All the Neighbors

Required Files: possibleLocationsSmall.json, possibleLocationsMedium.json, possibleLocationsLarge.json, possibleLocationsSmallOutput.json

A local game design company, Really Augmented, is developing an augmented reality game with a database of multiple cartesian coordinate pairs. The developers need help determining the closest point for each pair and outputting this information in a new json file. The final file for the program is expected to have no more than 100,000 coordinates. You have been provided with three different test files having 100 coordinates, 5000 coordinates, and 100,000 coordinates.

Develop a program that will allow the user to select an input file and create an output file formatted like the example provided. Once this process has been completed, Really Augmented would be interested in some additional features. They would like to be able to add new coordinates (or remove existing ones) and then have the output file updated for that new point without having the run time of running the original file. They would like to have the option to create another output file that has the three closest points for each uploaded point instead of just one. They would also like to see two types of maps: an overall map of the points as well as a localized map for any given point highlighting the two nearest identified neighbors.

User Stories:

As a user, I want to run the program and input the small json file and get the output of the small output file.
As a user, I want to provide the medium and large files and receive a solution for all points within ninety seconds.

Stretch Goals:

Stretch Goals will be ignored if User Stories are not complete.

As a user, I want to be able to remove points from the uploaded list and receive an updated output file.
As a user, I want to be able to add points to the updated list and receive an updated output file.
As a user, I want an overall visual representation of all the points provided in the file.
As a user, I want a localized visual representation of a single point on the map as well as the location of its nearest two neighbors.
As a user, I would like to have an option to create a second output file that provides the nearest three points for all points.

Contest ID:_____

When you are done with both of the programs, submit your code on the USB flash drive provided to the test proctor. The programs are graded with the following attributes in mind, first using the initial user stories and (if at least 50% of user stories points are satisfied) then using the stretch goals.

- Completeness (90 pts) - Does the program run an entire use case? Does it generally accomplish the task assigned?
- Correctness of Output (80 pts) - Does the program produce the output requested in the format requested?
- Validation of Input (70 pts) - Does the program check for input that could crash the program/cause it to provide incorrect or unexpected output?
- Internal Documentation (50 pts) - Is commenting provided? Could another developer come along and add functionality to your program with ease?
- Efficiency of Code (60 pts) - Does your program waste resources while running? Do you use optimal algorithms over brute force code?
- Quality of Work (25 pts) - Is this code professional? Externally, is the output to the user professional? Internally, is the code presented in a professional and organized manner?

_____ *Resume Provided (10 pts)* _____ *Professional Dress (10 pts)*

```
import json
import numpy as np
from sklearn.neighbors import NearestNeighbors
import matplotlib.pyplot as plt

class CoordinateManager:
    def __init__(self, coordinates):
        self.coordinates = np.array(coordinates)
        self.kd_tree = NearestNeighbors(n_neighbors=3, algorithm='kd_tree').fit(self.coordinates)

    def find_closest_points(self):
        # Find the nearest two neighbors for each point
        distances, indices = self.kd_tree.kneighbors(self.coordinates)
        closest_points = {}
        for i, (dist, idx) in enumerate(zip(distances, indices)):
            closest_points[tuple(self.coordinates[i])] = [tuple(self.coordinates[j]) for j in idx[1:3]]  #
Exclude the point itself
        return closest_points

    def add_coordinate(self, coordinate):
        # Add a new coordinate and update the kd-tree
        self.coordinates = np.append(self.coordinates, [coordinate], axis=0)
        self.kd_tree = NearestNeighbors(n_neighbors=3, algorithm='kd_tree').fit(self.coordinates)

    def remove_coordinate(self, coordinate):
        # Remove a coordinate and update the kd-tree
        self.coordinates = self.coordinates[np.all(self.coordinates != coordinate, axis=1)]
        self.kd_tree = NearestNeighbors(n_neighbors=3, algorithm='kd_tree').fit(self.coordinates)

    def generate_map(self):
        # Plot the coordinates
        plt.scatter(self.coordinates[:, 0], self.coordinates[:, 1], color='blue')
        plt.title("Overall Map of Coordinates")
        plt.show()

    def generate_localized_map(self, point):
        # Plot the selected point and its two nearest neighbors
        distances, indices = self.kd_tree.kneighbors([point])
        nearest_points = self.coordinates[indices[0][1:3]]
        plt.scatter(self.coordinates[:, 0], self.coordinates[:, 1], color='blue', label='All Points')
        plt.scatter(nearest_points[:, 0], nearest_points[:, 1], color='red', label='Nearest Neighbors')
        plt.scatter(point[0], point[1], color='green', label='Selected Point')
        plt.title(f"Localized Map for {point}")
        plt.legend()
```

```
        plt.show()

def load_coordinates(file_path):
    with open(file_path, 'r') as f:
        data = json.load(f)
    return data['coordinates']

def save_output(file_path, closest_points):
    with open(file_path, 'w') as f:
        json.dump(closest_points, f, indent=4)

def main():
    # 1. Load the input coordinates
    file_path = input("Enter the path to the input JSON file: ")
    coordinates = load_coordinates(file_path)

    # 2. Process the coordinates
    manager = CoordinateManager(coordinates)
    closest_points = manager.find_closest_points()

    # 3. Save the output to a new file
    output_file = input("Enter the path to save the output JSON file: ")
    save_output(output_file, closest_points)

    # 4. Display options for visualization
    if input("Do you want to see the overall map? (y/n): ").lower() == 'y':
        manager.generate_map()

    if input("Do you want to see a localized map for a point? (y/n): ").lower() == 'y':
        x = float(input("Enter the x-coordinate of the point: "))
        y = float(input("Enter the y-coordinate of the point: "))
        manager.generate_localized_map([x, y])

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

==========

```
import json
import math

def euclidean_distance(point1, point2):
    """Compute the Euclidean distance between two points (x1, y1) and (x2, y2)."""
```

```
    return math.sqrt((point2[0] - point1[0]) ** 2 + (point2[1] - point1[1]) ** 2)

class CoordinateManager:
    def __init__(self, coordinates):
        self.coordinates = coordinates

    def find_closest_points(self):
        """Find the two closest points for each coordinate."""
        closest_points = {}
        for i, point1 in enumerate(self.coordinates):
            min_distances = [(float('inf'), None), (float('inf'), None)]  # To store (distance, index)
            for j, point2 in enumerate(self.coordinates):
                if i != j:
                    dist = euclidean_distance(point1, point2)
                    if dist < min_distances[1][0]:
                        if dist < min_distances[0][0]:
                            min_distances[1] = min_distances[0]
                            min_distances[0] = (dist, j)
                        else:
                            min_distances[1] = (dist, j)
            closest_points[tuple(point1)] = [tuple(self.coordinates[min_distances[0][1]]),
                                             tuple(self.coordinates[min_distances[1][1]])]
        return closest_points

    def add_coordinate(self, coordinate):
        """Add a new coordinate to the list and recompute closest points."""
        self.coordinates.append(coordinate)

    def remove_coordinate(self, coordinate):
        """Remove a coordinate and recompute closest points."""
        if coordinate in self.coordinates:
            self.coordinates.remove(coordinate)
        else:
            print(f"Coordinate {coordinate} not found.")

def load_coordinates(file_path):
    """Load the coordinates from a JSON file."""
    with open(file_path, 'r') as file:
        data = json.load(file)
    return data['coordinates']

def save_output(file_path, closest_points):
    """Save the closest points information to a JSON file."""
    with open(file_path, 'w') as file:
```

```
        json.dump(closest_points, file, indent=4)

def main():
    # 1. Load the input coordinates
    file_path = input("Enter the path to the input JSON file: ")
    coordinates = load_coordinates(file_path)

    # 2. Process the coordinates to find closest points
    manager = CoordinateManager(coordinates)
    closest_points = manager.find_closest_points()

    # 3. Save the output to a new file
    output_file = input("Enter the path to save the output JSON file: ")
    save_output(output_file, closest_points)

    # 4. Update options for adding/removing coordinates
    while True:
        update_action = input("Do you want to add or remove a coordinate? (add/remove/none):
").strip().lower()
        if update_action == "add":
            x = float(input("Enter x-coordinate to add: "))
            y = float(input("Enter y-coordinate to add: "))
            manager.add_coordinate([x, y])
            closest_points = manager.find_closest_points()
            save_output(output_file, closest_points)
        elif update_action == "remove":
            x = float(input("Enter x-coordinate to remove: "))
            y = float(input("Enter y-coordinate to remove: "))
            manager.remove_coordinate([x, y])
            closest_points = manager.find_closest_points()
            save_output(output_file, closest_points)
        elif update_action == "none":
            break
        else:
            print("Invalid option. Please choose 'add', 'remove', or 'none'.")

    print("Process complete.")

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