2 Closest pair

Given a set of points in a plane, $p = \{(x_0, y_0),(x_1, y_1), \ldots ,(x_{n-1}, y_{n-1})\}$, consider the problem of determining the closest pair of points; i.e., pairs $p_i = (x_i , y_i)$ and $p_j = (x_j , y_j)$ such that the distance between them $d_{ij} = (x_i - x_j)^2 + (y_i - y_j)^2$ is minimized.

Clearly the naive idea of computing the $n(n-1)/2$ distances and finding the minimum gives an $O(n^2)$ algorithm. Here is an improvement using divide and conquer.

As a pre-processing step, the input array is sorted according to x coordinates.

1. Find a value x such that half the points have $x_i <= x$ and half have $x_i > x$ (roughly; you may partition about the median on their x coordinates to do this). On this basis split into two groups L and R.

2. Recursively compute the closest pair in L and in R. Let these pairs be $p_L, q\ L \in L$ and $p_R, q_R \in R$, with distances d_L and d_R respectively. Let the smaller of these two distances be d.

3. It remains to be seen if there is a point in L and a point in R that are less than d distance apart from each other.

4. Now, go through this sorted list, and for each point, compare its distance to the seven subsequent points in the list (why?). Let p_M, q_M be the closest pair found this way.

5. The answer is one of the three pairs $\{p_L, q_L\}$, $\{p_R, q_R\}$ and $\{p_M, q_M\}$, whichever is closest.

Final Project Assignment.

② Closest Pair → Tic Tac Toe (own).

were given a set of points in a plane

$$P = \{(x_0\ y_0), (x_1\ y_1) \cdots (x_{n-1}\ y_{n-1})\}.$$

Problem: To determine the closest pair of points.

$p_i = (x_i\ y_i)$ $\quad$ $p_j (x_j\ y_j)$

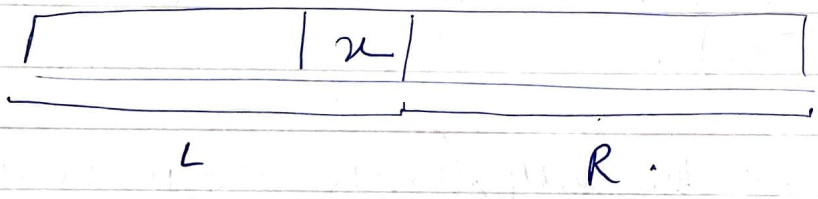
S.t the distance b/w them is.

$$d_{ij} = (x_i - x_j)^2 + (y_i - y_j)^2 \text{ is minimized.}$$

" As a pre-processing step, the input array is sorted according to x coordinates".

1. Find value x s.t

$x_i \leq x$ & $x_i > x$



Rough idea of a diagram to express the split into half and half on the basis of the x co-ordinates.

I'll be using my own variables to differentiate b/w the question & my answers.

Find the middle point in the sorted array.

Take $P[n/2]$ as the Middle point.

Then: Divide the given array in two halves.

L being $P[0] \rightarrow P[n/2]$ (left subarray)

R being $P[n/2+1] \rightarrow P[n-1]$ (right subarray)

Now, we could use the brute-force method or whose it would be

$O(n^2)$

ie: The naive method to solve and code out the question.

Here we

invariant: Count inversions in an array to calculate the distances of every pair of points we have.

So for n number of points we would need to measure

$\frac{n(n-1)}{2}$ distances.

this would mean $O(n^2)$ algorithm.

From this we think of a better way to solve which would be $O(n \cdot \lg n)$.

We pre-sort and split the array. (merge-sort method)

Note: Merge sort is faster than brute-force sorting.

In merge sort,

- We split the array
- sort the subarrays as a recursive step
- Compare the numbers in two subarrays and pick the youngest ~~(ie smallest)~~ (like... nearest)
- and this repeats until both subarrays have been checked through.

Each recursive step is $O(n)$ ∴

the ultimate algorithm is at $O(n \lg n)$.

2. Recursively compute the closest pair in L and in R where.

$p_L q_L \in L$ distance d_L $\quad$ $p_R q_R \in R$ distance d_R. Respectively.

From this let the smaller of these distances be d.

So here what were doing is basically recursively computing the closest pair in L and R.

3. It remains to be seen if there is a point in L and a point in R that are less than d distance apart from other.

For this now we will discard all points with

$x_i < x - d$ and $x_i > x + d$ } From this we will sort the remaining points with 2d

We'll do this by their y coordinates.

Now we need to consider the pairs s.t one point in the pair is from L and the other is from R

Now lets imagine a line passing through $P[n/2]$. From this we'll find all points whose x coordinate is closer than d to said line.

We will build a new assay of all such points. and sort it according to the y coordinates.

This step becomes $O(n \log n)$.

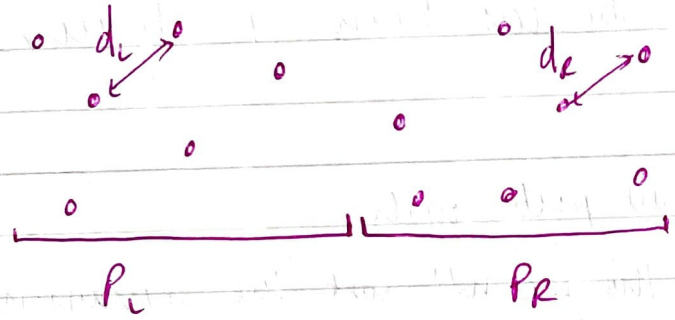
Now for step 3, it can be optimized to $O(n)$ by recursively sorting and merging.

4. Now go through this sorted list, and for each point, compare its distance to the seven subsequent points in the list...

Why are we doing this?

Let p_m q_m be the closest pair found this way.

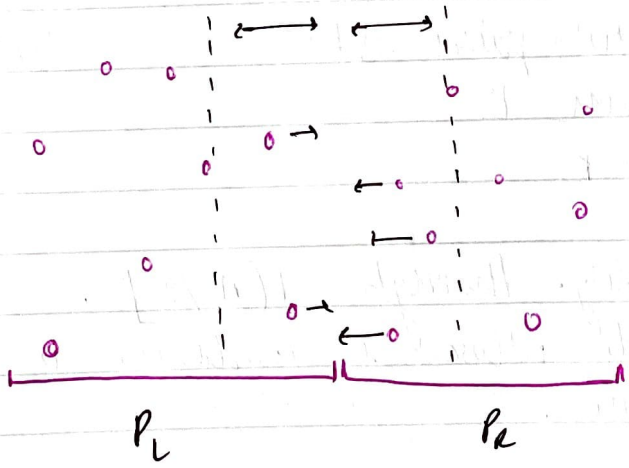
Lets look at a visual diagram for an explanation.



$d = \min(d_L, d_R)$

Imagine the dots are stars in the sky and we're looking for the smallest d b/w the stars.

Left side subarray Right side subarray.



#1 will give an example with numbers to explain this further.

5. The answer is one of the three pairs
$\{p_L q_L\}$ $\{p_R q_R\}$ $\{p_m q_m\}$.
whichever is closest.

6 finally we return the minimum of d and the distance b/w the two closest points.

Now Doing it with an example :

- We can follow the previously written theory or even the code (it helps) to understand just the steps and concept of what we're trying to do.

Closest pair of points in a plane using divide & conquer.

Basically we have to find two points s.t the distance between them is minimum among any other two points in the plane.

The distance formula being

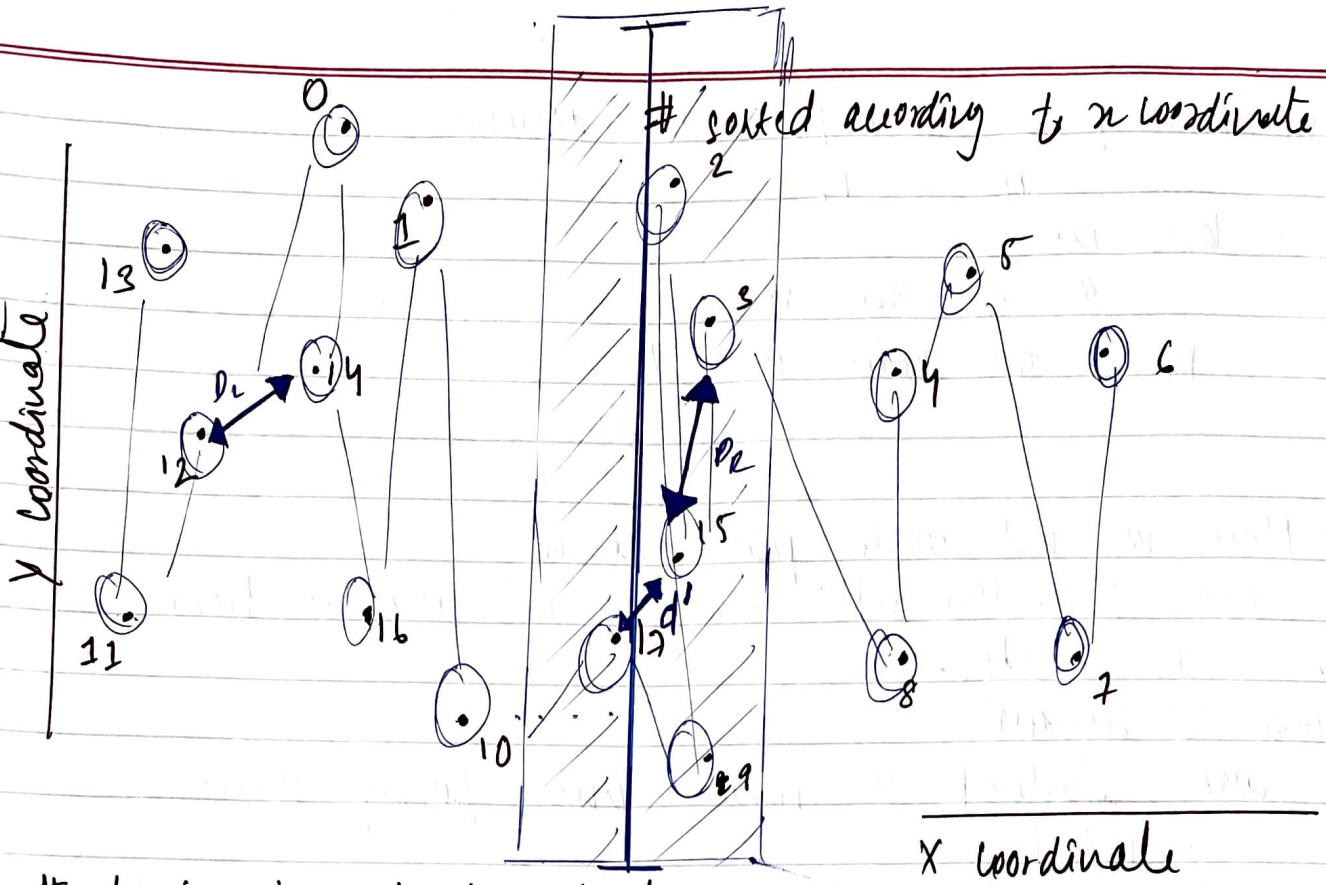
$$|pq| = \sqrt{(px - qx)^2 + (py - qy)^2}$$

Here px and py are the x and y coordinates of point p. &
qx and qy are the x and y coordinates of point q.

Taking a look at the algorithm.

Input : An array of n points P[]
Output : The smallest distance b/w 2 points in the given array.

$$P[] = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17\}$$



drawing is not to scale!

① The middle point in the (pre) sorted assay : sorted according to x coordinates.
P[n/2] a the middle point.

$$P(n/2) = P[9] = \underline{17}.$$

as we have

$$P[\,] = \{13, 11, 12, 0, 14, 16, 1, 10, (17), 9, 2, 15, 3, 8, 4, 5, 7, 6\}$$

② From here we divide the assay into two halves.

$P_L[\,] = \{13, 11, 12, 0, 14, 16, 1, 10, 17\} \rightarrow P[0] \rightarrow P[n/2]$

$P_R[\,] = \{9, 2, 15, 3, 8, 4, 5, 7, 6\} \rightarrow P[n/2+1] \rightarrow P(n-1]$

17 is the point of split belonging to P_L.

③ From here we recursively find the smallest distances in both of the subarrays. d_L and d_R (are the distances). To find → minimum of d_L and d_R. letting the minimum be d.

$$d = \min(d_L, d_R)$$

So we've found the distance the closest pair of points in both halves seperately

For this specific example lets assume

$D_L < D_R$

so

D_L is the value of d

(4) Now we will consider pairs - as in one from the left half and another from the right half.

(look at diagram)

We have considered the vertical line passing through $P[n/2]$

- Find all points whose x coordinates are closer than d to this middle vertical line.

And

Build an array strip[] of all such points.

strip[] = {17, 9, 2, 15, 3}.

We will find the distances only in this "strip[]" now as others distances would be greater than d.

(5) We sort this new array strip according to the y coordinates

(Since its y axis the order would basically be top to bottom).

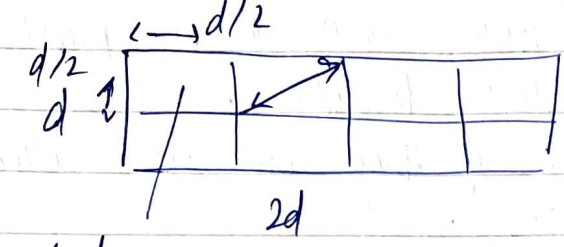
strip[] = {2, 3, 15, 17, 9}.

(6) To find the smallest distance d in strip [ ]

Now, in the worst case scenario it would take

$$O(n^2).$$

But we can optimize it here.
We need not consider each point in the strip with all the remaining points in the strip.
Instead
We will only consider those points which are less than d distance from that point.
For so, lets consider a 2d by d rectangle -



note figure not to scale.

$$\frac{d}{\sqrt{2}} < d$$

Now, no two points can belong to the same d/2 square because the maximum distance between two points on the same square can be

$$d/\sqrt{2}$$

which is the length of the diagonal
and we already know that the minimum distance between 2 points in the same ~~box~~, half is d.

So one point belongs to one square, at max.
And so, the 2d by d rectangle can have at most 8 points and any other point on the strip needs to be compared to only 7 other points.

Let d' be the smallest distance between the pair of points in the strip.

So, in this example that is

d' = distance b/w 15 and 17.

(7) Hence, the closest pair of points in the plane is the minimum of d and d'

(return min (d, d'))

```
if d' < d
    return d'
else
    return d
```

Now, we have considered each point with every other point for the smallest distance. (even if they lie on different sides).

So, we have successfully computed the distance between the closest pair of points.

The code.

```
class Point():
    def __init__(self, x, y):
        self.x = x
        self.y = y
"""
```

We have used a class to represent a Point in a 2D plane.
Now, to create a class we use the keyword class.

We've given the name Point to the class that's been created.

the __init__() function assigns values to object properties.

Now, the self parameter is a reference to the current instance of the class.
So, we use it to access variables that belong to the class.
Note: We don't actually have to name it "self" however since I am learning and exploring at the same time I've chosen to just keep it as self.

Example:

```
class Person
    def __init__(self, name, age):
        self.name = name
        self.age = age
p1 = Person("John", 36)
```

#from this we can print p1.name, p1.age as we have the p1 entry.

'''

We know the basic distance formula so we will define that

'''

```
def dist(p1, p2):
    return ((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y)) ** 0.5
```

or to write it more clearly.

```
def dist(p1, p2):
    return ((p1.x - p2.x)**2 + (p1.y - p2.y)**2) ** 0.5
```

'''

Basically square of p1.x - p2.x +
square of p1.y - p2.y
whole.
square root.

'''

""" float('inf') is used for setting a variable with an infinitely large value.

in this case it's +ve infinity.
-ve would be ('-inf')

So using the Brute force method to return the smallest distance between two points in P[] of size n we have

"""

```
def bf(P, n):    # bf ie Brute Force
    min_val = float('inf')
    for i in range(n):
        for j in range(i+1, n):
            if dist(P[i], P[j]) < min_val:
                min_val = dist(P[i], P[j])
    return min_val
```

""" Now we have the function ~~defined~~ explained in step (4).

```
def stripclosest(strip, size, d):
    # Initialize the minimum value to be d
    min_val = d
    for i in range(size):
        j = i+1
        while j < size and (strip[j].y - strip[i].y) <
        min_val:
            min_val = dist(strip[i], strip[j])
            j = j+1
    return min_val
```

we are picking all points one by one and trying the next
until the difference between y coordinates is smaller than d

"""
In python, lambda is a keyword used to define anonymous functions.

Let's take an example to understand this concept.
Take the sorted() function.

```
>>> sorted(['Some', 'words', 'sort', 'differently'])
['Some', 'differently', 'sort', 'words'].
```

This sorts uppercased words before words that are lowercased.
Now, using key keyword we can change each entry so it'll be sorted differently.
We could lowercase all the words before sorting.

```
>>> def lowercased(word): return word.lower()
>>> sorted(['Some', 'words', 'sort', 'differently'], key = lowercased)
['differently', 'Some', 'sort', 'words']
```

Here we had to create a different function. We could not inline the def lowercased() line into the sorted() expression

A lambda can be specified directly, inline in the sorted expression.

```
>>> sorted(['Some', 'words', 'sort', 'differently'], key = lambda word:
                                                    word.lower())
"""
```

"""Now were using a recursive function to find the smallest distance.
Where, our array P contains all points sorted according to x coordinate """.

```
def smallestdistance(P, Q, n):
    # if there's less than or equal to 3 points, brute force

    if n <= 3:
        return bf(P, n)
    # Find the middle point
    mid = n // 2
    midPoint = P[mid]
    # create copies of the left and right branch
    Pl = P[:mid]
    Pr = P[mid:]

# Now we can refer back to the explained algorithm where
# we consider the line passing and calculate the smallest
# distance dl on the left and dr on the right.

    dl = smallestdistance(Pl, Q, mid)
    dr = smallestdistance(Pr, Q, n-mid)

# we need to actually find the smaller of the two distances now
# we cannot just presume.

    d = min(dl, dr)  # we will def min function aswell.


# next page cont. code with indentation.
```

```
# now we'll continue to find the distance of the points
# which are < d

stripP = [ ]
stripQ = [ ]

Lr = Pl + Pr

for i in range(n):
    if abs(Lr[i].x - midPoint.x) < d:
        stripP.append(Lr[i])
    if abs(Q[i].x - midPoint.x) < d:
        stripQ.append(Q[i])   # we will define abs

stripP.sort(key = lambda point : point.y)

min_a = min(d, stripcloset(stripP, len(stripP), d))

min_b = min(d, stripcloset(stripQ, len(stripQ), d))

# now we have to return the smaller of the two.

return min(min_a, min_b)
```

""" The concept of deepcopy
A deepcopy constructs a new compound object. As in, objects that contain other objects.
It then recursively inserts these copies into the variable that we've 'said' .deepcopy to.
It inserts into it what was found in the original """

So now we're writing the main function that finds the smallest # distance

```
def closest(P, n):

    P.sort(key = lambda point : point.x)

    Q = copy.deepcopy(P)
    Q.sort(key = lambda point : point.y)

# we will use the recursive function smallestdistance() that we # had already made to find so

    return smallestdistance(P, Q, n)

""" Now we can just place our main driver code with P[] the points and n, ie the length of P """

P = [Point(2,3), Point(12,30), Point(40,50), Point(5,1),
     Point(12,10), Point(3,4)]
n = len(P)

print(closest(P, n))
```

eee

In our code when we're using the function min() we're only looking for the smaller between two numbers/variables/lists/arrays etc eeee

So even within our code where we use the function min() — we can also use an alternative saying

```
if   x > y :
     return y
else :
     return x
```

Because here imagine x = 5 y = 4

if x > y , here 5 > 4 (True)

∴ It returns y

(or in some other cases it stores the value of y in another variable).

Say x = 3 y = 7

if x > y . , 3 > 7 (False.)

So it would go to the else condition and return x, ie 3.

FOR this case we do not need to define (or use an inbuilt function min())

we can simply compare with conditions

,,,

"""

Now the abs() function.

To put it in the most simple words abs() function just returns a positive number

"""

```
i = -20
abs(i)
```

// Output
20.

```
floatingnumber = -30.33
abs(floatingnumber)
```

// floating, can have decimals etc
Output
30.33.

Now, we're using this function because in our question were taking all distances of the points to be positive.

The abs() function is not just limited to this use however so far in our code this is the only use we have for it. Therefore.

We could even define our own abs() function (like we did for min()) to be

```
if n < 0:
    return n * (-1)
else:
    return n
```

"""

eee Here I realise we are using .sort ie the built-in sorting algorithm.

Now we can use merge sort for our code (as we are presuming the time complexity of that for our algorithm).

```
def sort(arr):
    if len(arr) > 1:
        mid = len(arr)//2          # Finding the mid of the array
        L = arr[:mid]              # dividing the elements into
        R = arr[mid:]              # Left and Right

        sort(L)
        sort(R)
        i = j = k = 0              # assigning variable initial 0 value.

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i = i + 1
            else:
                arr[k] = R[j]
                j = j + 1
            k = k + 1
```

#PTO code continued with indentation.

```
while i < len(L):
    arr[k] = L[i]
    i = i + 1
    k = k + 1

while j < len(R):
    arr[k] = R[j]
    j = j + 1
    k = k + 1
```

```
# The code to print the list
def printlist(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()
```

```
# And now the driver code where we mention the array
arr = [ ]
printlist(arr)
sort(arr)
printlist(arr)
```

So first it prints the unsorted list
Then it prints the sorted list.

The Efficiency

Q. How efficient is this ~~prog~~ algorithm?.

let Time complexity be $T(n)$.
We have the sorting algo that we used.
Let that be $O(n \log n)$

first we divide all points into 2 subarrays so.
$2T(n/2)$.

and, after dividing we sorted in $O(n \log n)$ Time.

and we found the closest points in the pair in $O(n)$ Time.

$\therefore T(n) = O(n(\log n)^2)$.

```
#I have given a line by line explanation and understanding of my code in my pdf of the assignment

import copy

#A class to represent a Point in 2D plane

class Point():
    def __init__(self, x, y):
        self.x = x
        self.y = y

#dist function to find the distance between two points

def dist(p1, p2):
    return ((p1.x - p2.x) * (p1.x - p2.x) +(p1.y - p2.y) * (p1.y - p2.y))**0.5

#assert: A Brute Force method to return the smallest distance between two points in P[] of size n

def bf(P, n):
    min_val = float('inf')
    for i in range(n):
        for j in range(i + 1, n):
            if dist(P[i], P[j]) < min_val:
                min_val = dist(P[i], P[j])

    return min_val

#assert: A function to find the distance between the closest points of strip of given size.
#All points in strip[] are sorted according to y coordinate.

def stripClosest(strip, size, d):

    #assert: Initialize the minimum distance as d
    min_val = d


    #INV: Pick all points one by one and try the next points till the difference between y coordinates is smaller than d.

    for i in range(size):
        j = i + 1
        while j < size and (strip[j].y - strip[i].y) < min_val:
            min_val = dist(strip[i], strip[j])
            j += 1

    return min_val

#INV: A recursive function to find the smallest distance. The array P contains all points sorted according to x coordinate

def smallestdistance(P, Q, n):

    #assert: If there are <= 3 points, then use brute force

    if n <= 3:
        return bf(P, n)

    #Find the middle point

    mid = n // 2
    midPoint = P[mid]

    #keep a copy of left and right subarrays/ sections that we've made

    Pl = P[:mid]
    Pr = P[mid:]

    #Consider the vertical line passing through the middle point calculate smallest distance dl and dr

    dl = smallestdistance(Pl, Q, mid)
    dr = smallestdistance(Pr, Q, n - mid)
```

```
 #Find the smaller of two distances

   d = min(dl, dr)

   #assert: Build an array strip[] that contains points close (closer than d) to the line passing through the middle point
   #INV: Look and solve for values with length < d

   stripP = []
   stripQ = []
   lr = Pl + Pr

   #To put it in the most simple words, so far in this code at least the function of abs is just to return a positive value
   #Same case as min(), we could just have an if else statement
   #if lr[i].x - midPoint.x <0:    create_a_new_variable_to_check_condition_with= (lr[i].x - midPoint.x)* (-1)
   #else:  we'd move to the next line.

   for i in range(n):
      if abs(lr[i].x - midPoint.x) < d:
         stripP.append(lr[i])
      if abs(Q[i].x - midPoint.x) < d:
         stripQ.append(Q[i])

#In python lambda is a keyword used to define anonymous functions. It can be specified directly, Inline in (for example
here) the sorted function
#We are "pre-sorting", now we have used an inbuilt function but what we can use is a merge sort algorithm and define a
.sort() function.
#I have specified all defined function which are in this code inbuilt functions and given explanations in my written pdf.

   stripP.sort(key = lambda point: point.y)
   min_a = min(d, stripClosest(stripP, len(stripP), d))
   min_b = min(d, stripClosest(stripQ, len(stripQ), d))


   #Return the minimum of d and self.closest distance is strip[]
   #We are using an inbuilt function here but we could actually not do that and not even define a new min
   #This is because we are just comparing two variables
   #So we could have if min_a<min_b:   return min_a else:  return min_b (an if else statement)
   #we could do this for all comparisons where we use min.

   return min(min_a,min_b)

#The main function that finds the smallest distance.
#Performs the initial sorting and makes the initial invocation (the execution of a program or function) of the recursive
function.

#Deepcopy: It constructs a new compound object. As in objects that contain other objects.
#It then recursively inserts these copies into the variable that we .deepcopy to.
#It basically inserts into it what was found in the original

def closest(P, n):
   P.sort(key = lambda point: point.x)
   Q = copy.deepcopy(P)
   Q.sort(key = lambda point: point.y)

   #Use recursive function smallestdistance() to find the smallest distance

   Return smallestdistance(P, Q, n)

#Driver code

P = [Point(4, 23), Point(21, 32), Point(4, 5), Point(7, 9), Point(42, 10), Point(37, 2)]
n = len(P)
print(closest(P, n))
```

Citations

I am adding all the links I referred to that helped with my understanding and solving of the question.

1. Divide and Conquer: Princeton
   https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pearson/05DivideAndConquer-2x2.pdf

3. Towards Data Science: Algorithms
   https://towardsdatascience.com/basic-algorithms-finding-the-closest-pair-5fbef41e9d55

6. Finding the closest pair
   https://tildesites.bowdoin.edu/~ltoma/teaching/cs3250-CompGeom/spring17/Lectures/cg-closestPair.pdf

9. GFG Algorithms and Time Complexity
   https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/

Time complexity.

Let Time complexity of the above algorithm be T(n).

Let us assume that we use an O(nLogn) sorting algorithm. The code uses quicksort which can be O(n^2) in the worst case.
The above algorithm divides all points in two sets and recursively calls for two sets.
After dividing, it finds the strip in O(n) time, sorts the strip in O(nLogn) time and finally finds the closest points in strip in O(n) time.

So T(n) can expressed as follows

$T(n) = 2T(n/2) + O(n) + O(nLogn) + O(n)$
$T(n) = 2T(n/2) + O(nLogn)$
$T(n) = T(n \times Logn \times Logn)$

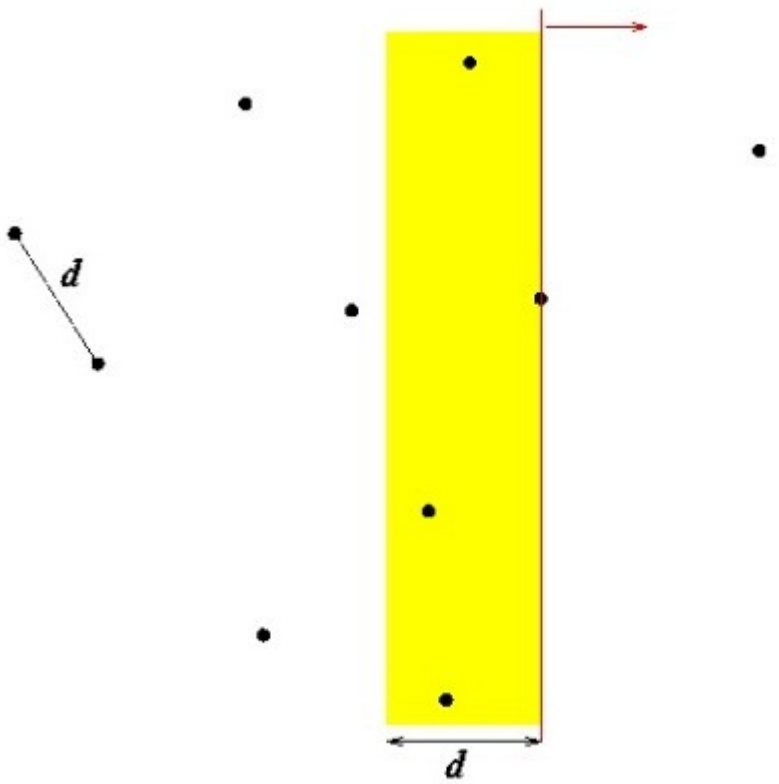
Proof Of Correctness for Closest Pair

Base Case

Let {p1,...,pn} be the set of input points sorted by their x-coordinates.
When the sweep line hits p2, then the pair (p1,p2) will be the current closest pair with distance d=dist(p1,p2).

Furthermore, we know that if p1 is one of the points that makes up the closest pair for the whole set, then the other point must be p2, since no other points in the set are closer to p1.

Let D be the strip of width d just to the left of the sweep line (where d is the current minimum distance).



Plane sweep technique for closest pair. Sweep line shown in red.

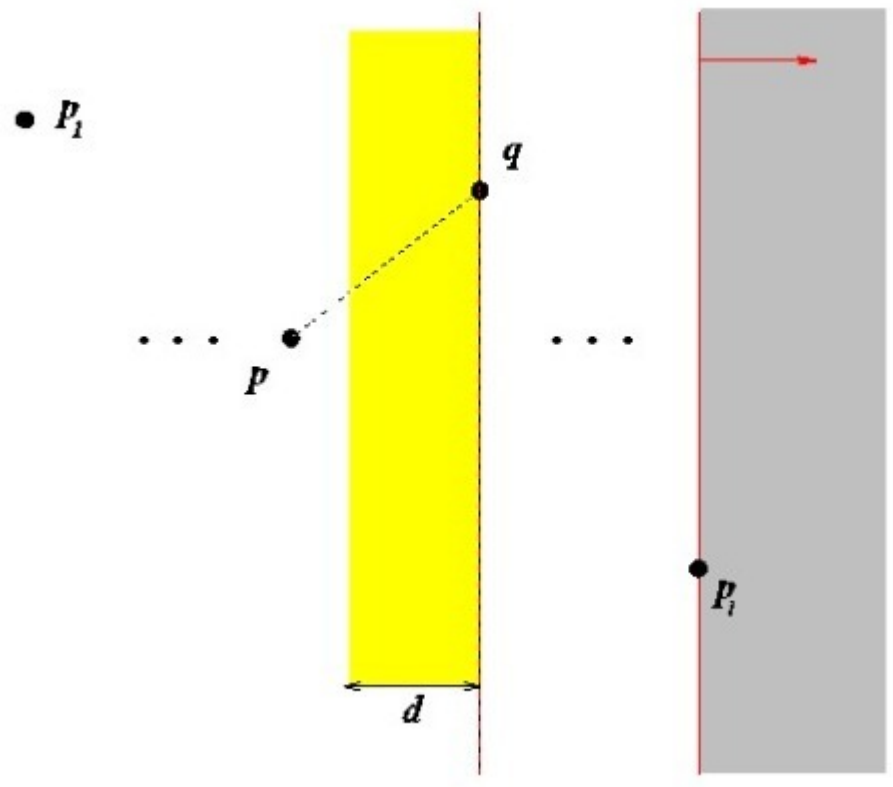
Induction Hypothesis

We will now prove that when the sweep completes processing any intermediate point pi, then the current d is the minimum distance between all of the points p1,...,pi.

We will prove this by assuming its opposite.

Induction Step

Suppose there are two points p and q to the left of pi such that dist(p,q) is less than d that the algorithm missed, and assume that q is further to the right (has a greater x-coordinate) than p then consider what occured when the sweep line was at q.
At this time the strip D was of width at least d (since it takes a smaller distance to change it).There are two cases for the position of p:



[CASE 1]: p is in the strip
In this case, p would have been within the bounding box checked by the algorithm (since it is within less than d from q as per our assumption), and hence would not have been missed.

[CASE 2]: p is not in the strip
in this case p must be to the left of the strip (since we assumed that it is to the left of q, but this would violate our assumption that p is less than d away from q.

Hence our assumption must be incorrect, there is no pair of points to the left of the sweep line less than d apart from each other.

Now, if we include the current point pi, the algorithm will check and see if there is a point to the left of the sweep line within d of it and update accordingly.
Hence when the sweep completes processing at a given point pi, then d is the distance between the closest pair among the points p1,...,pi.

Applying this result to the last point in the list shows that the algorithm is correct.

Citation for proof
https://www.cs.mcgill.ca/~cs251/ClosestPair/ClosestPairPS.html#proof
https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/