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*

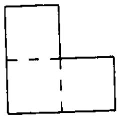


Figure 5.1.1 A tromino.

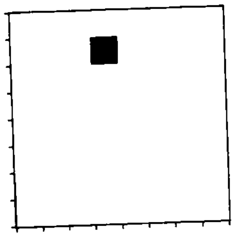


Figure 5.1.2 A deficient 8×8 board. The missing square is shown in black. The gap between successive marks along the sides is one unit.

(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.

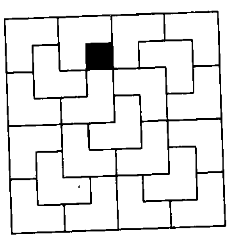


Figure 5.1.3 A tiling of a deficient 8×8 board with trominoes. □

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

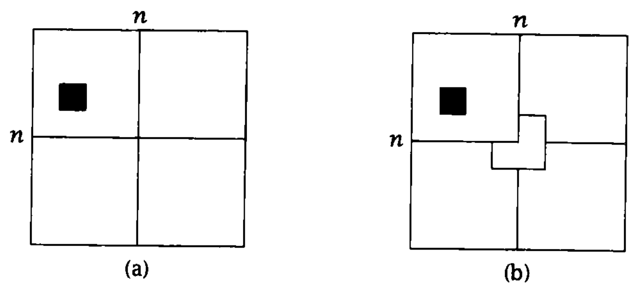


Figure 5.1.4 Using divide and conquer to tile a deficient $n \times n$ board with trominoes. In (a), the original $n \times n$ board is divided into four $n/2 \times n/2$ subboards. The subboard containing the missing square is then tiled recursively. A tromino is placed as shown in (b) so that each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

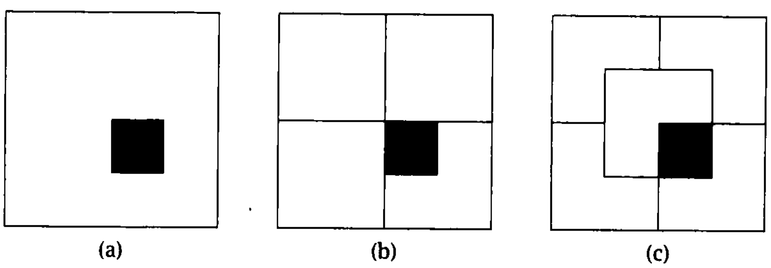


Figure 5.1.5 Tiling the deficient 4×4 board shown in (a). First, the board is divided into four 2×2 subboards as shown in (b). The subboard that contains the missing square is recursively tiled; in this case, the deficient 2×2 board is a tromino. Next, we place a tromino as shown in (c) so that each 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. □

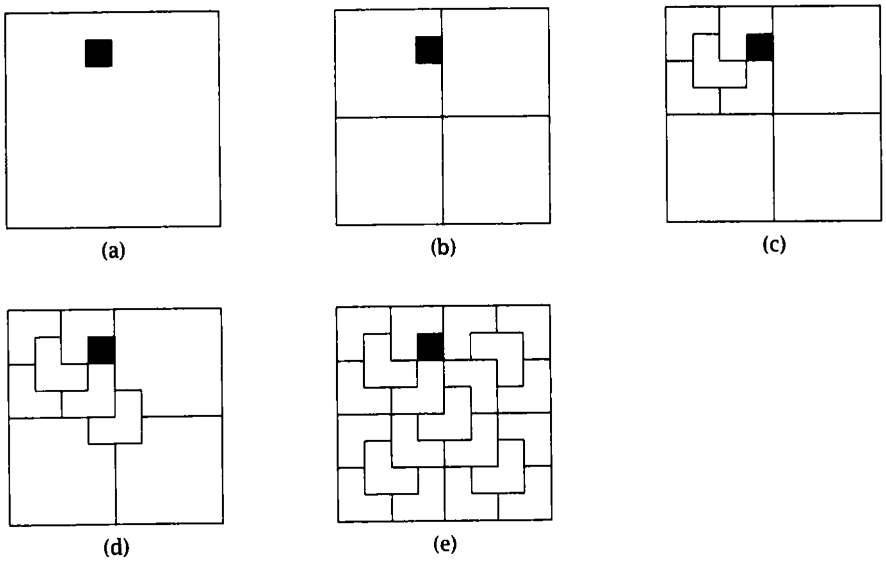


Figure 5.1.6 Tiling the deficient 8×8 board shown in (a). First, the board is divided into four 4×4 subboards as shown in (b). The subboard that contains the missing square is recursively tiled as shown in (c). Next, we place a tromino as shown in (d) so that each of its 1×1 squares lies in one of the 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.

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

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

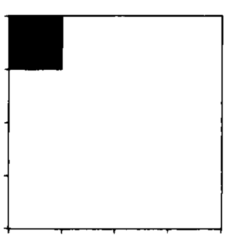
trominoes on the board. Therefore, the time required by Algorithm 5.1.4 is $\Theta(n^2)$.

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.