Finding the Closest Pair of Points

Problem: Given n ordered pairs (x_1 , y_1), (x_2 , y_2), ... , (x_n , y_n), find the distance between the two points in the set that are closest together.

The brute force algorithm is as follows:

Iterate through all possible pairs of points, calculating the distance between each of these pairs. Any time you see a distance shorter than the shortest distance seen, update the shortest distance seen.

Since computing the distance between two points takes O(1) time, and there are a total of $\frac{n(n-1)}{2} = \theta(n^2)$ distinct pairs of points, it follows that the running time of this algorithm is $\theta(n^2)$.

Can we do better?

Here's the idea:

1) Split the set of n points into two halves by a vertical line. (We can do this by sorting all the points by their x-coordinate and then picking the middle point and drawing a vertical line just to the left or right of it.)
2) Recursively call the function to solve the problem on both sets of points.
3) Return the smaller of the two values.

What's the problem with this idea?

The problem is that the actual shortest distance between any two of the original points MIGHT BE between a point from the first set and a point in the second set! Consider this situation:

```
                                   |
*                          *       |  *                      *
                                   |
              A                             B
```

Here we have four points separated into groups A and B. Let the points be labeled 1, 2, 3 and 4 from left to right. Notice that the minimum of the distances between points 1 and 2 and then points 3 and 4 is NOT the actual shortest possible distance between points, since points 2 and 3 are much closer to one another.

Thus, we must adapt our approach:

In step 3, we can "save" the smaller of the two values, (we'll call this δ), then, we have to check to see if there are points (one in each group) that are closer than δ apart.

Do we need to search through all possible pairs of points from the two different sides?

Probably not. We must only consider points that are within a distance of δ to our dividing line.

```
                                                   δ
                                   |  --------------------------
*                          *       |  *                      *
                                   |
              A                             B
         ------------------------------------------------
                  δ                        δ
```

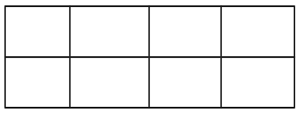
Still, however, one could construct a case where ALL the points on each side are within δ of the vertical line:

```
* | *
* | *
  |
* | *
* | *
```

So, technically speaking, this case is as bad as our original idea where we'd have to compare each pair of points to one another from the different groups.

But, wait, is it really necessary to compare each point on one side with every other point on every other side???

Consider the following rectangle around the dividing line that is constructed by eight δ/2 x δ/2 squares. Note that the diagonal of each square is $\delta/\sqrt{2}$**, which is less than δ. Since each square lies on a single side of the dividing line, at most one point lies in each box, because if two points were within a single box the distance between those two points would be less than δ, contradicting the definition of δ. This means, that there are at MOST 7 other points that could possibly be a distance of less than δ apart from a given point, that have a greater y coordinate than that point. (We assume that our point is on the bottom row of this grid; we draw the grid that way.)**



Now, we come to the issue of how do we know WHICH 7 points to compare a given point with???
The idea is as follows: as you are processing the points recursively, SORT them based on the y-coordinate. Then, for a given point within the strip, you only need to compare with the next seven points!

Here is a pseudocode version of the algorithm. (I have simplified the pseudocode from the book so that it's easier to get an overall understanding of the flow of the algorithm.)

```
closest_pair(p) {
 mergesort(p, 1, n) // n is number of points
 return rec_cl_pair(p, 1, 2)
}

rec_cl_pair(p, i, j) {

 if (j - i < 3)  { \\ If there are three points or less...
    mergesort(p, i, j)
    return shortest_distance(p[i], p[i+1], p[i+2])
 }

 xval = p[(i+j)/2].x
 deltaL = rec_cl_pair(p, i, (i+j)/2)
 deltaR = rec_cl_pair(p, (i+j)/2+1, j)
 delta = min(deltaL, deltaR)
 merge(p, i, k, j)

 v = vert_strip(p, xval, delta)

 for k=1 to size(v)-1
   for s = (k+1) to min(t, k+7)
     delta = min(delta, dist(v[k], v[s]))
 return delta
```

}

Let's trace through an example with the following 9 points:

(0, 0), (1, 6), (2, 8), (2, 3), (3, 4), (5, 1), (6, 7), (7, 4) and (8, 0).

These are already sorted these by the x-coordinate. We will split the points in half, with the first four in the first half and the last five in the last half.

So, we must recursively call rec_cl_pair on the set

{(0, 0), (1, 6), (2, 8), (2, 3)}

This results in two more recursive calls, to rec_cl_pair on the sets {(0, 0), (1, 6)} and {(2, 8), (2, 3)}. The first call simply returns the distance $\sqrt{37}$ and sorts the points by y-coordinate which leaves them unchanged. The second call returns 5 and swaps the order of the points (2, 3), (2, 8) in the array p.

Then, delta is set to 5, and the points are all merged by y coordinate: (0, 0), (2, 3), (1, 6), (2, 8). (All of these are then copied into the v array since all points are within a distance of 5 of the line x=1.)

Since we have less than 7 points, all points are compared and it is discovered that (1, 6) and (2, 8) are a distance of $\sqrt{5}$ apart.

Now, let's call rec_cl_pair on the set

{(3, 4), (5, 1), (6, 7), (7, 4), (8, 0)}

We first make two recursive calls to the sets: {(3, 4), (5, 1)} and {(6, 7), (7, 4), (8, 0)}. The first will result in deltaL being set to $\sqrt{13}$ and the points getting sorted by y: {(5, 1), (3, 4)}.

The second results in delta being set to $\sqrt{10}$ and the points getting sorted by y: {(8, 0), (7, 4), (6, 7)}. We then merge all of these points: {(8, 0), (5, 1), (3, 4), (7, 4), (6, 7)}. All of these points get copied into v.

Then, we discover that NO pair of points beats the original deltaR of $\sqrt{10}$. This is then the closest distance between any pair of points within the second set of data.

Finally, we must merge ALL the points together by y-coordinate:

(0, 0), (8, 0), (5, 1), (2, 3), (3, 4), (7, 4) , (1, 6), (6, 7), (2, 8)

this time, we only pick those points that are within $\sqrt{10}$ of the line x=2 to copy into v. These points are:

(0, 0), (5, 1), (2, 3), (3, 4), (1, 6), (2, 8)

Now, we scan through all pairs to discover that the shortest distance between any of the two points is $\sqrt{2}$.