Project Title: Closest Pair of Points: Algorithm Analysis and Implementation

Student Name: Shreyansh Pravinkumar Sharma

1. **Problem Definition:** The **Closest Pair of Points** problem is a fundamental problem in computational geometry. Given a set of points in a two-dimensional plane, the objective is to efficiently determine the pair of points that are closest to each other in terms of Euclidean distance.

   Input: A set(array) of points = { $_0$, $_1$, . . . , P_{n-1}} of 'n' distinct points in the 2D plane, where each point P_i has coordinates (x_i , y_i).
   Output: Indices i and j such that the Euclidian distance between Pi and Pj is the smallest among all the pair of the points,

   To analyze the empirical performance of the algorithms, the following input sizes are considered:

   - n = 10,000; 20,000; … ; 100,000
   - For each input size, the experiment is repeated 10 times with different randomly generated and distinct point sets to ensure statistical reliability.

Following are few Real-World Applications that can utilized this problem:

- **Geographical Information Systems (GIS):** Finding the closest pair of cities or landmarks.
- **Astronomy:** Identifying stars or celestial objects that are closest together in space.
- **Collision Detection in Games/Simulations:** Determining the nearest entities that may interact.
- **Clustering Algorithms:** As a preprocessing step for hierarchical clustering.
- **Networking:** Optimizing routes or connections between physically closest servers or nodes.
- **Machine Learning:** Used in nearest neighbor classification and clustering techniques.

2. Algorithms and Running Time:

Algorithm 1: **Brute Force Approach**

Pseudocode:

```
BRUTE-FORCE-CLOSEST-PAIR(P)
// P is a list of n points, n ≥ 2, P1 = (x1, y1),..., Pn = (xn, yn)
  n = |P|   //number of points
  minDist = ∞
  for i = 1 to n-1
    for j = i+1 to n
      dist = DISTANCE(P[i], P[j])  //Euclidian distance
      if dist < minDist
        minDist = dist
        index1 = i; index2 = j
  return index1, index2, minDist
```

Running Time Analysis: Brute Force Algorithm

- The brute-force algorithm checks all possible pairs of points.
- Number of pairs in a set of 'n' points: $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = (n^2 - n)/2$
- For each pair, it calculates the Euclidean distance.
- **Time complexity: $O(n^2)$**

Algorithm 2: **Divide and Conquer Approach**

Pseudocode:

```
Algorithm Closest-Pair(P)
        // Returns the closest pair of points in P
        // Sort points by x and y coordinates
        construct Px and Py      // TC: O(n log n)
        return Closest-Pair-Rec (Px, Py)  // Call the recursive function


Algorithm Closest-Pair-Rec(Px, Py)   // Returns the closest pair of points
        // Px is the set of points sorted by x-coordinate
        // Py is the set of points sorted by y-coordinate
```

```
// Base case
if |P| ≤ 3 then
        find the closest pair by measuring all pairwise distances
        return closest pair

// Divide step
construct Qx, Qy, Rx, Ry     // O(n)

// Conquer step (recursive calls)
(q0*, q1*) = Closest-Pair-Rec(Qx, Qy)
(r0*, r1*) = Closest-Pair-Rec(Rx, Ry)

// Combine step
δ = min(d(q0*, q1*), d(r0*, r1*))
x* = maximum x-coordinate of a point in set Q
L = {(x, y): x = x*}

// Find points in the strip
S = points in P within distance δ of L
construct Sy (S sorted by y-coordinate)          // O(n) time

// Check for closer pairs in the strip
for each point s ∈ Sy do                    // O(n)
        compute the distance from s to each of the next 15 points in Sy

let s, s' be the pair with the minimum distance

// Return the closest pair found
if d(s, s') < δ then
        return (s, s')
else if d(q0*, q1*) < d(r0*, r1*) then
        return (q0*, q1*)
else
        return (r0*, r1*)
```

Running Time Analysis: Divide and conquer Approach

- Sorting takes $O(n \log n)$ time.
- Dividing points into two halves: O(n)
- Each recursive call divides the problem in half, and the merge step processes only a linear number of points (strip).
- Two recursive calls on sets of size n/2: 2T(n/2)
- The comparison in the strip is optimized by comparing at most 15 points per point.
- Recurrence relation: T(n) = 2T(n/2) + O(n)
- **Time Complexity: O (n log$n$)**
- Much faster and scalable than brute-force for large datasets.

3. Experimental Results:

Tables

➢ **Computing constant c1 for Algorithm 1: Brute force Approach**

Table Alg-1:

n	Theoretical_RT n^2	Empirical_RT (ms)	Ratio = (Empirical_RT)/(Theoretical_RT)	Predicted_RT = c1 x Theoretical_RT
10,000	10^8	33.14	$r_1 = 3.314 \times 10^{-7}$	33.14
20,000	4×10^8	129.57	$r_2 = 3.239 \times 10^{-7}$	132.56
30,000	9×10^8	293.08	$r_3 = 3.256 \times 10^{-7}$	298.26
40,000	16×10^8	528.21	$r_4 = 3.301 \times 10^{-7}$	530.24
50,000	25×10^8	818.52	$r_5 = 3.274 \times 10^{-7}$	828.50
60,000	36×10^8	1187.52	$r_6 = 3.299 \times 10^{-7}$	1193.04
70,000	49×10^8	1600.39	$r_7 = 3.266 \times 10^{-7}$	1623.86
80,000	64×10^8	2102.44	$r_8 = 3.285 \times 10^{-7}$	2120.96
90,000	81×10^8	2661.19	$r_9 = 3.285 \times 10^{-7}$	2684.33
100,000	100×10^8	3288.03	$r_{10} = 3.288 \times 10^{-7}$	3313.99

(values in the table are rounded off to 2-3 decimal places)

Computing c1:

c1 = max (r1, r2, …., r10)
c1 = max (3.314×10^{-7}, 3.239×10^{-7}, 3.256×10^{-7}, … , 3.288×10^{-7})

Therefore, **$c1 = 3.314 \times 10^{-7}$**

- **Computing c2 for Algorithm 2: Divide and Conquer Approach**

Table Alg-2:

n	Theoretical_RT (n log *n*)	Empirical_RT (ms)	Ratio = (Empirical_RT)/(Theoretical_RT)	Predicted_RT = c1 x Theoretical_RT
10,000	1.33×10^5	6.27	4.72×10^{-5}	6.277
20,000	2.86×10^5	12.86	4.50×10^{-5}	13.4992
30,000	4.46×10^5	18.50	4.15×10^{-5}	21.0512
40,000	6.12×10^5	26.47	4.33×10^{-5}	28.8864
50,000	7.80×10^5	30.27	3.88×10^{-5}	36.816
60,000	9.52×10^5	38.93	4.09×10^{-5}	44.9344
70,000	1.13×10^6	48.59	4.31×10^{-5}	53.336
80,000	1.30×10^6	55.29	4.24×10^{-5}	61.36
90,000	1.48×10^6	58.29	3.94×10^{-5}	69.856
100,000	1.66×10^6	63.04	3.80×10^{-5}	78.352

Calculating c2:

C2 = max(r1, r2, r3, …, r10)

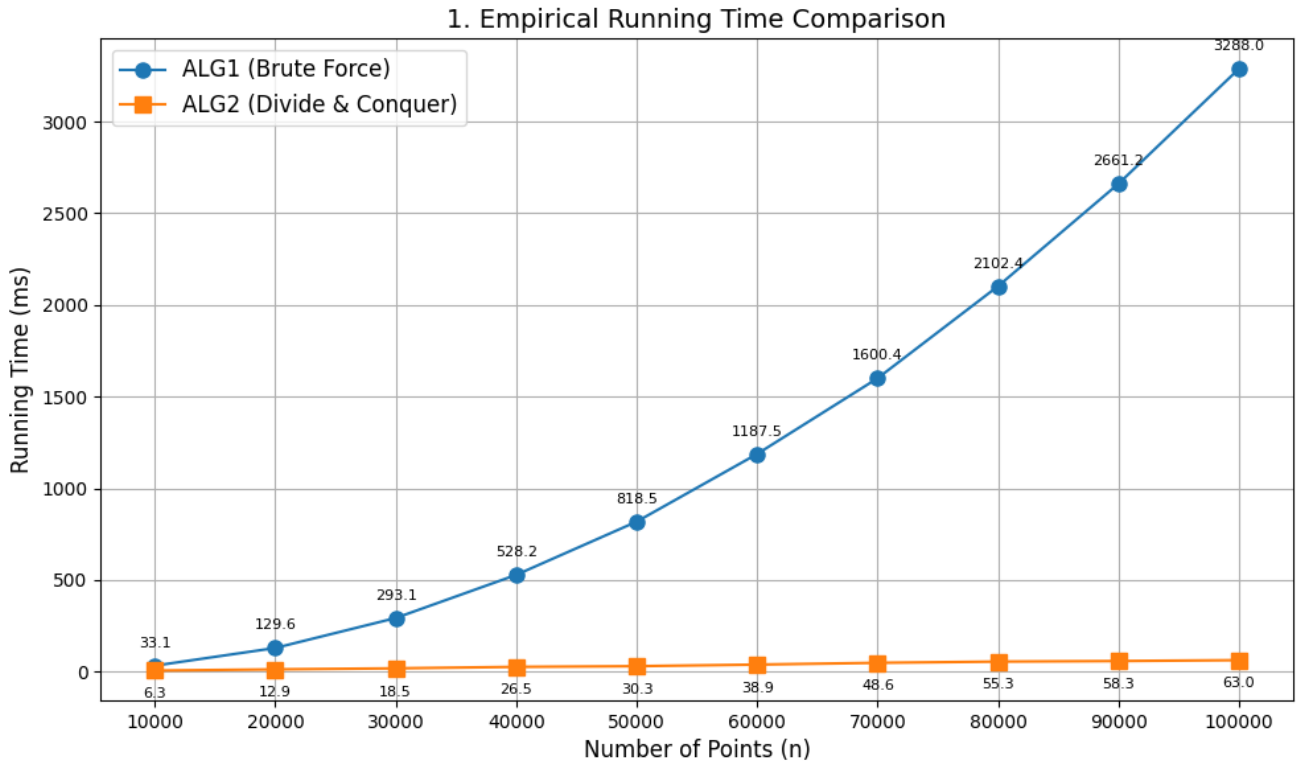
C2 = max(4.72×10^{-5}, 4.50×10^{-5}, …, 3.80×10^{-5})

Therefore, **$c2 = 4.72 \times 10^{-5}$**

Graphs

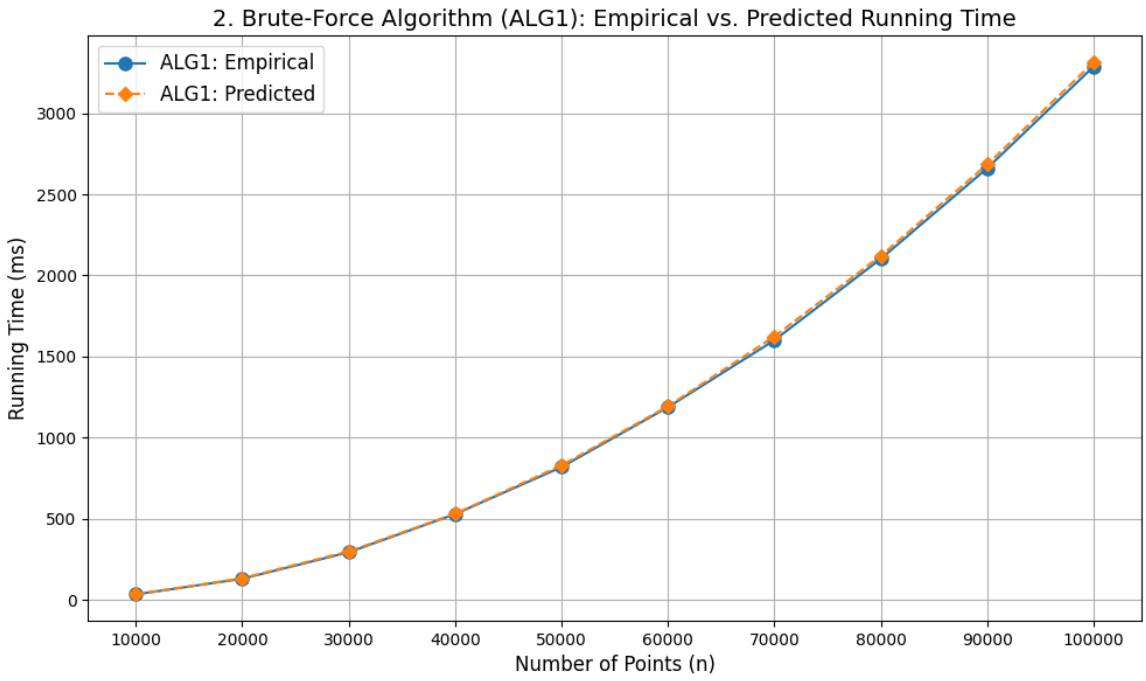
1. **Empirical Running Time Comparison**

- Plots average empirical running time of each algorithm against all input values.

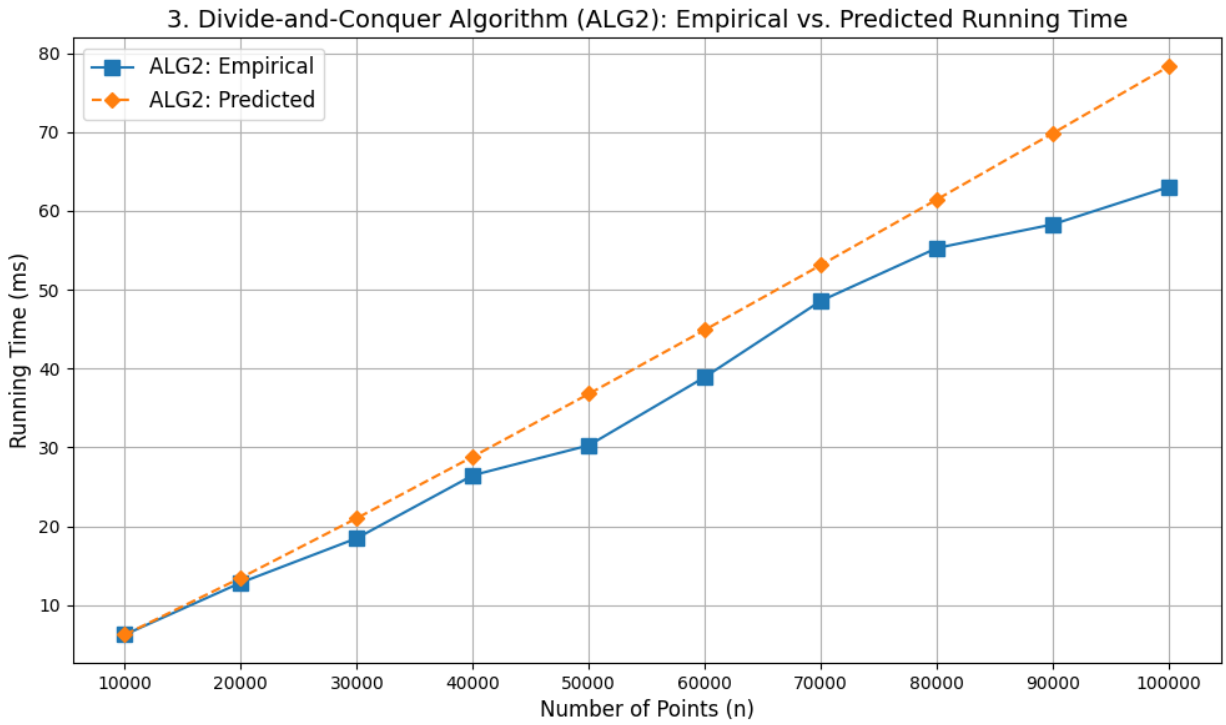


2. Brute-Force Algorithm (ALG1): Empirical vs. Predicted Running Time

- Plotting Empirical running time and actual running time of Brute force approach against input values.



3. Divide-and-Conquer Algorithm (ALG2): Empirical vs. Predicted Running Time



4. **Conclusions**:

- **Theoretical vs. Empirical Consistency**

The experimental data demonstrates strong consistency between theoretical predictions and empirical measurements for both algorithms. For the brute force algorithm (ALG1), which has a theoretical complexity of $O(n^2)$, the predicted and empirical running times align remarkably well across all input sizes (10,000 to 100,000 points), as evidenced by the nearly overlapping lines in the second graph. The constant factor $c_1 = 3.314 \times 10^{-7}$ accurately scales the theoretical values to match observed performance.

Similarly, for the divide-and-conquer algorithm (ALG2) with theoretical complexity of $O(n \log n)$, the predicted values generally follow the same growth pattern as the empirical measurements, although with slightly more deviation than ALG1. The constant factor $c_2 = 4.72 \times 10^{-5}$ provides a reasonable approximation of actual performance, with some overestimation at larger input sizes.

- **Performance Comparison**

The first graph clearly illustrates the dramatic performance difference between the two algorithms. At n=100,000, the brute force approach requires approximately 3288 ms, while the divide-and-conquer method completes in just 63 ms a speedup factor of about 52x. This performance gap widens as input size increases, confirming the theoretical advantage of $O(n \log n)$ over $O(n^2)$.

- **Scalability Analysis**

The empirical results validate the theoretical scalability predictions. The brute force algorithm shows a quadratic growth pattern, making it impractical for large datasets. In contrast, the divide-and-conquer approach scales much more efficiently, with a nearly linear appearance on the graph despite its n log n complexity. At n=10,000, both algorithms perform relatively efficiently, but by n=50,000, the brute force approach already requires about 27 times more processing time.

5. **Project Demo**:

Video Link: https://youtu.be/UdLw-7ThyZY

Source Code Link: https://github.com/Shrey-ansh10/closest-pair-of-points

(Implementation of both algorithms is in ClosestPair.cpp)

6. **Reference**:

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. *Algorithm Design*, J. Kleinberg and E. Tardos, Addison-Wesley Publishing Company, 2006.
3. de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). Computational Geometry: Algorithms and Applications. Springer-Verlag.
4. Preparata, F. P., & Shamos, M. I. (1985). Computational Geometry: An Introduction. Springer-Verlag.
5. Bentley, J. L., & Shamos, M. I. (1976). Divide-and-conquer in multidimensional space. Proceedings of the Eighth Annual ACM Symposium on Theory of Computing, 220-230.
6. Lecture Slides.