**18CSC204J-Design and analysis of Algorithm**

**AIR TRAFFIC CONTROL**

A MINI PROJECT REPORT

*Submitted by*

LAKSHAY VIJAY [ RA2011003011157]

DISHITA SIBAL [ RA2011003011162]

ZAYD HASSAN [RA2011003011178]

*Under the guidance of*

Ms. M. REVATHI

(Assistant Professor, Department of Computer Science & Engineering)

*in partial fulfilment for the award of the degree*

*of*

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY



S.R.M. Nagar, Kattankulathur, Kancheepuram District

SRM UNIVERSITY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this project report titled “AIR TRAFFIC CONTROL ” is the bonafide work of “ LAKSHAY VIJAY[RA2011003011157] DISHITA SIBAL [ RA2011003011162] and ZAYD HASSAN [RA2011003011178] ”, who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE SIGNATURE

Ms. M. REVATHI Dr M.PUSHPLATHA HEAD OF THE GUIDE HEAD OF THE DEPARTMENT Assistant Professor Dept. of Computer Science& Engineering Dept. of Computer Science & Engineering

Signature of the Internal Examiner Signature of the External Examiner

# TABLE OF CONTENTS

1. Abstract 1
2. Problem Statement 2
3. Closest pair examples 3
4. Algorithm 4
5. Working 6
6. Pseudocode 9
7. Time complexity 11
8. Program[code] 12
9. Output 17
10. Conclusion 18
11. Reference 19

***Abstract:***

*The* ***closest pair of points problem*** *or* ***closest pair problem*** *is a problem of computational geometry: given n {\displaystyle n} points in metric space, find a pair of points with the smallest distance between them. The closest pair problem for points in the Euclidean plane was among the first geometric problems that were treated at the origins of the systematic study of the computational complexity of geometric algorithms.*

*An algorithm with optimal time complexity O (n log n) for solving the Closest-Pair problem in the planar case appeared for the first time in 1975, in a computational geometry classic paper by* ***Ian Shamos****. This algorithm was based on the*

*Voronoi polygons. We used a divide-and-conquer approach which we generalized from one-dimension in order to solve the problem. Here we will discuss another O(nlogn) that attacks the problem in a different way. In the course of solving the duplicate-grouping problem, we describe a new universal class of hash functions of independent interest. It is shown that* ***both of the foregoing problems can be solved by randomized algorithms that use O(n) space and finish in O(n) time with probability tending to 1 as n grows to infinity****.*

***Topic:Air–Traffic control***

***Problem Statement:***

*With the increase number of low altitude aircrafts used in deliveries, it became necessary to have a management system that puts the downsides such as the increase number of collisions, energy waste and delivery costs into consideration. In air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision.*

**

***The closest pair 2 other examples-***

* ***Testing -*** *we have a set of points which are e.g. some laboratories owned by some company; the company has built a teleporter; it can be reconstructed only in these labs because e.g. their walls were built with gold and it's necessary for a teleporter to be encapsulated in gold room for some reason; but a teleporter produces a magnetic field and if they build 2 teleporters in one faculty the fields will interfere and they won't work; and ofc they want to test them and it would be good to first test them for as small distance as possible; so they have to find two labs which are closest to each other*
* ***Genetics*** *- points are people; we want to find two people the has the closest DNA (it doesn't concert us how to calculate the distance between two people); why? because e.g., they want to find out how close we can be to each other and what is the least number of nucleobases we have to change in a person to make him sufficiently different*

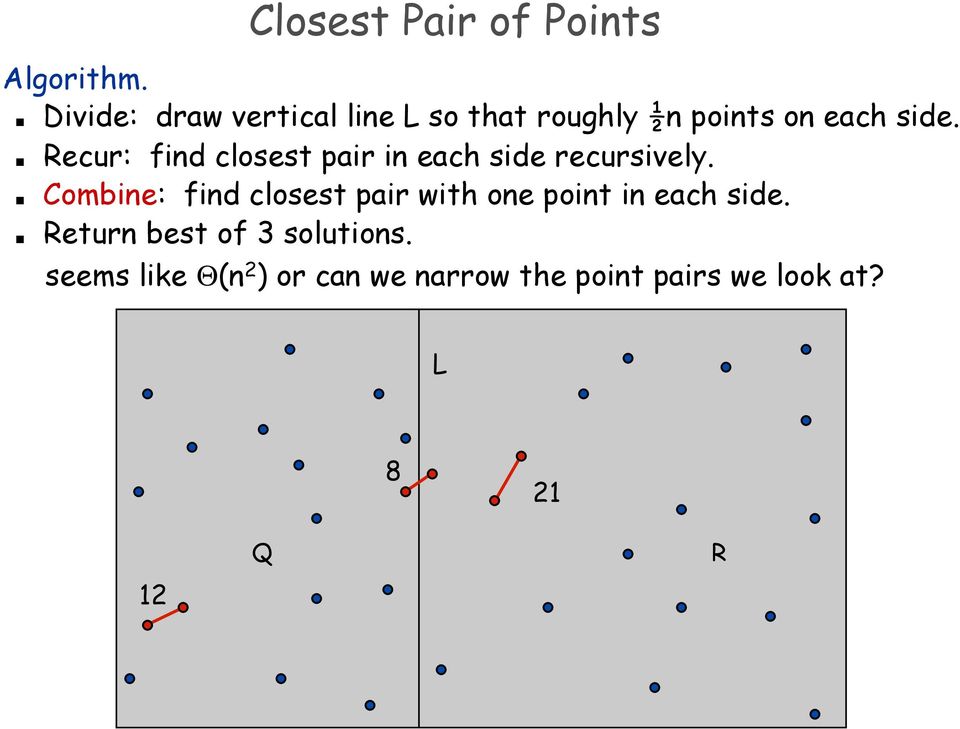
***Algorithm:***

***Merge Sort Method-Finding Closest Pair:***

*We sort the arrays with an algorithm called merge-sort, which is faster than brute-force sorting algorithms. The merge-sort algorithm splits the array, sorts the subarrays (as a recursive step), compares the youngest numbers in two subarrays and picks the younger, and repeats it until both subarrays are exhausted. Each of the recursive step’s costs just*Θ(n) *so that the total cost of the algorithm stays at*Θ(nlogn).

*def* ***merge\_sort****(array, coordinate=0):  
   
 length = len(array) if length == 1:  
 return array if length == 2:  
 if array[0][coordinate] > array[1][coordinate]:  
 return np.array([array[1], array[0]])  
 else:  
 return array  
   
 elif length > 2:  
 array\_l = array[:length//2]  
 array\_r = array[length//2:]  
 array\_l\_sorted =* ***merge\_sort****(array\_l, coordinate)  
 array\_r\_sorted =* ***merge\_sort****(array\_r, coordinate)  
   
 l\_length = len(array\_l)  
 r\_length = len(array\_r) l = 0  
 r = 0  
   
 sorted\_list = []  
   
 for i in range(length):  
 if r == r\_length:  
 sorted\_list.append(array\_l\_sorted[l])  
 l += 1 elif l == l\_length:  
 sorted\_list.append(array\_r\_sorted[r])  
 r += 1   
   
 elif array\_l\_sorted[l][coordinate] > array\_r\_sorted[r][coordinate]:  
 sorted\_list.append(array\_r\_sorted[r])  
 r += 1  
   
 elif array\_l\_sorted[l][coordinate] < array\_r\_sorted[r][coordinate]:  
 sorted\_list.append(array\_l\_sorted[l])  
 l += 1  
   
 return np.array(sorted\_list)*

***WORKING:***



Chart, scatter chart

Description automatically generated

*In the example shown in the right-hand side, the closest pair****within****subarrays is determined in the right subarray (note that the point on the dashed line belongs to the left subarray) and its distance is*d*.*

*If the closest pair exists across the left and right subarrays, the points should be within the range of*d*from the dashed line dividing the array into the two subarrays. Therefore, we can look at the subset within the shaded range.*

*Second, we sort the subset we obtained in the previous step by Y coordinate. We show that we must look at sets of only eight consecutive points each in the sorted subset.*

*Chart, scatter chart

Description automatically generated*

*As shown in the figure, the maximum number of points that can exist in*2d\**d rectangle across right and left subarrays is eight (points on the dashed line duplicate; two belong to the left subarray and another two are in the right).*

*Third, we look at each set of eight consecutive points in the subset sorted on the Y coordinate. If we find a pair whose distance is less than*d*, it means the closest pair exists across the subarrays.*

*This step costs*O(n)*, and the total cost of this recursive algorithm stays at*O(nlgn).

***Pseudocode:***

***minDist = infinity***

***for each p in P:***

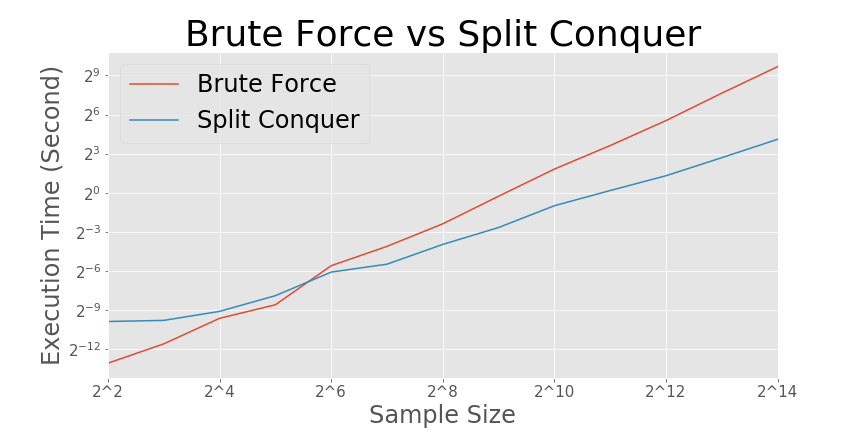
***for each q in P:***

***if p ≠ q and dist(p,q)< minDist:***

***minDist = dist(p,q)***

***closestPair = (p,q)***

***return closestPair***



***TIME COMPLEXITY:***

***The time complexity of this algorithm will be O (n log n).***

***Program:***

*#include <iostream>*

*#include <float.h>*

*#include <stdlib.h>*

*#include <math.h>*

***using******namespace*** *std;*

*// A structure to represent a Point in 2D plane*

***struct*** *Point*

*{*

***int*** *x, y;*

*};*

*// Needed to sort array of points according to X coordinate*

***int*** *compareX(****const******void****\* a,* ***const******void****\* b)*

*{*

*Point \*p1 = (Point \*)a,  \*p2 = (Point \*)b;*

***return*** *(p1->x != p2->x) ? (p1->x - p2->x) : (p1->y - p2->y);*

*}*

*// Needed to sort array of points according to Y coordinate*

***int*** *compareY(****const******void****\* a,* ***const******void****\* b)*

*{*

*Point \*p1 = (Point \*)a,   \*p2 = (Point \*)b;*

***return*** *(p1->y != p2->y) ? (p1->y - p2->y) : (p1->x - p2->x);*

*}*

*// A utility function to find the distance between two points*

***float*** *dist(Point p1, Point p2)*

*{*

***return******sqrt****( (p1.x - p2.x)\*(p1.x - p2.x) +*

*(p1.y - p2.y)\*(p1.y - p2.y)*

*);*

*}*

*// A Brute Force method to return the smallest distance between two points*

*// in P[] of size n*

***float*** *bruteForce(Point P[],* ***int*** *n)*

*{*

***float*** *min = FLT\_MAX;*

***for*** *(****int*** *i = 0; i < n; ++i)*

***for*** *(****int*** *j = i+1; j < n; ++j)*

***if*** *(dist(P[i], P[j]) < min)*

*min = dist(P[i], P[j]);*

***return*** *min;*

*}*

*// A utility function to find a minimum of two float values*

***float*** *min(****float*** *x,* ***float*** *y)*

*{*

***return*** *(x < y)? x : y;*

*}*

*// A utility function to find the distance between the closest points of*

*// strip of a given size. All points in strip[] are sorted according to*

*// y coordinate. They all have an upper bound on minimum distance as d.*

*// Note that this method seems to be a O(n^2) method, but it's a O(n)*

*// method as the inner loop runs at most 6 times*

***float*** *stripClosest(Point strip[],* ***int*** *size,* ***float*** *d)*

*{*

***float*** *min = d;  // Initialize the minimum distance as d*

*// Pick all points one by one and try the next points till the difference*

*// between y coordinates is smaller than d.*

*// This is a proven fact that this loop runs at most 6 times*

***for*** *(****int*** *i = 0; i < size; ++i)*

***for*** *(****int*** *j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)*

***if*** *(dist(strip[i],strip[j]) < min)*

*min = dist(strip[i], strip[j]);*

***return*** *min;*

*}*

*// A recursive function to find the smallest distance. The array Px contains*

*// all points sorted according to x coordinates and Py contains all points*

*// sorted according to y coordinates*

***float*** *closestUtil(Point Px[], Point Py[],* ***int*** *n)*

*{*

*// If there are 2 or 3 points, then use brute force*

***if*** *(n <= 3)*

***return*** *bruteForce(Px, n);*

*// Find the middle point*

***int*** *mid = n/2;*

*Point midPoint = Px[mid];*

*// Divide points in y sorted array around the vertical line.*

*// Assumption: All x coordinates are distinct.*

*Point Pyl[mid];   // y sorted points on left of vertical line*

*Point Pyr[n-mid];  // y sorted points on right of vertical line*

***int*** *li = 0, ri = 0;  // indexes of left and right subarrays*

***for*** *(****int*** *i = 0; i < n; i++)*

*{*

***if*** *((Py[i].x < midPoint.x || (Py[i].x == midPoint.x && Py[i].y < midPoint.y)) && li<mid)*

*Pyl[li++] = Py[i];*

***else***

*Pyr[ri++] = Py[i];*

*}*

*// Consider the vertical line passing through the middle point*

*// calculate the smallest distance dl on left of middle point and*

*// dr on right side*

***float*** *dl = closestUtil(Px, Pyl, mid);*

***float*** *dr = closestUtil(Px + mid, Pyr, n-mid);*

*// Find the smaller of two distances*

***float*** *d = min(dl, dr);*

*// Build an array strip[] that contains points close (closer than d)*

*// to the line passing through the middle point*

*Point strip[n];*

***int*** *j = 0;*

***for*** *(****int*** *i = 0; i < n; i++)*

***if*** *(****abs****(Py[i].x - midPoint.x) < d)*

*strip[j] = Py[i], j++;*

*// Find the closest points in strip.  Return the minimum of d and closest*

*// distance is strip[]*

***return*** *stripClosest(strip, j, d);*

*}*

*// The main function that finds the smallest distance*

*// This method mainly uses closestUtil()*

***float*** *closest(Point P[],* ***int*** *n)*

*{*

*Point Px[n];*

*Point Py[n];*

***for*** *(****int*** *i = 0; i < n; i++)*

*{*

*Px[i] = P[i];*

*Py[i] = P[i];*

*}*

***qsort****(Px, n,* ***sizeof****(Point), compareX);*

***qsort****(Py, n,* ***sizeof****(Point), compareY);*

*// Use recursive function closestUtil() to find the smallest distance*

***return*** *closestUtil(Px, Py, n);*

*}*

*// Driver program to test above functions*

***int*** *main()*

*{*

*Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};*

***int*** *n =* ***sizeof****(P) /* ***sizeof****(P[0]);*

*cout << "The smallest distance is " << closest(P, n);*

***return*** *0;*

*}*

***Output***

***The smallest distance is 1.41421***

***CONCLUSION:***

After the completion of the project, it can be concluded that this project successfully calculates the minimum distance between airplanes in order to avoid crashes. We have spread the calculation of distance on a smaller scale but the same concept can be used in the controlling the air traffic of larger area . It uses the concept of closest pair using merge sort algorithm.