CHAPTER 5
Divide and Conquer

A **divide-and-conquer algorithm** proceeds as follows. If the problem is small, it is solved directly. If the problem is large, the problem is divided into two or more parts called *subproblems*. Each subproblem is then solved after which solutions to the subproblems are combined into a solution to the original problem. The divide-and-conquer technique is also used to solve the subproblems; that is, the subproblems are further divided into subproblems, which are divided into subproblems, and so on. Eventually, small problems result that can be solved directly. The solutions to the various subproblems are then combined into a solution to the original problem. *Recursion* is often used to solve a subproblem. As an example, an array of two or more elements can be sorted by using a divide-and-conquer algorithm in which the original array is divided into two parts. If either part consists of one element, that part is already sorted. Parts containing two or more elements are sorted recursively. Finally, the two sorted arrays are merged into a single sorted array. The sorting algorithm is called *mergesort* and is discussed in Section 5.2.

We begin in Section 5.1 by introducing the divide-and-conquer technique with a tiling problem. After discussing mergesort (Section 5.2), we turn to a geometry problem that has an elegant divide-and-conquer solution (Section 5.3). The chapter concludes with a divide-and-conquer algorithm for multiplying matrices (Section 5.4), which is asymptotically faster than the algorithm derived directly from the definition of matrix multiplication.

In succeeding chapters, we will again have occasion to use the divide-and-conquer technique (see, e.g., Section 6.2, Quicksort, and Section 6.5, Selection).

5.1 A Tiling Problem

A *right tromino*, hereafter called simply a *tromino*, is an object made up of three 1×1 squares, as shown in Figure 5.1.1. We call an $n \times n$ board, with one 1×1 square (on the unit grid lines) removed, a *deficient board*

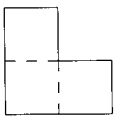


Figure 5.1.1 A tromino.

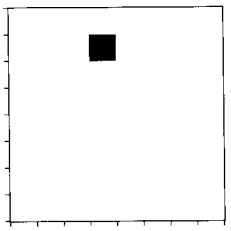


Figure 5.1.2 A deficient 8×8 board. The missing square is shown in black. The gap between successive marks along the sides is one unit.

(see Figure 5.1.2). Our tiling problem can then be stated as follows: Given a deficient $n \times n$ board, where n is a power of 2, tile the board with trominoes. By a *tiling* of the board with trominoes, we mean an exact covering of the board by trominoes without having any of the trominoes overlap each other or extend outside the board.

Example 5.1.1. Figure 5.1.3 shows a tiling of a deficient 8×8 board with trominoes.

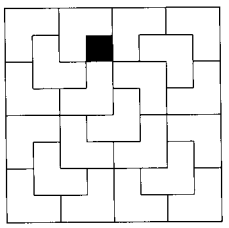


Figure 5.1.3 A tiling of a deficient 8×8 board with trominoes. □

Suppose that we are given a deficient $n \times n$ board, where n is a power of 2. If $n = 2$, we can tile the board because the board *is* a tromino (see Figure 5.1.1). Suppose that $n > 2$. A divide-and-conquer approach to solving the tiling problem begins by dividing the original problem (tile the $n \times n$ board) into subproblems (tile smaller boards). We divide the original board into four $n/2 \times n/2$ subboards [see Figure 5.1.4(a)]. Since n is a power of 2, $n/2$ is also a power of 2. The subboard that contains the missing square [in Figure 5.1.4(a), the upper-left subboard] is a deficient $n/2 \times n/2$ subboard, so we can recursively tile it. The other three $n/2 \times n/2$ subboards are not deficient, so we cannot directly recursively tile these subboards. However, if we place a tromino as shown in Figure 5.1.4(b) so that each of its 1×1 squares lies in

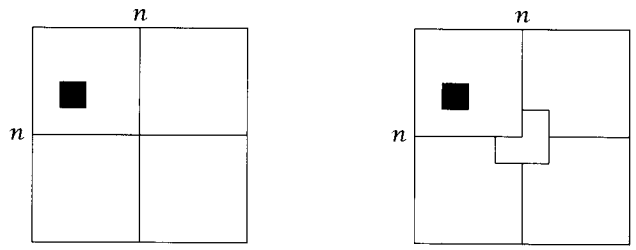


Figure 5.1.4 Using divide and conquer to tile a deficient $n \times n$ board with trominoes. In (a), the original $n \times n$ board is divided into four $n/2 \times n/2$ subboards. The subboard containing the missing square is then tiled recursively. A tromino is placed as shown in (b) so that each of its 1×1 squares lies in one of the three remaining subboards. These 1×1 squares are then considered as missing. The remaining subboards are then tiled recursively.

one of the three remaining subboards, we can consider each of these 1×1 squares as missing in the remaining subboards. We can then recursively tile these deficient subboards. Our tiling problem is solved.

Example 5.1.2. Figure 5.1.5 shows how our algorithm tiles a deficient 4×4 board.

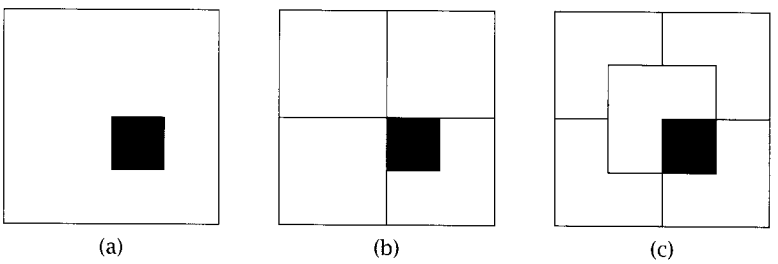


Figure 5.1.5 Tiling the deficient 4×4 board shown in (a). First, the board is divided into four 2×2 subboards as shown in (b). The subboard that contains the missing square is recursively tiled; in this case, the deficient 2×2 board is a tromino. Next, we place a tromino as shown in (c) so that each of its 1×1 squares lies in one of the three remaining subboards. Each of these 1×1 squares is considered as missing in the remaining subboards. We can then recursively tile these deficient subboards. Again, each of the deficient 2×2 boards is a tromino, so the problem is solved. □

Example 5.1.3. Figure 5.1.6 (next page) shows how our algorithm tiles a deficient 8×8 board. □

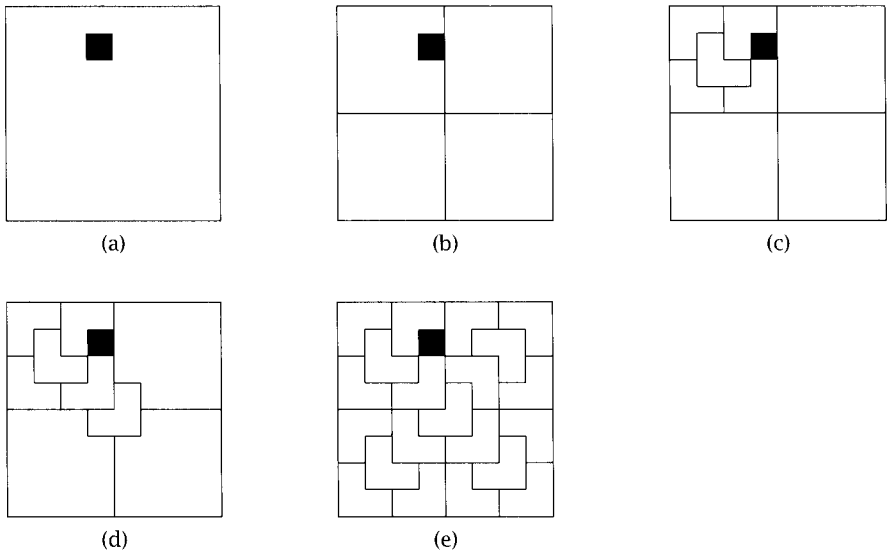


Figure 5.1.6 Tiling the deficient 8×8 board shown in (a). First, the board is divided into four 4×4 subboards as shown in (b). The subboard that contains the missing square is recursively tiled as shown in (c). Next, we place a tromino as shown in (d) so that each of its 1×1 squares lies in one of the three remaining subboards. Each of these 1×1 squares is considered as missing in the remaining subboards. We can then recursively tile each of these deficient 4×4 subboards as shown in (e). The problem is solved.

$\mathcal{WWW}$ We formally state our tiling algorithm as Algorithm 5.1.4.

Algorithm 5.1.4 Tiling a Deficient Board with Trominoes. This algorithm constructs a tiling by trominoes of a deficient $n \times n$ board where n is a power of 2.

Input Parameters: n, a power of 2 (the board size);
the location L of the missing square
Output Parameters: None

```
tile(n, L) {
  if (n == 2) {
    // the board is a right tromino T
    tile with T
    return
  }
  divide the board into four n/2 × n/2 subboards
  place one tromino as in Figure 5.1.4(b)
  // each of the 1 × 1 squares in this tromino is considered as missing
  let m1, m2, m3, m4 denote the locations of the missing squares
```

tile$(n/2, m_1)$
tile$(n/2, m_2)$
tile$(n/2, m_3)$
tile$(n/2, m_4)$
}

In Algorithm 5.1.4, "tile with T" can be interpreted in many ways. It could mean printing the location and orientation of T, or it could mean drawing T using a graphics system (see Exercises 5.1 and 5.2). In any case, we assume that "tile with T" takes constant time. We also assume that dividing the board, placing the tromino as in Figure 5.1.4(b), and computing m_1, m_2, m_3, m_4 each takes constant time. It follows that the time required by Algorithm 5.1.4 is proportional to the number of trominoes placed on the board. Since the number of 1×1 squares on a deficient $n \times n$ board is $n^2 - 1$ and each tromino occupies three squares, Algorithm 5.1.4 places

$$\frac{n^2 - 1}{3} = \Theta(n^2)$$

trominoes on the board. Therefore, the time required by Algorithm 5.1.4 is $\Theta(n^2)$.

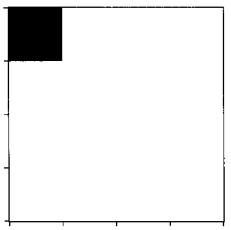
If we can tile a deficient $n \times n$ board, where n is not necessarily a power of 2, then the number of squares, $n^2 - 1$, must be divisible by 3. Chu and Johnsonbaugh (see Chu, 1986) showed that the converse is true, except when n is 5. More precisely, if $n \neq 5$, any deficient $n \times n$ board can be tiled with trominoes if and only if 3 divides $n^2 - 1$ (see Exercises 11 and 12). Some deficient 5×5 boards can be tiled and some cannot (see Exercises 6 and 7).

Some real-world problems can be modeled as tiling problems. One example is the *VLSI layout problem*—the problem of packing many components on a computer chip (see Wong, 1986). (VLSI is short for Very Large Scale Integration.) The problem is to tile a rectangle of minimum area with the desired components. The components are sometimes modeled as rectangles and L-shaped figures similar to trominoes. In practice, other constraints are imposed such as the proximity of various components that must be interconnected and restrictions on the ratios of width to height of the resulting rectangle.

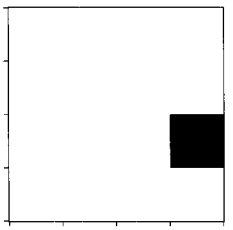
Exercises

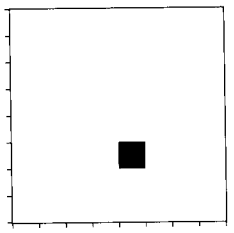
In Exercises 1–4, show how Algorithm 5.1.4 tiles the given deficient board.

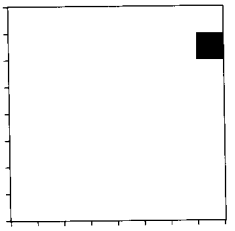
1S.



2.



3. 

4S. 

5S. Let c_n denote the time required by Algorithm 5.1.4. Write a recurrence relation and an initial condition for c_n. Show that $c_n = \Theta(n^2)$.

6. Give a tiling of a 5×5 board with trominoes in which the upper-left square is missing.

7. Show a deficient 5×5 board that is impossible to tile with trominoes. Prove that your board cannot be tiled with trominoes.

8S. Show how to tile with trominoes any $2i \times 3j$ board with no squares missing, where i and j are positive integers.

9. Show how to tile any deficient 7×7 board with trominoes.

10. Show how to tile any deficient 11×11 board with trominoes. *Hint:* Subdivide the board into overlapping 7×7 and 5×5 boards and two 6×4 boards. Then, use Exercises 6, 8, and 9.

11S. Write an algorithm that tiles any deficient $n \times n$ board with trominoes if n is odd, $n > 5$, and 3 divides $n^2 - 1$. *Hint:* Use the hint for Exercise 10.

12. Write an algorithm that tiles any deficient $n \times n$ board with trominoes if n is even, $n > 8$, and 3 divides $n^2 - 1$. *Hint:* Use Algorithm 5.1.4 with $n = 4$ and Exercises 8 and 11.

13. A *3D-septomino* is a three-dimensional $2 \times 2 \times 2$ cube with one $1 \times 1 \times 1$ corner cube removed. A *deficient cube* is an $n \times n \times n$ cube with one $1 \times 1 \times 1$ cube removed. Give an algorithm to tile a deficient $n \times n \times n$ cube with 3D-septominoes when n is a power of 2. (An unsolved problem is to determine which deficient cubes can be tiled with 3D-septominoes.)

A straight tromino *is an object made up of three squares in a row. Exercises 14–16 deal with straight trominoes.*

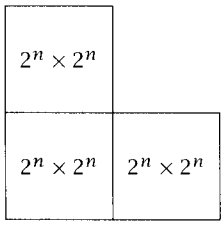
14S. Which deficient 4×4 boards can be tiled with straight trominoes?

15. Which deficient 5×5 boards can be tiled with straight trominoes?

16. Which deficient 8×8 boards can be tiled with straight trominoes?

17S. This exercise and the one that follows are due to Anthony Quas.

A $2^n \times 2^n$ *L-shape*, $n \geq 0$, is a figure of the form



with no missing squares. Write an algorithm to tile any $2^n \times 2^n$ L-shape with trominoes.

18. Use the preceding exercise to give a different algorithm to tile any $2^n \times 2^n$ deficient board with trominoes.

5.2 Mergesort

WWW **Mergesort** divides the array to be sorted into two nearly equal parts. Each part is then sorted. The two sorted halves are then *merged* into one sorted array. We begin by discussing the merge algorithm.

Example 5.2.1 Merging Two Sorted Arrays. Suppose that the goal is to merge the sorted arrays

| 14 | 20 | 36 |

| 10 | 12 | 30 | 40 | 44 |

.

We begin by examining the first element in each array:

| 14 | 20 | 36 |

| 10 | 12 | 30 | 40 | 44 |

.

Since $10 < 14$ and the arrays are *sorted*, 10 is the smallest element. It is copied into the output array

| 10 | ··· |

,

and we move to the next item in the second array

| 14 | 20 | 36 |

| 10 | 12 | 30 | 40 | 44 |

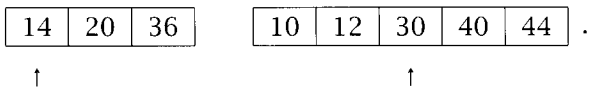
.

Since $12 < 14$, 12 is copied into the output array

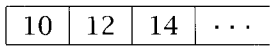
| 10 | 12 | ··· |

,

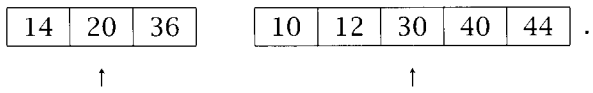
and we move to the next item in the second array



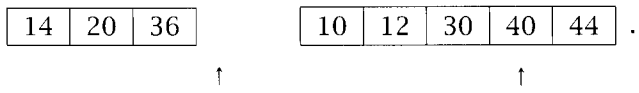
This time 30 > 14, so 14 is copied into the output array



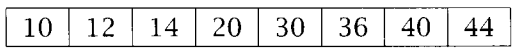
and we move to the next item in the first array



After the next values, 20, 30, and 36, are copied to the output array, we have



Since all of the data from the first array have been copied to the output array, we conclude by copying the remainder of the second array to the output array to obtain



The merge is complete. □

We state the merge algorithm as Algorithm 5.2.2.

Algorithm 5.2.2 Merge. This algorithm receives as input indexes i, m, and j, and an array a, where $a[i], \ldots, a[m]$ and $a[m+1], \ldots, a[j]$ are each sorted in nondecreasing order. These two nondecreasing subarrays are merged into a single nondecreasing array.

Input Parameters: a, i, m, j
Output Parameter: a

```
merge(a, i, m, j) {
  p = i // index in a[i], ..., a[m]
  q = m + 1 // index in a[m + 1], ..., a[j]
  r = i // index in a local array c
  while (p ≤ m && q ≤ j) {
    // copy smaller value to c
    if (a[p] ≤ a[q]) {
      c[r] = a[p]
      p = p + 1
    }
    else {
      c[r] = a[q]
```

```
        q = q + 1
      }
      r = r + 1
   }
   // copy remainder, if any, of first subarray to c
   while (p ≤ m) {
        c[r] = a[p]
        p = p + 1
        r = r + 1
   }
   // copy remainder, if any, of second subarray to c
   while (q ≤ j) {
        c[r] = a[q]
        q = q + 1
        r = r + 1
   }
   // copy c back to a
   for r = i to j
      a[r] = c[r]
}
```

In Algorithm 5.2.2, at every iteration of every while loop, an element of a is copied into c; thus, each element in the array a is accessed once. Thus, if a has n elements, the total time to execute the while loops is $\Theta(n)$. Also, the for loop executes in time $\Theta(n)$. Thus, the time to execute merge is $\Theta(n)$.

Having written the merge algorithm, mergesort is straightforward.

Algorithm 5.2.3 Mergesort. This algorithm sorts the array $a[i], \ldots, a[j]$ in nondecreasing order. It uses the merge algorithm (Algorithm 5.2.2).

Input Parameters: a, i, j
Output Parameter: a

```
mergesort(a, i, j) {
   // if only one element, just return
   if (i == j)
      return
   // divide a into two nearly equal parts
   m = (i + j)/2
   // sort each half
   mergesort(a, i, m)
   mergesort(a, m + 1, j)
   // merge the two sorted halves
   merge(a, i, m, j)
}
```

We next consider the worst-case time of mergesort (Algorithm 5.2.3). We let t_n denote the worst-case time for mergesort to sort an array of n elements.

Suppose that an array of size $n > 1$ is input to mergesort. After the array is divided into two nearly equal parts, the sizes of the two parts are $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. By definition, the worst-case times to sort these two halves are $t_{\lceil n/2 \rceil}$ and $t_{\lfloor n/2 \rfloor}$. Since the time to merge the two halves is $2n$ (counting the number of times an array element is copied in the merge algorithm), we obtain the recurrence relation

$$t_n = t_{\lceil n/2 \rceil} + t_{\lfloor n/2 \rfloor} + 2n.$$

An induction proof (see Exercise 12) shows that t_n is nondecreasing. Therefore,

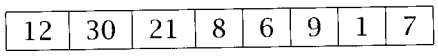
$$\begin{aligned} t_n &= t_{\lceil n/2 \rceil} + t_{\lfloor n/2 \rfloor} + 2n \\ &\leq t_{\lceil n/2 \rceil} + t_{\lceil n/2 \rceil} + 2n \\ &= 2t_{\lceil n/2 \rceil} + 2n. \end{aligned}$$

By the Main Recurrence Theorem (Theorem 2.4.7), $t_n = O(n \lg n)$. Similarly,

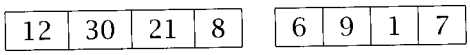
$$t_n \geq 2t_{\lfloor n/2 \rfloor} + 2n.$$

Again, by the Main Recurrence Theorem, $t_n = \Omega(n \lg n)$. Therefore, $t_n = \Theta(n \lg n)$, and the worst-case time of mergesort is $\Theta(n \lg n)$.

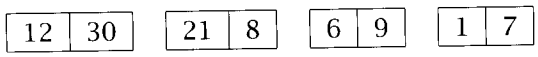
Example 5.2.4. We show how mergesort sorts the array



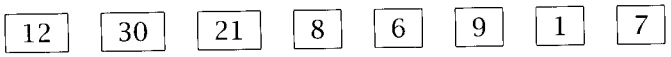
The array is first divided into two equal parts



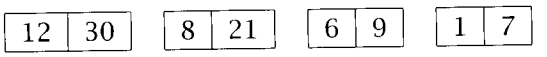
Each part is then sorted by mergesort. The process begins by dividing each part into equal parts



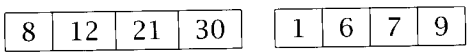
and then each of these parts into equal parts



This subdividing process now ends because each part contains only one item. Each pair is then merged



Each of these pairs is then merged



Finally these pairs are merged

1	6	7	8	9	12	21	30

to obtain the sorted array. □

Stable Sorts

Suppose that duplicates occur in the input to a sorting algorithm; specifically, suppose that in the input array a we have $a[i] = a[j]$ with $i < j$. Let i' be the index in the (sorted) output where $a[i]$ is located, and let j' be the index in the (sorted) output where $a[j]$ is located. The sorting algorithm is **stable** if for all such i and j, we have $i' < j'$. In words, a sorting algorithm is stable if the relative positions of items with duplicate keys are unchanged by the algorithm. Mergesort is stable because, when merge (Algorithm 5.2.2) encounters equal keys, it copies the one with the smaller index to the output array.

Stability may be important if the items to sort comprise several fields, but the key involves only some subset of the fields.

Example 5.2.5. Suppose that weather data are entered daily and that a record comprises the date and the maximum temperature for that date. Example data might be:

Mar 1; 51
Mar 2; 49
Mar 3; 52
Mar 4; 58
Mar 5; 52
Mar 6; 56
Mar 7; 52
Mar 8; 51

If the data are sorted in nondecreasing order of maximum temperature using a stable sort, the output would be

Mar 2; 49
Mar 1; 51
Mar 8; 51
Mar 3; 52
Mar 5; 52
Mar 7; 52
Mar 6; 56
Mar 4; 58

Because the sort is stable, dates with equal temperatures are listed in chronological order, which might be helpful in analyzing the data. □

In-Place Merge and Mergesort

Merge (Algorithm 5.2.2) receives an array that contains two sorted subarrays and merges them into a second array, which is then sorted. The reader may

wonder whether a second array is necessary. The answer is "No." Several versions of merge have been designed so that the merging is done in place. By "in place," we mean that only one extra cell (in addition to the input array) is allowed, together with $O(1)$ storage for handling array indexing. One of the most efficient in-place merge algorithms is due to Huang and Langston (see Huang, 1988). In practice, in-place merging is much slower than merging using an extra array.

In-place merging gives rise to an in-place version of mergesort. However, since the merging is slower than merging using an extra array, in practice, the resulting version of mergesort runs much slower than Algorithm 5.2.3.

An alternative to using an in-place version of merge in mergesort is to obtain an in-place version of mergesort directly (see, e.g., Katajainen, 1996). The key observation that makes a direct in-place version of mergesort possible is that if we have an array a consisting of a sorted subarray of size m followed by a sorted subarray of size n, then the subarrays can be merged into a using extra storage, say an array b, of size $\min\{m, n\}$. For example, if $m \leq n$, we copy the first m elements of a into b and then merge b and the last n elements of a into a using the standard merge algorithm (see Exercise 13).

Now suppose that we have an array of size n. The idea of in-place mergesort is to sort the first $n/2$ elements using the last part of the array as the extra storage. (When we do so, we have to be careful to *swap* data rather than overwrite data; for details, see Katajainen, 1996.) Next, we sort the $n/4$ elements following the first $n/2$ elements, using the last fourth of the array as the extra storage. We then merge the two sorted subarrays using the last fourth of the array as the extra storage. We then repeatedly sort half of the unsorted end of the array and merge it with the already sorted first part of the array, using the end of the array as the extra storage until the entire array is sorted. In practice, an optimized version of this algorithm runs much faster than in-place mergesort using in-place merging; however, experiments by Katajainen, et al., show that the optimized version is about 50 percent slower than mergesort implemented with a version of merge that uses a second array (as in Algorithm 5.2.3).

Exercises

Show how merge *merges the arrays in Exercises 1–4.*

1S. 14 24 27
17 26 54

2. 14 24 27 28 31 45 47 51
7 10 11 29 31

3. 7 10 11 29 31
47 50 71 79 101

4S. 7 10 11 29 31
9 28 71 79 101

Show how mergesort *sorts each array in Exercises 5–8.*

5S. 14 40 31 28 3 15 17 51

6. 3 14 15 17 28 31 40 51

7. 51 40 31 28 17 15 14 3

8S. 23 23 23 23 23 23 23 23

9S. Write a nonrecursive version of mergesort. Make your algorithm as efficient as you can.

10. Give a formal proof using mathematical induction that mergesort is stable.

11. Suppose we merge two sorted *linked* lists of sizes m and n by changing reference fields in the list. Show that the best-case time is $\Theta(\min\{m,n\})$ and the worst-case time is $\Theta(m+n)$. Give examples of input that produce the best-case time and the worst-case time.

12S. Let t_n be the worst-case time of mergesort. Show that t_n is nondecreasing.

13. Write an algorithm that, given an array a consisting of a sorted subarray of size m followed by a sorted subarray of size n, merges them into a using an extra array b of size $\min\{m,n\}$.

5.3 Finding a Closest Pair of Points

A lumber store stocks several different kinds of plywood. Each is classified according to several features (e.g., number of knotholes per square foot, smoothness). In Figure 5.3.1, the different kinds of plywood are plotted as points in the plane, where the x-coordinate measures the number of knotholes per square foot, and the y-coordinate measures the smoothness. Notice that the distance between two points gives a dissimilarity measure: If the distance is small, the types of plywood represented by the points are similar; if the distance is large, the types of plywood represented by the points are dissimilar. The store decides to reduce the number of different kinds of plywood in stock by identifying two types of plywood that are most similar and eliminating one of them. The problem then is to identify a closest pair of points and eliminate a type of plywood that corresponds to one of the points. (We say *a* closest pair since it is possible that several pairs achieve the same minimum distance.) Our distance measure is ordinary Euclidean distance.

WWW The **closest-pair problem** furnishes an example of a problem from computational geometry. **Computational geometry** is concerned with the design

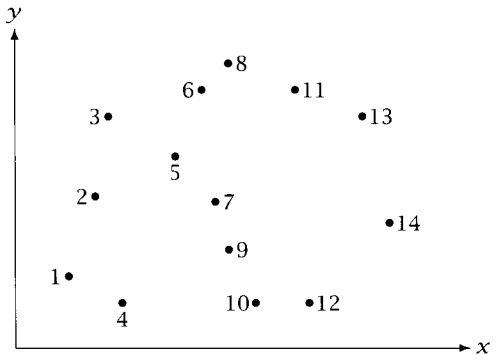


Figure 5.3.1 Types of plywood. Each type of plywood is plotted as a point in the plane; the x-coordinate measures the number of knotholes per square foot, and the y-coordinate measures the smoothness. The closest pair of points (6 and 8) represents the most similar types of plywood.

and analysis of algorithms to solve geometry problems. Efficient geometric algorithms are useful in fields such as pattern recognition, computer graphics, statistics, image processing, and very-large-scale-integration (VLSI) design.

One way to solve the closest-pair problem is to list the distance between each pair and choose the minimum in this list of distances. Since there are $n(n-1)/2 = \Theta(n^2)$ pairs, this "list-all" algorithm's time is $\Theta(n^2)$. We can do better; we will give a divide-and-conquer closest-pair algorithm whose worst-case time is $\Theta(n \lg n)$.

We begin by finding a vertical line l that divides the points into two nearly equal parts (see Figure 5.3.2). [If n is even, we divide the points into parts each having $n/2$ points. If n is odd, we divide the points into parts, one having $(n+1)/2$ points and the other having $(n-1)/2$ points.]

We then recursively solve the problem for each of the parts. We let δ_L be the distance between a closest pair in the left part; we let δ_R be the distance between a closest pair in the right part; and we let

$$\delta = \min\{\delta_L, \delta_R\}.$$

Unfortunately, δ may not be the distance between a closest pair from the original set of points because a pair of points, one from the left part and the other from the right part, might be closer (see Figure 5.3.2). Thus we must consider distances between points on opposite sides of the line l.

We first note that if the distance between a pair of points is less than δ, the points must lie in the vertical strip of width 2δ centered at l (see Figure 5.3.2). (Any point not in this strip is at least δ away from every point on the other side of l.) Thus, we can restrict our search for a pair closer than δ to points in this strip.

If there are n points in the strip and we check *all* pairs in the strip, the worst-case time to process the points in the strip is $\Theta(n^2)$. In this case the

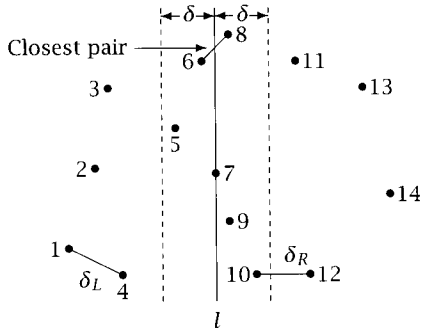


Figure 5.3.2 n points in the plane. The problem is to find a closest pair. For this set, the closest pair is 6 and 8. Line l divides the points into two approximately equal parts. The closest pair in the left half is 1 and 4, which is δ_L apart. The closest pair in the right half is 10 and 12, which is δ_R apart. Any pair (e.g., 6 and 8) closer together than $\delta = \min\{\delta_L, \delta_R\}$ must lie in the vertical strip of width 2δ centered at l.

worst-case time of our algorithm will be $\Omega(n^2)$, which is at least as bad as exhaustive search; thus we must avoid checking all pairs in the strip.

We order the points in the strip in nondecreasing order of the y-coordinates. We then examine the points in this order. When we examine a point p in the strip, any point following p whose distance to p is less than δ must lie strictly within or on the base of the rectangle of height δ whose base contains p and whose vertical sides are at a distance δ from l (see Figure 5.3.3). We need not compute the distance between p and points below p. These distances would already have been considered since we are examining the points in nondecreasing order of their y-coordinates. (We are only interested in distances from p to other points within the rectangle that are on the other side of l; however, our algorithm checks the distance from p to *all* other points within the rectangle.) We will show that this rectangle contains at most eight points, including p itself; so if we compute the distances between p and the next seven points in the strip, we can be sure that we will compute the distance between p and all points in the rectangle. Of course, if fewer than seven points follow p in the list, we compute the distances

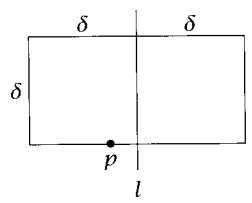


Figure 5.3.3 Any point whose y-coordinate is greater than or equal to p's y-coordinate and whose distance to p is less than δ must lie within the rectangle.

between p and all of the remaining points. By restricting the search in the strip in this way, the time spent processing the points in the strip is $O(n)$. (Since there are at most n points in the strip, the time spent processing the points in the strip is at most $7n$.)

We show that the rectangle of Figure 5.3.3 contains at most eight points. Figure 5.3.4 shows the rectangle of Figure 5.3.3 divided into eight equal squares. Notice that the length of a diagonal of a square is

$$\left[\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2\right]^{1/2} = \frac{\delta}{\sqrt{2}} < \delta;$$

thus each square contains at most one point. (A point on l is assigned to the square on the side of l to which the point belongs. Other points on edges of squares are arbitrarily assigned to adjoining squares.) Therefore the $2\delta \times \delta$ rectangle contains at most eight points.

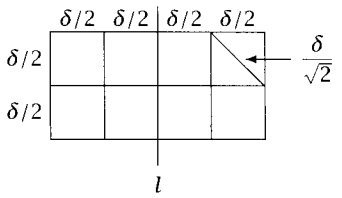


Figure 5.3.4 The large rectangle contains at most eight points because each square contains at most one point.

Example 5.3.1. We show how the closest-pair algorithm finds a closest pair for the input of Figure 5.3.2.

The algorithm begins by finding a vertical line l that divides the points into two equal parts

$$S_1 = \{1, 2, 3, 4, 5, 6, 7\}, \qquad S_2 = \{8, 9, 10, 11, 12, 13, 14\}.$$

For this input there are many possible choices for the dividing line. The particular line chosen here happens to go through point 7.

Next we recursively solve the closest-pair problem for S_1 and S_2. The closest pair of points in S_1 is 1 and 4. We let δ_L denote the distance between points 1 and 4. The closest pair of points in S_2 is 10 and 12. We let δ_R denote the distance between points 10 and 12. We let

$$\delta = \min\{\delta_L, \delta_R\} = \delta_R.$$

We next order the points in the vertical strip of width 2δ centered at l in nondecreasing order of their y-coordinates:

$$10, \quad 9, \quad 7, \quad 5, \quad 6, \quad 8.$$

We then examine the points in this order. We compute the distances between each point and the remaining points.

We first compute the distances from 10 to each of 9, 7, 5, 6, and 8. Since each of these distances exceeds δ, at this point we have not found a closer pair.

We next compute the distances from 9 to each of 7, 5, 6, and 8. Since the distance between points 9 and 7 is less than δ, we have discovered a closer pair. We update δ to the distance between points 9 and 7.

We next compute the distances from 7 to each of 5, 6, and 8. Since each of these distances exceeds δ, at this point we have not found a closer pair.

We next compute the distances from 5 to each of 6 and 8. Since each of these distances exceeds δ, at this point we have not found a closer pair.

We next compute the distance from 6 to 8. Since the distance between 6 and 8 is less that δ, we have discovered a closer pair. We update δ to the distance between points 6 and 8. Since there are no more points in the strip to consider, the algorithm terminates. The closest pair is 6 and 8, and the distance between them is δ. □

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 5.2) 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.

We can now formally state the closest-pair algorithm. To simplify the description, our version outputs only the distance between a closest pair of points, but not the points themselves. We leave this enhancement as an exercise (see Exercise 10).

Algorithm 5.3.2 Finding the Distance Between a Closest Pair of Points. This algorithm finds the distance between a closest pair of points. The input is an array $p[1], \ldots, p[n]$ of $n \geq 2$ points. If p is a point, $p.x$ is the x-coordinate of p, and $p.y$ is the y-coordinate of p. The function *merge* is Algorithm 5.2.2 and *mergesort* is Algorithm 5.2.3. The function *merge* uses as the key the y-coordinate of the point. The function *mergesort* uses as the key either the x- or y-coordinate of the point; the comments indicate which. The function $dist(p, q)$ returns the Euclidean distance between points p and q.

Input Parameter: p
Output Parameters: None

```
closest_pair(p) {
  n = p.last
  mergesort(p, 1, n) // sort by x-coordinate
  return rec_cl_pair(p, 1, n)
}

// rec_cl_pair assumes that the input is sorted by x-coordinate.
// At termination, the input is sorted by y-coordinate.

rec_cl_pair(p, i, j) {
  if (j − i < 3) {
    mergesort(p, i, j) // sort by y-coordinate
    // find the distance delta between a closest pair
    delta = dist(p[i], p[i + 1])
    if (j − i == 1) // two points
      return delta
    // three points
    if (dist(p[i + 1], p[i + 2]) < delta)
      delta = dist(p[i + 1], p[i + 2])
    if (dist(p[i], p[i + 2]) < delta)
      delta = dist(p[i], p[i + 2])
    return delta
  }
  k = (i + j)/2
  l = p[k].x
  deltaL = rec_cl_pair(p, i, k)
  deltaR = rec_cl_pair(p, k + 1, j)
  delta = min(deltaL, deltaR)
  // p[i], ..., p[k] is now sorted by y-coordinate, and
  // p[k + 1], ..., p[j] is now sorted by y-coordinate.
  merge(p, i, k, j)
  // p[i], ..., p[j] is now sorted by y-coordinate.

  // store points in the vertical strip in v.
  t = 0
  for k = i to j
    if (p[k].x > l − delta && p[k].x < l + delta) {
      t = t + 1
      v[t] = p[k]
    }
  // look for closer pairs in the strip by comparing
  // each point in the strip to the next 7 points.
  for k = 1 to t − 1
    for s = k + 1 to min(t, k + 7)
      delta = min(delta, dist(v[k], v[s]))
  return delta
}
```

We show that the worst-case time of the closest-pair algorithm (Algorithm 5.3.2) is $\Theta(n \lg n)$.

Theorem 5.3.3. *The worst-case time of Algorithm 5.3.2 is* $\Theta(n \lg n)$.

Proof. The function *closest_pair* begins by sorting the points by x-coordinate, which takes time $\Theta(n \lg n)$ in the worst case. Next, *closest_pair* calls *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 calls itself with input sizes $\lfloor n/2 \rfloor$ and $\lfloor (n+1)/2 \rfloor$. Merging the points, extracting the points in the strip, and checking the distances in the strip each takes time $O(n)$. Thus, we obtain the recurrence relation

$$a_n \leq a_{\lfloor n/2 \rfloor} + a_{\lfloor (n+1)/2 \rfloor} + cn, \quad n > 3,$$

whose solution satisfies (see the Main Recurrence Theorem, Theorem 2.4.7)

$$a_n = O(n \lg n).$$

Sorting the points by x-coordinate takes time $\Theta(n \lg n)$, and the worst-case time of *rec_cl_pair* is $O(n \lg n)$; thus, the worst-case time of *closest_pair* is $\Theta(n \lg n)$. ■

It can be shown (see Exercise 15) that there are at most six points in the rectangle of Figure 5.3.3 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 (see Exercise 13). 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 (see Exercise 12).

Making reasonable assumptions about what kinds of computations are allowed and using a decision tree model together with advanced methods, Preparata (see Preparata, 1985: Theorem 5.2, page 188) shows that Algorithm 5.3.2 is asymptotically optimal.

Exercises

In Exercises 1 and 2, describe how the closest-pair algorithm finds a closest pair of points for the given input.

1S. (8,4), (3,11), (12,10), (5,4), (1,2), (17,10), (8,7), (8,9), (11,3), (1,5), (11,7), (5,9), (1,9), (7,6), (3,7), (14,7)

2. (10,1), (7,7), (3,13), (6,10), (16,4), (10,5), (7,13), (13,8), (4,4), (2,2), (1,8), (10,13), (7,1), (4,8), (12,3), (16,10), (14,5), (10,9)

3S. In order to terminate the recursion in the closest-pair algorithm, if there are three or fewer points in the input, we find a closest pair directly. Why can't we replace "three" by "two"?

4. Show that the worst-case time of *rec_cl_pair* for input of size n is $\Theta(n \lg n)$.

5. What can you conclude about input to the closest-pair algorithm when the output is zero for the distance between a closest pair?

6S. Give an example of input for which the closest-pair algorithm puts some points on the dividing line l into the left half and other points on l into the right half.

7. Explain why, in some cases, when dividing a set of points by a vertical line into two nearly equal parts, it is necessary for the line to contain some of the points.

8. Show that there are at most four points in the lower half of the rectangle of Figure 5.3.4.

9S. What would the worst-case time of the closest-pair algorithm be if instead of merging $p[i], \ldots, p[k]$ and $p[k+1], \ldots, p[j]$, we used mergesort to sort $p[i], \ldots, p[j]$?

10. Write a closest-pair algorithm that finds one closest pair as well as the distance between the pair of points.

11. Write an algorithm that finds the distance between a closest pair of points on a (straight) line.

12S. Give an example of input for which comparing each point in the strip with the next two points gives incorrect output.

13. Give an example to show that it is possible to place six points in the rectangle of Figure 5.3.3 when the base is included and the other sides are excluded.

14. When we compute the distances between a point p in the strip and points following it, can we stop computing distances from p if we find a point q such that the distance between p and q exceeds δ? Explain.

$\star$**15S.** Show that there are at most six points in the rectangle of Figure 5.3.3 when the base is included and the other sides are excluded.

5.4 Strassen's Matrix Product Algorithm

If A is a matrix, we let A_{ij} denote the entry in row i, column j. Recall that the **matrix product** of A and B, where A is an $m \times p$ matrix (i.e., A has m

rows and p columns) and B is a $p \times n$ matrix, is defined as the $m \times n$ matrix C, where

$$C_{ij} = \sum_{k=1}^{p} A_{ik}B_{kj}, \qquad 1 \le i \le m,\ 1 \le j \le n.$$

Matrix multiplication is a fundamental operation on matrices and is used in many different contexts. For example, if A is the adjacency matrix of a simple graph G with vertices $1, \ldots, n$, entry ij in

$$A^k = \underbrace{A \cdot A \cdots A}_{k\ A\text{'s}}$$

is equal to the number of paths of length k from vertex i to vertex j (see Exercise 7). An adjacency matrix is an example of a *square matrix*—a matrix in which the number of rows is equal to the number of columns. In the remainder of this section, we deal only with square matrices. Algorithm 5.4.1 computes the product of two square matrices directly from the definition.

Algorithm 5.4.1 Matrix Product. This algorithm computes the product C of the $n \times n$ matrices A and B directly from the definition of the matrix product.

Input Parameters: A, B
Output Parameter: C

```
matrix_product(A,B,C) {
  n = A.last
  for i = 1 to n
    for j = 1 to n {
      C[i][j] = 0
      for k = 1 to n
        C[i][j] = C[i][j] + A[i][k] * B[k][j]
    }
}
```

Since Algorithm 5.4.1 performs n^3 multiplications and n^3 additions, it requires $\Theta(n^3)$ arithmetic operations. For many years, $\Theta(n^3)$ was considered to be the minimum number of arithmetic operations, and it was quite a surprise when a more efficient algorithm was discovered.

Consider a divide-and-conquer approach to computing the matrix product. The input is A and B—two $n \times n$ matrices, where n is a power of 2. If $n > 1$, we divide each of A and B into four $n/2 \times n/2$ matrices

$$A = \begin{pmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} \\ \mathbf{a}_{21} & \mathbf{a}_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} \mathbf{b}_{11} & \mathbf{b}_{12} \\ \mathbf{b}_{21} & \mathbf{b}_{22} \end{pmatrix},$$

and then compute the matrix product as

$$AB = \begin{pmatrix} \mathbf{a}_{11}\mathbf{b}_{11} + \mathbf{a}_{12}\mathbf{b}_{21} & \mathbf{a}_{11}\mathbf{b}_{12} + \mathbf{a}_{12}\mathbf{b}_{22} \\ \mathbf{a}_{21}\mathbf{b}_{11} + \mathbf{a}_{22}\mathbf{b}_{21} & \mathbf{a}_{21}\mathbf{b}_{12} + \mathbf{a}_{22}\mathbf{b}_{22} \end{pmatrix}. \tag{5.4.1}$$

[Exercise 1 is to verify that equation (5.4.1) correctly computes AB.] Each of the terms $\mathbf{a}_{ik}\mathbf{b}_{kj}$ is itself a *matrix* product and is computed recursively, unless $\mathbf{a}_{ik}$ and $\mathbf{b}_{kj}$ are 1×1.

Let c_n denote the number of multiplications and additions required by an algorithm based on equation (5.4.1) to compute the product of two $n \times n$ matrices. Computing the products of the eight $n/2 \times n/2$ submatrices requires $8c_{n/2}$ additions and multiplications, and combining them requires $4(n/2)^2 = n^2$ additions. Thus, we obtain the recurrence relation

$$c_n = 8c_{n/2} + n^2.$$

Unfortunately, the solution of this recurrence relation is $c_n = \Theta(n^3)$ (see the Main Recurrence Theorem, Theorem 2.4.7), which is no better asymptotically than Algorithm 5.4.1. Strassen (see Strassen, 1969) showed how to compute the expressions in equation (5.4.1) using only *seven* matrix products and $\Theta(n^2)$ additions and subtractions to combine them. The recurrence relation for this technique is

$$c_n = 7c_{n/2} + f(n),$$

where $f(n) = \Theta(n^2)$. The solution satisfies $c_n = \Theta(n^{\lg 7}) = \Theta(n^{2.807})$ (see the Main Recurrence Theorem, Theorem 2.4.7). Strassen's algorithm is, then, asymptotically faster than Algorithm 5.4.1.

Strassen's algorithm computes the quantities

$$\begin{aligned}
\mathbf{q}_1 &= (\mathbf{a}_{11} + \mathbf{a}_{22}) * (\mathbf{b}_{11} + \mathbf{b}_{22}) \\
\mathbf{q}_2 &= (\mathbf{a}_{21} + \mathbf{a}_{22}) * \mathbf{b}_{11} \\
\mathbf{q}_3 &= \mathbf{a}_{11} * (\mathbf{b}_{12} - \mathbf{b}_{22}) \\
\mathbf{q}_4 &= \mathbf{a}_{22} * (\mathbf{b}_{21} - \mathbf{b}_{11}) \\
\mathbf{q}_5 &= (\mathbf{a}_{11} + \mathbf{a}_{12}) * \mathbf{b}_{22} \\
\mathbf{q}_6 &= (\mathbf{a}_{21} - \mathbf{a}_{11}) * (\mathbf{b}_{11} + \mathbf{b}_{12}) \\
\mathbf{q}_7 &= (\mathbf{a}_{12} - \mathbf{a}_{22}) * (\mathbf{b}_{21} + \mathbf{b}_{22}).
\end{aligned}$$

We can verify (see Exercise 6) that

$$AB = \begin{pmatrix} \mathbf{q}_1 + \mathbf{q}_4 - \mathbf{q}_5 + \mathbf{q}_7 & \mathbf{q}_3 + \mathbf{q}_5 \\ \mathbf{q}_2 + \mathbf{q}_4 & \mathbf{q}_1 + \mathbf{q}_3 - \mathbf{q}_2 + \mathbf{q}_6 \end{pmatrix}. \tag{5.4.2}$$

Computing the $\mathbf{q}$'s and combining them to obtain AB requires only seven matrix products and $\Theta(n^2)$ additions and subtractions. Therefore, Strassen's algorithm requires only $\Theta(n^{\lg 7}) = \Theta(n^{2.807})$ arithmetic operations.

Various schemes have been proposed to extend Strassen's algorithm so that it can compute the product of $n \times n$ matrices where n is not a power of 2. One technique is to expand the size of the matrices by adding extra rows and columns of zeros—for example, adding rows and columns of zeros to make the size a power of 2. A more complicated scheme along the same lines was suggested by Strassen himself (see Strassen, 1969).

Strassen's algorithm requires considerable overhead and is practical only for large values of n (e.g., $n \geq 50$). For modest values of n, Algorithm 5.4.1 typically runs faster than Strassen's algorithm.

An algorithm by Coppersmith and Winograd (see Coppersmith, 1987) runs in time $\Theta(n^{2.376})$ and, so, is asymptotically faster than Strassen's algorithm. Since the product of two $n \times n$ matrices contains n^2 terms, *any* algorithm that multiplies two $n \times n$ matrices requires at least $\Omega(n^2)$ arithmetic operations. At the present time, no sharper lower bound is known for the matrix-product problem.

Exercises

1S. Prove that equation (5.4.1) correctly computes the matrix product.

2. Show how the $\Theta(n^3)$ recursive matrix-product algorithm based on equation (5.4.1) computes

$$\begin{pmatrix} 3 & 1 & 0 & 2 \\ 4 & -1 & 1 & 1 \\ 1 & 1 & -2 & 0 \\ -1 & -1 & 1 & 4 \end{pmatrix} \begin{pmatrix} 2 & -5 & 1 & 2 \\ 0 & -3 & -1 & 3 \\ 1 & 2 & 3 & 5 \\ 3 & 1 & -2 & 3 \end{pmatrix}.$$

3. Estimate the number of additions and multiplications the $\Theta(n^3)$ recursive matrix-product algorithm based on equation (5.4.1) performs. How does this compare with Algorithm 5.4.1?

4S. Show how Strassen's algorithm computes

$$\begin{pmatrix} 3 & 1 \\ 4 & -1 \end{pmatrix} \begin{pmatrix} 2 & -5 \\ 6 & -3 \end{pmatrix}.$$

5. Show how the extension of Strassen's algorithm that adds an extra column and row of zeros computes

$$\begin{pmatrix} 1 & 3 & -2 \\ 4 & -1 & 6 \\ 1 & 3 & -3 \end{pmatrix} \begin{pmatrix} -2 & 5 & 1 \\ 6 & -3 & 3 \\ -3 & 1 & 2 \end{pmatrix}.$$

6. Verify equation (5.4.2).

7S. Suppose that A is the adjacency matrix of a simple graph G with vertices $1, \ldots, n$. Show that entry ij in A^k is equal to the number of paths of length k from vertex i to vertex j.

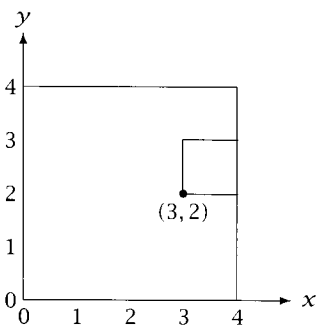
Notes

Algorithm 5.1.4 is due to Solomon W. Golomb (Golomb, 1954). Golomb introduced *polyominoes*, of which trominoes are a special case. A *polyomino of order s* consists of s squares joined at the edges. A tromino is a polyomino of order 3. Three squares in a row form the only other type of polyomino of order 3. No one has yet found a simple formula for the number of polyominoes of order s. Numerous problems using polyominoes have been devised (see Martin, 1991).

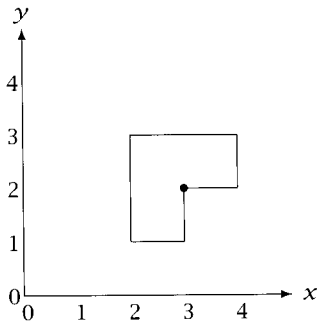
Preparata, 1985, and Edelsbrunner, 1987, are good references on the closest-pair problem and computational geometry, in general. These references also give algorithms to solve the closest-pair problem in an arbitrary number of dimensions. A more recent computational geometry reference is de Berg, 1997.

Chapter Exercises

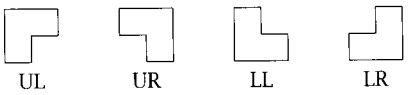
5.1. Implement Algorithm 5.1.4 in the following way. Assume that the board to be tiled is in the usual xy-coordinate system. Assume further that the board to be tiled is designated by the coordinates of its lower-left corner, and that the missing square is also designated by the coordinates of its lower-left corner. The following figure shows a 4×4 board located at $(0, 0)$ whose missing square is located at $(3, 2)$:



Assume also that the location of a tromino is given by the coordinates of its inner corner. For example, the tromino in the figure



is located at $(3,2)$. Finally, assume that the orientations of the trominoes are designated as follows:



(The terminology arises from the orientation of the trominoes if they are placed in the corners of a square: LL is lower left, UL is upper left, etc.) The output from a board of size 4 at $(0,0)$, with the missing square at $(3,2)$, might be

```
2 2 LL
1 3 UL
3 3 UL
3 1 LR
1 1 LL
```

5.2. Implement Algorithm 5.1.4 so that it draws the tiling.

5.3. Which rectangles can be tiled with trominoes?

5.4. Which deficient rectangles can be tiled with trominoes?

5.5. Write an $O(n \lg n)$ algorithm that receives as input an array a of n real numbers and a value *val*. (The array is not necessarily sorted.) The algorithm returns true if there are distinct indexes i and j such that $a[i] + a[j] = val$ and false otherwise.

5.6. Write an $O(n \lg n)$ algorithm that receives as input two n-element arrays a and b of real numbers and a value *val*. (The arrays are not necessarily sorted.) The algorithm returns true if there are indexes i and j such that $a[i] + b[j] = val$ and false otherwise.

5.7. Write an $O(n)$ algorithm that computes the union $A \cup B$ of two n-element sets A and B of real numbers. The sets are represented as arrays sorted in nondecreasing order. (There are no duplicates in either array.) The output is an array sorted in nondecreasing order that represents the union. (There are no duplicates in the output array either.)

5.8. Repeat Exercise 5.7 with "union" replaced by "intersection."

5.9. Write an $O(n \lg m)$ algorithm that computes the union $A \cup B$ of an m-element set A and an n-element set B, where $m \leq n$. The sets A and B contain real numbers and are represented as arrays. (The arrays are not necessarily sorted, and there are no duplicates in either array.) The output is an array that represents the union. (The output array is not necessarily sorted, and it contains no duplicates.)

5.10. Repeat Exercise 5.9 with "union" replaced by "intersection."

5.11. Explain how to tweak the input so that *any* sorting algorithm becomes a stable sorting algorithm.

5.12. Implement merge (Algorithm 5.2.2) and an in-place merge (see Section 5.2 for a reference) and compare the times for various array sizes.

5.13. Implement mergesort (Algorithm 5.2.3), mergesort using an in-place merge, and an in-place mergesort (see Section 5.2 for references for in-place merge and in-place mergesort). Compare the times for arrays of various sizes containing sorted and unsorted data.

5.14. Write an $O(n \lg n)$ algorithm that finds the distance δ between a closest pair of points in the plane and, if $\delta > 0$, also finds *all* pairs δ apart.

5.15. Write an $O(n \lg n)$ algorithm that finds the distance δ between a closest pair of points in the plane and *all* pairs less than 2δ apart.

5.16. Write an $O(n \lg n)$ algorithm that finds a closest pair of points in the plane in which Euclidean distance is replaced by

$$\text{dist}(p, q) = |p_x - q_x| + |p_y - q_y|,$$

where $p = (p_x, p_y)$ and $q = (q_x, q_y)$. This distance is known as the *taxicab* or *Manhattan distance* because $\text{dist}(p, q)$ is equal to the number of blocks between points p and q if we are moving within a street system comprising square blocks and p and q are at the corners of blocks.

5.17. Write a version of Strassen's algorithm (see Section 5.4) that multiplies $n \times n$ matrices, where n is any positive integer. What is the time of your algorithm?