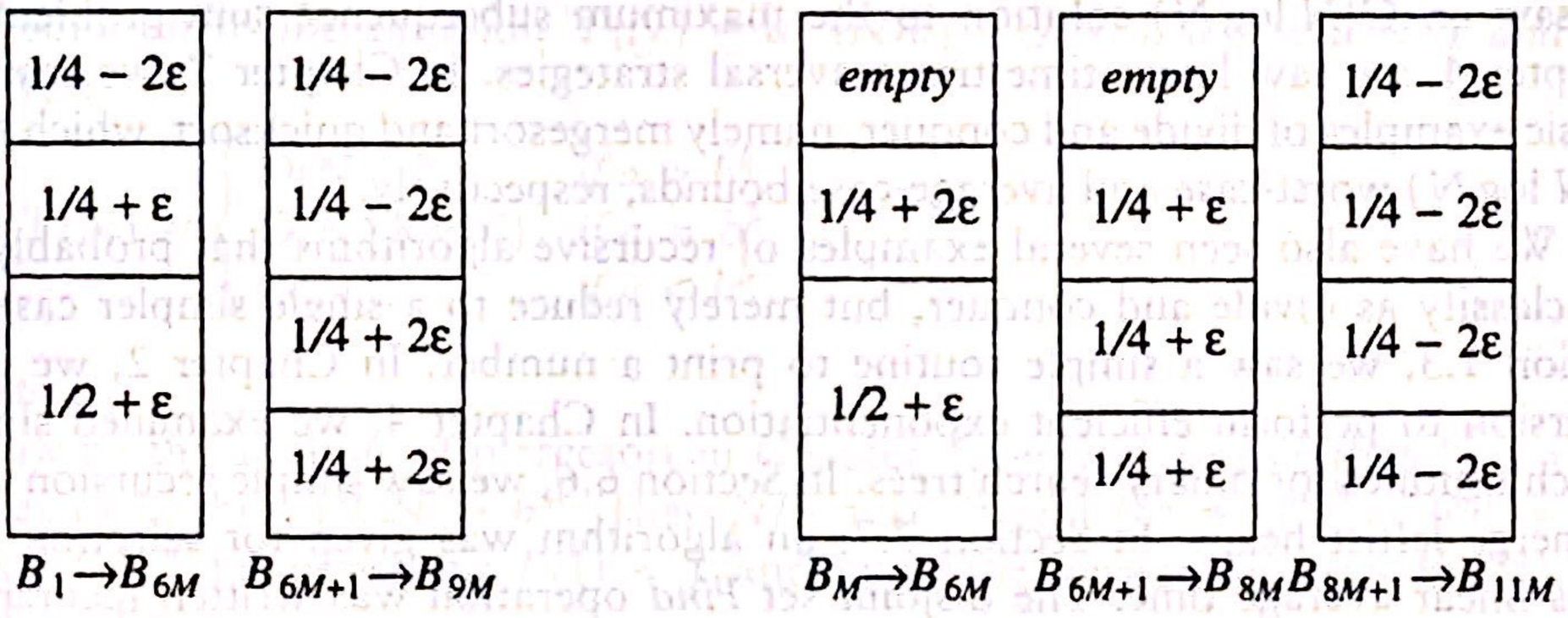


Figure 10.28 Example where first fit decreasing uses $11M$ bins, but only $9M$ bins are required

THEOREM 10.5.

Let M be the optimal number of bins required to pack a list I of items. Then first fit decreasing never uses more than $\frac{11}{9}M + 4$ bins. There exist sequences such that first fit decreasing uses $\frac{11}{9}M$ bins.

PROOF:

The upper bound requires a very complicated analysis. The lower bound is exhibited by a sequence consisting of $6M$ elements of size $\frac{1}{2} + \epsilon$, followed by $6M$ elements of size $\frac{1}{4} + 2\epsilon$, followed by $6M$ elements of size $\frac{1}{4} + \epsilon$, followed by $12M$ elements of size $\frac{1}{4} - 2\epsilon$. Figure 10.28 shows that the optimal packing requires $9M$ bins, but first fit decreasing uses $11M$ bins.

In practice, first fit decreasing performs extremely well. If sizes are chosen uniformly over the unit interval, then the expected number of extra bins is $\Theta(\sqrt{M})$. Bin packing is a fine example of how simple greedy heuristics can give good results.

10.2. Divide and Conquer

Another common technique used to design algorithms is *divide and conquer*. Divide and conquer algorithms consist of two parts:

Divide: Smaller problems are solved recursively (except, of course, base cases).

Conquer: The solution to the original problem is then formed from the solutions to the subproblems.

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. We generally insist that the subproblems be disjoint (that is, essentially nonoverlapping). Let us review some of the recursive algorithms that have been covered in this text.

We have already seen several divide and conquer algorithms. In Section 2.4.3, we saw an $O(N \log N)$ solution to the maximum subsequence sum problem. In Chapter 4, we saw linear-time tree traversal strategies. In Chapter 7, we saw the classic examples of divide and conquer, namely mergesort and quicksort, which have $O(N \log N)$ worst-case and average-case bounds, respectively.

We have also seen several examples of recursive algorithms that probably do not classify as divide and conquer, but merely reduce to a single simpler case. In Section 1.3, we saw a simple routine to print a number. In Chapter 2, we used recursion to perform efficient exponentiation. In Chapter 4, we examined simple search routines for binary search trees. In Section 6.6, we saw simple recursion used to merge leftist heaps. In Section 7.7, an algorithm was given for selection that takes linear average time. The disjoint set *Find* operation was written recursively in Chapter 8. Chapter 9 showed routines to recover the shortest path in Dijkstra's algorithm and other procedures to perform depth-first search in graphs. None of these algorithms are really divide and conquer algorithms, because only one recursive call is performed.

We have also seen, in Section 2.4, a very bad recursive routine to compute the Fibonacci numbers. This could be called a divide and conquer algorithm, but it is terribly inefficient, because the problem really is not divided at all.

In this section, we will see more examples of the divide and conquer paradigm. Our first application is a problem in *computational geometry*. Given N points in a plane, we will show that the closest pair of points can be found in $O(N \log N)$ time. The exercises describe some other problems in computational geometry which can be solved by divide and conquer. The remainder of the section shows some extremely interesting, but mostly theoretical, results. We provide an algorithm which solves the selection problem in $O(N)$ worst-case time. We also show that 2 N-bit numbers can be multiplied in $o(N^2)$ operations and that two $N \times N$ matrices can be multiplied in $o(N^3)$ operations. Unfortunately, even though these algorithms have better worst-case bounds than the conventional algorithms, none are practical except for very large inputs.

10.2.1. Running Time of Divide and Conquer Algorithms

All the efficient divide and conquer algorithms we will see divide the problems into subproblems, each of which is some fraction of the original problem, and then perform some additional work to compute the final answer. As an example, we have seen that mergesort operates on two problems, each of which is half the size of the original, and then uses $O(N)$ additional work. This yields the running time equation (with appropriate initial conditions)

$$T(N) = 2T(N/2) + O(N)$$

We saw in Chapter 7 that the solution to this equation is $O(N \log N)$. The following theorem can be used to determine the running time of most divide and conquer algorithms.

THEOREM 10.6.

The solution to the equation $T(N) = aT(N/b) + \Theta(N^k)$, *where* $a \geq 1$ *and* $b > 1$, *is*

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{if } a > b^k \\ O(N^k \log N) & \text{if } a = b^k \\ O(N^k) & \text{if } a < b^k \end{cases}$$

PROOF:

Following the analysis of mergesort in Chapter 7, we will assume that N is a power of b; thus, let $N = b^m$. Then $N/b = b^{m-1}$ and $N^k = (b^m)^k = b^{mk} = b^{km} = (b^k)^m$. Let us assume $T(1) = 1$, and ignore the constant factor in $\Theta(N^k)$. Then we have

$$T(b^m) = aT(b^{m-1}) + (b^k)^m$$

If we divide through by a^m, we obtain the equation

$$\frac{T(b^m)}{a^m} = \frac{T(b^{m-1})}{a^{m-1}} + \left\{\frac{b^k}{a}\right\}^m \tag{10.3}$$

We can apply this equation for other values of m, obtaining

$$\frac{T(b^{m-1})}{a^{m-1}} = \frac{T(b^{m-2})}{a^{m-2}} + \left\{\frac{b^k}{a}\right\}^{m-1} \tag{10.4}$$

$$\frac{T(b^{m-2})}{a^{m-2}} = \frac{T(b^{m-3})}{a^{m-3}} + \left\{\frac{b^k}{a}\right\}^{m-2} \tag{10.5}$$

$$\cdots$$

$$\frac{T(b^1)}{a^1} = \frac{T(b^0)}{a^0} + \left\{\frac{b^k}{a}\right\}^1 \tag{10.6}$$

We use our standard trick of adding up the telescoping equations (10.3) through (10.6). Virtually all the terms on the left cancel the leading terms on the right, yielding

$$\frac{T(b^m)}{a^m} = 1 + \sum_{i=1}^{m} \left\{\frac{b^k}{a}\right\}^i \tag{10.7}$$

$$= \sum_{i=0}^{m} \left\{\frac{b^k}{a}\right\}^i \tag{10.8}$$

Thus

$$T(N) = T(b^m) = a^m \sum_{i=0}^{m} \left\{\frac{b^k}{a}\right\}^i \tag{10.9}$$

If $a > b^k$, then the sum is a geometric series with ratio smaller than 1. Since the sum of infinite series would converge to a constant, this finite sum is also bounded by a constant, and thus Equation (10.10) applies:

$$T(N) = O(a^m) = O(a^{\log_b N}) = O(N^{\log_b a}) \tag{10.10}$$

If $a = b^k$, then each term in the sum is 1. Since the sum contains $1 + \log_b N$ terms and $a = b^k$ implies that $\log_b a = k$,

$$\begin{aligned} T(N) &= O(a^m \log_b N) = O(N^{\log_b a} \log_b N) = O(N^k \log_b N) \\ &= O(N^k \log N) \end{aligned} \tag{10.11}$$

Finally, if $a < b^k$, then the terms in the geometric series are larger than 1, and the second formula in Section 1.2.3 applies. We obtain

$$T(N) = a^m \frac{(b^k/a)^{m+1} - 1}{(b^k/a) - 1} = O(a^m (b^k/a)^m) = O((b^k)^m) = O(N^k) \tag{10.12}$$

proving the last case of the theorem.

As an example, mergesort has $a = b = 2$ and $k = 1$. The second case applies, giving the answer $O(N \log N)$. If we solve three problems, each of which is half the original size, and combine the solutions with $O(N)$ additional work, then $a = 3$, $b = 2$ and $k = 1$. Case 1 applies here, giving a bound of $O(N^{\log_2 3}) = O(N^{1.59})$. An algorithm that solved three half-sized problems, but required $O(N^2)$ work to merge the solution, would have an $O(N^2)$ running time, since the third case would apply.

There are two important cases that are not covered by Theorem 10.6. We state two more theorems, leaving the proofs as exercises. Theorem 10.7 generalizes the previous theorem.

THEOREM 10.7.

The solution to the equation $T(N) = aT(N/b) + \Theta(N^k \log^p N)$, *where* $a \geq 1$, $b > 1$, *and* $p \geq 0$ *is*

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{if } a > b^k \\ O(N^k \log^{p+1} N) & \text{if } a = b^k \\ O(N^k \log^p N) & \text{if } a < b^k \end{cases}$$

THEOREM 10.8.

If $\sum_{i=1}^{k} \alpha_i < 1$, *then the solution to the equation* $T(N) = \sum_{i=1}^{k} T(\alpha_i N) + O(N)$ *is* $T(N) = O(N)$.

10.2.2. Closest-Points Problem

The input to our first problem is a list P of points in a plane. If $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, then the Euclidean distance between p_1 and p_2 is $[(x_1 - x_2)^2 + (y_1 - y_2)^2]^{1/2}$. We are required to find the closest pair of points. It is possible that two points have the same position; in that case that pair is the closest, with distance zero.

If there are N points, then there are $N(N-1)/2$ pairs of distances. We can check all of these, obtaining a very short program, but at the expense of an $O(N^2)$ algorithm. Since this approach is just an exhaustive search, we should expect to do better.

Let us assume that the points have been sorted by x coordinate. At worst, this adds $O(N \log N)$ to the final time bound. Since we will show an $O(N \log N)$ bound for the entire algorithm, this sort is essentially free, from a complexity standpoint.

Figure 10.29 shows a small sample point set P. Since the points are sorted by x coordinate, we can draw an imaginary vertical line that partitions the point set into two halves, P_L and P_R. This is certainly simple to do. Now we have almost exactly the same situation as we saw in the maximum subsequence sum problem in Section 2.4.3. Either the closest points are both in P_L, or they are both in P_R, or one is in P_L and the other is in P_R. Let us call these distances d_L, d_R, and d_C. Figure 10.30 shows the partition of the point set and these three distances.

We can compute d_L and d_R recursively. The problem, then, is to compute d_C. Since we would like an $O(N \log N)$ solution, we must be able to compute d_C with only $O(N)$ additional work. We have already seen that if a procedure consists of two half-sized recursive calls and $O(N)$ additional work, then the total time will be $O(N \log N)$.

Let $\delta = \min(d_L, d_R)$. The first observation is that we only need to compute d_C if d_C improves on δ. If d_C is such a distance, then the two points that define d_C must be within δ of the dividing line; we will refer to this area as a *strip*. As shown in Figure 10.31, this observation limits the number of points that need to be considered (in our case, $\delta = d_R$).

There are two strategies that can be tried to compute d_C. For large point sets that are uniformly distributed, the number of points that are expected to be in the strip is very small. Indeed, it is easy to argue that only $O(\sqrt{N})$ points are in the strip on average. Thus, we could perform a brute force calculation on these points in $O(N)$ time. The pseudocode in Figure 10.32 implements this strategy, assuming the C convention that the points are indexed starting at 0.

Figure 10.29 A small point set

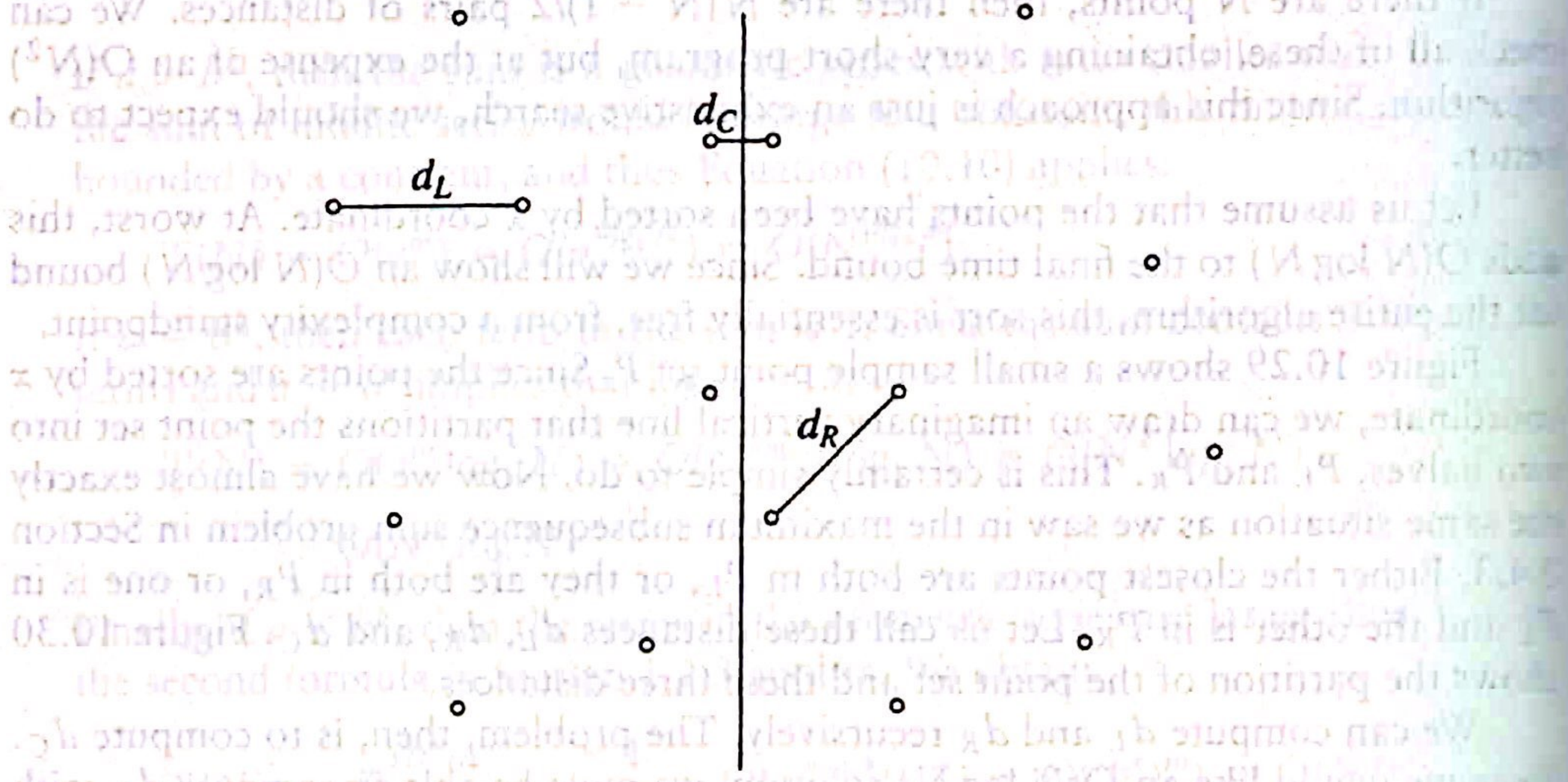


Figure 10.30 P partitioned into P_L and P_R; shortest distances are shown

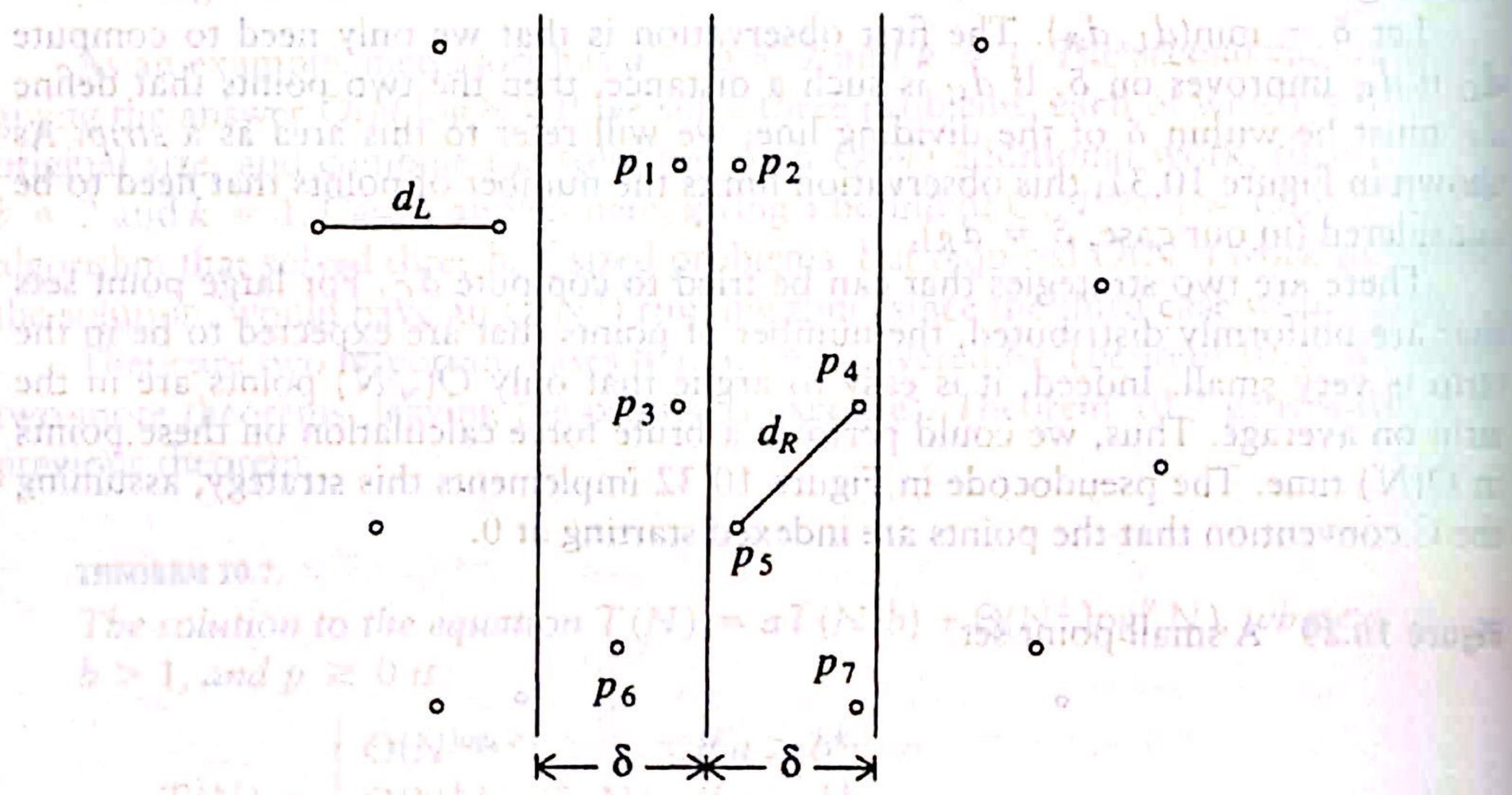


Figure 10.31 Two-lane strip, containing all points considered for d_C strip

```
/* Points are all in the strip */

for( i = 0; i < NumPointsInStrip; i++ )
    for( j = i + 1; j < NumPointsInStrip; j++ )
        if( Dist(Pi, Pj) < δ )
            δ = Dist(Pi, Pj);
```

Figure 10.32 Brute force calculation of $min(\delta, D_C)$

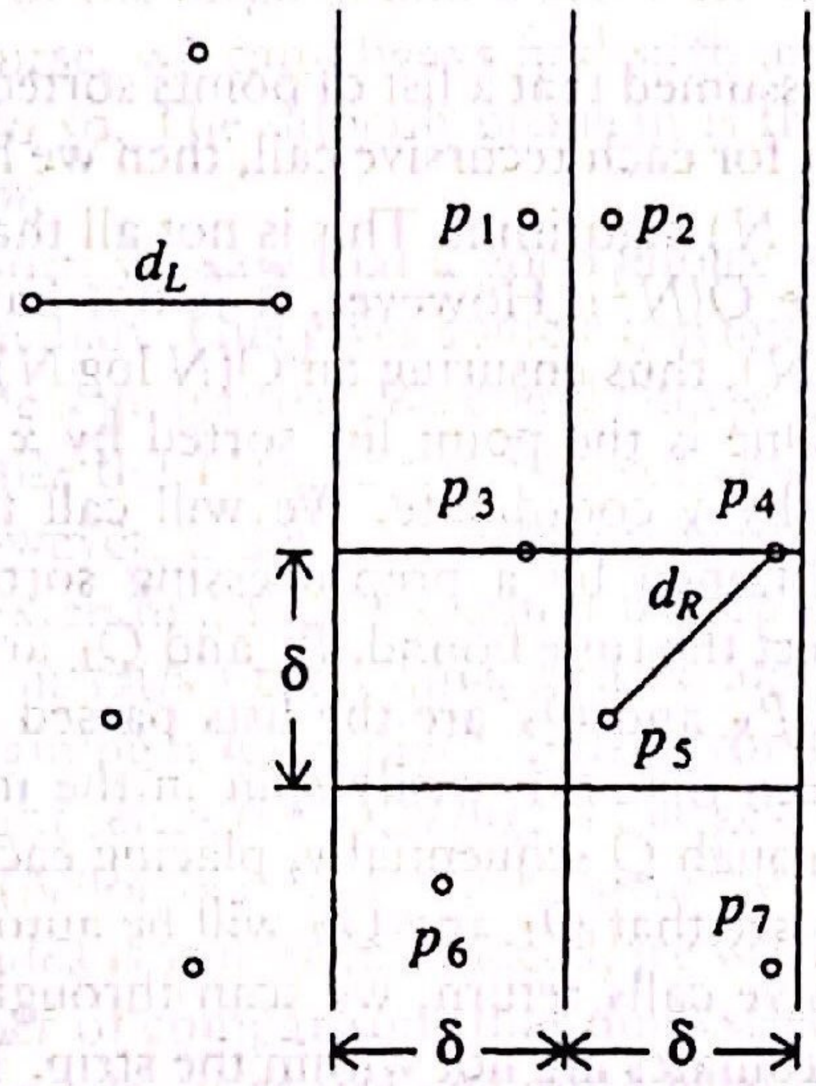
```
/* Points are all in the strip and sorted by y coordinate */

for( i = 0; i < NumPointsInStrip; i++ )
    for( j = i + 1; j < NumPointsInStrip; j++ )
        if( P_i and P_i 's coordinates differ by more than δ  )
            break;          /* Go to next P_i . */
        else
        if( Dist(P_i, P_j) < δ )
            δ = Dist(P_i, P_j);
```

Figure 10.33 Refined calculation of $min(\delta, D_C)$

In the worst case, all the points could be in the strip, so this strategy does not always work in linear time. We can improve this algorithm with the following observation: The y coordinates of the two points that define d_C can differ by at most δ. Otherwise, $d_C > \delta$. Suppose that the points in the strip are sorted by their y coordinates. Therefore, if p_i and p_j's y coordinates differ by more than δ, then we can proceed to p_{i+1}. This simple modification is implemented in Figure 10.33.

This extra test has a significant effect on the running time, because for each p_i only a few points p_j are examined before p_i's and p_j's y coordinates differ by more than δ and force an exit from the inner *for* loop. Figure 10.34 shows, for instance, that for point p_3, only the two points p_4 and p_5 lie in the strip within δ vertical distance.

Figure 10.34 Only p_4 and p_5 are considered in the second *for* loop



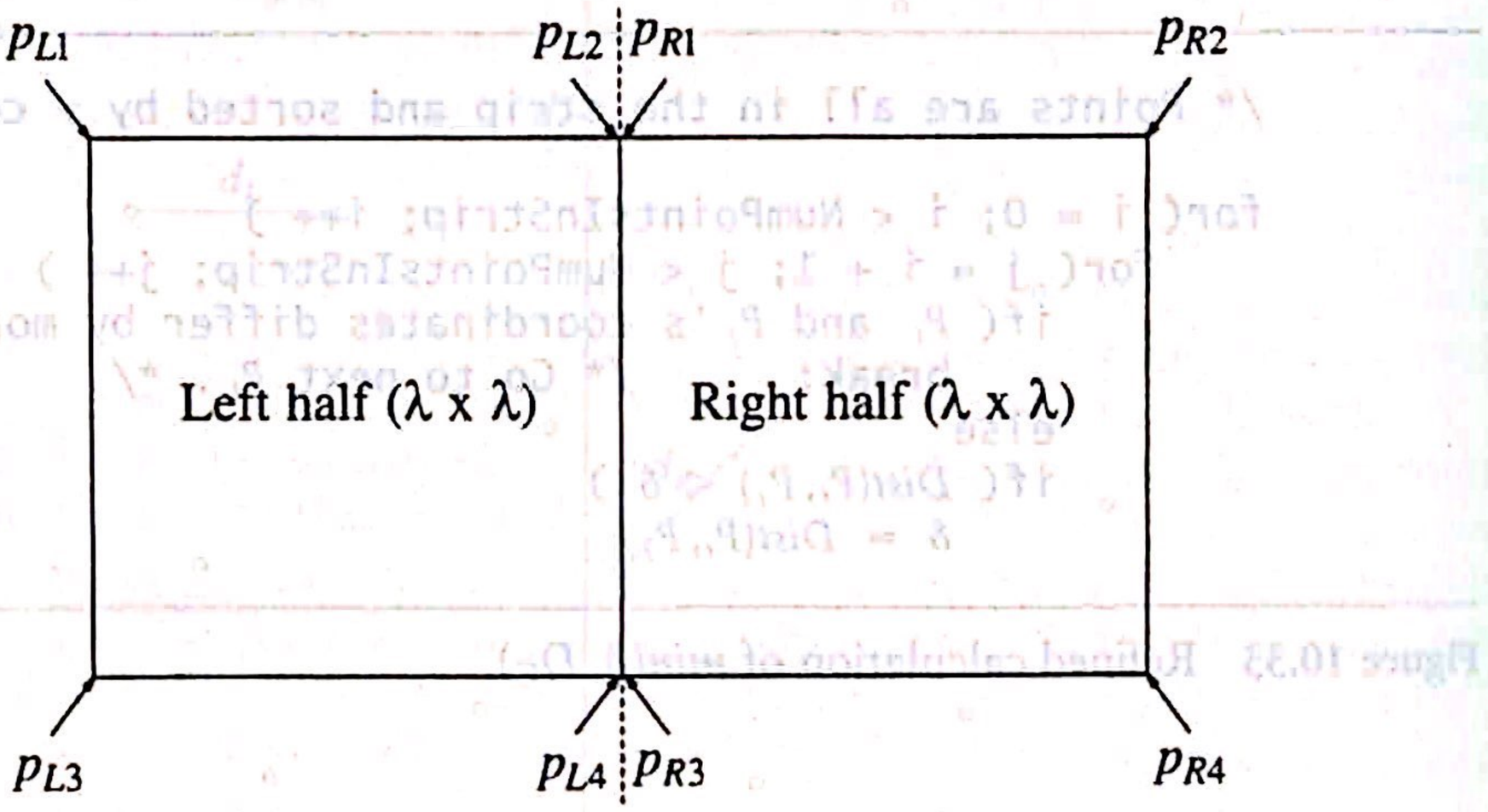


Figure 10.35 At most eight points fit in the rectangle; there are two coordinates shared by two points each

In the worst case, for any point p_i, at most 7 points p_j are considered. This is because these points must lie either in the δ by δ square in the left half of the strip or in the δ by δ square in the right half of the strip. On the other hand, all the points in each δ by δ square are separated by at least δ. In the worst case, each square contains four points, one at each corner. One of these points is p_i, leaving at most seven points to be considered. This worst-case situation is shown in Figure 10.35. Notice that even though p_{L2} and p_{R1} have the same coordinates, they could be different points. For the actual analysis, it is only important that the number of points in the λ by 2λ rectangle be $O(1)$, and this much is certainly clear.

Because at most seven points are considered for each p_i, the time to compute a d_C that is better than δ is $O(N)$. Thus, we appear to have an $O(N \log N)$ solution to the closest-points problem, based on the two half-sized recursive calls plus the linear extra work to combine the two results. However, we do not quite have an $O(N \log N)$ solution yet.

The problem is that we have assumed that a list of points sorted by y coordinate is available. If we perform this sort for each recursive call, then we have $O(N \log N)$ extra work: this gives an $O(N \log^2 N)$ algorithm. This is not all that bad, especially when compared to the brute force $O(N^2)$. However, it is not hard to reduce the work for each recursive call to $O(N)$, thus ensuring an $O(N \log N)$ algorithm.

We will maintain two lists. One is the point list sorted by x coordinate, and the other is the point list sorted by y coordinate. We will call these lists P and Q, respectively. These can be obtained by a preprocessing sorting step at cost $O(N \log N)$ and thus does not affect the time bound. P_L and Q_L are the lists passed to the left-half recursive call, and P_R and Q_R are the lists passed to the right-half recursive call. We have already seen that P is easily split in the middle. Once the dividing line is known, we step through Q sequentially, placing each element in Q_L or Q_R as appropriate. It is easy to see that Q_L and Q_R will be automatically sorted by y coordinate. When the recursive calls return, we scan through the Q list and discard all the points whose x coordinates are not within the strip. Then Q contains