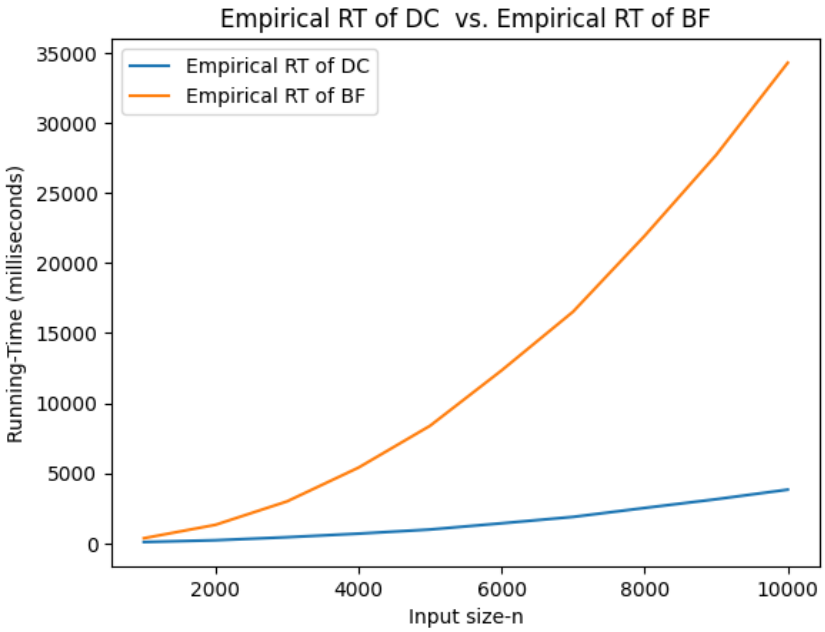
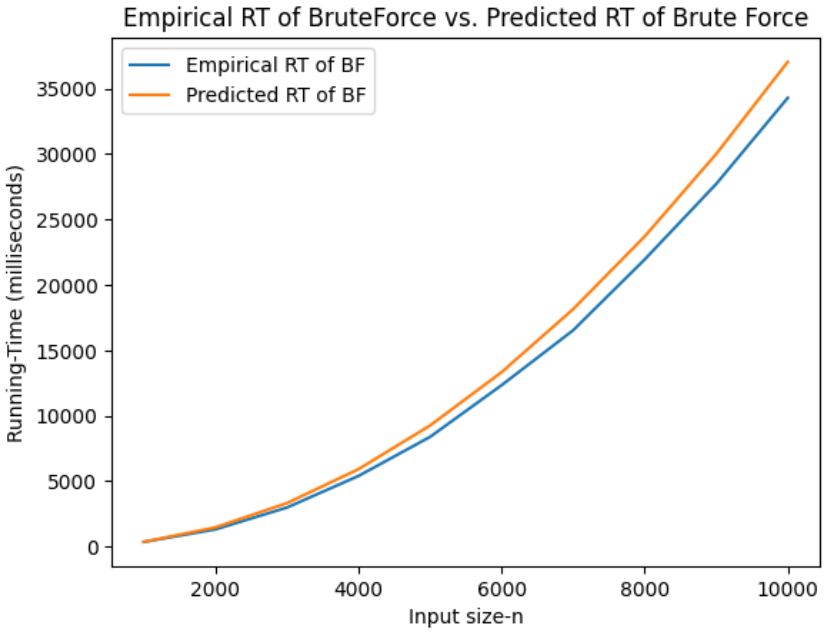
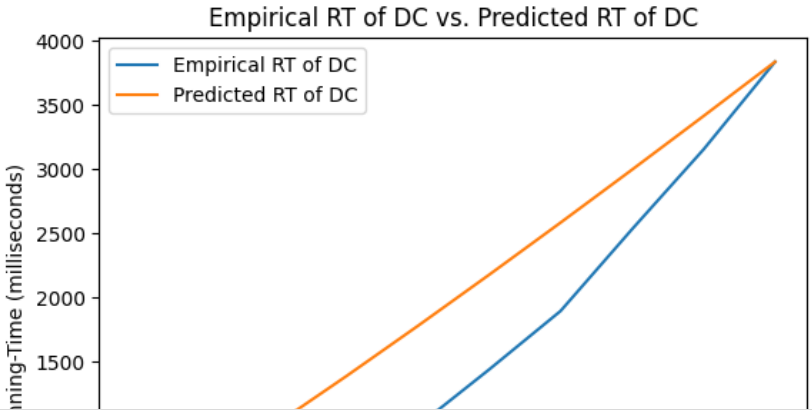
```
import csv
import matplotlib.pyplot as plt
import random
import time
import math

# Function to generate random points within a given range
def generate_random_points(num_points, min_value, max_value):
    point_set = []
    for po in range(num_points):
        point_set.append((random.randint(min_value, max_value), random.randint(min_value, max_value)))
    return point_set

# Function to calculate the square of the distance between two points
def square_distance(point1, point2):
    return ((point1[0] - point2[0]) ** 2 + (point1[1] - point2[1]) ** 2)

# Brute force method to find the closest pair of points
def closest_pair_brute_force(input_points):
    min_dist = float('inf')
    closest_pair = (None, None)

    for i in range(len(input_points)):
        for j in range(i+1, len(input_points)):
            if square_distance(input_points[i], input_points[j]) < min_dist:
                min_dist = square_distance(input_points[i], input_points[j])
                closest_pair = (input_points[i], input_points[j])

    return closest_pair, min_dist

# Recursive function for the divide and conquer approach to find the closest pair
def closest_pair_divide_conquer_rec(px, py):
    if len(px) <= 3:
        return closest_pair_brute_force(px)
    mid = len(px) // 2
    qx = px[:mid]
    rx = px[mid:]
    qy=[]
    ry=[]
    for p in py:
      if(p in qx):
        qy.append(p)
    for p in py:
      if(p in rx):
        ry.append(p)
    (p1, q1), dist1 = closest_pair_divide_conquer_rec(qx, qy)
    (p2, q2), dist2 = closest_pair_divide_conquer_rec(rx, ry)
    delta = min(dist1, dist2)
    best_pair = (p1, q1) if dist1 <= dist2 else (p2, q2)
    mid_x = px[mid][0]
    s_y=[]
    for x in py :
      if (abs(x[0]-mid_x)) <= delta:
        s_y.append(x)
    for i in range(len(s_y) - 1):
        for j in range(i + 1, min(i + 15, len(s_y))):
            p, q = s_y[i], s_y[j]
            dst = square_distance(p, q)
            if dst < delta:
                best_pair = p, q
                delta = dst
    return best_pair, delta

# Functions to use as sort keys for sorting points by x and y coordinates
def x_sort_key(point):
  return point[0]
def y_sort_key(point):
    return point[1]

# Function to find the closest pair using the divide and conquer approach
def closest_pair_divide_conquer(P) :
  Px = sorted(P, key=x_sort_key)  # Sort by x-coordinate
  Py = sorted(P, key=y_sort_key)  # Sort by y-coordinate
  return closest_pair_divide_conquer_rec(Px, Py)

# Function to write the results of the algorithms to a CSV file
```

```
def write_results_to_file(list_sizes, efficient_times, brute_force_times):
    bf_max_ratio = 0
    dc_max_ratio = 0

    # Find the max ratios first
    for i in range(len(list_sizes)):
        list_size = list_sizes[i]
        bf_time_complexity = list_size ** 2
        dc_time_complexity = list_size * math.log2(list_size)
        bf_actual_run_time = brute_force_times[i]
        dc_actual_run_time = efficient_times[i]
        bf_ratio = bf_actual_run_time / bf_time_complexity
        dc_ratio = dc_actual_run_time / dc_time_complexity
        if bf_ratio > bf_max_ratio:
            bf_max_ratio = bf_ratio
        if dc_ratio > dc_max_ratio:
            dc_max_ratio = dc_ratio

    with open("comparison.csv", "w", newline="") as csvfile:
        field_names = [
            "Size of List",
            "Theoretical Time Complexity using Brute Force",
            "Theoretical Time Complexity using Divide and Conquer",
            "Actual Run Time using Brute Force",
            "Actual Run Time using Divide and Conquer",
            "Predicted Runtime using Brute Force",
            "Predicted Runtime using Divide and Conquer",
            "Ratio for Brute Force",
            "Ratio for Divide and Conquer"
        ]
        writer = csv.DictWriter(csvfile, fieldnames=field_names)
        writer.writeheader()

        for i in range(len(list_sizes)):
            list_size = list_sizes[i]
            bf_time_complexity = list_size ** 2
            dc_time_complexity = list_size * math.log2(list_size)
            bf_actual_run_time = brute_force_times[i]
            dc_actual_run_time = efficient_times[i]
            bf_ratio = bf_actual_run_time / bf_time_complexity
            dc_ratio = dc_actual_run_time / dc_time_complexity

            # calculate predicted runtimes
            bf_predicted_run_time = bf_max_ratio * bf_time_complexity
            dc_predicted_run_time = dc_max_ratio * dc_time_complexity

            writer.writerow({
                "Size of List": list_size,
                "Theoretical Time Complexity using Brute Force": bf_time_complexity,
                "Theoretical Time Complexity using Divide and Conquer": dc_time_complexity,
                "Actual Run Time using Brute Force": bf_actual_run_time,
                "Actual Run Time using Divide and Conquer": dc_actual_run_time,
                "Predicted Runtime using Brute Force": bf_predicted_run_time,
                "Predicted Runtime using Divide and Conquer": dc_predicted_run_time,
                "Ratio for Brute Force": bf_ratio,
                "Ratio for Divide and Conquer": dc_ratio
            })

    return bf_max_ratio, dc_max_ratio

# Function to plot graphs comparing the empirical and predicted run times of the algorithms
def plot_graphs(input_sizes, efficient_times, brute_force_times, c1, c2):

    efficient_times = [i*1000 for i in efficient_times] # Convert to milliseconds
    brute_force_times = [i*1000 for i in brute_force_times] # Convert to milliseconds

    # Empirical RT of Algo-1 and Empirical RT of Algo-2
    plt.figure()
    plt.plot(input_sizes, efficient_times, label="Empirical RT of DC")
    plt.plot(input_sizes, brute_force_times, label="Empirical RT of BF")
    plt.xlabel("Input size-n")
    plt.ylabel("Running-Time (milliseconds)")
    plt.legend()
    plt.title("Empirical RT of DC  vs. Empirical RT of BF")
    plt.show()

    # Empirical RT of Algo-1 and Predicted RT of Algo-1
```

```
        plt.figure()
        plt.plot(input_sizes, brute_force_times, label="Empirical RT of BF")
        plt.plot(input_sizes, [c1 * n ** 2*1000 for n in input_sizes], label="Predicted RT of BF")
        plt.xlabel("Input size-n")
        plt.ylabel("Running-Time (milliseconds)")
        plt.legend()
        plt.title("Empirical RT of BruteForce vs. Predicted RT of Brute Force")

        # Empirical RT of Algo-2 and Predicted RT of Algo-2
        plt.figure()
        plt.plot(input_sizes, efficient_times, label="Empirical RT of DC")
        plt.plot(input_sizes, [c2 * n * math.log2(n)*1000 for n in input_sizes], label="Predicted RT of DC")
        plt.xlabel("Input size-n")
        plt.ylabel("Running-Time (milliseconds)")
        plt.legend()
        plt.title("Empirical RT of DC vs. Predicted RT of DC")

    # Main execution block
    if __name__ == "__main__":
        sizes = [i for i in range(1000, 10001, 1000)] # Sizes from 10^3 to 10^4 in steps of 10^3
        bf_times = []
        dc_times = []
        c1 = c2 = 1
        num_runs = 10

        for size in sizes:
            temp_bf_times = []
            temp_dc_times = []

            points = generate_random_points(size, 0, 1e9)
            for _ in range(num_runs):
                start_time = time.time()
                closest_pair_brute_force(points)
                temp_bf_times.append(time.time() - start_time)

                start_time = time.time()
                closest_pair_divide_conquer(points)
                temp_dc_times.append(time.time() - start_time)

            bf_times.append(sum(temp_bf_times) / num_runs)
            dc_times.append(sum(temp_dc_times) / num_runs)

        c1,c2 = write_results_to_file(sizes, dc_times, bf_times)
        plot_graphs(sizes, dc_times, bf_times, c1, c2)
```







✓ 24m 26s completed at 11:47 AM