link = tk.Button(root, text="Open Recipe Link", bg="darksalmon", font=("Calisto MT", 13),
                             command=open_recipe_link)
button_open_link.pack(pady=10)

button_quit = tk.Button(root, text="Quit", bg="darksalmon", font=("Calisto MT", 13), command=root.destroy)
button_quit.pack(pady=10)

root.mainloop()
```

Buttons for finding new recipes, searching saved recipes, opening recipe links, and quitting the application are created. These buttons are packed into the GUI window with specified padding. The main event loop (root.mainloop()) is initiated to keep the GUI running and responsive.

3. Output:





4. Individual contributions of each group member in the project:

Manal Aamir : Formation Of data base and functions
Tayyaba Arshad: Compilation of project, Formation of GUI, Collection of Data
Collective: Formation of functions and proof reading.

5. Future Expansions:

1. **Enhanced Recipe Sorting and Filtering:**
   - Implement advanced sorting and filtering options for recipes, allowing users to narrow down their search based on dietary preferences, cooking time, or nutritional content.
2. **User Accounts and Cloud Storage:**
   - Introduce user accounts to enable personalized recipe collections. Consider integrating cloud storage for users to access their saved recipes from different devices.
3. **Recipe Rating and Reviews:**
   - Incorporate a rating and review system to allow users to share feedback on recipes. This feature enhances community engagement and provides valuable insights for other users.
4. **Integration with Smart Devices:**
   - Explore integration with smart home devices to enhance the cooking experience. This could include voice-activated commands, step-by-step cooking instructions, and synchronization with smart kitchen appliances.
5. **Multi-Language Support:**
   - Expand the application's reach by adding multi-language support. This will cater to a diverse user base and make the application more accessible globally.