**Problem Definition**

**Problem Name:** Closest Pair of Points

**Problem Statement:**  
Given n distinct points P1,P2,…,Pn in a 2D plane, find the pair of points (Pi,Pj) with the minimum Euclidean distance between them.

**Input:**  
A set of n points where each point Pi=(xi,yi) with integer coordinates.

**Output:**  
Indices i,j such that distance between Pi and Pj is minimized.

**Real-world Applications:**

* Pattern recognition
* Geographic information systems (GIS)
* Robotics and navigation
* Clustering and data mining

**Algorithms & Running Time Analysis**

**Algorithm 1:** Brute-Force (ALG1)

BruteForceClosestPoints(P)

// P is a list of n points, n ≥ 2, P1 = (x1, y1),…, Pn = (xn, yn)

// Returns the index1 and index2 of the closest pair of points

dmin = ∞

for i = 1 to n - 1

for j = i + 1 to n

d = sqrt((xi - xj)^2 + (yi - yj)^2)

if d < dmin

dmin = d

index1 = i

index2 = j

return index1, index2

**Time Complexity:**

* The outer loop runs n−1 times
* The inner loop runs from i+1 to n, giving roughly n(n−1)/2 iterations

**Worst-case runtime:** Θ(n2)

**Best-case runtime:** Also Θ(n2) - always checks all pairs

**Algorithm 2:** Divide-and-Conquer (ALG2)

DivideAndConquerClosestPoints(P)

// Input: P is a list of n points in 2D

// Returns the closest pair of points

1. Sort the points P by x-coordinate → Px

2. Sort the points P by y-coordinate → Py

3. return ClosestPair(Px, Py)

Function ClosestPair(Px, Py)

if |Px| ≤ 3 then

return BruteForceClosestPoints(Px)

Let Qx and Rx be the left and right halves of Px

Let midpoint = Px[n/2].x

Let Qy = points in Py with x ≤ midpoint

Let Ry = points in Py with x > midpoint

(p1, q1) = ClosestPair(Qx, Qy)

(p2, q2) = ClosestPair(Rx, Ry)

δ = min(distance(p1, q1), distance(p2, q2))

(p3, q3) = ClosestSplitPair(Px, Py, δ)

return the pair among (p1,q1), (p2,q2), (p3,q3) with the smallest distance

Function ClosestSplitPair(Px, Py, δ)

Let Sy = points within δ of the midpoint line (sorted by y)

for i = 1 to length(Sy)

for j = i+1 to i+7

if j ≤ length(Sy)

compute distance and update minimum if needed

return closest split pair

**Time Complexity:**

Let T(n) be the time to compute the closest pair for n points.

* Dividing the points into two halves: O(n)
* Two recursive calls: 2T(n/2)
* Merging and split pair check: O(n)

**Worst-case runtime:** Θ(nlogn)   
**Best-case runtime:** Still Θ(nlogn) - since it always recurses

**Experimental Results**

**Table ALG1 –** Computing constant c₁ for Brute Force

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | TheoreticalRT n² | EmpiricalRT (msec) | Ratio | Predicted RT |
| 1×10⁴ | **1×10⁸** | **63.52** | **63.52×10⁻⁸** | **63.52** |
| 2×10⁴ | **4×10⁸** | **227.12** | **56.78×10⁻⁸** | **254.08** |
| 3×10⁴ | **9×10⁸** | **500.08** | **55.56×10⁻⁸** | **571.69** |
| 4×10⁴ | **16×10⁸** | **900.15** | **56.26×10⁻⁸** | **1016.33** |
| 5×10⁴ | **25×10⁸** | **1380.44** | **55.22×10⁻⁸** | **1588.02** |
| 6×10⁴ | **36×10⁸** | **2012.45** | **55.90×10⁻⁸** | **2286.75** |
| 7×10⁴ | **49×10⁸** | **2731.82** | **55.75×10⁻⁸** | **3112.52** |
| 8×10⁴ | **64×10⁸** | **3579.47** | **55.93×10⁻⁸** | **4065.33** |
| 9×10⁴ | **81×10⁸** | **4566.79** | **56.38×10⁻⁸** | **5145.19** |
| 10×10⁴ | **100×10⁸** | **5670.44** | **56.70×10⁻⁸** | **6352.08** |

**Table ALG2 –** Computing constant c₂ for Divide and Conquer

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | TheoreticalRT n·log₂(n) | EmpiricalRT (msec) | Ratio | Predicted RT |
| 1×10⁴ | **132877** | **8.31** | **6.25×10⁻⁵** | **8.31** |
| 2×10⁴ | **285754** | **9.60** | **3.36×10⁻⁵** | **17.87** |
| 3×10⁴ | **446180** | **14.53** | **3.26×10⁻⁵** | **27.90** |
| 4×10⁴ | **611508** | **19.15** | **3.13×10⁻⁵** | **38.24** |
| 5×10⁴ | **780482** | **25.48** | **3.26×10⁻⁵** | **48.80** |
| 6×10⁴ | **952360** | **30.04** | **3.15×10⁻⁵** | **59.55** |
| 7×10⁴ | **1126655** | **35.96** | **3.19×10⁻⁵** | **70.45** |
| 8×10⁴ | **1303017** | **42.91** | **3.29×10⁻⁵** | **81.48** |
| 9×10⁴ | **1481187** | **50.07** | **3.38×10⁻⁵** | **92.62** |
| 10×10⁴ | **1660964** | **55.23** | **3.33×10⁻⁵** | **103.86** |

**A graph with orange line and black text

AI-generated content may be incorrect.**

**A graph with orange lines and numbers

AI-generated content may be incorrect.**

**A graph of a number of points

AI-generated content may be incorrect.**

**Conclusions**

The experimental results strongly support the theoretical analysis of the two algorithms:

**ALG1 – Brute-Force Approach (Θ(n²)):**

* This algorithm performs an exhaustive pairwise comparison of all points, resulting in a quadratic growth in runtime.
* The empirical results confirmed this: as n doubled, the runtime roughly quadrupled.
* It becomes inefficient for larger inputs (e.g., n > 30,000), where runtimes exceeded several seconds.
* This aligns precisely with the expected theoretical performance of Θ(n²).

**ALG2 – Divide-and-Conquer Approach (Θ(n log n)):**

* This algorithm leverages recursive spatial partitioning and only compares relevant candidate pairs near the partition boundary.
* Empirical results show a much more scalable growth pattern, consistent with the logarithmic multiplier.
* ALG2 significantly outperforms ALG1 for all values of n, particularly when n exceeds 20,000.
* The predicted runtime curve based on theoretical complexity closely follows the actual measurements, confirming its asymptotic efficiency.

**Constants and Variability:**

* Hidden constants c₁ and c₂ were extracted from the ratio of empirical to theoretical runtime.
* Minor fluctuations were observed due to system noise and Java execution overhead, but the trends were stable.
* Graphs show a consistent and expected gap between the performance of ALG1 and ALG2.

**Final Verdict:**

* For small n, both algorithms are feasible, though ALG1 is simpler to implement.
* For moderate to large n, ALG2 is clearly preferable due to its significantly faster execution time and better scalability.
* This experiment validates both theoretical complexity analysis and practical performance expectations.

**References**

1. Kleinberg, J., & Tardos, É. (2006). *Algorithm Design*. Pearson.
2. Course Notes & Lectures – COT 6405: Analysis of Algorithms.
3. ProgrammingProject.pdf and ProgrammingProject\_Additional\_Explanations.pdf.
4. Java Platform Standard Edition Documentation. Oracle Inc.