Closest-Points Problem

Aditi Sanjay Bagora

IIT Hyderabad

10/09/2021

Advanced Data Structures and Algorithms (CS6013)

Rogers Mathew

Abstract

The report contains pseudo-code and description of the implemented algorithms used to solve the closest-points problem, correctness of the algorithm and running time complexities of the algorithm

Table of Contents

Table of Contents

1 Introduction

1.1 Problem Description

Closest-Points Problem: We know that the Euclidean distance between any two points on a plane, say (a_1, b_1) and (a_2, b_2) is ($(a_2-a_1)^2 + (b_2-b_1)^2)^{1/2}$. Write a program that takes as input from keyboard the value of a positive integer *N*, and the coordinates of *N* points on the 2-dimensional plane, namely (x_1, y_1),(x_2, y_2), . . . ,(x_n, y_n). The program then finds a closest pair of points and outputs this pair of points along with the distance between them. It is possible that two points have the same position; in that case, that pair is the closest, with distance zero. The program should print the output on the standard output. You may round real numbers to two decimal places.

Input/output format explained using an example:

How many points are there on the 2D plane? 3

Enter the coordinates of Point 1

x_1 : 3

y_1 : 2

Enter the coordinates of Point 2

x_2 : 2

y_2 : 0

Enter the coordinates of Point 3

x_3 : 3

y_3 : 3.5

The closest pair of points are (3, 2) and (3, 3.5). The distance between them is 1.5 units.

2 Background

2.1 Euclidian Distance

We use Euclidian distance metric to compute distance (d) between points P1(x1, y1), P2 (x2, y2) as

$$\mathbf{d = \sqrt{[(x_2 - x_1)^2 + (y_2 - y_1)^2]}.}$$

2.2 Divide & Conquer Strategy

To calculate closest points pair by brute it takes $O(n^2)$ time . To optimize the time complexity to

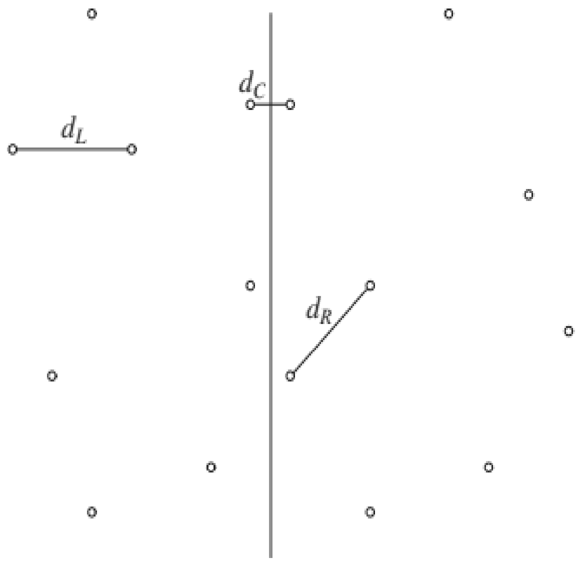
$O(N \log N)$ we use the divide and conquer strategy.

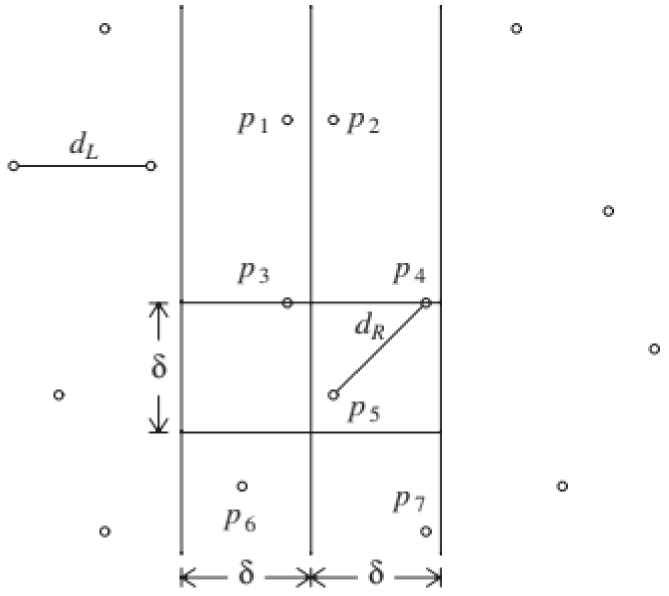
3 Methodology

The idea of main algorithm is similar to max_subset_problem discussed in class. To find closest points we use distance metric as point with minimum distance will be the closest point. So now the focus is to find two points such that the distance between them is minimum in O (NlogN) time. So we define a vertical line that divides the points almost equally in two halves. Now, both the points with minimum distance can lie to the left or to the right or a point can lie in the left and another can lie at the right side. The algorithm denotes these the distances as d_L, d_R, d_C respectively.

The implementation takes points P_1,P_2,P_3….P_n where n is the number of points given by the user. Each point contains it's corresponding x and y co-ordinates in 2D space. The points are initialized with the values given by the user and are sorted in ascending order of their x co-ordinate using modified version of merge sort which takes array of points as input and sorts them based on the axis(either x or y) specified in Θ(*N* log *N*). So, we maintain two lists of points P_x, P_y.

We first find x co-ordinate of the midpoint of P_x and segregate P_y such that all the points in the P_y whose x co-ordinate is less than or equal to x co-ordinate of the midpoint lies to left partition and others lie in the right partition as shown in Figure1. To find d_L, d_R the input is divided recursively until the size of the partition becomes less than or equal to 3. For size<=3 minimum distance can be easily found by comparison of the distances. We get d_L, d_R recursively then we find ɗ =minimum (d_L, d_R).





Left_Partition x=m Right_Partition

Figure 1[1]: It shows the distances d_L, d_R, d_C created

and how the input is divided.

Figure 2[1]: Shows 2D representation of the strip

We know that the minimum distance between points in both the halves is ɗ. So we compute d_C iff it reduces ɗ . In order to do that we create a strip that contains point within ɗ distance from the midpoint in that iteration. To create that strip we simply compare the x co-ordinate of the point P_i and the midpoint if P_i lies within the range of $midpoint_x$ + ɗ or $midpoint_x$ - ɗ it is added to strip as shown in Figure 2. Now we try to find distance between the points in the strip such that the distance is less than ɗ.

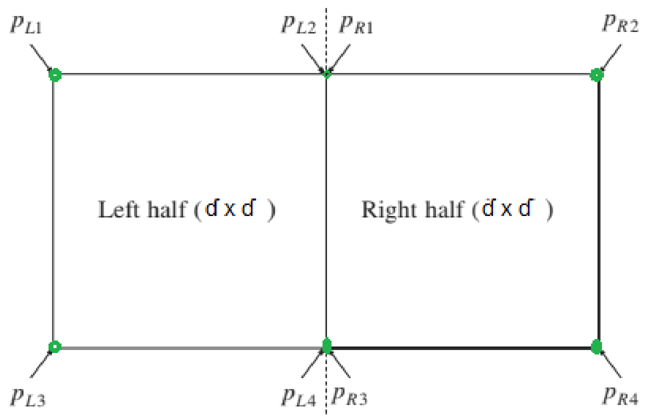


Figure 3[1]: The figure shows that at most 8 points can lie within a rectangle of (ɗ x 2 *ɗ)

To further reduce the comparisons we discard the points where the y co-ordinates of points P_i, P_j in the strip differ by more than delta as if difference is more than ɗ then d_C>= ɗ . This condition restricts the number of comparisons to be within a rectangle of (ɗ,2 *ɗ) .The point has to lie ɗ- ɗ square either on right or left and by the condition points differ by at least ɗ. So in a square of side ɗ maximum number of points can be at most 4. Therefore for any point P_i maximum 7 comparisons are needed i.e. O (1) and to compute d_C this needs to be done for all in the strip thus taking O(n) time.

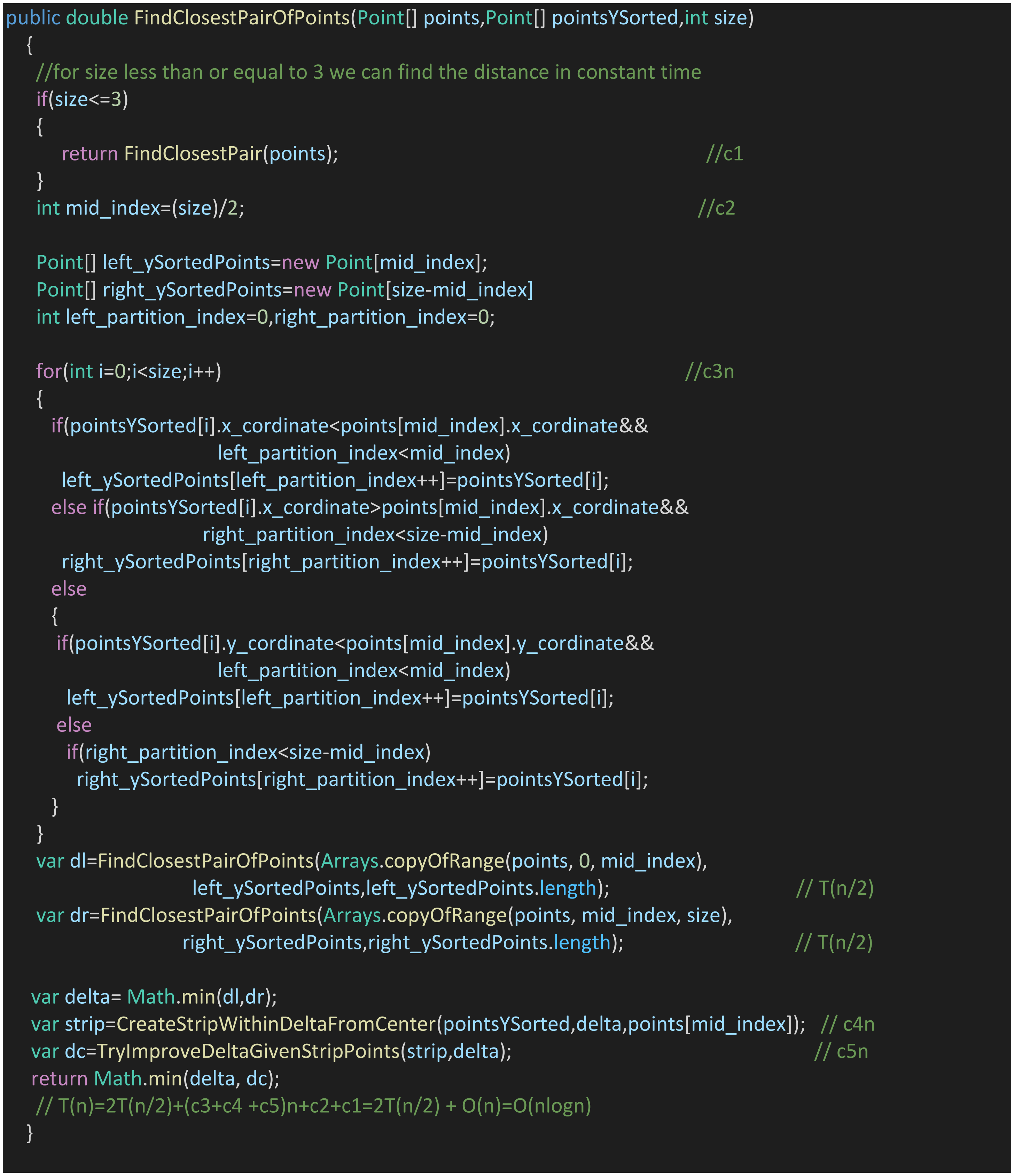
3.1 Pseudo Code:

3.1.1 Merge Sort: This merge sort works on array of points and axis determines the axis (x or y) to sort the array in ascending order.

```
public static void merge_sort(Point[] array,int lower_index,int upper_index,int axis)
{
  if(lower_index==upper_index)                    //Base case for 1 element
    return;

  int mid_index=(int)((lower_index+upper_index)/2);

  merge_sort(array, lower_index, mid_index,axis);       //T(n/2)
  merge_sort(array, mid_index+1, upper_index,axis);     //T(n/2)

  // Combine the arrays from low to mid,mid+1 to upper in ascending order for each division of input
  combine(array,lower_index,mid_index,upper_index,axis); //O(n)
}
```

```
private void combine(Point[] array, int lower_index, int mid_index, int upper_index,int axis)
{
 Point[] left_partition=new Point[mid_index+1-lower_index]; Point[] right_partition=new Point[upper_index-
mid_index];
    for(int i=lower_index,k=0;i<=mid_index;++i,++k)             // [c3*(n/2)]
       left_partition[k]=array[i];
    for(int i=mid_index+1,k=0;i<=upper_index;++i,++k)           // [c4*(n/2)]
       right_partition[k]=array[i];
    int left_partition_index=0;int right_partition_index=0
    for(int i=lower_index;i<=upper_index;++i)                         // [c5*n]
    {
      if(left_partition_index==left_partition.length &&
       right_partition_index<right_partition.length)
      {
        array[i]=right_partition[right_partition_index++];

      }
      else if(right_partition_index==right_partition.length
          && left_partition_index<left_partition.length)
      {
        array[i]=left_partition[left_partition_index
      }
      else
      {  Compare for x if axis is 0 and for y is axis is 1
        if(axis==0)
        {
         if(left_partition[left_partition_index].x_cordinate<=
           right_partition[right_partition_index].x_cordinate)
         {
          array[i]=left_partition[left_partition_index++];
         }
         else
        {
         array[i]=right_partition[right_partition_index++];
        }
       }
       else
       {
           if(left_partition[left_partition_index].y_cordinate<=
           right_partition[right_partition_index].y_cordinate)
           {
            array[i]=left_partition[left_partition_index++];
           }
           else
           {
            array[i]=right_partition[right_partition_index++];
           }
       }
      }
    }

  }
```

3.1.2 FindClosestPairOfPoints: This is a recursive method that takes two presorted array of points on x and y axis respectively using the above mentioned merge sort algorithm and the size of array sorted on y axis.



```
public double FindClosestPairOfPoints(Point[] points,Point[] pointsYSorted,int size)
  {
   //for size less than or equal to 3 we can find the distance in constant time
   if(size<=3)
   {
      return FindClosestPair(points);                                        //c1
   }
   int mid_index=(size)/2;                                                   //c2

   Point[] left_ySortedPoints=new Point[mid_index];
   Point[] right_ySortedPoints=new Point[size-mid_index]
   int left_partition_index=0,right_partition_index=0;

   for(int i=0;i<size;i++)                                                  //c3n
   {
     if(pointsYSorted[i].x_cordinate<points[mid_index].x_cordinate&&
                             left_partition_index<mid_index)
      left_ySortedPoints[left_partition_index++]=pointsYSorted[i];
     else if(pointsYSorted[i].x_cordinate>points[mid_index].x_cordinate&&
                           right_partition_index<size-mid_index)
      right_ySortedPoints[right_partition_index++]=pointsYSorted[i];
     else
     {
      if(pointsYSorted[i].y_cordinate<points[mid_index].y_cordinate&&
                             left_partition_index<mid_index)
       left_ySortedPoints[left_partition_index++]=pointsYSorted[i];
      else
       if(right_partition_index<size-mid_index)
        right_ySortedPoints[right_partition_index++]=pointsYSorted[i];
     }
   }
   var dl=FindClosestPairOfPoints(Arrays.copyOfRange(points, 0, mid_index),
                        left_ySortedPoints,left_ySortedPoints.length);                  // T(n/2)
   var dr=FindClosestPairOfPoints(Arrays.copyOfRange(points, mid_index, size),
                       right_ySortedPoints,right_ySortedPoints.length);                 // T(n/2)

  var delta= Math.min(dl,dr);
  var strip=CreateStripWithinDeltaFromCenter(pointsYSorted,delta,points[mid_index]);   // c4n
  var dc=TryImproveDeltaGivenStripPoints(strip,delta);                                 // c5n
  return Math.min(delta, dc);
   // T(n)=2T(n/2)+(c3+c4 +c5)n+c2+c1=2T(n/2) + O(n)=O(nlogn)
  }
```

Figure 4: FindClosestPairPoints main method

```
  private double FindClosestPair(Point[] points)
  {
   for(int i=0;i<points.length;++i)
      for(int j=i+1;j<points.length;++j)
          {
              var distance= CalculateEuclidianDistance(points[i], points[j]);
              if(distance<min_partition_distance)
              {
                min_partition_distance=distance;
                closest_point_Pair[0]=points[i];closest_point_Pair[1]=points[j];
              }
          }
       return min_partition_distance;
  }
```

```
public Point[] CreateStripWithinDeltaFromCenter(Point[] points,double delta,Point mid_point)
  {
       var center_x=mid_point.x_cordinate;
       ArrayList<Point> strip=new ArrayList<Point>();
       for(int i=0;i<points.length;i++)                                    //c1n
       {
         if(Math.abs(points[i].x_cordinate-center_x)<=delta)
            strip.add(points[i]);
       }
       Point[] stripPoints= new Point[strip.size()];
       strip.toArray(stripPoints);
       return stripPoints;
  }
```
Figure 5: Creation of the strip main method

```
  public double TryImproveDeltaGivenStripPoints(Point[] points,double delta)
  {
     min_partition_distance=delta;
     for(int i=0;i<points.length;i++)        //O(n)
      for(int j=i+1;j<points.length;++j)      //O(1)
       {
          if(IsYDiffMoreThanDelta(points[i], points[j], delta))
            break;
          double distance=CalculateEuclidianDistance(points[i], points[j]);
          if(min_partition_distance>distance)                      {
            dc=min_partition_distance=distance;
            closest_point_Pair[0]=points[i];closest_point_Pair[1]=points[j];
          }
       }
     return dc;
  }
```
Figure 6: Find dc and try to improve delta.

```
FindClosestPairPoints(xsorted,ysorted,size)
{
     If(size<=3)
       Find Closest Points and return;
   Mid=size/2;
   Create a leftysorted array with points less than or equal to xsorted[Mid].x;
   Create a rightysorted array with points greater than or equal to xsorted[Mid].x
   Dl= FindClosestPairPoints(xsorted(0,Mid), leftysorted,size);
   Dr= FindClosestPairPoints(xsorted(Mid+1,size), rightysorted,size);
   Delta=min(Dl,Dr)
   Strip=CreateStrip(ysorted, Delta, xsorted[Mid])
   Finddcminimizedelta(Strip,Delta)
   Return min(delta,dc);
}
CreateStrip(array,delta,mid-point)
{
   For every point in array
   If(Abs(array[i].x-mid-point.x)<=delta)
    Add array[i] to strip
  Return strip
}

Finddcminimizedelta(Strip,Delta){

min_distance=delta
for every point Pi,Pj in strip where Pi!=Pj && |Pi.y-Pi.y|<delta
if(distance between Pi&Pj<min_distance)
{
  min_distance=dc= distance between Pi&Pj
 }
return dc;
}
```

3.2 Correctness:

Correctness proof of the algorithm mentioned in Fig.4 (FindClosestPairPoints).

We need to find the closest points among all the given points $P_1(x_1, y_1), P_2(x_2, y_2), \ldots, P_n(x_n, y_n)$.The idea is to sort the points according to x axis and dividing the points into two sub problems of size n/2(Such that points on each sub partition is sorted on y axis and the left partition contains points to the left side of the vertical line shown in Fig. 1. and right side contains points at right side) recursively till the sub problems are of size<=3. For size <=3, brute force i.e. for every pair of point distance is calculated using Euclidian distance formula and minimum of all the distances is returned. Recursion (Fig.4) takes x-sorted and y-sorted array and their size as input. At each step of recursion the x-sorted array is divided into halves from mid-point and y-sorted arrays are arranged as shown in Fig.1.

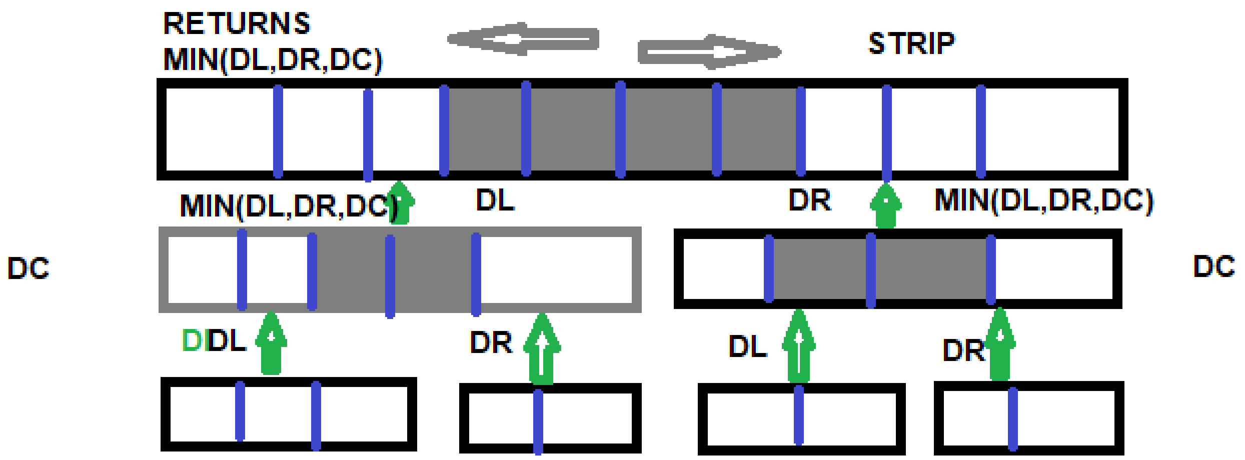


Fig.7: Working of recursion for this problem.

The algorithm calculates and returns distances for each sub partition as dl and dr and the minimum is found called as delta and then at combine stage the strip is created to find pair of points whose distance is lesser than min(dl,dr). So the closest points P_x and P_y may lie in one of these three cases

1. both in the left side
2. both to the right side
3. One is left and another in right.

Case 1 and Case 2 can be solved recursively(as shown in Fig. 7) and it is bound to find the closest pair as we use brute force to solve the smallest problems i.e. it considers all the combinations and finds the minimum. Then we find minimum of (dl,dr)=delta

For Case 3 we need to compute dc iff it minimizes delta i.e. we need to calculate dc only if there lies two points which have lesser distance than delta. The idea of finding dc is similar to our max_subset_problem where we looked at either sides of the mid-point and try to find if it increases the sum, if it does we output that subset. Here we try to search for a point in either side of the vertical line(line here means x=c where c is the co-ordinate which we get by dividing x-sorted array from mid or as shown in Fig. 1). ***Since, we need to minimize delta any point which is not in the range of [midpoint$_x$ - delta, midpoint$_x$ + delta] need not be considered as distance will be greater than delta***.

Consider any points P_i (x_i,y_i) and $P_j(x_j,y_j)$ where P_j is the midpoint and P_i is a point such that $|(x_j-x_i)|>$ delta.(1)

$$(x_j - x_i)^2 > \text{delta}^2 \qquad \text{by squaring both the sides}$$

Let $(y_j-y_i) = c$ where c is a real number

$D = ((x_j-x_i)^2 + c^2)^{1/2}$ by Euclidian distance formula

where D is the distance between the points and is greater than delta for simplicity assume c=0

Then $D = (x_j-x_i)$ and $(x_j-x_i) >$delta

Now we know that even if c=0 D>delta so for any real number c the distance will be greater than delta. Even If c<0 the distance formula adds c^2 which is always positive. therefore D >delta given (1)

Similarly for any two points $P_i (x_i,y_i)$ and $P_j(x_j,y_j)$ such that $|(y_j-y_i)|>$ delta

$(y_j-y_i)^2 >$ delta^2 by squaring both the sides

Let $(x_2-x_1) = c$ where c is a real number

Then $d = ((y_2-y_1)^2 + c^2)^{1/2}$ by Euclidian distance formula

d> delta where d is the distance between the points. (2)

Hence, the algorithm includes only the points within the range [midpoint$_x$ - delta, midpoint$_x$ + delta] and call that a strip as shown in Fig. 2. How do we ensure the strip is made correctly? *The algorithm (Fig.5) shows that for each point in the y sorted array we check if the point lies in the range and if it does the algorithm adds it to the list in that order*. ***So they remain sorted on y and also include all the points that lie in the strip as it checks for every point in the strip***.

We try to find distance between every pair of point lying in the strip. But we know that if *for two points P_i and P_j in the strip where $P_i! = P_j$. If $|(y_j-y_i)|>$ delta then distance is also greater than delta* hence we filter such P_j when comparing i.e. ***we compare only when $|(y_j-y_i)|<=$ delta***

These conditions creates a rectangle of dimension delta x 2*delta as shown in Fig. 3. It creates 2 squares of side delta on either side of the vertical line. Now, *we know the point has to lie in this left square or right square as for the pair one must lie in left half and other in right* **by definition of case 3.So, we know that point has to be in that square of size delta.**

We know that on each side of vertical line(left/right) the minimum distance between any two points is delta which is min(dl, dr) from case 1 and 2 and as the square lies entirely on one of these sides distance between any two points within the square cannot be less than delta. ***Hence each pair of points within the square is at least delta distance apart.***

Thus, ***in a square of size delta points can only be at the corners i.e. in a square there can be 4 points at most and for the rectangle there can be at most 8 points***. So for each P_i we need to check with at most 7 points which can be done in constant time. Thus for each P_i there exist at most 7 P_j for calculation of distance. Now, we know that the inner loop in Fig.6. runs in constant time for each P_i. Hence, the time complexity is O (n).

The algorithm filter the points so how do we ensure that the closest pair in the strip is always captured?

The algorithm filter the points such that all the points within the delta x 2*delta region that can minimize delta are considered and ***the algorithm check for every pair of points within that rectangle which is again brute force as shown in Fig.6 brute force ensures that distance between every pair in the concerned rectangle is calculated and if distance<delta then returns the distance (dc) and the pair of points***. The algorithm only

reduce the number of points in consideration as *we already know that for points outside this rectangle the delta will not be minimized(from (1) and (2)).*

Now, we know from the above arguments that the algorithm calculates the minimum distance for case 1 case 2 and case 3 correctly. As the idea of implementation is similar to max_subset problem, we know that if the algorithm correctly finds the closest points for these three cases then it can find closest pair of points for any arbitrary points correctly.

The method takes input as two arrays which are already sorted using merge sort correctness and complexity of the merge sort are already known.

3.3 Time Complexity:

Time Complexity of the algorithm mentioned in Fig.4 (FindClosestPairPoints).

The method contains calls to other methods and 2 recursion calls to itself.

1. ***FindClosestPair***
   It starts with the base case where if the number of points is less than equal to 3 brute force is used i.e. for each pair of points distance is calculate now there can be at most 3 points and number of comparisons needed to compute distances between each pair of 3 points is 3 which is a constant and time taken to compute distance is also constant. Hence, the method FindClosestPair takes constant time to calculate let that be c1.
2. Calculation of mid-point also takes constant time, creation of left and right y sorted array also takes constant time, Calculation of min (dl,dr) takes constant time. Let sum of all these constants be c2.
3. To create the point array as shown in Fig.1 we take y sorted array and compare x co-ordinate with the midpoint_x and if it is less than midpoint_x add it to left array else we add it to right array. This takes a traversal of all the points and comparison takes constant time hence the loop takes c3 * n time.
4. ***FindClosestPairPoints***
   Then there are two recursive calls to the same function with input of size n/2 hence it takes 2T (n/2).
5. ***CreateStripWithinDeltaFromCenter (Fig.5)***
   The method starts with the midpoint and runs a for loop from 0 to n. This loop can run for c1*n time. We can consider rest of the sentences to be done in constant time c3 .Hence Complexity for this method = n*(c1) + c3= O (n)
6. ***TryImproveDeltaGivenStripPoints (Fig. 6.)***
   From our discussion of correctness proof we know that for each P_i in the strip there are at most 7 P_js to be compared. The inner loop runs for at most 7 times and everything written inside the inner loop i.e. calculating distance, comparisons and storing can be done in constant time c1. So inner loop takes at most 7*c1 time that is constant time and it executes for each Pi in the strip. Let number of points in the strip be n then the outer loop runs for n*(7*c1) i.e. c2n =O (n)

Now, the combined complexity of the method ***FindClosestPairPoints*** for size n

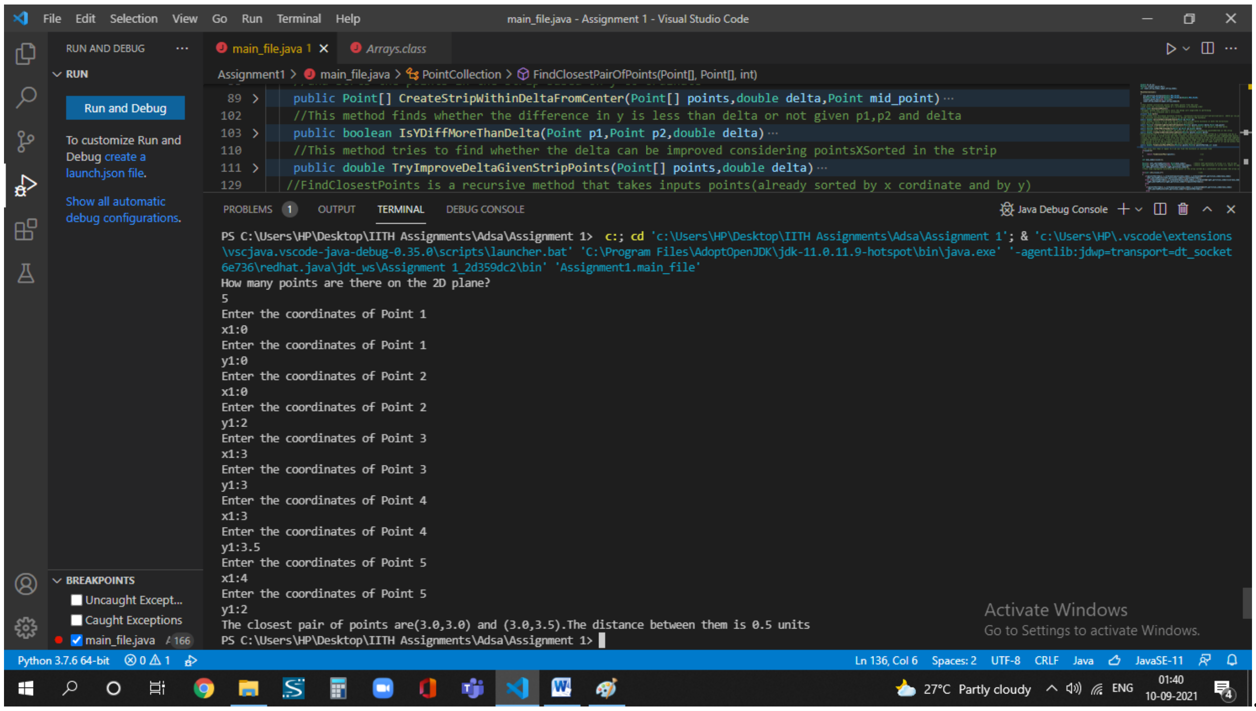
$$T(n) = c1 + c2 + c3 * n + 2\,T(n/2) + O(n) + O(n)$$

$$= 2\,T(n/2) + c'\,n$$

$$T(n) \in \Theta(n * \log n)$$

Now, we know complexity of method ***FindClosestPairPoints*** € Θ (n*log n). But the time taken for initialization of points and sorting of array on x and y axis is not considered. Initialization take O (n) as every point needs to be initialized and as merge sort is used for sorting the array and it runs two times i.e. for sorting based on x and for sorting based on y that takes 2*n*log n where n is the number of points hence the complexity of initialization and sorting = 2*n*log n +c*n = Θ (n*log n). Therefore the complete program runs in Θ (n*log n) time.

4 Results and discussion:



```
public Point[] CreateStripWithinDeltaFromCenter(Point[] points,double delta,Point mid_point)···
//This method finds whether the difference in y is less than delta or not given p1,p2 and delta
public boolean IsYDiffMoreThanDelta(Point p1,Point p2,double delta)···
//This method tries to find whether the delta can be improved considering pointsXSorted in the strip
public double TryImproveDeltaGivenStripPoints(Point[] points,double delta)···
//FindClosestPoints is a recursive method that takes inputs points(already sorted by x cordinate and by y)
```

```
PS C:\Users\HP\Desktop\IITH Assignments\Adsa\Assignment 1>  c:; cd 'c:\Users\HP\Desktop\IITH Assignments\Adsa\Assignment 1'; & 'c:\Users\HP\.vscode\extensions
\vscjava.vscode-java-debug-0.35.0\scripts\launcher.bat' 'C:\Program Files\AdoptOpenJDK\jdk-11.0.11.9-hotspot\bin\java.exe' '-agentlib:jdwp=transport=dt_socket
6e736\redhat.java\jdt_ws\Assignment 1_2d359dc2\bin' 'Assignment1.main_file'
How many points are there on the 2D plane?
5
Enter the coordinates of Point 1
x1:0
Enter the coordinates of Point 1
y1:0
Enter the coordinates of Point 2
x1:0
Enter the coordinates of Point 2
y1:2
Enter the coordinates of Point 3
x1:3
Enter the coordinates of Point 3
y1:3
Enter the coordinates of Point 4
x1:3
Enter the coordinates of Point 4
y1:3.5
Enter the coordinates of Point 5
x1:4
Enter the coordinates of Point 5
y1:2
The closest pair of points are(3.0,3.0) and (3.0,3.5).The distance between them is 0.5 units
PS C:\Users\HP\Desktop\IITH Assignments\Adsa\Assignment 1> 
```

main_file.java - Assignment 1 - Visual Studio Code
RUN AND DEBUG
Run and Debug
To customize Run and Debug create a launch.json file.
Show all automatic debug configurations.
BREAKPOINTS
Uncaught Except...
Caught Exceptions
main_file.java
Assignment1 > main_file.java > PointCollection > FindClosestPairOfPoints(Point[], Point[], int)
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
Java Debug Console
Activate Windows
Go to Settings to activate Windows.
Python 3.7.6 64-bit
Ln 136, Col 6 Spaces: 2 UTF-8 CRLF Java JavaSE-11
27°C Partly cloudy ENG 01:40 10-09-2021

.

5 Acknowledgements

I would like to express my gratitude towards Rogers Mathew (IIT Hyderabad) for their kind co-operation and encouragement which helped me in completion of this assignment. I would like to express my special gratitude and thanks for providing me the reference material it helped me understanding the concept better.

.

6 References

.

[1]Figure 10.30, 10.34, 10.35 Section 10.2.2 of the book "Data Structures and Algorithm Analysis in C++ (DSAAC)", by MARK ALLEN WEISS.