"Analysis of the Closest Pair of Points Problem

Brute-Force vs Divide-and-Conquer "

HARSHITHA MOHAN – Z23814651

Problem Definition

The objective of this project is to analyze and compare the computational efficiency of two algorithms that solve the Closest Pair of Points Problem. The Closest Pair of Points Problem is defined as follows: Given a set of n points P1, P2,…,Pn in a 2-dimensional plane, the goal of this project is to find two points in a plane that are closest to each other based on the Euclidean distance. We are given n points, each with x and y coordinates. The task is to find which two points have the smallest distance between them.

The distance between any two points $P_i(x_i, y_i)$ and $P_j(x_j, y_j)$ is calculated using:

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

The **input size (n)** is the total number of points given to the program. In this project, we test the algorithms with different input sizes, starting from **10,000 points** and going up to **100,000 points**, increasing by **10,000** each time.

Real-World Applications

1. Navigation and Mapping Systems: The closest pair of points problem helps in GPS and mapping applications to find nearby locations, such as the nearest gas station or restaurant to a user's position.
2. Data Clustering and Machine Learning: In data analysis, this problem is used to group similar data points together based on distance. It helps algorithms identify which data points are close in value or behavior.
3. Computer Graphics and Game Development: It is used in computer graphics and gaming to detect when two objects are close to each other or about to collide, improving motion and animation accuracy.
4. Robotics and Sensor Networks: Robots and sensors often use this logic to detect nearby obstacles or other robots to avoid collisions or to coordinate movements.
5. Air Traffic and Logistics: In air traffic control or shipping, finding the closest pair can help detect possible conflicts or plan efficient routes between locations.

Algorithms and Running Time Analysis

ALG1: Brute-Force Algorithm

The Brute-Force Algorithm is the simplest way to solve the Closest Pair of Points problem. It compares the distance between every possible pair of points and keeps track of the smallest distance found. Since it checks all combinations, it guarantees the correct result, but it requires a large number of computations as the number of points increases.

Pseudocode

```
BruteForceClosestPair(P[1..n])
   min_dist = infinity
   for i = 1 to n-1
      for j = i+1 to n
         dist = EuclideanDistance(P[i], P[j])
         if dist < min_dist
            min_dist = dist
            closest_pair = (P[i], P[j])
   return closest_pair
EuclideanDistance(Point p1, Point p2)
   return sq.root((p2.x - p1.x)^2 + (p2.y - p1.y)^2)
```

Run Time Analysis

- The brute force algorithm computes the distance between every pair of points, resulting in $O(n^2)$
- distance computations.
- Each distance computation requires constant time, so the overall time complexity of the brute force algorithm is **$O(n^2)$**.

ALG2: Divide and Conquer Algorithm

The Divide-and-Conquer Algorithm improves efficiency by dividing the set of points into two halves, solving the problem separately for each half, and then combining the results. After finding the closest pairs in both halves, it checks only the points near the dividing line to find any closer pairs that cross the boundary.

Pseudocode

```
ClosestPairRecursive(P[1..n])
   if n ≤ 3
      return BruteForceClosestPair(P)
   mid = n / 2
   left_pair = ClosestPairRecursive(P[1..mid])
   right_pair = ClosestPairRecursive(P[mid+1..n])
   min_dist = min(left_pair.distance, right_pair.distance)
   mid_strip = points within min_dist of the line passing through P[mid]
   strip_pair = ClosestPairInStrip(mid_strip, min_dist)
   if strip_pair.distance < min_dist
      return strip_pair
   else if left_pair.distance < right_pair.distance
      return left_pair
   else
      return right_pair

ClosestPairInStrip(mid_strip, min_dist)
   min_dist = infinity
   for i = 1 to size(mid_strip)-1
      for j = i+1 to min(size(mid_strip), i+7)
         dist = EuclideanDistance(mid_strip[i], mid_strip[j])
         if dist < min_dist
            min_dist = dist
            closest_pair = (mid_strip[i], mid_strip[j])
   return closest_pair

BruteForceClosestPair(P)
   min_dist = infinity
   for i = 1 to size(P)-1
      for j = i+1 to size(P)
         dist = EuclideanDistance(P[i], P[j])
         if dist < min_dist
            min_dist = dist
            closest_pair = (P[i], P[j])
   return closest_pair
EuclideanDistance(Point p1, Point p2)
   return sq.root((p2.x - p1.x)^2 + (p2.y - p1.y)^2)
```

Run Time Analysis

- The divide and conquer algorithm first sort the points by their x-coordinate, which takes. θ (nlogn) time.
- The recursion tree has a height of logn with each level performing O(n) work.

- The closest pair in the strip is found in O(n) time using a linear scan.
- Overall, the time complexity of the divide and conquer algorithm is **θ (nlogn)**.

Experimental Results

Table 1 - ALG1: Brute-Force Algorithm

n	Theoretical rt (n^2)	Empirical rt	Ratio r = empirical rt / theoretical rt	Predicted rt max(r) * n^2
10000	100000000	6484.73	0.0000648473	7261.7
20000	400000000	25817.60	0.0000645440	29046.8
30000	900000000	58200.74	0.0000646674	65355.3
40000	1600000000	103720.32	0.0000648252	116187.2
50000	2500000000	163965.69	0.0000655862	181542.5
60000	3600000000	241252.34	0.0000670145	261421.2
70000	4900000000	344379.46	0.0000702815	355823.3
80000	6400000000	455351.31	0.0000711486	464748.8
90000	8100000000	577654.94	0.0000713154	588197.7
100000	10000000000	726170.08	0.0000726170	726170.0

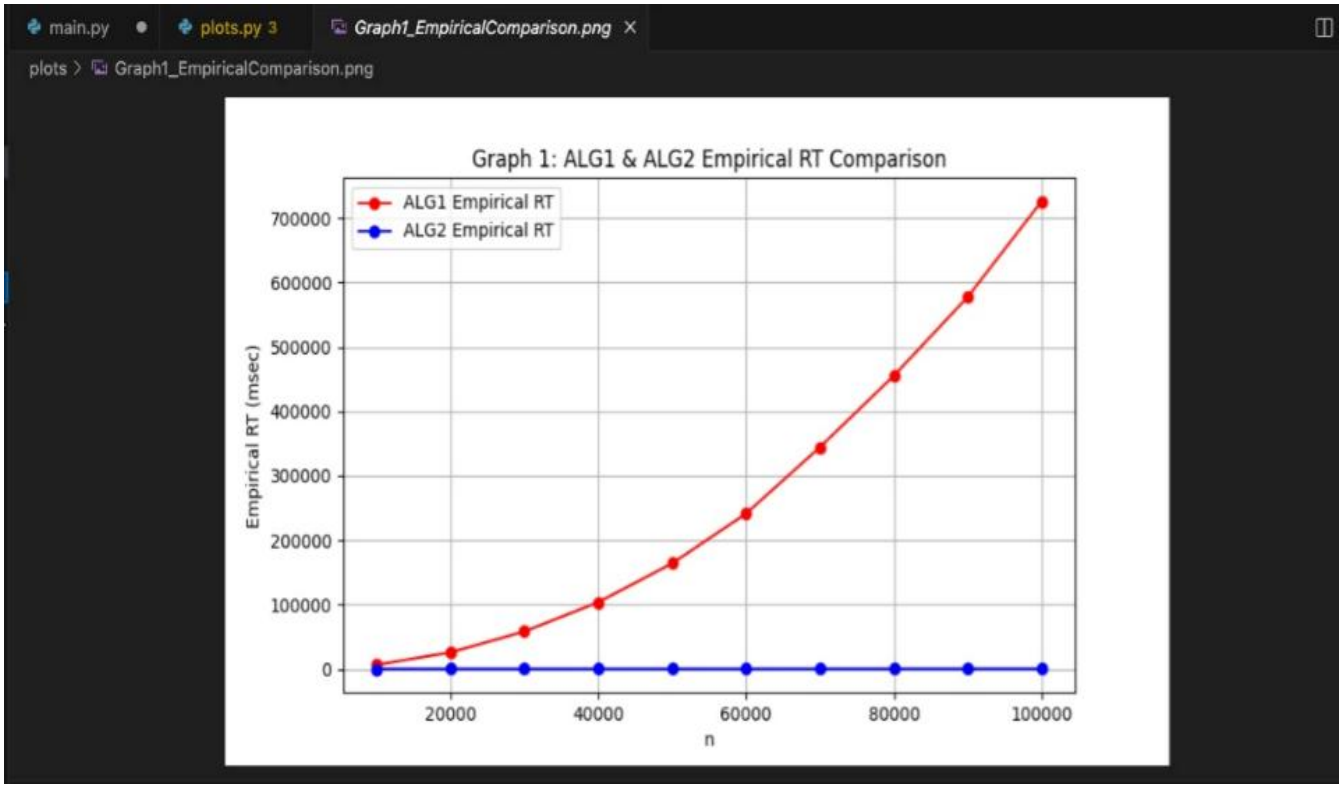
C1 = max (Ratio) = 0. 0000726170

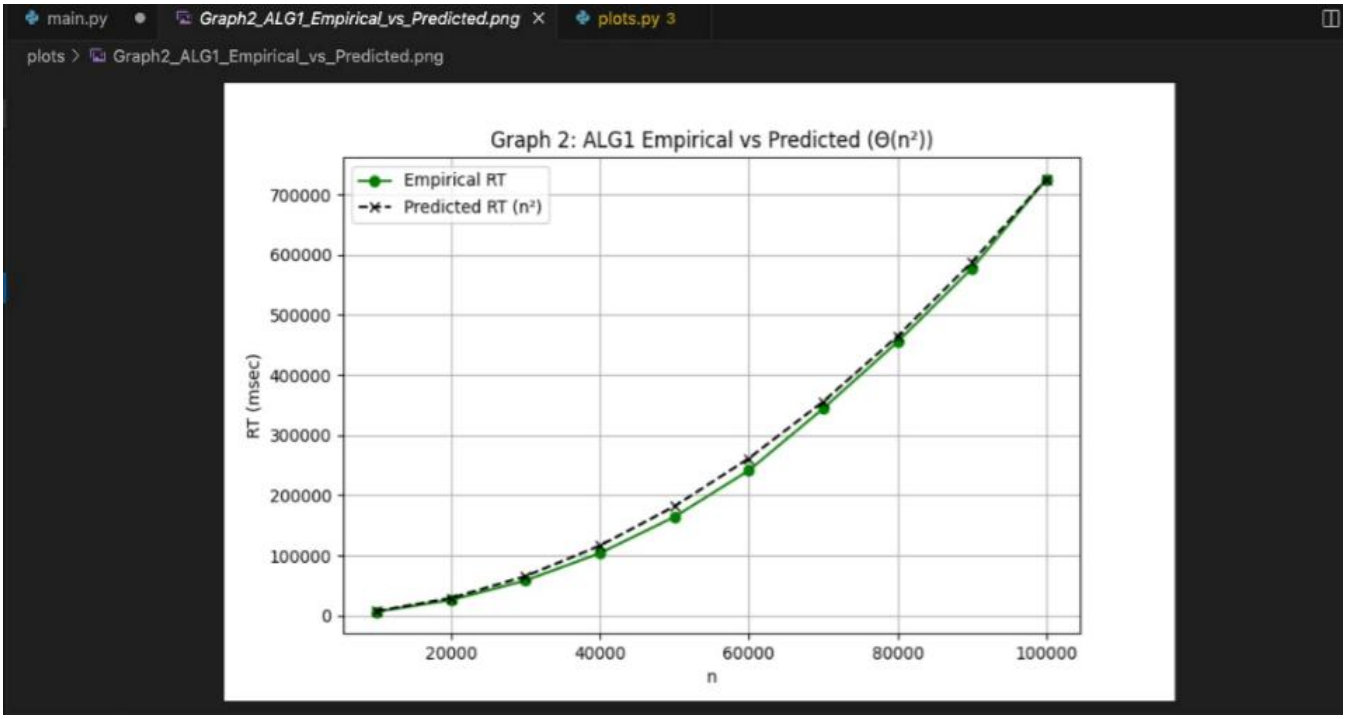
Table 2 - ALG2: Divide and Conquer Algorithm

n	Theoretical rt (nlogn)	Empirical rt	Ratio r = empirical rt / theoretical rt	Predicted rt max(r) * n^2
10000	132877.12	19.56	0.000147	20.104
20000	285754.24	40.54	0.000141	43.235
30000	446180.24	65.13	0.000145	67.508
40000	611508.49	86.92	0.000142	92.523
50000	780482.02	106.78	0.000136	118.090
60000	952360.49	138.85	0.000145	144.096
70000	1126654.71	170.46	0.000151	170.467
80000	1303016.99	190.57	0.000146	197.151
90000	1481187.36	210.33	0.000142	224.109
100000	1660964.04	238.54	0.000143	251.310

C2 = max (Ratio) = 0. 000151

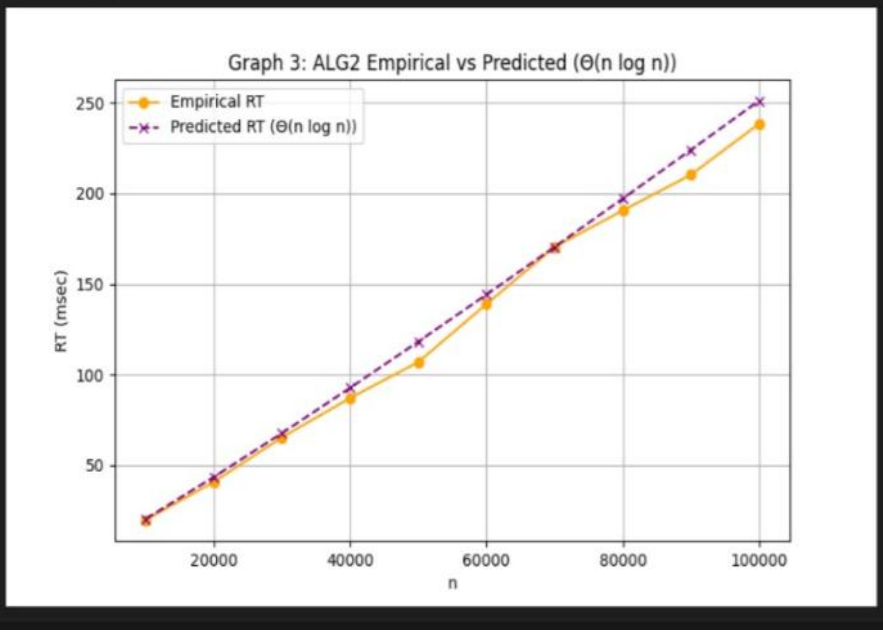
Graphs


main.py
plots.py 3
Graph1_EmpiricalComparison.png
plots > Graph1_EmpiricalComparison.png
Graph 1: ALG1 & ALG2 Empirical RT Comparison
ALG1 Empirical RT
ALG2 Empirical RT
Empirical RT (msec)
0
100000
200000
300000
400000
500000
600000
700000
20000
40000
60000
80000
100000
n


main.py
Graph2_ALG1_Empirical_vs_Predicted.png
plots.py 3
plots > Graph2_ALG1_Empirical_vs_Predicted.png
Graph 2: ALG1 Empirical vs Predicted (Θ(n²))
Empirical RT
Predicted RT (n²)
RT (msec)
0
100000
200000
300000
400000
500000
600000
700000
20000
40000
60000
80000
100000
n

main.py ● Graph3_ALG2_Empirical_vs_Predicted.png × plots.py 3

plots > Graph3_ALG2_Empirical_vs_Predicted.png

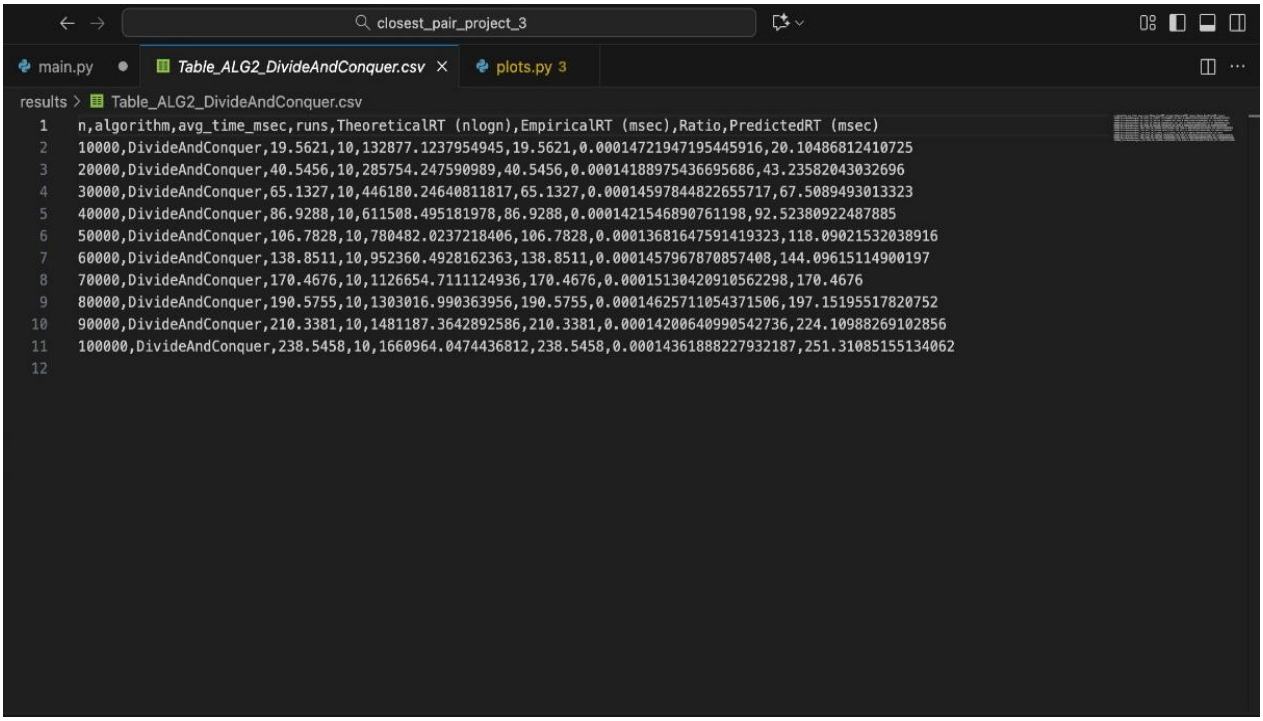


closest_pair_project_3

main.py ● Table_ALG1_BruteForce.csv × plots.py 3

results > Table_ALG1_BruteForce.csv

```
1   n,algorithm,avg_time_msec,runs,TheoreticalRT (n²),EmpiricalRT (msec),Ratio,PredictedRT (msec)
2   10000,BruteForce,6484.7389,10,100000000,6484.7389,6.4847389e-05,7261.700801
3   20000,BruteForce,25817.6073,10,400000000,25817.6073,6.454401825e-05,29046.803204
4   30000,BruteForce,58200.7446,10,900000000,58200.7446,6.4667494e-05,65355.307209
5   40000,BruteForce,103720.3248,10,1600000000,103720.3248,6.4825203e-05,116187.212816
6   50000,BruteForce,163965.6976,10,2500000000,163965.6976,6.558627904e-05,181542.520025
7   60000,BruteForce,241252.3496,10,3600000000,241252.3496,6.701454155555555e-05,261421.228836
8   70000,BruteForce,344379.4608,10,4900000000,344379.4608,7.02815226122449e-05,355823.339249
9   80000,BruteForce,455351.3196,10,6400000000,455351.3196,7.11486436875e-05,464748.851264
10  90000,BruteForce,577654.9432,10,8100000000,577654.9432,7.131542508641975e-05,588197.764881
11  100000,BruteForce,726170.0801,10,10000000000,726170.0801,7.261700801e-05,726170.0801
12
```


closest_pair_project_3
main.py
Table_ALG2_DivideAndConquer.csv
plots.py 3
results > Table_ALG2_DivideAndConquer.csv
n,algorithm,avg_time_msec,runs,TheoreticalRT (nlogn),EmpiricalRT (msec),Ratio,PredictedRT (msec)
10000,DivideAndConquer,19.5621,10,132877.1237954945,19.5621,0.00014721947195445916,20.10486812410725
20000,DivideAndConquer,40.5456,10,285754.247590989,40.5456,0.00141889754366956 86,43.23582043032696
30000,DivideAndConquer,65.1327,10,446180.24640811817,65.1327,0.00014597844822655717,67.5089493013323
40000,DivideAndConquer,86.9288,10,611508.495181978,86.9288,0.0001421546890761198,92.52380922487885
50000,DivideAndConquer,106.7828,10,780482.0237218406,106.7828,0.00013681647591419323,118.09021532038916
60000,DivideAndConquer,138.8511,10,952360.4928162363,138.8511,0.00014579678708574 08,144.09615114900197
70000,DivideAndConquer,170.4676,10,1126654.7111124936,170.4676,0.00015130420910562298,170.4676
80000,DivideAndConquer,190.5755,10,1303016.990363956,190.5755,0.00014625711054371506,197.15195517820752
90000,DivideAndConquer,210.3381,10,1481187.3642892586,210.3381,0.00014200640990542736,224.10988269102856
100000,DivideAndConquer,238.5458,10,1660964.0474436812,238.5458,0.00014361888227932187,251.31085155134062

Implementation Code: Brute-Force and Divide and Conquer Algorithm

```
import random
import time
import math
import csv
import os

# Creating Random Points
def generate_points(n):
    points = set()
    while len(points) < n:
        x = random.randint(0, 32767)
        y = random.randint(0, 32767)
        points.add((x, y))
    return list(points)

# Calculating Euclidean distance
def distance(p1, p2):
    return math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

# Brute-Force Algorithm
def brute_force_closest_pair(points):
    min_dist = float('inf')
    pair = None
    for i in range(len(points)):
        for j in range(i + 1, len(points)):
            dist = distance(points[i], points[j])
            if dist < min_dist:
                min_dist = dist
                pair = (i, j)
    return pair
```

```
# Divide and Conquer Algorithm
def closest_pair_dc(points):
    if len(points) <= 3:
        return brute_force_closest_pair(points)

    mid = len(points) // 2
    mid_point = points[mid]

    dl = closest_pair_dc(points[:mid])
    dr = closest_pair_dc(points[mid:])

    if distance(points[dl[0]], points[dl[1]]) < distance(points[dr[0]], points[dr[1]]):
        dmin = distance(points[dl[0]], points[dl[1]])
        min_pair = dl
    else:
        dmin = distance(points[dr[0]], points[dr[1]])
        min_pair = dr

    strip = [p for p in points if abs(p[0] - mid_point[0]) < dmin]
    strip.sort(key=lambda point: point[1])

    for i in range(len(strip)):
        for j in range(i + 1, len(strip)):
            if (strip[j][1] - strip[i][1]) < dmin:
                dist = distance(strip[i], strip[j])
                if dist < dmin:
                    dmin = dist
                    min_pair = (points.index(strip[i]), points.index(strip[j]))
            else:
                break
    return min_pair
```

Calculating Runtime and Main Function

```
# Calculating Runtime
def measure_runtime(func, points):
    start_time = time.time()
    func(points)
    return (time.time() - start_time) * 1000


# Main function
def main():
    ns = [10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000]
    m = 10

    os.makedirs("results", exist_ok=True)
    csv_path = "results/avg_runtimes.csv"

    with open(csv_path, "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["n", "algorithm", "avg_time_msec", "runs"])

    for n in ns:
        bf_times = []
        dc_times = []

        for iteration in range(m):
            points = generate_points(n)
            points.sort(key=lambda point: point[0])

            bf_time = measure_runtime(lambda pts: brute_force_closest_pair(pts),
points)     dc_time = measure_runtime(lambda pts: closest_pair_dc(pts), points)

            bf_times.append(bf_time)
            dc_times.append(dc_time)

            print(f"n={n}, iteration={iteration+1}, "
                  f"BruteForce={bf_time:.2f} ms, Divide&Conquer={dc_time:.2f} ms")

        avg_bf = sum(bf_times) / m
        avg_dc = sum(dc_times) / m

        with open(csv_path, "a", newline="") as f:
            writer = csv.writer(f)
            writer.writerow([n, "BruteForce", f"{avg_bf:.4f}", m])
            writer.writerow([n, "DivideAndConquer", f"{avg_dc:.4f}", m])

        print(f" Average results n={n}: "
              f"AVG BruteForce={avg_bf:.2f} ms, AVG Divide&Conquer={avg_dc:.2f} ms")


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

Graph Plotting Function

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import os

def main():
    os.makedirs("plots", exist_ok=True)
    os.makedirs("results", exist_ok=True)

    df = pd.read_csv("results/avg_runtimes.csv")

    df["avg_time_msec"] = pd.to_numeric(df["avg_time_msec"], errors="coerce")

    bf = df[df["algorithm"] == "BruteForce"].copy()
    dnc = df[df["algorithm"] == "DivideAndConquer"].copy()

    bf["TheoreticalRT (n²)"] = bf["n"] ** 2
    bf["EmpiricalRT (msec)"] = bf["avg_time_msec"]
    bf["Ratio"] = bf["EmpiricalRT (msec)"] / bf["TheoreticalRT (n²)"]

    bf = bf.dropna(subset=["Ratio"])

    if not bf.empty:
        c1 = bf["Ratio"].max()
    else:
        c1 = np.nan
    bf["PredictedRT (msec)"] = c1 * bf["TheoreticalRT (n²)"]

    bf.to_csv("results/Table_ALG1_BruteForce.csv", index=False)
    print(f"c1 (max ratio) = {c1}")

    dnc["TheoreticalRT (nlogn)"] = dnc["n"] * np.log2(dnc["n"])
    dnc["EmpiricalRT (msec)"] = dnc["avg_time_msec"]
    dnc["Ratio"] = dnc["EmpiricalRT (msec)"] / dnc["TheoreticalRT (nlogn)"]

    dnc = dnc.dropna(subset=["Ratio"])

    if not dnc.empty:
        c2 = dnc["Ratio"].max()
    else:
        c2 = np.nan
    dnc["PredictedRT (msec)"] = c2 * dnc["TheoreticalRT (nlogn)"]

    dnc.to_csv("results/Table_ALG2_DivideAndConquer.csv", index=False)
    print(f"c2 (max ratio) = {c2}")

    # Graphs
    plt.figure(figsize=(8, 5))
    plt.plot(bf["n"], bf["EmpiricalRT (msec)"], 'o-', label="ALG1 Empirical RT", color="red")
    plt.plot(dnc["n"], dnc["EmpiricalRT (msec)"], 'o-', label="ALG2 Empirical RT", color="blue")
    plt.xlabel("n")
    plt.ylabel("Empirical RT (msec)")
    plt.title("Graph 1: ALG1 & ALG2 Empirical RT Comparison")
    plt.legend()
    plt.grid(True)
    plt.savefig("plots/Graph1_EmpiricalComparison.png")


    plt.figure(figsize=(8, 5))
    plt.plot(bf["n"], bf["EmpiricalRT (msec)"], 'o-', label="Empirical RT", color="green")
    plt.plot(bf["n"], bf["PredictedRT (msec)"], 'x--', label="Predicted RT (n²)", color="black")
    plt.xlabel("n")
    plt.ylabel("RT (msec)")
    plt.title("Graph 2: ALG1 Empirical vs Predicted (Θ(n²))")
    plt.legend()
    plt.grid(True)
    plt.savefig("plots/Graph2_ALG1_Empirical_vs_Predicted.png")

    plt.figure(figsize=(8, 5))
    plt.plot(dnc["n"], dnc["EmpiricalRT (msec)"], 'o-', label="Empirical RT", color="orange")
    plt.plot(dnc["n"], dnc["PredictedRT (msec)"], 'x--', label="Predicted RT (Θ(n log n))",
color="purple")
    plt.xlabel("n")
    plt.ylabel("RT (msec)")
    plt.title("Graph 3: ALG2 Empirical vs Predicted (Θ(n log n))")
    plt.legend()
    plt.grid(True)
    plt.savefig("plots/Graph3_ALG2_Empirical_vs_Predicted.png")


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

Conclusion

Based on the experimental results, both the **Brute-Force** and **Divide-and-Conquer** algorithms performed just as their theoretical time complexities predicted.

For the Brute-Force Algorithm (ALG1), the running time increased very quickly as the number of points grew. This is expected, since its time complexity is $O(n^2)$, meaning it checks every possible pair of points to find the closest two. When the number of points doubles, the number of comparisons roughly quadruples, which explains the steep rise in execution time. The experimental data clearly shows this pattern, confirming that the algorithm becomes much slower for large datasets. Even so, it's simple to implement and always gives the correct answer, making it useful for smaller datasets or when accuracy is more important than speed.

The Divide-and-Conquer Algorithm (ALG2), in contrast, ran much faster and handled larger inputs more efficiently. Its performance closely followed the expected O(n log n) behavior. By dividing the points into smaller groups and only checking the ones near the dividing line, it avoids a lot of unnecessary comparisons. The measured times were very close to the predicted values, showing that this algorithm scales well and remains efficient even as the number of points increases. This makes it a better choice for real-world tasks like mapping, spatial analysis, or computer graphics, where data sizes are large and quick results are important.

Overall, the **empirical (observed)** results are consistent with the **theoretical (predicted)** time complexities of both algorithms. The divide-and-conquer approach clearly outperforms the brute-force method in terms of speed and efficiency, making it the preferred solution for large-scale problems involving many data points.

Project Demo: https://youtu.be/a3-97b97Ow0?si=bq8ScC5ncX8d4EFR

Github : https://github.com/harshi2520/Project---Closest-Pair-of-Points-

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, *Algorithms*, New York, NY, USA: McGraw-Hill Education, 2021.

[3] R. Klein, "Computational geometry: Algorithms and applications in spatial data processing," *IEEE Computer Graphics and Applications*, vol. 42, no. 2, pp. 50–63, Mar.–Apr. 2022.

[4] GeeksforGeeks, "Closest Pair of Points using Divide and Conquer algorithm," [Online]. Available: https://www.geeksforgeeks.org/closest-pair-of-points/.

[5] COT 6405: Analysis of Algorithms – Lecture Notes, Fall 2025, Florida Atlantic University.