CLOSEST PAIR OF POINTS

Meghananjali Remala
Z23701551

Problem Statement:

Given n points P1, P2, …, Pn in a plane represented by their x and y coordinates. To find the closest pair of points in the list and calculate the distance between them using Brute-Force and Divide and Conquer algorithms. The distance between two points (x1, y1) and (x2, y2) is defined as the Euclidean distance.

Input Size:

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

Real World Applications:

- **Sensor Networks:** In sensor networks, where multiple sensors are deployed in an area to collect data, the closest pair of points can be used to determine the sensors that are physically closest to each other.

- **Robotics:** In robotics and autonomous vehicles, the closest pair of points can be used to calculate the distance between the robot and other objects in its environment. This is useful for obstacle avoidance and path planning.

Algorithms:

1. **Brute Force Algorithm:**

The brute force algorithm for the closest pair of points problem is a straightforward and intuitive approach to find the minimum distance between any pair of points in a given list. Given a list of 2D points represented by their x and y coordinates, the algorithm iterates through each point and compares it with all other points in the list to calculate the distance between them. It returns the points that have minimum distance.

Pseudo Code:

```
import math

def brute_force_closest_pair(points):
    closest_pair = None
    min_distance = float('inf')
    # Iterate over each point in the list
    for i, p1 in enumerate(points):
        # Compare with all other points in the list (excluding p1)
        for j in range(i + 1, len(points)):
            p2 = points[j]
            # Calculate the distance between p1 and p2
            distance = math.sqrt((p2[0] - p1[0])**2 + (p2[1] - p1[1])**2)

            # Update the closest pair and minimum distance if a smaller distance is found
            if distance < min_distance:
                min_distance = distance
                closest_pair = (p1, p2)

    return closest_pair, min_distance
```

Runtime Analysis:

The algorithm involves a nested loop structure, where for each point p1 in the list, it compares it with all other points p2 (excluding itself) using a second loop. This results in a total of n*(n-1) comparisons. However, as 'n' becomes significantly large, the -1 term becomes insignificant, and the time complexity can be approximated to O(n^2).

To be precise, the number of comparisons performed by the algorithm is given by the sum of the first (n-1) positive integers, which is (n*(n-1))/2. Asymptotically, this is equivalent to O(n^2) .

Therefore, the time complexity is O(n^2).

2. Divide And Conquer Algorithm:

The algorithm works by breaking the problem into smaller subproblems, solving them recursively, and then merging the results to find the overall solution. The algorithm proceeds to divide 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 two divided subproblems.

Next comes the merging step. The algorithm considers a "strip" region around the middle x-coordinate, with a width equal to the minimum distance found in the two subproblems. Within this strip, only points that fall within a certain range need to be considered, reducing the number of points to be examined.To simplify the merging process, the points in the strip are sorted based on their y-coordinates. The algorithm then iterates through the 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.

After all the recursive calls and merging processes are complete, the algorithm will have found the closest pair of points and their distance.

PseudoCode:

```
def closest_pair_divide_conquer_rec(p_x, p_y):
    ln_x = len(p_x)  # It's quicker to assign variable
    if ln_x <= 3:
        return brute_force(p_x)  # A call to bruteforce comparison
    mid = ln_x // 2  # Division without remainder, need int
    Qx = p_x[:mid]  # Two-part split
    Rx = p_x[mid:]

    # Determine midpoint on x-axis
    midpoint = p_x[mid][0]
    Qy = list(filter(lambda x: x[0] <= midpoint, p_y))  # filter left side
    Ry = list(filter(lambda x: x[0] > midpoint, p_y))  # filter right side

    # Call recursively both arrays after split
    (p1, q1, dist1) = closest_pair_divide_conquer_rec(Qx, Qy)
    (p2, q2, dist2) = closest_pair_divide_conquer_rec(Rx, Ry)

    # Determine smaller distance between points of 2 arrays
    if dist1 <= dist2:
        d = dist1
        mn = (p1, q1)
    else:
        d = dist2
        mn = (p2, q2)

    # Call function to account for points on the boundary
    (p3, q3, dist3) = closest_split_pair(p_x, p_y, d, mn)

   # Determine smallest distance for the array
```

```
if d <= dist3:
    return mn[0], mn[1], d
else:
    return p3, q3, dist3
```

Runtime Analysis:

1.The division step involves sorting the points, which takes O(n log n) time.

2.The conquer step involves recursively solving two smaller problems of size n/2, which, based on the recurrence relation T(n) = 2T(n/2) + O(n), gives us a time complexity of O(n log n) in the recursive case.

3.The combine step involves finding the closest split pair, which can be done in linear time O(n). By summing up these steps, 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:

T(n) = 2T(n/2) + O(n)

By comparing to the master's 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. So by the Master Theorem, the time complexity of the algorithm is Θ(n^c * log^(k+1)(n)) = Θ(n * log(n)), i.e., 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

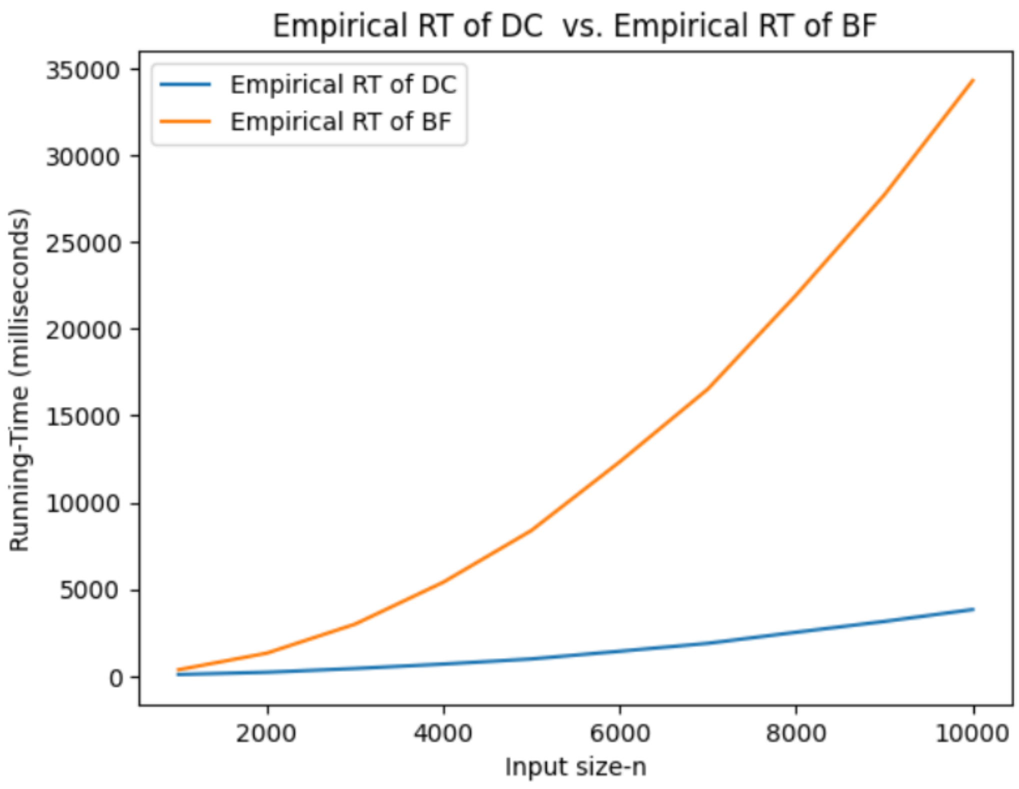
Table for Brute Force Algorithm:

Input Size	Theoretical TC using BF	Empirical RT of algo using BF	Ratio of algo using BF	Predicted RT for BF
1000	1000000	0.370304656	3.70E-07	0.370304656
2000	4000000	1.325185466	3.31E-07	1.481218624
3000	9000000	2.994357538	3.33E-07	3.332741904
4000	16000000	5.406629539	3.38E-07	5.924874496
5000	25000000	8.389118457	3.36E-07	9.257616401
6000	36000000	12.33767381	3.43E-07	13.33096762
7000	49000000	16.53028209	3.37E-07	18.14492815
8000	64000000	21.93804421	3.43E-07	23.69949799
9000	81000000	27.70742769	3.42E-07	29.99467714
10000	100000000	34.2800601	3.43E-07	37.0304656

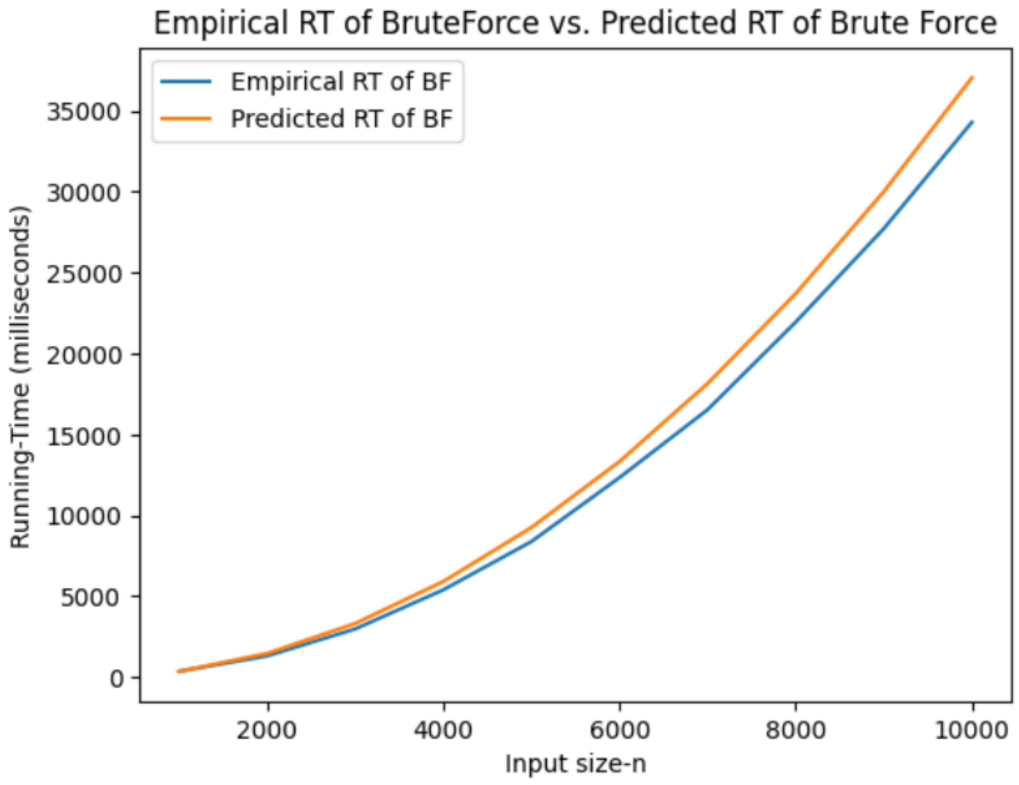
Table for Divide And Conquer Algorithm:

Input Size	Theoretical TC using DC	Empirical RT of algo using DC	Ratio of algo using DC	Predicted RT for DC
1000	9965.784285	0.097530222	9.79E-06	0.287603579
2000	21931.56857	0.225239849	1.03E-05	0.632925361
3000	34652.24036	0.440807343	1.27E-05	1.000032518
4000	47863.13714	0.696921706	1.46E-05	1.381287128
5000	61438.5619	0.991699743	1.61E-05	1.773061687
6000	75304.48071	1.434298325	1.90E-05	2.173219644
7000	89411.97445	1.891089439	2.12E-05	2.580349237
8000	103726.2743	2.529218078	2.44E-05	2.993447067
9000	118221.3836	3.15141046	2.67E-05	3.411762896
10000	132877.1238	3.834714389	2.89E-05	3.834714389

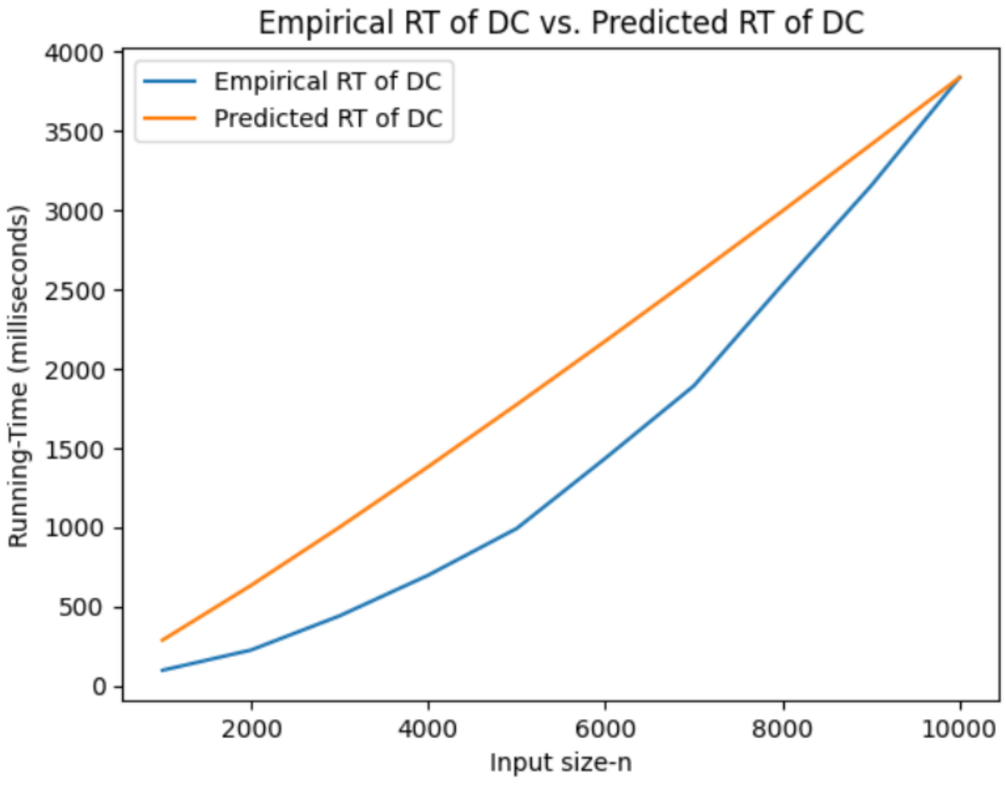
Graph-1:



Graph-2:



Graph-3:



Conclusion:

The Closest Pair of Points problem can be solved using either the Brute Force method or the Divide and Conquer approach, each with its distinct pros and cons. The Brute Force method, characterized by its simplicity, involves a pairwise comparison of each point with all others, which leads to a quadratic time complexity of O(n^2). While easy to implement, this method becomes increasingly inefficient as the number of points, n, increases.

In contrast, the Divide and Conquer approach, although more complex to implement, offers a significant improvement in efficiency. By breaking the problem down into smaller subsets, this method reduces the time complexity to O(n log n), making it a more efficient choice for larger datasets. However, the Divide and Conquer method does require additional space (O(n)) to store sorted points and make recursive function calls, unlike the Brute Force method, which only requires constant space (O(1)).

Therefore, when choosing a method, one must consider the size of the dataset, the available computational resources, and the ease of implementation. For smaller datasets, the simplicity of the Brute Force method may be beneficial, but for larger datasets, the efficiency of the Divide and Conquer approach would likely outweigh the additional complexity.

Programming Language:

Python programming Language is used.

Google Collab:
https://colab.research.google.com/drive/1LnzJE2L3oY7JLzDamixtVEDsn3Tm8x79?usp=sharin
g

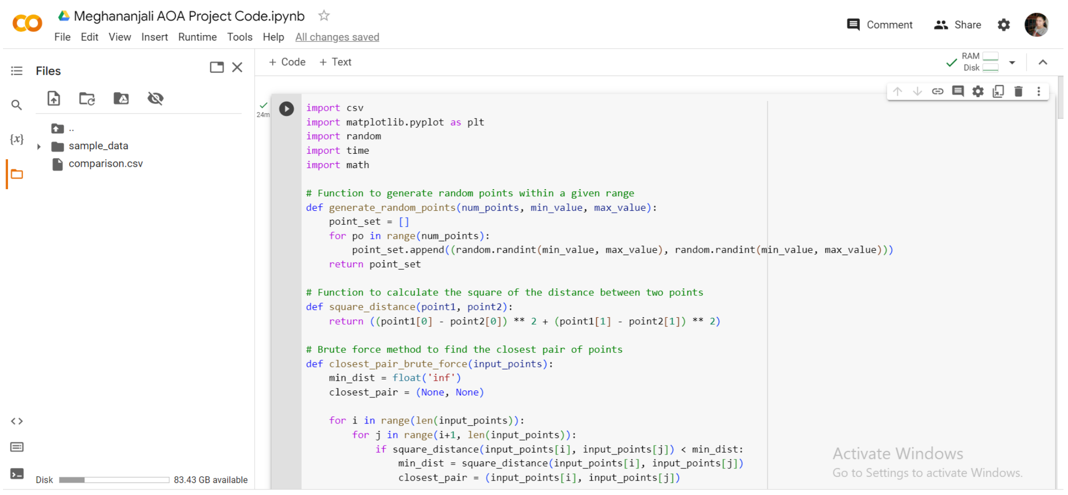
Source Code:

```
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]
```

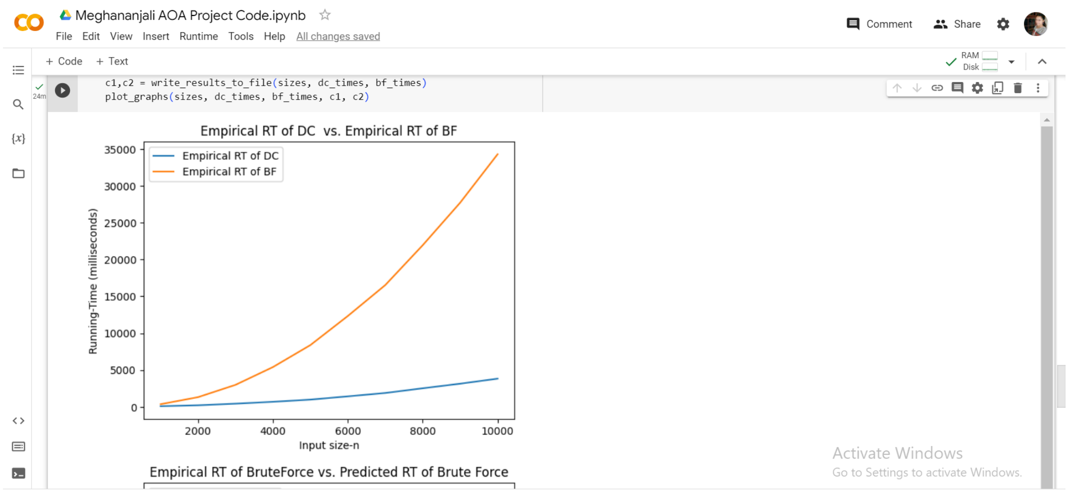
```
def y_sort_key(point):
    return point[1]

# 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)
```

Snapshots of Google Collab:



Meghananjali AOA Project Code.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
Files
sample_data
comparison.csv
+ Code + Text
RAM
Disk
Disk 83.43 GB available
Activate Windows
Go to Settings to activate Windows.

```
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])
```



Meghananjali AOA Project Code.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
RAM
Disk
Empirical RT of DC vs. Empirical RT of BF
Empirical RT of DC
Empirical RT of BF
Running-Time (milliseconds)
35000
30000
25000
20000
15000
10000
5000
0
2000
4000
6000
8000
10000
Input size-n
Empirical RT of BruteForce vs. Predicted RT of Brute Force
Activate Windows
Go to Settings to activate Windows.

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

Project Demo:

https://drive.google.com/file/d/1rjpmlcNJXVbj0XdnDUh08f1lyFYQXpFI/view?usp=sharing

(or)

https://clipchamp.com/watch/1jM9pGTrn7O

References:

1. Introduction to Algorithms, 3rd edition, by T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, The MIT Press, 2009, ISBN: 0262033844.
2. Algorithm Design, J. Kleinberg and E. Tardos, Addison Wesley, 2006.