Closest Pairs Problem

The next Divide-and-Conquer problem that we consider is an important one that comes from a family of problems in "computational geometry." The problem is very simple to state:

Given n distinct points in the plane, find the pair of points whose distance is a minimum.

Applications:

1. Air traffic control software might want to quickly determine which two of n aircraft are closest so as to identify those planes that deserve highest priority.

2. Grades of lumber are characterized by smoothness and frequency of knotholes. A lumber store might want to eliminate a line of lumber. It would make sense to determine which two lines are closest to being identical and eliminating one of these.

Naïve algorithm: Calculate all possible distances using the distance formula $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ and then choose the minimum distance.

Complexity of naïve algorithm:

First point—compute distance to n-1 other points
Second point—compute distance to n-2 other points
…
(n-1)st point—compute distance to 1 other point

This leads to a familiar sum: $1 + 2 + ... + n - 1 = \frac{(n-1)\cdot n}{2}$ which is $\Theta(n^2)$

Once again, the divide-and-conquer technique gives us a faster-than-quadratic solution to a problem that seems inherently quadratic. The basic idea is fairly simple, but the "patching" together of solutions to the subproblems requires a fairly sophisticated analysis.

The basic idea:

Divide: Find a vertical line that divides the set of points into two sets of size n/2, namely the points to the left of the line and those to the right. Using recursion, find the smallest distance d_l in the left group and the smallest distance d_r in the right group.

Put back together: Consider d_l and d_r and choose the minimum of these two values—call this value d. Unfortunately, d may not be the minimum for the original set because the minimum distance might involve a point in the left set and a point in the right set.

The challenge: We must determine if there is a pair of points, one from the left set and one from the right set, whose distance is less than d. If we can do this in **linear time**, then we would have a recurrence relation that looks like mergesort:

$$T(n) = 2T\left(n/2\right) + \Theta(n) = \ldots 2^k T\left(n/2^k\right) + \Theta(kn) = \Theta(n \log n)$$

Some preliminary steps:

The trivial case for the recursion is three points or less because we want to make sure that in any division, there is a pair of points on each side of the dividing line.

How do we divide the points quickly into two groups? We'll sort (using an nlogn sort!) by x-coordinate.

It turns out that it will also be helpful to maintain the points sorted by y-coordinate as well—we'll see why soon. So we'll also sort by y-coordinate, keeping that list of points in a second array.

What points in the left and right sets must we compare?

Suppose there are two points, one in the left set and one in the right set, whose distance is less than d. Then these two points must lie in the strip of width 2d centered on the dividing line. So when we piece the solution back together, we must consider only points in this strip. Of course, if most of the points are in fact in this strip, a brute-force approach might still be quadratic.

Note that if we test the points from “bottom-up”, which we can do since we have a list sorted by y-coordinate, we only need to check the points “above” the point we are testing. So naïvely, we have something that looks like nested loops:

```
for (i=1; i<=n-1; ++i)
  for (j=i+1; j<=n; ++j)
    Compare distance from point $p_i$ to $p_j$    //only if points are in the “strip”
                            //points are stored in increasing order of y-coordinate
```

Our job is to reduce the running time of these nested loops.

The key observation:

Suppose p_i is in the left set and suppose it is exactly d units away from the dividing line (worst case). Then any point that we need to compare p_i to must have a y-coordinate in the 2d-x-d rectangle, where p_i is the point on the lower-left corner. Let’s divide this rectangle into 8 subsquares, each of dimension d/2-x-d/2. Now, there can’t be more than one point in any of these smaller squares, because if there were, the distance between them would be less than d. (This is the pigeonhole principle coming into play here.) And since p_i is in one of the squares, there are at most only 7 other points for which we need to calculate distances. So the inner loop above becomes constant in length and the total running time of examining the points in the strip is $\Theta(n)$. (7n, to be precise).

A final subtlety:

The algorithmic complexity of this approach depends on needing lists of points, at each recursive call, that are sorted by x-coordinate and also by y-coordinate. The first thing to observe is that if we sort at each recursive call, the complexity of the algorithm goes up from $\Theta(n\log n)$ to $\Theta(n\log^2 n)$.

Here are the details of a familiar recurrence argument:

$$W(n) = 2W\left(n/2\right) + n\log n = 2[2W\left(n/2^2\right) + \frac{n}{2}\log\left(n/2\right)] + n\log n = 2^2 W\left(n/2^2\right) + n\log\left(n/2\right) + n\log n$$

=…

$$2^k W\left(n/2^k\right) + n\log\left(n/2^{k-1}\right) + n\log\left(n/2^{k-2}\right) + ... + n\log\left(n/2^1\right) + n\log n =$$

$$n\cdot 1 + n[1+2+...+(k-1)+k] = n + n\left[\frac{k(k+1)}{2}\right] = n + n\left[\frac{(\log n)(\log n+1)}{2}\right] = \Theta(n\log^2 n)$$

If we presort by x-coordinate, then it's easy to pass to the recursive subcalls the appropriate left and right sublists, still sorted by x-coordinate.

For example, suppose our list of points were:
X = [(1,10), (2,15), (3,5), (4,30), (5,3), (6,18), (7,20), (8,8)]

It is easy to see that when we take the left half and the right half and pass to the recursive routines, the lists are still sorted by x-coordinate.

We just need to make sure that we can also pass the recursive calls the appropriate "left-sided" points and the appropriate "right-sided" points, still sorted by y-coordinate. But we can do this in linear time by essentially performing the reverse of the merge operation:

Y = [(5,3), (3,5), (8,8), (1,10), (2,15), (6,18), (7,20), (4,30)]

Using 4 as the x-coordinate cutoff point for deciding "right versus left", we can in linear time produce the following two sets:

Y_L = [(3,5), (1,10), (2,15), (4,30)] and Y_R = [(5,3), (8,8), (6,18), (7,20)]

Finally, when we return from the recursive calls, we can examine the points in ascending y-coordinate order and create a new list which contains points only in the strip of width 2d. These are the points that would be examined in the nested loops considered previously. Of course, this can be done in linear time.