Chapter 2
Divide-and-conquer algorithms

The divide-and-conquer strategy solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of problem
2. Recursively solving these subproblems
3. Appropriately combining their answers

The real work is done piecemeal, in three different places: in the partitioning of problems into subproblems; at the very tail end of the recursion, when the subproblems are so small that they are solved outright; and in the gluing together of partial answers. These are held together and coordinated by the algorithm's core recursive structure.

As an introductory example, we'll see how this technique yields a new algorithm for multiplying numbers, one that is much more efficient than the method we all learned in elementary school!

2.1 Multiplication
The mathematician Carl Friedrich Gauss (1777–1855) once noticed that although the product of two complex numbers

\[(a + bi)(c + di) = ac - bd + (bc + ad)i\]

seems to involve four real-number multiplications, it can in fact be done with just three: \(ac\), \(bd\), and \((a + b)(c + d)\), since

\[bc + ad = (a + b)(c + d) - ac - bd.\]

In our big-\(O\) way of thinking, reducing the number of multiplications from four to three seems wasted ingenuity. But this modest improvement becomes very significant when applied recursively.

Let's move away from complex numbers and see how this helps with regular multiplication. Suppose \(x\) and \(y\) are two \(n\)-bit integers, and assume for convenience that \(n\) is a power of 2 (the more general case is hardly any different). As a first step toward multiplying \(x\) and \(y\), split each of them into their left and right halves, which
are \( n/2 \) bits long:

\[
\begin{align*}
x &= x_0 \quad \quad x_1 \\
y &= y_0 \quad \quad y_1 \\
&= 2^{n/2} x_1 + x_0 \quad 2^{n/2} y_1 + y_0.
\end{align*}
\]

For instance, if \( x = 10110110_2 \) (the subscript 2 means "binary") then \( x_0 = 1011_2 \), \( x_1 = 0110_2 \), and \( x = 1011_2 \times 2^4 + 0110_2 \). The product of \( x \) and \( y \) can then be rewritten as

\[
xy = (2^{n/2} x_1 + x_0)(2^{n/2} y_1 + y_0) = 2^n x_0 y_1 + 2^{n/2} (x_0 y_1 + x_1 y_0) + x_1 y_1.
\]

We will compute \( xy \) via the expression on the right. The additions take linear time, as do the multiplications by powers of 2 (which are merely left-shifts). The significant operations are the four \( n/2 \)-bit multiplications, \( x_0 y_1, x_1 y_0, x_0 y_0, x_1 y_1 \); these we can handle by four recursive calls. Thus our method for multiplying \( n \)-bit numbers starts by making recursive calls to multiply these four pairs of \( n/2 \)-bit numbers (four subproblems of half the size), and then evaluates the preceding expression in \( O(n) \) time. Writing \( T(n) \) for the overall running time on \( n \)-bit inputs, we get the recurrence relation

\[
T(n) = 4T(n/2) + O(n).
\]

We will soon see general strategies for solving such equations. In the meantime, this particular one works out to \( O(n^2) \), the same running time as the traditional grade-school multiplication technique. So we have a radically new algorithm, but we haven't yet made any progress in efficiency. How can our method be sped up?

This is where Gauss's trick comes to mind. Although the expression for \( xy \) seems to demand four \( n/2 \)-bit multiplications, as before just three will do: \( x_0 y_1, x_1 y_0, \) and \( (x_0 + x_1)(y_1 + y_0) \), since \( x_0 y_1 + x_1 y_0 = (x_0 + x_1)(y_1 + y_0) - x_0 y_1 - x_1 y_0 \). The resulting
\[(u) \frac{O + (1 + \frac{2}{u})T \geq (u)T}{\text{Actually, the recurrence should read}}\]

\[\frac{(u)O \times \left(\frac{2}{u}\right)}{(u)O \times \left(\frac{2}{u}\right)} = \frac{(u)}{(u)}\]

The idea is to separate and combine the subproblems. Therefore, the total time spent at depth \(k\) is

For each subproblem, a larger amount of work is done in the remaining subproblems. Each of these subproblems, each of size \(\frac{n}{u^k}\), is expected to produce the same order of magnitude of recursion. Therefore, the height of the tree is \(\log u\). The branching factor is \(2\) in each problem. We therefore get recursion down to size \(1\), and so the recursion ends. Averaging these results, we see that the subproblems get halved in size at each level. The next access level, therefore, is \(\log u\). The tree access level of the lower level of recursion is similar to the shape of the tree. From this, we can conclude that the structure is similar to the structure of the recursion.

Thus, the running time can be derived by looking at the structure of the algorithm, pattern of recursion, and the running time of \(O(1)\). Therefore, the running time of \((u)O + (1 + \frac{2}{u})T \geq (u)T\) is shown in Figure 2.1. Let us now compute the improved running time of the algorithm, shown in Figure 2.1.

```
#define \(d + \frac{\epsilon}{u}\) for \(u \geq 1\)
return \(d \times \left(\frac{d}{d} - \frac{d}{d}\right) + \left(\frac{d}{d}\right)\times\left(\frac{d}{d}\right)\)
\[\text{Multiplier = } \frac{d}{d}\]
\[\text{Multiplier = } \frac{d}{d}\]
\[\text{Multiplier = } \frac{d}{d}\]
```

\(\text{Input: } n, \text{ positive integers } x \text{ and } y\)

\(\text{Output: } \text{The product of } x \text{ and } y\)

Function \(\text{Multiply}(x, y, n, \text{leftmost, rightmost, } u, \text{leftmost, rightmost, } u)\)

Figure 2.1 A divide-and-conquer algorithm for integer multiplication.
Figure 2.2 Divide-and-conquer integer multiplication. (a) Each problem is divided into three subproblems. (b) The levels of recursion.

At the very top level, when \( k = 0 \), this works out to \( O(n) \). At the bottom, when \( k = \log_2 n \), it is \( O(3^{\log_2 n}) \), which can be rewritten as \( O(n^{\log_3 3}) \) (do you see why?). Between these two endpoints, the work done increases geometrically from \( O(n) \) to \( O(n^{\log_3 3}) \), by a factor of \( 3/2 \) per level. The sum of any increasing geometric series is, within a constant factor, simply the last term of the series: such is the rapidity of the increase (Exercise 0.2). Therefore the overall running time is \( O(n^{\log_3 3}) \), which is about \( O(n) \).

In the absence of Gauss's trick, the recursion tree would have the same height, but the branching factor would be 4. There would be \( 4^{\log_2 n} = n^2 \) leaves, and therefore the running time would be at least this much. In divide-and-conquer algorithms, the number of subproblems translates into the branching factor of the recursion tree; small changes in this coefficient can have a big impact on running time.

A practical note: it generally does not make sense to recurse all the way down to 1 bit. For most processors, 16- or 32-bit multiplication is a single operation, so by the time the numbers get into this range they should be handed over to the built-in procedure.

Finally, the eternal question: Can we do better? It turns out that even faster algorithms for multiplying numbers exist, based on another important divide-and-conquer algorithm: the fast Fourier transform, to be explained in Section 2.6.