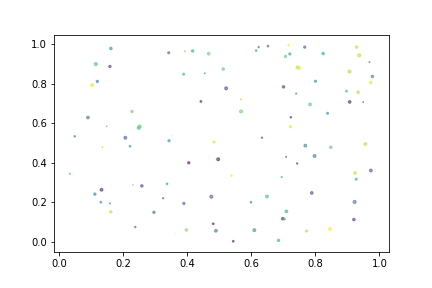
**QUESTION STATEMENT:**

**Find the closest pair of points:**

Generate a random pair of 100 values. Apply the closest pair algorithm to find the closest pair (distance). A C++ Implementation from scratch implementation with neat documentation is expected.



Random pairs on a plane Example

The time complexity of the algorithm should be O(N log N).

**CLOSEST PAIR ALGORITHM:**

The **closest pair of points problem** or **closest pair problem** is a problem of [computational geometry](https://en.wikipedia.org/wiki/Computational_geometry): given *n* points in [metric space](https://en.wikipedia.org/wiki/Metric_space), find a pair of points with the smallest distance between them. An algorithm of finding distances between all pairs of points in a space of dimension *d* and selecting the minimum requires [O](https://en.wikipedia.org/wiki/Big_O_notation)(*n*2) time. It turns out that the problem may be solved in O(*n* log *n*) time in a [Euclidean space](https://en.wikipedia.org/wiki/Euclidean_space) or [L](https://en.wikipedia.org/wiki/Lp_space" \o "Lp space)[p](https://en.wikipedia.org/wiki/Lp_space" \o "Lp space)[space](https://en.wikipedia.org/wiki/Lp_space" \o "Lp space) of fixed dimension d. In the [algebraic decision tree](https://en.wikipedia.org/wiki/Algebraic_decision_tree) [model of computation](https://en.wikipedia.org/wiki/Model_of_computation), the O(*n* log *n*) algorithm is optimal, by a reduction from the [element uniqueness problem](https://en.wikipedia.org/wiki/Element_uniqueness_problem). In the computational model that assumes that the [floor function](https://en.wikipedia.org/wiki/Floor_function) is computable in constant time the problem can be solved in O(*n* log log *n*) time. If we allow randomization to be used together with the floor function, the problem can be solved in O(*n*) time.

**PROGRAM LOGIC:**

All the points are divided into two halves and the smallest distance is found out recursively. The minimum of two smallest distances is taken and named as ***dist***. An array ***stp[]*** is created which stores all the points that are nearly d distance away from the middle point. The smallest distance is found out in ***stp[]*** and is returned along with the minimum of ***dist***.

**PROGRAM CODE:**

#include <iostream>

#include <cfloat>

#include <cstdlib>

#include <cmath>

using namespace std;

struct poi {

double poi1, poi2;

};

inline int Comp\_poi1(const void\* x, const void\* b)

{

poi \*p1 = (poi \*)x, \*pnt2 = (poi \*)b;

return (p1->poi1 - pnt2->poi1);

}

inline int Comp\_poi2(const void\* x, const void\* y)

{

poi \*pnt1 = (poi \*)x, \*pnt2 = (poi \*)y;

return (pnt1->poi2 - pnt2->poi2);

}

inline double Distance(poi pnt1, poi pnt2) { // Calculate the distance between two points

return sqrt( (pnt1.poi1 - pnt2.poi1)\*(pnt1.poi1 - pnt2.poi1) +

(pnt1.poi2 - pnt2.poi2)\*(pnt1.poi2 - pnt2.poi2) );

}

double S\_Distance(poi P[], int n, poi &pnt1, poi &pnt2) {

double min = DBL\_MAX;

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

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

if (Distance(P[i], P[j]) < min) {

min = Distance(P[i], P[j]);

pnt1.poi1 = P[i].poi1, pnt1.poi2 = P[i].poi2;

pnt2.poi1 = P[j].poi1, pnt2.poi2 = P[j].poi2;

}

return min;

}

inline double Minimum(double poi1, double poi2) { // Find minimum between two values

return (poi1 < poi2)? poi1 : poi2;

}

double Closest\_dist\_Spoint(poi stp[], int s, double dist, poi &pnt1, poi &pnt2) { // Calculate distance beween the closest points

double Minimum = dist; // Initialize the minimum distance as dist

qsort(stp, s, sizeof(poi), Comp\_poi2);

for (int i = 0; i < s; ++i)

for (int j = i+1; j < s && (stp[j].poi2 - stp[i].poi2) < Minimum; ++j)

if (Distance(stp[i],stp[j]) < Minimum) {

Minimum = Distance(stp[i], stp[j]);

pnt1.poi1 = stp[i].poi1, pnt1.poi2 = stp[i].poi2;

pnt2.poi1 = stp[j].poi1, pnt2.poi2 = stp[j].poi2;

}

return Minimum;

}

double Closest\_dist(poi P[], poi stp[], int n, poi &pnt1, poi &pnt2) { // Calculate smallest distance.

static poi pt1, pt2, pt3, pt4;

if (n <= 3)

return S\_Distance(P, n, pt1, pt2);

int medium = n/2; // Calculate the mid point

poi mediumPoint = P[medium];

double D\_Left = Closest\_dist(P, stp, medium, pt1, pt2); // D\_Left: left of medium point

double D\_Right = Closest\_dist(P + medium, stp, n-medium, pt3, pt4); // D\_Right: right side of the medium point

if(D\_Left < D\_Right) {

pnt1.poi1 = pt1.poi1; pnt1.poi2 = pt1.poi2; // Store the pair that has smaller distance

pnt2.poi1 = pt2.poi1; pnt2.poi2 = pt2.poi2;

} else {

pnt1.poi1 = pt3.poi1; pnt1.poi2 = pt3.poi2;

pnt2.poi1 = pt4.poi1; pnt2.poi2 = pt4.poi2;

}

double min\_dist = Minimum(D\_Left, D\_Right);

int j = 0;

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

if (abs(P[i].poi1 - mediumPoint.poi1) < min\_dist)

stp[j++] = P[i];

double min\_dist\_strip = Closest\_dist\_Spoint(stp, j, min\_dist, pt1, pt2);

double F\_Min = min\_dist;

if(min\_dist\_strip < min\_dist) {

pnt1.poi1 = pt1.poi1; pnt1.poi2 = pt1.poi2;

pnt2.poi1 = pt2.poi1; pnt2.poi2 = pt2.poi2;

F\_Min = min\_dist\_strip;

}

return F\_Min;

}

int main() {

poi P[] = {{4, 1}, {15, 20}, {30, 40}, {8, 4}, {13, 11}, {5, 6}};

poi pnt1 = {DBL\_MAX, DBL\_MAX}, pnt2 = {DBL\_MAX, DBL\_MAX}; // Closest pair of points in array

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

qsort(P, n, sizeof(poi), Comp\_poi1);

poi \*stp = new poi[n];

cout << "The closest distance of point in array is: " << Closest\_dist(P, stp, n, pnt1, pnt2) << endl;

cout << "The closest pair of point in array: (" << pnt1.poi1 << "," << pnt1.poi2 << ") and ("

<< pnt2.poi1 << "," << pnt2.poi2 << ")" << endl;

delete[] stp;

return 0;

}

**OUTPUT:**

