Analysis of Algorithms Project

THE CLOSEST PAIR OF POINTS

~ Rajeev Pondala

Problem Statement:

Finding the Closest Pair of Points Problem given n points P1, P2, …, Pn in a plane, find the pair of points that is closest together. Return the indexes i and j of the two closest points Pi and Pj.

Input Size:

'n' is the input size. 'n' is ranging from 10^3 to 10^4 in the steps of 1000.

Relevance in the real world Applications:

Location-based Services (LBS):

In applications like GPS navigation systems or ride-sharing apps, finding the closest pair of points helps in determining the nearest points of interest, such as restaurants, gas stations, or ATMs.

Supply Chain Management:

logistics and supply chain management, finding the closest pair of points can optimize routes for delivery vehicles, reducing fuel consumption and delivery times.

E-commerce:

Online retail platforms use the closest pair of points algorithm to suggest nearby stores or warehouses for faster delivery of products to customers.

Algorithms:

1.Brute Force Algorithm:

The brute force algorithm is a straightforward approach to solving a problem that systematically tries all possible solutions. It involves generating all possible solutions, evaluating each one, and then selecting the best solution. While this method is conceptually simple and easy to implement, it is often inefficient, especially for large problem instances, because it requires examining every possible solution.

Pseudo Code:

```
import math
function closest_pair_brute_force(input_points)
   min_dist = infinity
   closest_pair = (None, None)

   for i from 0 to length(input_points) - 1 do
      point1 = input_points[i]
      for j from i + 1 to length(input_points) - 1 do
         point2 = input_points[j]
         dist = distance(point1, point2)
         if dist < min_dist then
            min_dist = dist
            closest_pair = (point1, point2)

   return closest_pair, min_dist
```

Runtime Analysis:

The algorithm involves a nested loop structure. For each point `p1` in the list, it compares it with all other points `p2` (excluding itself) using a second loop.

The outer loop runs from 0 to `n - 1`, where `n` is the number of input points and The inner loop runs from `i + 1` to `n - 1` for each `i` in the outer loop.

This results in a total of (frac{{n times (n - 1)}}{2}) comparisons.
Asymptotically, this is equivalent to (O(n^2)\).
The number of comparisons performed by the algorithm is given by the sum of the first (n - 1) positive integers, which is (frac{{n times (n - 1)}}{2}).
Although the time complexity can be approximated to (O(n^2)), as `n` becomes significantly large, the `- 1` term becomes insignificant.
The algorithm's time complexity is (O(n^2)).

2.Divide And Conquer Algorithm:

The algorithm employs a divide-and-conquer strategy, breaking down the problem into smaller sub-problems, solving them recursively, and then merging the results to find the overall solution. Initially, the algorithm divides the list of points into two roughly equal halves based on their x-coordinates. It then recursively finds the closest pair of points in each of these divided sub-problems.

Following the division, comes the merging step. The algorithm defines a 'strip' region around the middle x-coordinate, with a width equal to the minimum distance found in the

two sub-problems. Within this strip, only points falling within a certain range are considered, significantly reducing the number of points to be examined. To streamline the merging process, the points in the strip are sorted based on their y-coordinates. The algorithm then iterates through these sorted strip points, comparing the distances between adjacent points. Since the number of points in the strip is relatively small, this process is efficient. If a pair of points in the strip has a smaller distance than the minimum distance found so far, the algorithm updates the minimum distance and the corresponding closest pair.

Once all the recursive calls and merging processes are complete, the algorithm identifies the closest pair of points and their distance."

PseudoCode:

```
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 = sorted([p for p in py if p in Qx], key=lambda x: x[1])
    Ry = sorted([p for p in py if p in Rx], key=lambda x: x[1])

    (p1, q1), dist1 = closest_pair_divide_conquer_rec(Qx, Qy)
    (p2, q2), dist2 = closest_pair_divide_conquer_rec(Rx, Ry)

    delta = min(dist1, dist2)
    closer_pair = (p1, q1) if dist1 <= dist2 else (p2, q2)

    Sy = [point for point in py if abs(point[0] - px[mid][0]) < delta]

    for i in range(len(Sy)):
        for j in range(i+1, min(i+15, len(Sy))):
            p, q = Sy[i], Sy[j]
            dst = distance(p, q)
            if dst < delta:
                closer_pair = p, q
                delta = dst

    return closer_pair, delta
```

Runtime Analysis:

1. Division Step: Sorting the points takes (O(n \log n)) time.

2.Conquer Step: Recursively solving two smaller problems of size (n/2) each. The recurrence relation for this step is (T(n) = 2T(n/2) + O(n)), which gives us a time complexity of (O(n log n)) in the recursive case.

3. Finding the closest split pair can be done in linear time, (O(n)).

The overall time complexity of the Divide and Conquer algorithm for the Closest Pair of Points problem is (O(n log n)).

The recurrence relation for the algorithm is

Comparing this to the Master Theorem:

T(n) = aT (n/b) + f(n)
In this case, (a = b = 2), so (log_b(a) = log_2(2) = 1). Since (f(n) = O(n)), we're in case 2 with (c = 1) and (k = 0). Therefore, by the Master Theorem, the time complexity of the algorithm is (Theta(n^c log^{k+1}(n)) = Theta(n log n)), which is (O(n log n)).

Experiment Results:

Input size:

The input values are from 10^3 ,2*(10^3), 3*(10^3),.....10*(10^3)

C1 = max (r1, r2, …., r10)
PredictedRT = C1∗ TheoreticalRT

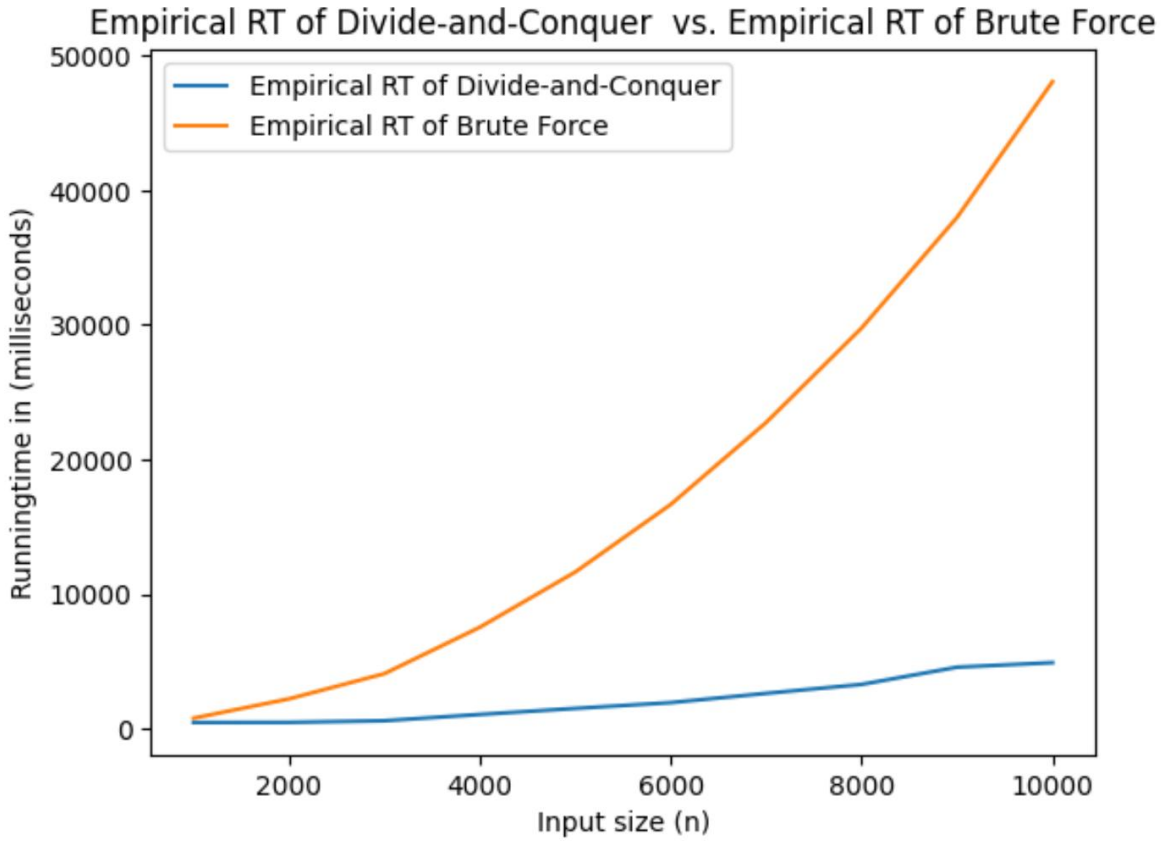
Brute Force Algorithm:

Input Size (n)	Theoretical TC using BF	Empirical RT of algo using BF	Ratio of algo using BF	Predicted RT for BF
1000	1000000	0. 49663763	4.966e-07	0.462184379
2000	4000000	1.774048328	4.345e-07	1.848737519
3000	9000000	4.064314627	4.515e-07	4.159659941
4000	16000000	7.645903587	4.778e-07	7.394950077
5000	25000000	11.88294799	4.753e-07	11.55460945
6000	36000000	16.63459725	4.620e-07	16.63863773
7000	49000000	22.80722856	4.654e-07	22.64703461
8000	64000000	29.80772902	4.657e-07	29.57980030
9000	81000000	37.37292206	4.613e-07	37.43693476
10000	100000000	46.21843798	4.621e-07	46.21843798

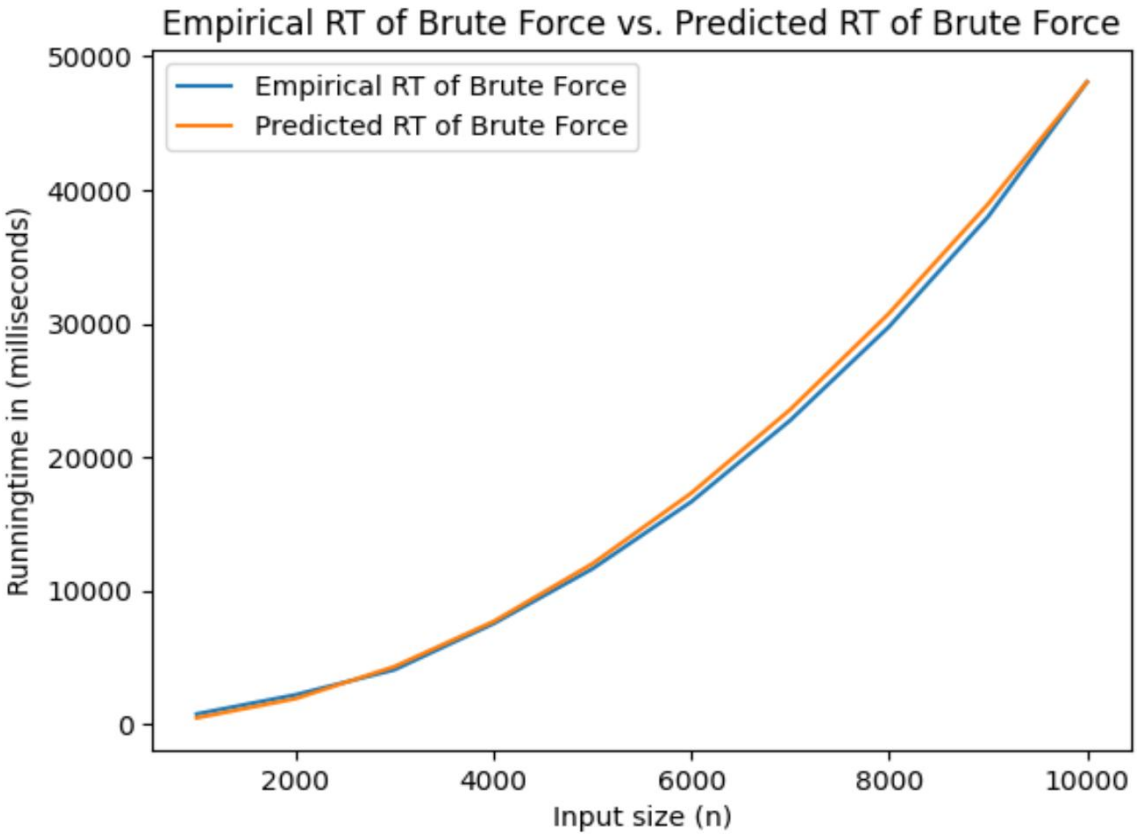
Divide and Conquer Algorithm:

Input Size (n)	Theoretical TC using DC	Empirical RT of Algorithm using DC	Ratio of Algorithm using DC	Predicted RT for DC
1000	9965.784266	0.118161582	1.185e-05	0.381319937
2000	21931.56856	0.437108969	1.993e-05	0.839165700
3000	34652.24035	0.589266061	1.700e-05	1.325895658
4000	47863.13713	1.071921968	2.239e-05	1.831380524
5000	61438.5618	1.474009108	2.399e-05	2.350818349
6000	75304.48071	1.918865299	2.548e-05	2.881368794
7000	89411.97444	2.619920635	2.930e-05	3.421162599
8000	103726.2742	3.372230315	3.251e-05	3.968869408
9000	118221.3835	3.974033951	3.361e-05	4.523494515
10000	132877.1237	5.084265828	3.826e-05	5.084265821

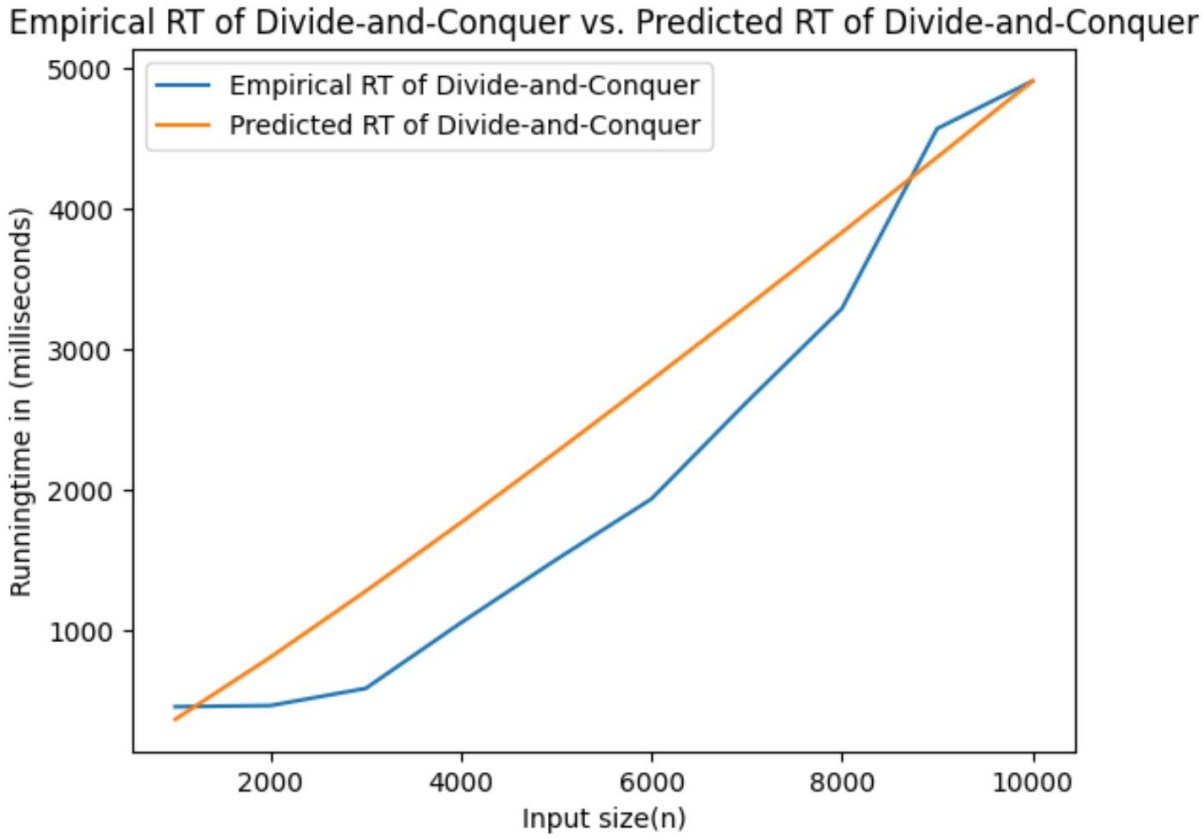
Graph-1:



Graph-2:



Graph-3:



Conclusion:

The Closest Pair of Points problem is a classic problem in computational geometry. It involves finding the pair of points in a set that are closest to each other in terms of Euclidean distance. This problem can be approached using either the Brute Force method or the Divide and Conquer approach, each with its own set of advantages and disadvantages.

Brute Force Method:

The Brute Force method is the simplest approach to solving the Closest Pair of Points problem. It involves a pairwise comparison of each point with all others to find the minimum distance. The algorithm iterates through each point in the set and calculates its distance to every other point, keeping track of the minimum distance found.

While the Brute Force method is easy to understand and implement, its simplicity comes at the cost of efficiency. The time complexity of the Brute Force method is O(n^2), where n is the number of points in the set. This means that as the number of points increases, the running time of the algorithm grows quadratically. As a result, the Brute Force method becomes increasingly inefficient for large datasets.

One advantage of the Brute Force method is that it requires only constant space complexity, O(1). This means that it does not require any additional memory beyond what is needed to store the input data.

Divide and Conquer Method:

The Divide and Conquer approach offers a significant improvement in efficiency compared to the Brute Force method. This method breaks the problem down into smaller subsets, recursively solving the problem for each subset, and then combining the results to find the overall solution.

In the case of the Closest Pair of Points problem, the Divide and Conquer approach involves dividing the set of points into two equal halves, finding the closest pair of points in each half, and then considering pairs of points that are split across the two halves.

The key advantage of the Divide and Conquer approach is its improved time complexity. The time complexity of this method is O(n log n), which is significantly better than the O(n^2) time complexity of the Brute Force method. This makes the Divide and Conquer approach a much more efficient choice for large datasets.

However, the Divide and Conquer method does have some drawbacks. It requires additional space complexity of O(n) to store the sorted points and make recursive function calls. This can be a limiting factor for very large datasets or in situations where memory resources are constrained.

When choosing between the Brute Force and Divide and Conquer methods for solving the Closest Pair of Points problem, several factors must be considered:

Size of the Dataset: For smaller datasets, the simplicity of the Brute Force method may be advantageous, especially if ease of implementation is a priority.

Efficiency: For larger datasets, the improved efficiency of the Divide and Conquer approach is likely to outweigh its additional complexity. The O(n log n) time complexity of this method makes it much more suitable for handling large amounts of data.

Available Computational Resources: The Divide and Conquer method requires more memory resources due to its O(n) space complexity. Therefore, the available memory resources should be taken into account when choosing a method.

In summary, while the Brute Force method is simpler to implement, the Divide and Conquer approach offers a significant improvement in efficiency, particularly for larger datasets. The choice of method should be based on the specific requirements of the problem and the available computational resources.

Colab Link:
https://colab.research.google.com/drive/1UZqKjPrndIfvOxCIkhio_O5NxnA2E4p9?usp=sharing

Source Code

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

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

# Function to generate random points
def generate_random_points(np, mini, maxi):
    ps = [(random.randint(mini, maxi), random.randint(mini, maxi)) for _ in range(np)]
    return ps

# Brute force method to find the closest pair
def closest_pair_brute_force(input_points):
    min_dist = float('inf')
    closest_pair = (None, None)
    for i, point1 in enumerate(input_points):
        for point2 in input_points[i+1:]:
            dist = distance(point1, point2)
            if dist < min_dist:
                min_dist = dist
                cp = (point1, point2)
    return cp, min_dist

# Recursive function for the divide and conquer
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 = sorted([p for p in py if p in Qx], key=lambda x: x[1])
    Ry = sorted([p for p in py if p in Rx], key=lambda x: x[1])
    (p1, q1), dist1 = closest_pair_divide_conquer_rec(Qx, Qy)
    (p2, q2), dist2 = closest_pair_divide_conquer_rec(Rx, Ry)
    delta = min(dist1, dist2)
    closer_pair = (p1, q1) if dist1 <= dist2 else (p2, q2)

    Sy = [point for point in py if abs(point[0] - px[mid][0]) < delta]

    for i in range(len(Sy)):
        for j in range(i+1, min(i+15, len(Sy))):
            p, q = Sy[i], Sy[j]
            dst = distance(p, q)
            if dst < delta:
                closer_pair = p, q
                delta = dst

    return closer_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)
    Py = sorted(P, key=y_sort_key)
    return closest_pair_divide_conquer_rec(Px, Py)
```

```
#Function to print the results
def print_results(list_sizes, productive_times, brute_force_times):
    bruteforce_maximum_ratio = 0
    divideconquer_maximum_ratio = 0

# Finding the maximum ratios
    for list_size, bruteforce_empirical_RT, dc_emprical_RT in zip(list_sizes, brute_force_times, productive_times):
     bruteforce_time_complexity = list_size ** 2
    divideconquer_time_complexity = list_size * math.log2(list_size)

    bruteforce_ratio = bruteforce_empirical_RT / bruteforce_time_complexity
    divideconquer_ratio = dc_emprical_RT / divideconquer_time_complexity

    bruteforce_maximum_ratio = max(bruteforce_maximum_ratio, bruteforce_ratio)
    divideconquer_maximum_ratio = max(divideconquer_maximum_ratio, divideconquer_ratio)
     for i in range(len(list_sizes)):
            list_size = list_sizes[i]
            bruteforce_time_complexity = list_size ** 2
            divideconquer_time_complexity = list_size * math.log2(list_size)
            bruteforce_empirical_RT = brute_force_times[i]
            dc_emprical_RT = productive_times[i]
            bruteforce_ratio = bruteforce_empirical_RT / bruteforce_time_complexity
            divideconquer_ratio = dc_emprical_RT / divideconquer_time_complexity

 # calculate predicted RT's

            bf_predicted_run_time = bruteforce_maximum_ratio * bruteforce_time_complexity
            dc_predicted_run_time = divideconquer_maximum_ratio * divideconquer_time_complexity
            print({
               "n": list_size,
               "Theoretical RT n^2 Brute Force": bruteforce_time_complexity,
               "Emprical RT Brute Force": bruteforce_empirical_RT,
               "Predicted RT Brute Force": bf_predicted_run_time,
               "Ratio of Brute Force": bruteforce_ratio,
            })
            print({
               "n": list_size,
               "Theoretical RT n^2 Divide and Conquer": divideconquer_time_complexity,
               "Emprical RT Divide and Conquer": dc_emprical_RT,
               "Predicted RT Divide and Conquer": dc_predicted_run_time,
               "Ratio Divide and Conquer": divideconquer_ratio
            })
    return bruteforce_maximum_ratio, divideconquer_maximum_ratio


# Function  plot graphs comparing the empirical and predicted run times
def plot_graphs(input_sizes, empirical_running_times, brute_force_times, C1, C2):

    empirical_running_times = [i*1000 for i in empirical_running_times]
    brute_force_times = [i*1000 for i in brute_force_times]

    # Empirical Runningtime of Algo-1 and Empirical Runningtime of Algo-2
    plt.figure()
    plt.plot(input_sizes,empirical_running_times, label="Empirical RT of Divide-and-Conquer")
    plt.plot(input_sizes, brute_force_times, label="Empirical RT of Brute Force")
    plt.xlabel("Input size (n)")
    plt.ylabel("Runningtime in (milliseconds)")

    plt.title("Empirical RT of Divide-and-Conquer  vs. Empirical RT of Brute Force")
    plt.legend()
    plt.show()

    # Empirical Runningtime of Algo-1 and Predicted Runningtime of Algo-1
    plt.figure()
    plt.plot(input_sizes, brute_force_times, label="Empirical RT of Brute Force")
    plt.plot(input_sizes, [C1 * n ** 2*1000 for n in input_sizes], label="Predicted RT of Brute Force")
    plt.xlabel("Input size (n)")
    plt.ylabel("Runningtime in (milliseconds)")

    plt.title("Empirical RT of Brute Force vs. Predicted RT of Brute Force")
    plt.legend()
    # Empirical Runningtime of Algo-2 and Predicted Runningtime of Algo-2
    plt.figure()
    plt.plot(input_sizes, empirical_running_times, label="Empirical RT of Divide-and-Conquer")
    plt.plot(input_sizes, [C2 * n * math.log2(n)*1000 for n in input_sizes], label="Predicted RT of Divide-and-Conquer")
    plt.xlabel("Input size(n)")
```

```
    plt.ylabel("Runningtime in (milliseconds)")

    plt.title("Empirical RT of Divide-and-Conquer vs. Predicted RT of Divide-and-Conquer")
    plt.legend()

# Generate random points and measuring executions
if __name__ == "__main__":
    # Input sizes
    sizes = [i for i in range(100, 1001, 100)]
    bruteforce_times = []
    divideconquer_times = []
    number_of_running = 10

    for size in sizes:
        temp_bruteforce_times = []
        temp_divideconquer_times = []

        # Generate random points for the given size
        points = generate_random_points(size, 0, 1e9)

        # Measure execution time for brute force algorithm
        initial_time = time.time()
        for _ in range(number_of_running):
            closest_pair_brute_force(points)
        temp_bruteforce_times.append((time.time() - initial_time) / number_of_running)

        # Measure execution time for divide and conquer algorithm
        initial_time = time.time()
        for _ in range(number_of_running):
            closest_pair_divide_conquer(points)
        temp_divideconquer_times.append((time.time() - initial_time) / number_of_running)

        # Calculate the average execution time for each algorithm
        bruteforce_times.append(sum(temp_bruteforce_times))
        divideconquer_times.append(sum(temp_divideconquer_times))

    C1, C2 = print_results(sizes, divideconquer_times, bruteforce_times)
    plot_graphs(sizes, divideconquer_times, bruteforce_times, C1, C2)
```

Project Video Recording Link:

https://drive.google.com/file/d/11Jp6PjR6TNply7Ts5xAfQ3cM7uSmyQiD/view?usp=sharing

References:

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). The MIT Press. ISBN: 0262033844.

2. Kleinberg, J., & Tardos, É. (2006). Algorithm Design. Addison Wesley.