Before we give a formal statement of the closest-pair algorithm, there are several technical points to resolve.

In order to terminate the recursion, we check the number of points in the input and if there are three or fewer points, we find a closest pair directly. Dividing the input and using recursion only if there are four or more points ensures that each of the two parts contains at least one pair of points and, therefore, that there is a closest pair in each part.

Before invoking the recursive procedure, we sort the entire set of points by $x$-coordinate. This makes it easy to divide the points into two nearly equal parts.

We use mergesort (see Section 5.3) to sort by $y$-coordinate. However, instead of sorting each time we examine points in the vertical strip, we assume as in mergesort that each half is sorted by $y$-coordinate. Then we simply merge the two halves to sort all of the points by $y$-coordinate.

We can now formally state the closest-pair algorithm. To simplify the description, our version outputs the distance between a closest pair but not a closest pair. We leave this enhancement as an exercise (Exercise 5).

### ALGORITHM 11.1.2 *Finding the Distance Between a Closest Pair of Points*

Input: $p_1, \ldots, p_n$ ($n \geq 2$ points in the plane)

Output: $\delta$, the distance between a closest pair of points

```
procedure closest_pair(p, n)
  sort p_1, ..., p_n by x-coordinate
  return(rec_cl_pair(p, 1, n))
end closest_pair

procedure rec_cl_pair(p, i, j)
  // The input is the sequence p_i, ..., p_j of points in the plane
  // sorted by x-coordinate.
  // At termination of rec_cl_pair, the sequence is sorted by
  // y-coordinate.
  // rec_cl_pair returns the distance between a closest pair
  // in the input.
  // Denote the x-coordinate of point p by p.x.
  // trivial case (3 or fewer points)
  if j - i < 3 then
    begin
    sort p_i, ..., p_j by y-coordinate
    directly find the distance δ between a closest pair
    return(δ)
    end
  // divide
  k := ⌊(i + j)/2⌋
  l := p_k.x
  δ_L := rec_cl_pair(p, i, k)
  δ_R := rec_cl_pair(p, k + 1, j)
  δ := min{δ_L, δ_R}
```

// $p_i, \ldots, p_k$ are now sorted by $y$-coordinate
// $p_{k+1}, \ldots, p_j$ are now sorted by $y$-coordinate
merge $p_i, \ldots, p_k$ and $p_{k+1}, \ldots, p_j$ by $y$-coordinate
// assume that the result of the merge is stored back in
// $p_i, \ldots, p_j$

// now $p_i, \ldots, p_j$ is sorted by $y$-coordinate

// store points in the vertical strip in $v$
$t := 0$
**for** $k := i$ **to** $j$ **do**
  **if** $p_k.x > l - \delta$ **and** $p_k.x < l + \delta$ **then**
    **begin**
    $t := t + 1$
    $v_t := p_k$
    **end**
// points in strip are $v_1, \ldots, v_t$
// look for closer pairs in strip
// compare each to next seven points
**for** $k := 1$ **to** $t - 1$ **do**
  **for** $s := k + 1$ **to** $\min\{t, k + 7\}$ **do**
    $\delta := \min\{\delta, \text{dist}(v_k, v_s)\}$
**return**($\delta$)
**end** *rec_cl_pair*

We show that the worst-case time of the closest-pair algorithm is $\Theta(n \lg n)$. The procedure *closest_pair* begins by sorting the points by $x$-coordinate. If we use an optimal sort (e.g., mergesort), the worst-case sorting time will be $\Theta(n \lg n)$. Next *closest_pair* invokes *rec_cl_pair*. We let $a_n$ denote the worst-case time of *rec_cl_pair* for input of size $n$. If $n > 3$, *rec_cl_pair* first invokes itself with input size $\lfloor n/2 \rfloor$ and $\lfloor (n+1)/2 \rfloor$. Each of merge, extracting the points in the strip, and checking the distances in the strip takes time $O(n)$. Thus we obtain the recurrence

$$a_n \le a_{\lfloor n/2 \rfloor} + a_{\lfloor (n+1)/2 \rfloor} + cn, \qquad n > 3.$$

This is the same recurrence that mergesort satisfies, so we conclude that *rec_cl_pair* has the same worst-case time $O(n \lg n)$ as mergesort. Since the worst-case time of the sorting of the points by $x$-coordinate is $\Theta(n \lg n)$ and the worst-case time of *rec_cl_pair* is $O(n \lg n)$, the worst-case time of *closest_pair* is $\Theta(n \lg n)$. In Section 11.2 we will show that any algorithm that finds a closest pair of points in the plane has worst-case time $\Omega(n \lg n)$; thus our algorithm is asymptotically optimal.

It can be shown (Exercise 10) that there are at most six points in the rectangle of Figure 11.1.2 when the base is included and the other sides are excluded. This result is the best possible since it is possible to place six points in the rectangle (Exercise 8). By considering the possible locations of the points in the rectangle, D. Lerner and R. Johnsonbaugh have shown that it suffices to compare each point in the strip with the next three points (rather than the next seven). This result is the best possible since checking the next two points does not lead to a correct algorithm (Exercise 7).

As another example, suppose that we have retained $p_1, \ldots, p_5$ (see Figure 11.3.9). This time, since $p_4, p_5, p$ make a right turn, we discard $p_5$. We then back up to examine $p_3, p_4, p$. Since these points also make a right turn, we discard $p_4$. We then back up to examine $p_2, p_3, p$. Since these points make a left turn, we retain $p_3$. We continue by examining the point following $p$. The pseudocode for Graham's Algorithm is given as Algorithm 11.3.6.

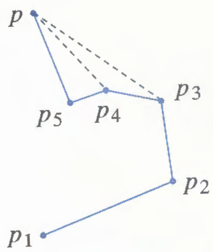


**FIGURE 11.3.9** A situation in the convex hull algorithm when point $p$ is examined. Before $p$ is examined, the convex hull of the points so far examined is $p_1, p_2, p_3, p_4, p_5$. Since $p_4, p_5, p$ make a right turn, $p_5$ is discarded. This leaves the points $p_3, p_4, p$, which also make a right turn; thus, $p_4$ is also discarded. This leaves the points $p_2, p_3, p$, which make a left turn; thus $p_3$ is retained. The current convex hull is $p_1, p_2, p_3, p$. The algorithm continues by examining the point following $p$.

### ALGORITHM 11.3.6 *Graham's Algorithm to Compute the Convex Hull*

This algorithm computes the convex hull of the points $p_1, \ldots, p_n$ in the plane. The $x$- and $y$-coordinates of the point $p$ are denoted $p.x$ and $p.y$, respectively.

Input: $p_1, \ldots, p_n$ and $n$

Output: $p_1, \ldots, p_k$ (the convex hull of $p_1, \ldots, p_n$) and $k$

```
procedure graham_scan(p, n, k)
  // trivial case
  if n = 1 then
    begin
    k := 1
    return
    end
  // find the point with minimum y-coordinate
  min := 1
  for i := 2 to n do
    if p_i.y < p_min.y then
      min := 1
  // Among all such points, find the one with minimum
  // x-coordinate
  for i := 1 to n do
    if p_i.y = p_min.y and p_i.x < p_min.x then
      min := i
  swap(p_1, p_min)
  // sort on angle from horizontal to p_1, p_i
  sort p_2, ..., p_n
  // p_0 is an extra point added to prevent the algorithm from
  // backing up forever
  p_0 := p_n
  // discard points not on the convex hull
  k := 2
  for i := 3 to n do
    begin
    while p_{k-1}, p_k, p_i do not make a left turn do
      // discard p_k
      k := k - 1
    k := k + 1
    swap(p_i, p_k)
    end
end graham_scan
```