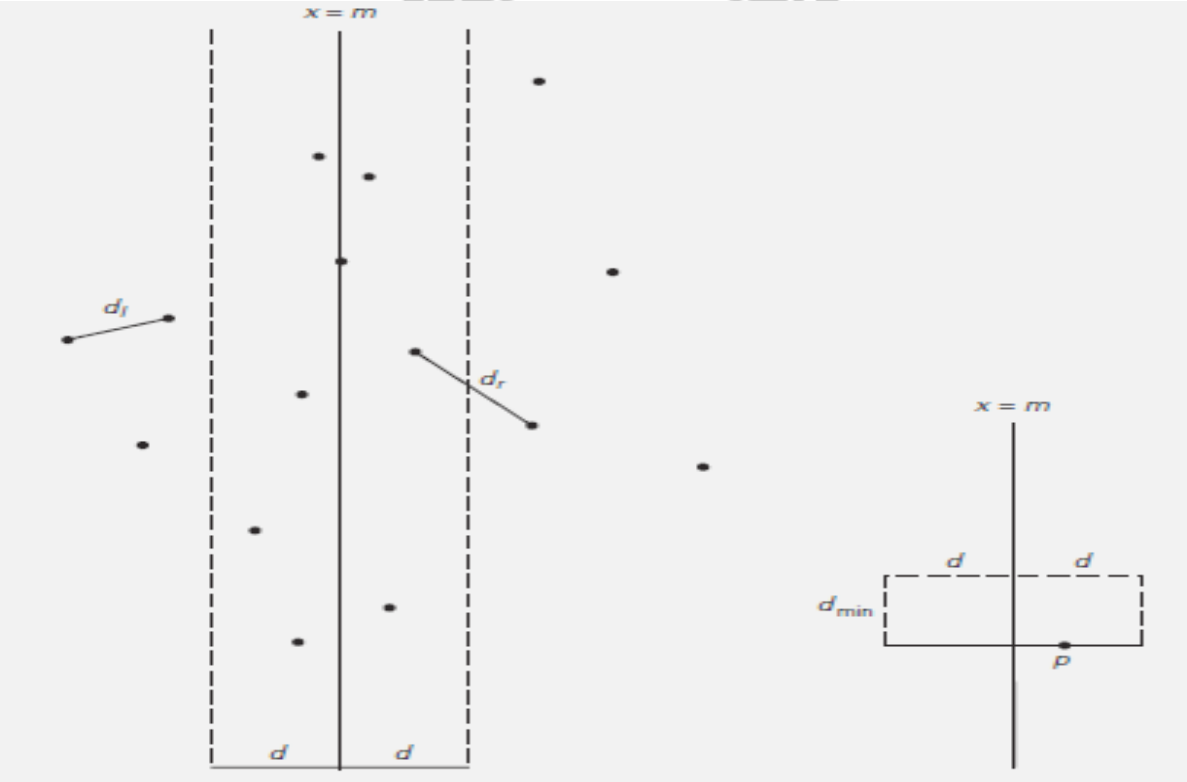
13. CLOSEST-PAIR AND CONVEX-HULL PROBLEMS.

The two-dimensional versions of the closest-pair problem and the convex-hull problem problems can be solved by brute-force algorithms in $\theta(n^2)$ and $O(n^3)$ time, respectively. The divide-and-conquer technique provides sophisticated and asymptotically more efficient algorithms to solve these problems.

The Closest-Pair Problem

Let P be a set of $n > 1$ points in the Cartesian plane. The points are ordered in non-decreasing order of their x coordinate. It will also be convenient to have the points sorted (by merge sort) in a separate list in non-decreasing order of the y coordinate and denote such a list by Q. If $2 \le n \le 3$, the problem can be solved by the obvious brute-force algorithm. If $n > 3$, we



can divide the points into two subsets Pl and Pr of $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ points, respectively, by drawing a vertical line through the median m of their x coordinates so that $\lceil n/2 \rceil$ points lie the left of or on the line itself, and $\lfloor n/2 \rfloor$ points lie to the right of or on the line. Then we can solve the closest- pair problem recursively for subsets P_l and P_r . Let d_l and d_r be the smallest distances between pairs of points in Pl and Pr, respectively, and let $d = \min\{d_l, d_r\}$.

FIGURE 2.13 (a) Idea of the divide-and-conquer algorithm for the closest-pair problem.

(a) Rectangle that may contain points closer than d_{min} to point p.

Note that d is not necessarily the smallest distance between all the point pairs because points of a closer pair can lie on the opposite sides of the separating line. Therefore, as a step combining the solutions to the smaller sub problems, we need to examine such points. Obviously, we can limit our attention to the points inside the symmetric vertical strip of width 2d around the separating line, since the distance between any other pair of points is at least d (Figure2.13a).

Let S be the list of points inside the strip of width 2d around the separating line, obtained from Q and hence ordered in non-decreasing order of their y coordinate. We will scan this list, updating the information about dmin, the minimum distance seen so far, if we encounter a closer pair of points. Initially, $d_{min} = d$, and subsequently $d_{min} \leq d$. Let p(x, y) be a point on this list.

For a point p (x, y) to have a chance to be closer to p than d_{min}, the point must follow p on list S and the difference between their y coordinates must be less than d_{min}.

Geometrically, this means that p must belong to the rectangle shown in Figure 2.13b. The principal insight exploited by the algorithm is the observation that the rectangle can contain just a few such points, because the points in each half (left and right) of the rectangle must be at least distance d apart.

It is easy to prove that the total number of such points in the rectangle, including p, does not exceed 8. A more careful analysis reduces this number to 6. Thus, the algorithm can consider no more than five next points following p on the list S, before moving up to the next point.

Here is pseudocode of the algorithm. We follow the advice given in to avoid computing square roots inside the innermost loop of the algorithm.

ALGORITHM *EfficientClosestPair(P, Q)*

```
//Solves the closest-pair problem by divide-and-conquer
//Input: An array P of n ≥ 2 points in the Cartesian plane sorted in non decreasing
//       order of their x coordinates and an array Q of the same points sorted in
//       non decreasing order of the y coordinates
//Output: Euclidean distance between the closest pair of points
if n ≤ 3
        return the minimal distance found by the brute-force algorithm
else
        copy the first ⌈n/2⌉ points of P to array Pl
        copy the same ⌈n/2⌉ points from Q to array
        Qlcopy the remaining ⌈n/2⌉ points of P to array
        Prcopy the same ⌈n/2⌉ points from Q to array Qr
        dl←EfficientClosestPair(Pl, Ql)
        dr←EfficientClosestPair(Pr, Qr)
        d ←min{dl, dr}
        m←P[⌈n/2⌉ − 1].x
        copy all the points of Q for which |x − m| < d into array S[0..num − 1]
        dminsq ←d2
        for i ←0 to num − 2 do
                k←i + 1
                while k ≤ num − 1 and (S[k].y− S[i].y)2< dminsq
                        dminsq ←min((S[k].x − S[i].x)2+ (S[k].y− S[i].y)2, dminsq)
                        k←k + 1
```


return ***sqrt(dminsq)***

The algorithm spends linear time both for dividing the problem into two problems half the size and combining the obtained solutions. Therefore, assuming as usual that n is a power of 2, we have the following recurrence for the running time of the algorithm:

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

where $f(n) \in \Theta(n)$. Applying the Master Theorem (with $a = 2$, $b = 2$, and $d = 1$), we get $T(n) \in \Theta(n \log n)$. The necessity to presort input points does not change the overall efficiency class if sorting is done by a $O(n \log n)$ algorithm such as merge sort. In fact, this is the best efficiency