REPORT – Assignment 1 – ADSA

Sauradeep Debnath, CS21MDS14025

OUTLINE :

Libraries needed – numpy, math

```
print(np.__version__)
```

```
1.20.1
```

FUNCTIONS, THEIR PSEUDO CODES & PROOF OF CORRECTNESS (wherever required) --

FUNCTION 1 - **custom_merge_sort_tuple()** – sorts using **merge sort.**

Works for either list of int/float e.g. [1,2,4.5]

Or list of tuples e.g. [(1.2, 8.9),(3.4, 8.9)]

Pseudo- code –

```
def custom_merge_sort_tuple(series):
    length =len(series)
    If length>1 :                                   # stop when length =1
            {
            Mid = floor( length/2)
            left_sub = series[mid:]
            custom_merge_sort_tuple(left_sub)        # recursive call
            right_sub =series[mid:]
            custom_merge_sort_tuple(right_sub)       # recursive call
            I,j,k=0,0,0
            for k in range(0, len(series)):
                    {if left_sub[i]<right_sub[j]:          # assign the smaller element to the sorted list
                            {series[k] = left_sub[i]
                            i=i+1
                            k=k+1}
                    else :
                            {series[k] = right_sub[j]
                            j=j+1
                            k=k+1}
```

if either of left or right sub array has remaining element(s) while the other runs out

input the elements from the remaining sub array into the "series" }

}

RUN TIME ANALYSIS – Skipped as per the instruction in the assignment

FUNCTION 2 - **get_input_points_2D()**

Gets the number of points & the co-ordinates inputs

Pseudo code –

N= no of points (input by user)

For index in range(0,N):

Take x & y inputs from the users

Add a serial number /Unique Identifier

Print the entire input list (along with the Unique Identifier)

FUNCTION 3 - **euclidean_distance_tuples()**

Calculates the Euclidean distances between two tuples of the form (x -co ordinate ,y- co ordinate, Serial No/system generated Unique ID)

FUNCTION 4 – findstrip()

Returns the y-sorted points whose x co ordinates are in the "strip" i.e. between the x -coordinate of the midpoint + /- DELTA (minimum distance)

FUNCTION 5 - brute_force()

Calculates the brute-force minimum – Used when the number of elements in a sub-series becomes <=3 .

FUNCTION 6 - closest_point_custom()

The **master function**. It calls various other functions to find the closest Pair of Points.

NOTE - THE FINAL DISTANCE IS ROUNDED UP TO 2 DECIMAL POINTS as suggested in the assignment

Input -- (x_sorted_points, y_sorted_points) -Sorted in Advance by Merge Sort (O(nlogn) time complexity)

(Runs only one time)

PSEUDO CODE –

NOTE – both x_sorted_points, y_sorted_points are a list of tuples of the format (x – coordinate, y-coordinate, Unique ID/Serial No)

```
def closest_point_custom (x_sorted_points, y_sorted_points):

    If the length of array>3:

            {

            Left_subarray = x_sorted_points[start:mid]

            left_minimum_distance, x1,y1,x2,y2 = closest_point_custom (Left_subarray, y_sorted_points)

            Right_subarray = x_sorted_points[mid:end]

            right_minimum_distance, x1,y1,x2,y2  = closest_point_custom (Right_subarray, y_sorted_points)

            delta = minimum_distance  = min (left_minimum_distance, right_minimum_distance)

            ### COMMENT - out of the above two- select the min distance & corresponding x1,y1,x2,y2

            y_sorted_strip= findstrip(x_sorted_points, y_sorted_points, mid, minimum_distance)

# COMMENT i.e. call FUNCTION findstrip-> to get the y-sorted points in the Strip ( i.e. points with x value in the range of x- # value of the mid element +/- delta )

            for i in range(0,len( y_sorted_strip)):

            {

            For j in range(i+1, length(y_sorted_strip)) :{

                    If yj – yi < delta

                        If the distance between a pair of points< prev minimum :

                                { update the minimum_distance  i.e. Delta & corresponding x1,y1,x2,y2}}

                    Else :

                                Break # EXIT the inner loop /j loop

            }

}
```

PROOF OF CORRECTNESS of Closest Pair of Points algo

BASE CASE - For n=2 and n=3 , we find the minimum distance by Brute force. Hence correct.

PROOF BY INDUCTION:

Let us suppose the algo works for a number of points n <N

We need to prove it works for number of points ,n = N

We divide the series into 2 parts – left **PL** & right **PR**. Each of size n = N/2 as shown in the diagram below from the book DSA in C++ by Mark Allen Weiss.

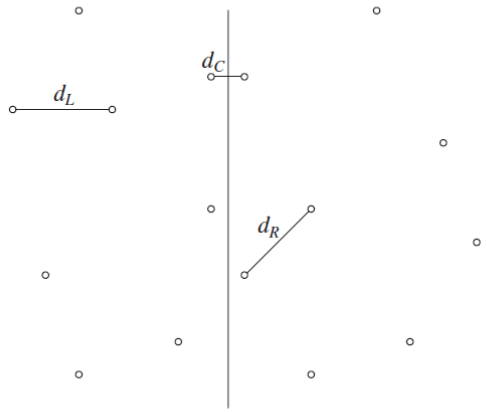


Figure 10.30 P partitioned into P_L and P_R; shortest distances are shown

We get **dL, dR** - the distance from each of these by calling our function **closest_point_custom** recursively.

Now since we assumed our algo works correctly for a count n<N → it will be correct for n = N/2 i.e. we are able to get the **dL, & dR** correctly.

We select the minimum of both of these **delta** = min (dL, dR)

Now the minimum distance points have to be either in Left, or right strip or both strips. Let us call the minimum distance between all the pair of points which are one in Left sub array , one in right subarray PR , another in left subarray PL ---- as **dC**

Since we are only interested in finding distance **dC** < **delta**--- For checking the minimum distance between both sub arrays, we only need to include the points whose x co ordinates lie within delta distance from the x-coordinate of the mid-point in the middle **"strip"**

Similarly, if any points have y co ordinates differing more than delta, we can ignore them as well (as illustrated in the same diagram from Mark's book below) . Let us call the points after these 2 step filtering as "selected strip".

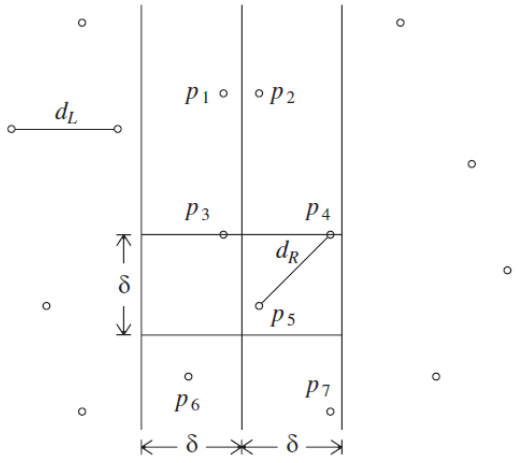


Figure 10.34 Only p_4 and p_5 are considered in the second `for` loop

We check all the distances between the points in “selected strip”—if distance(i,j) i.e. the distance between any pair of points (pi,pj) is lower than delta -> if yes, we replace the value of **delta** with distance(i,j)

Since the value of the minimum distance has to be either in left sub array PL, right sub array PR or the middle(one point in PL, another in PR)—the minimum value among (dL, dR, dC) –as computed by the above process- is the correct solution.

HENCE PROVED BY INDUCTION .

RUNNING TIME COMPLEXITY ANALYSIS of Closest Pair Point algo

Since we are dividing the problem into 2 sub problems of size n/2 – the time complexity can be denoted by :

T(n) = 2T(n/2) + f(n)

Where f(n) = the run time for separating the array into 2 parts (O(1)) + computing the value of dC (one point in PL, other in PR)

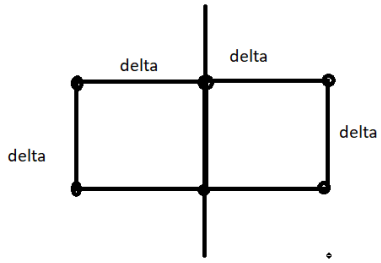
Usually for checking all the distance between n points would take n*(n-1)/2 ~n^2 = **O(n^2)** Operations.

But here, since we are doing two levels of filtering/selection:

1. We are only considering the points with x – coordinate in the range of mid -x -cordinate +/- **delta** (minimum distance) (i.e “strip”)
2. In the beginning of the problem we have x-sorted & y-sorted points.
   We select the points lying in the “strip” from y-sorted points, as soon as the distance of yj-yi exceeds delta, we exit the loop.
   Thus, effectively, there are only √n points in the strip in average.

Hence the brute force computation of the points in the “strip” – can be done in only O((√n)^2) = O(n) time.

This is because for each Pi, at worst, at worst only 7 Pi(s) are calculated. Because the points have to



Midline

Be Either in the left delta x delta square or in the right delta x delta square from the mid line- anything beyond is outside the strip. Also the points themselves have to be at least delta distance apart from each other. One of these points being Pi – only 7 Pj(s) at most need to be checked for minimum distance. That is the inner loop / j loop at most needs to run for 7 times.

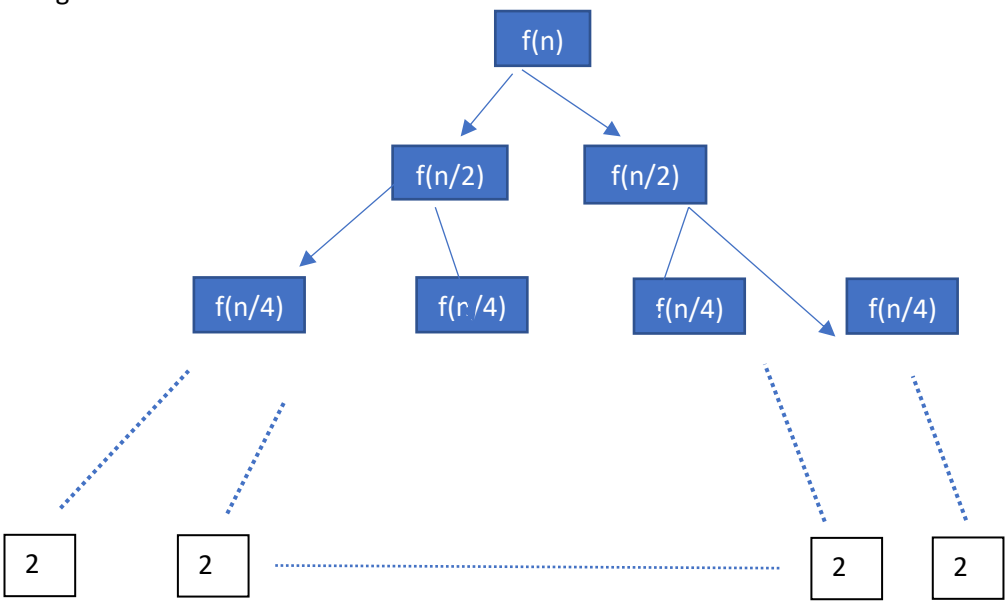
Thus this brute force checking at the y-sorted points in the strip only varies as O(n), not O(n^2)

Hence T(n) = 2T(n/2) + O(n) + c1 where c1 is a constant.

And we need to Divide by 2 till we get a n < 3

Since $2^x = n \Rightarrow x = \log_2 n$

The algo will run $\log_2 n$ times.



Since each step is O(n) – **the total time complexity is : T(n) = O(n$\log_2$n)** .

NOTE – Since the sorting by x & y-cordinate was done in the beginning itself & was a one time job – it added only O(n$\log_2$n) time complexity to the entire problem. Hence the total time complexity still remains O(n$\log_2$n)

RUN EXAMPLES-

How to run the code—the code from Example 3-

(base) C:\Users\saura\OneDrive\Documents\IIT Hyderabad\Assignments\ADSA\Assignment 1>**python ADSA_closest_pair_of_points.py**

How many points are there on the 2D plane?

6

Enter the coordinates of Point 1

the x co ordinate

2

the y co ordinate

3

Enter the coordinates of Point 2

the x co ordinate

4

the y co ordinate

5.5

Enter the coordinates of Point 3

the x co ordinate

3

the y co ordinate

7

Enter the coordinates of Point 4

the x co ordinate

-9

the y co ordinate

-2.2

Enter the coordinates of Point 5

the x co ordinate

-2

the y co ordinate

4

Enter the coordinates of Point 6

the x co ordinate

6

the y co ordinate

9

You have entered[(2.0, 3.0, 0), (4.0, 5.5, 1), (3.0, 7.0, 2), (-9.0, -2.2, 3), (-2.0, 4.0, 4), (6.0, 9.0, 5)] where the 3 elements of tuple are :

(x -co ordinate ,y- co ordinate, Serial No/system generated Unique ID)

x_sorted_points[(-9.0, -2.2, 3), (-2.0, 4.0, 4), (2.0, 3.0, 0), (3.0, 7.0, 2), (4.0, 5.5, 1), (6.0, 9.0, 5)]

y_sorted_points[(-9.0, -2.2, 3), (2.0, 3.0, 0), (-2.0, 4.0, 4), (4.0, 5.5, 1), (3.0, 7.0, 2), (6.0, 9.0, 5)]

minimum distance from Brute force (for validation)---> 1.8027756377319946

minimum distance as per Divide & Conquer algo is -

The closest pair of points are (3.0, 7.0) and (4.0, 5.5). The distance between them is 1.8 units.

SAMPLE RESULTS:

Example 1- 5 points –

```
You have entered[(2.3, -9.8, 0), (34.0, 32.0, 1), (45.0, 33.0, 2), (3.2, -7.5, 3), (4.0, 5.0, 4), (5.0, 6.0, 5)] where the 3 element
s of tuple are :
(x -co ordinate ,y- co ordinate, Serial No/system generated Unique ID)
x_sorted_points[(2.3, -9.8, 0), (3.2, -7.5, 3), (4.0, 5.0, 4), (5.0, 6.0, 5), (34.0, 32.0, 1), (45.0, 33.0, 2)]
y_sorted_points[(2.3, -9.8, 0), (3.2, -7.5, 3), (4.0, 5.0, 4), (5.0, 6.0, 5), (34.0, 32.0, 1), (45.0, 33.0, 2)]
minimum distance from Brute force ( for validation)---> 1.4142135623730951
minimum distance as per Divide & Conquer algo is -
The closest pair of points are (4.0, 5.0) and (5.0, 6.0). The distance between them is 1.41 units.

(base) C:\Users\saura\Downloads>
```

Example 2 – Same point repeated -i.e. Zero Distance

```
You have entered[(2.0, 3.0, 0), (4.0, 5.0, 1), (6.0, 7.0, 2), (2.0, 3.0, 3), (4.5, 5.5, 4), (4.79, 8.99, 5), (-9.9, -8.1, 6)] where
the 3 elements of tuple are :
(x -co ordinate ,y- co ordinate, Serial No/system generated Unique ID)
x_sorted_points[(-9.9, -8.1, 6), (2.0, 3.0, 3), (2.0, 3.0, 0), (4.0, 5.0, 1), (4.5, 5.5, 4), (4.79, 8.99, 5), (6.0, 7.0, 2)]
y_sorted_points[(-9.9, -8.1, 6), (2.0, 3.0, 3), (2.0, 3.0, 0), (4.0, 5.0, 1), (4.5, 5.5, 4), (6.0, 7.0, 2), (4.79, 8.99, 5)]
minimum distance from Brute force ( for validation)---> 0.0
minimum distance as per Divide & Conquer algo is -
The closest pair of points are (2.0, 3.0) and (2.0, 3.0). The distance between them is 0.0 units.

(base) C:\Users\saura\Downloads>
```

Example 3-

```
You have entered[(2.0, 3.0, 0), (4.0, 5.5, 1), (3.0, 7.0, 2), (-9.0, -2.2, 3), (-2.0, 4.0, 4), (6.0, 9.0, 5)] where the 3 elements o
f tuple are :
(x -co ordinate ,y- co ordinate, Serial No/system generated Unique ID)
x_sorted_points[(-9.0, -2.2, 3), (-2.0, 4.0, 4), (2.0, 3.0, 0), (3.0, 7.0, 2), (4.0, 5.5, 1), (6.0, 9.0, 5)]
y_sorted_points[(-9.0, -2.2, 3), (2.0, 3.0, 0), (-2.0, 4.0, 4), (4.0, 5.5, 1), (3.0, 7.0, 2), (6.0, 9.0, 5)]
minimum distance from Brute force ( for validation)---> 1.8027756377319946
minimum distance as per Divide & Conquer algo is -
The closest pair of points are (3.0, 7.0) and (4.0, 5.5). The distance between them is 1.8 units.

(base) C:\Users\saura\OneDrive\Documents\IIIT Hyderabad\Assignments\ADSA\Assignment 1>
```