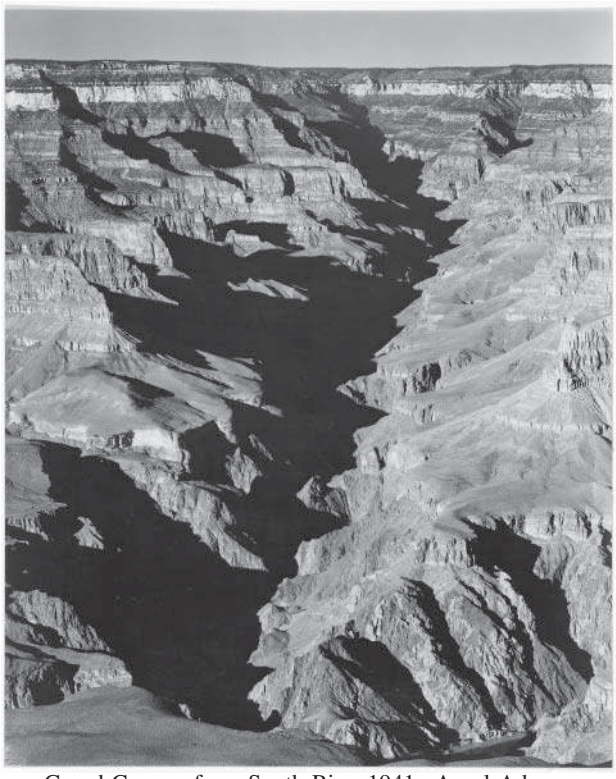
Chapter 11 Divide-and-Conquer



Grand Canyon from South Rim, 1941. Ansel Adams. U.S. government image. U.S. National Archives and Records Administration.

Contents

Imagine that you are getting ready to go on a vacation and are looking for a hotel. Suppose that for you, the only criteria that matter when you judge a hotel are the size of its pool and the quality of its restaurant, as measured by a well-known restaurant guide. You have gone to the restaurant guide website and several hotel websites and have discovered, for each of n hotels, the size of its pool and the quality of its restaurant.

With so many hotels to choose from, it would be useful if you could rule out some possibilities. For instance, if there is a hotel whose pool is smaller and restaurant is of lower quality than another hotel, then it can be eliminated as a possible choice, as both of your criteria for the first hotel are dominated by the second hotel. We can visualize the various trade-offs by plotting each hotel as a two-dimensional point, (x, y), where x is the pool size and y is the restaurant quality score. We say that such a point is a ***maximum*** point in a set if there is no other point, (x', y'), in that set such that $x \leq x'$ and $y \leq y'$. (See Figure 11.1.)

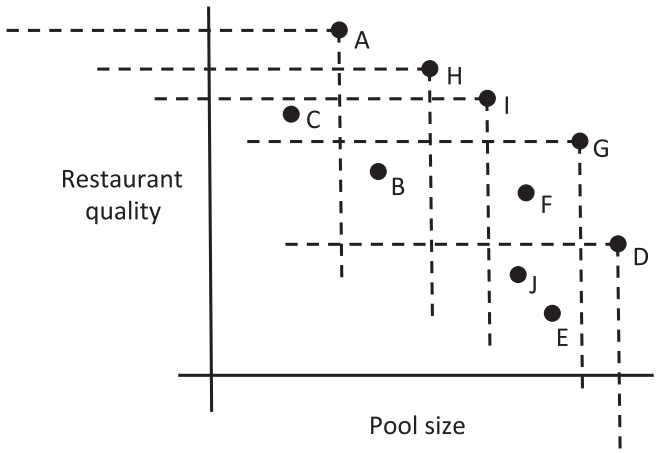


Figure 11.1: A set of hotels visualized as two-dimensional points, in terms of their pool size and restaurant quality scores. The points that are maxima are shown with dashed lines indicating the set of other points they dominate. For example, Hotel B is dominated by Hotels G, H, and I. The maxima set is $\{A, D, G, H, I\}$.

Given such a set of points, then, we are interested in designing an efficient algorithm that can identify the ***maxima set***. The other points can be pruned from consideration, since each of them is dominated by some other point. As we show in Section 11.4, the algorithmic technique we discuss in this chapter, ***divide-and-conquer***, is well suited for solving this problem. In addition, in this chapter, we discuss how to apply this technique to integer and matrix multiplication, and we give a general way, called the master theorem, for analyzing divide-and-conquer algorithms. Incidentally, we also discuss divide-and-conquer algorithms, merge-sort and quick-sort, in Chapter 8, and the fast Fourier transform, in Chapter 25.

11.1 Recurrences and the Master Theorem

The ***divide-and-conquer*** technique involves solving a particular computational problem by dividing it into one or more subproblems of smaller size, recursively solving each subproblem, and then "merging" or "marrying" the solutions to the subproblem(s) to produce a solution to the original problem.

We can model the divide-and-conquer approach by using a parameter n to denote the size of the original problem, and let $S(n)$ denote this problem. We solve the problem $S(n)$ by solving a collection of k subproblems $S(n_1)$, $S(n_2)$, ..., $S(n_k)$, where $n_i < n$ for $i = 1, \ldots, k$, and then merging the solutions to these subproblems.

For example, in the classic merge-sort algorithm (Section 8.1), $S(n)$ denotes the problem of sorting a sequence of n numbers. Merge-sort solves problem $S(n)$ by dividing it into two subproblems $S(\lfloor n/2 \rfloor)$ and $S(\lceil n/2 \rceil)$, recursively solving these two subproblems, and then merging the resulting sorted sequences into a single sorted sequence that yields a solution to $S(n)$. The merging step takes $O(n)$ time. Thus, the total running time of the merge-sort algorithm is $O(n \log n)$.

As with the merge-sort algorithm, the general divide-and-conquer technique can be used to build algorithms that have fast running times.

To analyze the running time of a divide-and-conquer algorithm, we often use a ***recurrence equation*** (Section 1.1.4). That is, we let a function $T(n)$ denote the running time of the algorithm on an input of size n, and we characterize $T(n)$ using an equation that relates $T(n)$ to values of the function T for problem sizes smaller than n. In the case of the merge-sort algorithm, we get the recurrence equation

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2, \end{cases}$$

for some constant $b \geq 1$, taking the simplifying assumption that n is a power of 2. In fact, throughout this section, we take the simplifying assumption that n is an appropriate power, so that we can avoid using floor and ceiling functions. Every asymptotic statement we make about recurrence equations will still be true, even if we relax this assumption (see Exercise C-11.5). As we observed in Chapter 8, we can show that $T(n)$ is $O(n \log n)$ in this case. In general, however, we may get a recurrence equation that is more challenging to solve than this one. Thus, it is useful to develop some general ways of solving the kinds of recurrence equations that arise in the analysis of divide-and-conquer algorithms.

The Iterative Substitution Method

One way to solve a divide-and-conquer recurrence equation is to use the ***iterative substitution*** method, which is more colloquially known as the "plug-and-chug" method. In using this method, we assume that the problem size n is fairly large and we then substitute the general form of the recurrence for each occurrence of the function T on the right-hand side. For example, performing such a substitution with the merge-sort recurrence equation yields the equation

$$\begin{aligned} T(n) &= 2(2T(n/2^2) + b(n/2)) + bn \\ &= 2^2T(n/2^2) + 2bn. \end{aligned}$$

Plugging the general equation for T in again yields the equation

$$\begin{aligned} T(n) &= 2^2(2T(n/2^3) + b(n/2^2)) + 2bn \\ &= 2^3T(n/2^3) + 3bn. \end{aligned}$$

The hope in applying the iterative substitution method is that at some point we will see a pattern that can be converted into a general closed-form equation (with T only appearing on the left-hand side). In the case of the merge-sort recurrence equation, the general form is

$$T(n) = 2^iT(n/2^i) + ibn.$$

Note that the general form of this equation shifts to the base case, $T(n) = b$, when $n = 2^i$, that is, when $i = \log n$, which implies

$$T(n) = bn + bn \log n.$$

In other words, $T(n)$ is $O(n \log n)$. In a general application of the iterative substitution technique, we hope that we can determine a general pattern for $T(n)$ and that we can also figure out when the general form of $T(n)$ shifts to the base case.

From a mathematical point of view, there is one point in the use of the iterative substitution technique that involves a bit of a logical "jump." This jump occurs at the point where we try to characterize the general pattern emerging from a sequence of substitutions. Often, as was the case with the merge-sort recurrence equation, this jump is quite reasonable. Other times, however, it may not be so obvious what a general form for the equation should look like. In these cases, the jump may be more dangerous. To be completely safe in making such a jump, we must fully justify the general form of the equation, possibly using induction. Combined with such a justification, the iterative substitution method is completely correct and an often useful way of characterizing recurrence equations. By the way, the colloquialism "plug-and-chug," used to describe the iterative substitution method, comes from the way this method involves "plugging" in the recursive part of an equation for $T(n)$ and then often "chugging" through a considerable amount of algebra in order to get this equation into a form where we can infer a general pattern.

The Recursion Tree

Another way of characterizing recurrence equations is to use the ***recursion tree*** method. Like the iterative substitution method, this technique uses repeated substitution to solve a recurrence equation, but it differs from the iterative substitution method in that, rather than being an algebraic approach, it is a visual approach. In using the recursion tree method, we draw a tree R where each node represents a different substitution of the recurrence equation. Thus, each node in R has a value of the argument n of the function $T(n)$ associated with it. In addition, we associate an ***overhead*** with each node v in R, defined as the value of the nonrecursive part of the recurrence equation for v. For divide-and-conquer recurrences, the overhead corresponds to the running time needed to merge the subproblem solutions coming from the children of v. The recurrence equation is then solved by summing the overheads associated with all the nodes of R. This is commonly done by first summing values across the levels of R and then summing up these partial sums for all the levels of R.

Example 11.1: *Consider the following recurrence equation:*

$$T(n) = \begin{cases} b & \text{if } n < 3 \\ 3T(n/3) + bn & \text{if } n \geq 3. \end{cases}$$

This is the recurrence equation that we get, for example, by modifying the merge-sort algorithm so that we divide an unsorted sequence into three equal-sized sequences, recursively sort each one, and then do a three-way merge of three sorted sequences to produce a sorted version of the original sequence. In the recursion tree R for this recurrence, each internal node v has three children and has a size and an overhead associated with it, which corresponds to the time needed to merge the subproblem solutions produced by v's children. We illustrate the tree R in Figure 11.2. Note that the overheads of the nodes of each level sum to bn. Thus, observing that the depth of R is $\log_3 n$, we have that $T(n)$ is $O(n \log n)$.

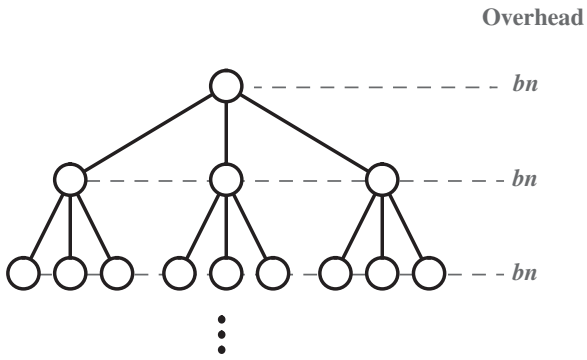


Figure 11.2: The recursion tree R used in Example 11.1, where we show the cumulative overhead of each level.

The Guess-and-Test Method

Another method for solving recurrence equations is the ***guess-and-test*** technique. This technique involves first making an educated guess as to what a closed-form solution of the recurrence equation might look like and then justifying that guess, usually by induction. For example, we can use the guess-and-test method as a kind of "binary search" for finding good upper bounds on a given recurrence equation. If the justification of our current guess fails, then it is possible that we need to use a faster-growing function, and if our current guess is justified "too easily," then it is possible that we need to use a slower-growing function. However, using this technique requires our being careful, in each mathematical step we take, in trying to justify that a certain hypothesis holds with respect to our current "guess." We explore an application of the guess-and-test method in the examples that follow.

Example 11.2: *Consider the following recurrence equation (assuming the base case $T(n) = b$ for $n < 2$):*

$$T(n) = 2T(n/2) + bn \log n.$$

This looks very similar to the recurrence equation for the merge-sort routine, so we might make the following as our first guess:

$$\text{First guess: } T(n) \leq cn \log n,$$

for some constant $c > 0$. We can certainly choose c large enough to make this true for the base case, so consider the case when $n \geq 2$. If we assume our first guess is an inductive hypothesis that is true for input sizes smaller than n, then we have

$$\begin{aligned} T(n) &= 2T(n/2) + bn \log n \\ &\leq 2(c(n/2)\log(n/2)) + bn \log n \\ &= cn(\log n - \log 2) + bn \log n \\ &= cn \log n - cn + bn \log n. \end{aligned}$$

But there is no way that we can make this last line less than or equal to $cn \log n$ for $n \geq 2$. Thus, this first guess was not sufficient. Let us therefore try

$$\text{Better guess: } T(n) \leq cn \log^2 n,$$

for some constant $c > 0$. We can again choose c large enough to make this true for the base case, so consider the case when $n \geq 2$. If we assume this guess as an inductive hypothesis that is true for input sizes smaller than n, then we have

$$\begin{aligned} T(n) &= 2T(n/2) + bn \log n \\ &\leq 2(c(n/2)\log^2(n/2)) + bn \log n \\ &= cn(\log^2 n - 2\log n + 1) + bn \log n \\ &= cn \log^2 n - 2cn \log n + cn + bn \log n \\ &\leq cn \log^2 n, \end{aligned}$$

provided $c \geq b$. Thus, we have shown that $T(n)$ is indeed $O(n \log^2 n)$ in this case.

We must take care in using this method. Just because one inductive hypothesis for $T(n)$ does not work, that does not necessarily imply that another one proportional to this one will not work.

Example 11.3: *Consider the following recurrence equation (assuming the base case $T(n) = b$ for $n < 2$):*

$$T(n) = 2T(n/2) + \log n.$$

This recurrence is the running time for the bottom-up heap construction discussed in Section 5.4, which we have shown is $O(n)$. Nevertheless, if we try to prove this fact with the most straightforward inductive hypothesis, we will run into some difficulties. In particular, consider the following:

$$\text{First guess: } T(n) \leq cn,$$

for some constant $c > 0$. We can choose c large enough to make this true for the base case, certainly, so consider the case when $n \geq 2$. If we assume this guess as an inductive hypothesis that is true for input sizes smaller than n, then we have

$$\begin{aligned} T(n) &= 2T(n/2) + \log n \\ &\leq 2(c(n/2)) + \log n \\ &= cn + \log n. \end{aligned}$$

But there is no way that we can make this last line less than or equal to cn for $n \geq 2$. Thus, this first guess was not sufficient, even though $T(n)$ is indeed $O(n)$. Still, we can show this fact is true by using

$$\text{Better guess: } T(n) \leq c(n - \log n),$$

for some constant $c > 0$. We can again choose c large enough to make this true for the base case; in fact, we can show that it is true any time $n < 8$. So consider the case when $n \geq 8$. If we assume this guess as an inductive hypothesis that is true for input sizes smaller than n, then we have

$$\begin{aligned} T(n) &= 2T(n/2) + \log n \\ &\leq 2c((n/2) - \log(n/2)) + \log n \\ &= cn - 2c \log n + 2c + \log n \\ &= c(n - \log n) - c \log n + 2c + \log n \\ &\leq c(n - \log n), \end{aligned}$$

provided $c \geq 3$ and $n \geq 8$. Thus, we have shown that $T(n)$ is indeed $O(n)$ in this case.

The guess-and-test method can be used to establish either an upper or lower bound for the asymptotic complexity of a recurrence equation. Even so, as the above example demonstrates, it requires that we have developed some skill with mathematical induction.

11.1.1 The Master Theorem

Each of the methods described above for solving recurrence equations is ad hoc and requires mathematical sophistication in order to be used effectively. There is, nevertheless, one method for solving divide-and-conquer recurrence equations that is quite general and does not require explicit use of induction to apply correctly. It is the ***master theorem***. The master theorem is a "cookbook" method for determining the asymptotic characterization of a wide variety of recurrence equations. Namely, it is used for recurrence equations of the form

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d, \end{cases}$$

where $d \geq 1$ is an integer constant, $a \geq 1$, $c > 0$, and $b > 1$ are real constants, and $f(n)$ is a function that is positive for $n \geq d$. Such a recurrence equation would arise in the analysis of a divide-and-conquer algorithm that divides a given problem into a subproblems of size at most n/b each, solves each subproblem recursively, and then "merges" the subproblem solutions into a solution to the entire problem. The function $f(n)$, in this equation, denotes the total additional time needed to divide the problem into subproblems and merge the subproblem solutions into a solution to the entire problem. Each of the recurrence equations given above uses this form, as do each of the recurrence equations used to analyze divide-and-conquer algorithms given earlier in this book. Thus, it is indeed a general form for divide-and-conquer recurrence equations.

The master theorem for solving such recurrence equations involves simply writing down the answer based on whether one of the three cases applies. Each case is distinguished by comparing $f(n)$ to the special function $n^{\log_b a}$ (we will show later why this special function is so important).

Theorem 11.4 [The Master Theorem]: *Let $f(n)$ and $T(n)$ be defined as above.*

1. *If there is a small constant $\epsilon > 0$, such that $f(n)$ is $O(n^{\log_b a - \epsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$.*
2. *If there is a constant $k \geq 0$, such that $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$.*
3. *If there are small constants $\epsilon > 0$ and $\delta < 1$, such that $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq \delta f(n)$, for $n \geq d$, then $T(n)$ is $\Theta(f(n))$.*

Case 1 characterizes the situation where $f(n)$ is polynomially smaller than the special function, $n^{\log_b a}$. Case 2 characterizes the situation when $f(n)$ is asymptotically close to the special function, and Case 3 characterizes the situation when $f(n)$ is polynomially larger than the special function.

Some Example Applications of the Master Theorem

We illustrate the usage of the master theorem with a few examples (with each taking the assumption that $T(n) = c$ for $n < d$, for constants $c \geq 1$ and $d \geq 1$).

Example 11.5: *Consider the recurrence*

$$T(n) = 4T(n/2) + n.$$

In this case, $n^{\log_b a} = n^{\log_2 4} = n^2$. *Thus, we are in Case 1, for* $f(n)$ *is* $O(n^{2-\epsilon})$ *for* $\epsilon = 1$. *This means that* $T(n)$ *is* $\Theta(n^2)$ *by the master theorem.*

Example 11.6: *Consider the recurrence*

$$T(n) = 2T(n/2) + n \log n,$$

which is one of the recurrences given above. In this case, $n^{\log_b a} = n^{\log_2 2} = n$. *Thus, we are in Case 2, with* $k = 1$, *for* $f(n)$ *is* $\Theta(n \log n)$. *This means that* $T(n)$ *is* $\Theta(n \log^2 n)$ *by the master theorem.*

Example 11.7: *Consider the recurrence*

$$T(n) = T(n/3) + n,$$

which is the recurrence for a geometrically decreasing summation that starts with n. *In this case,* $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$. *Thus, we are in Case 3, for* $f(n)$ *is* $\Omega(n^{0+\epsilon})$, *for* $\epsilon = 1$, *and* $af(n/b) = n/3 = (1/3)f(n)$. *This means that* $T(n)$ *is* $\Theta(n)$ *by the master theorem.*

Example 11.8: *Consider the recurrence*

$$T(n) = 9T(n/3) + n^{2.5}.$$

In this case, $n^{\log_b a} = n^{\log_3 9} = n^2$. *Thus, we are in Case 3, since* $f(n)$ *is* $\Omega(n^{2+\epsilon})$ *(for* $\epsilon = 1/2$*) and* $af(n/b) = 9(n/3)^{2.5} = (1/3)^{1/2} f(n)$. *This means that* $T(n)$ *is* $\Theta(n^{2.5})$ *by the master theorem.*

Example 11.9: *Finally, consider the recurrence*

$$T(n) = 2T(n^{1/2}) + \log n.$$

Unfortunately, this equation is not in a form that allows us to use the master theorem. We can put it into such a form, however, by introducing the variable $k = \log n$, *which lets us write*

$$T(n) = T(2^k) = 2T(2^{k/2}) + k.$$

Substituting into this the equation $S(k) = T(2^k)$, *we get that*

$$S(k) = 2S(k/2) + k.$$

Now, this recurrence equation allows us to use master theorem, which specifies that $S(k)$ *is* $O(k \log k)$. *Substituting back for* $T(n)$ *implies* $T(n)$ *is* $O(\log n \log \log n)$.

High-Level Justification of the Master Theorem

Rather than rigorously prove Theorem 11.4, we instead discuss the justification behind the master theorem at a high level.

If we apply the iterative substitution method to the general divide-and-conquer recurrence equation, we get

$$\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a(aT(n/b^2) + f(n/b)) + f(n) = a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\
&\quad \vdots \\
&= a^{\log_b n}T(1) + \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) \\
&= n^{\log_b a}T(1) + \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i),
\end{aligned}$$

where the last substitution is based on the identity,

$$a^{\log_b n} = n^{\log_b a}.$$

Indeed, this equation is where the special function comes from.

Given this closed-form characterization of $T(n)$, we can intuitively see how each of the three cases is derived:

- Case 1 comes from the situation when $f(n)$ is small and the first term above dominates.
- Case 2 denotes the situation when each of the terms in the above summation is proportional to the others, so the characterization of $T(n)$ is $f(n)$ times a logarithmic factor.
- Finally, Case 3 denotes the situation when the first term is smaller than the second and the summation above is a sum of geometrically-decreasing terms that start with $f(n)$; hence, $T(n)$ is itself proportional to $f(n)$.

The proof of Theorem 11.4 formalizes this intuition, but instead of giving the details of this proof, we present more applications of the master theorem in the remainder of the chapter, for the analysis of various divide-and-conquer algorithms, including integer and matrix multiplication and the maxima-set problem.

11.2 Integer Multiplication

We consider, in this subsection, the problem of multiplying ***big integers***, that is, integers represented by a large number of bits that cannot be handled directly by the arithmetic unit of a single processor. Multiplying big integers has applications to data security, where big integers are used in encryption schemes.

Given two big integers I and J represented with n bits each, we can easily compute $I + J$ and $I - J$ in $O(n)$ time. Efficiently computing the product $I \cdot J$ using the common grade-school algorithm requires, however, $O(n^2)$ time. In the rest of this section, we show that by using the divide-and-conquer technique, we can design a subquadratic-time algorithm for multiplying two n-bit integers.

Let us assume that n is a power of two (if this is not the case, we can pad I and J with 0's). We can therefore divide the bit representations of I and J in half, with one half representing the ***higher-order*** bits and the other representing the ***lower-order*** bits. In particular, if we split I into I_h and I_l and J into J_h and J_l, then

$$\begin{aligned} I &= I_h 2^{n/2} + I_l, \\ J &= J_h 2^{n/2} + J_l. \end{aligned}$$

Also, observe that multiplying a binary number I by a power of two, 2^k, is trivial—it simply involves shifting left (that is, in the higher-order direction) the number I by k bit positions. Thus, provided a left-shift operation takes constant time, multiplying an integer by 2^k takes $O(k)$ time.

Let us focus on the problem of computing the product $I \cdot J$. Given the expansion of I and J above, we can rewrite $I \cdot J$ as

$$I \cdot J = (I_h 2^{n/2} + I_l) \cdot (J_h 2^{n/2} + J_l) = I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l.$$

Thus, we can compute $I \cdot J$ by applying a divide-and-conquer algorithm that divides the bit representations of I and J in half, recursively computes the product as four products of $n/2$ bits each (as described above), and then merges the solutions to these subproducts in $O(n)$ time using addition and multiplication by powers of two. We can terminate the recursion when we get down to the multiplication of two 1-bit numbers, which is trivial. This divide-and-conquer algorithm has a running time that can be characterized by the following recurrence:

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ 4T(n/2) + cn, & \text{if } n \geq 2, \end{cases}$$

for some constant $c > 0$. We can then apply the master theorem to note that the special function $n^{\log_b a} = n^{\log_2 4} = n^2$ in this case; hence, we are in Case 1 and $T(n)$ is $\Theta(n^2)$. Unfortunately, this is no better than the grade-school algorithm.

A Better Algorithm

The master theorem gives us some insight into how we might improve this algorithm. If we can reduce the number of recursive calls, then we will reduce the complexity of the special function used in the master theorem, which is currently the dominating factor in our running time. Fortunately, if we are a little more clever in how we define subproblems to solve recursively, we can in fact reduce the number of recursive calls by one. In particular, consider the product

$$(I_h - I_l) \cdot (J_l - J_h) = I_h J_l - I_l J_l - I_h J_h + I_l J_h.$$

This is admittedly a strange product to consider, but it has an interesting property. When expanded out, it contains two of the products we want to compute (namely, $I_h J_l$ and $I_l J_h$) and two products that can be computed recursively (namely, $I_h J_h$ and $I_l J_l$). Thus, we can compute $I \cdot J$ as follows:

$$I \cdot J = I_h J_h 2^n + [(I_h - I_l) \cdot (J_l - J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l.$$

This computation requires the recursive computation of three products of $n/2$ bits each, plus $O(n)$ additional work. Thus, it results in a divide-and-conquer algorithm with a running time characterized by the following recurrence equation (for $n \geq 2$):

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ 3T(n/2) + cn, & \text{if } n \geq 2, \end{cases}$$

for some constant $c > 0$.

Theorem 11.10: *We can multiply two n-bit integers in $O(n^{\log_2 3})$ time, which is $O(n^{1.585})$.*

Proof: We apply the master theorem with the special function,

$$n^{\log_b a} = n^{\log_2 3}.$$

Thus, we are in Case 1, which implies that $T(n)$ is $\Theta(n^{\log_2 3})$, which is itself $O(n^{1.585})$. ∎

Using divide-and-conquer, we have designed an algorithm for integer multiplication that is asymptotically faster than the straightforward quadratic-time method. We can actually do even better than this, achieving a running time that is "almost" $O(n \log n)$, by using a more complex divide-and-conquer algorithm called the ***fast Fourier transform***, which we discuss in Chapter 25.

11.3 Matrix Multiplication

Suppose we are given two $n \times n$ matrices X and Y, and we wish to compute their product $Z = XY$, which is defined so that

$$Z[i,j] = \sum_{k=0}^{e-1} X[i,k] \cdot Y[k,j],$$

which is an equation that immediately gives rise to a simple $O(n^3)$-time algorithm.

Another way of viewing this product is in terms of submatrices. That is, let us assume that n is a power of two and let us partition X, Y, and Z each into four $(n/2) \times (n/2)$ matrices, so that we can rewrite $Z = XY$ as

$$\begin{pmatrix} I & J \\ K & L \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix}.$$

Thus,

$$\begin{aligned} I &= AE + BG \\ J &= AF + BH \\ K &= CE + DG \\ L &= CF + DH. \end{aligned}$$

We can use this set of equations in a divide-and-conquer algorithm that computes $Z = XY$ by computing I, J, K, and L from the subarrays A through G. By the above equations, we can compute I, J, K, and L from the eight recursively computed matrix products on $(n/2) \times (n/2)$ subarrays, plus four additions that can be done in $O(n^2)$ time. Thus, the above set of equations give rise to a divide-and-conquer algorithm whose running time $T(n)$ is characterized by the recurrence

$$T(n) = 8T(n/2) + bn^2,$$

for some constant $b > 0$. Unfortunately, this equation implies that $T(n)$ is $O(n^3)$ by the master theorem; hence, it is no better than the straightforward matrix multiplication algorithm.

Interestingly, there is an algorithm known as ***Strassen's Algorithm***, which organizes arithmetic involving the subarrays A through G so that we can compute I, J, K, and L using just seven recursive matrix multiplications. We begin Strassen's algorithm with seven submatrix products:

$$\begin{aligned} S_1 &= A(F - H) \\ S_2 &= (A + B)H \\ S_3 &= (C + D)E \\ S_4 &= D(G - E) \\ S_5 &= (A + D)(E + H) \\ S_6 &= (B - D)(G + H) \\ S_7 &= (A - C)(E + F). \end{aligned}$$

Given these seven submatrix products, we can compute I as

$$\begin{aligned} I &= S_5 + S_6 + S_4 - S_2 \\ &= (A+D)(E+H) + (B-D)(G+H) + D(G-E) - (A+B)H \\ &= AE + BG. \end{aligned}$$

We can compute J as

$$\begin{aligned} J &= S_1 + S_2 \\ &= A(F-H) + (A+B)H \\ &= AF - AH + AH + BH \\ &= AF + BH. \end{aligned}$$

We can compute K as

$$\begin{aligned} K &= S_3 + S_4 \\ &= (C+D)E + D(G-E) \\ &= CE + DE + DG - DE \\ &= CE + DG. \end{aligned}$$

Finally, we can compute L as

$$\begin{aligned} L &= S_1 - S_7 - S_3 + S_5 \\ &= A(F-H) - (A-C)(E+F) - (C+D)E + (A+D)(E+H) \\ &= CF + DH. \end{aligned}$$

Thus, we can compute $Z = XY$ using seven recursive multiplications of matrices of size $(n/2) \times (n/2)$. Thus, we can characterize the running time $T(n)$ as

$$T(n) = 7T(n/2) + bn^2,$$

for some constant $b > 0$. Thus, by the master theorem, we have the following:

Theorem 11.11: *We can multiply two $n \times n$ matrices in $O(n^{\log 7})$ time.*

Thus, with a fair bit of additional complication, we can perform the multiplication for $n \times n$ matrices in time $O(n^{2.808})$, which is $o(n^3)$ time. As admittedly complicated as Strassen's matrix multiplication is, there are actually much more complicated matrix multiplication algorithms, with running times as low as $O(n^{2.376})$.

11.4 The Maxima-Set Problem

Let us now return to the problem of finding a ***maxima set*** for a set, S, of n points in the plane. This problem is motivated from ***multi-objective optimization***, where we are interested in optimizing choices that depend on multiple variables. For instance, in the introduction we used the example of someone wishing to optimize hotels based on the two variables of pool size and restaurant quality. A point is a ***maximum*** point in S if there is no other point, (x', y'), in S such that $x \leq x'$ and $y \leq y'$. Points that are not members of the maxima set can be eliminated from consideration, since they are dominated by another point in S. Thus, finding the maxima set of points can act as a kind of filter that selects out only those points that should be candidates for optimal choices. Moreover, such a filter can be quite effective, as we explore in Exercise A-11.4.

Given a set, S, of n points in the plane, there is a simple divide-and-conquer algorithm for constructing the maxima set of points in S. If $n \leq 1$, the maxima set is just S itself. Otherwise, let p be the median point in S according to a lexicographic ordering of the points in S, that is, where we order based primarily on x-coordinates and then by y-coordinates if there are ties. If the points have distinct x-coordinates, then we can imagine that we are dividing S using a vertical line through p. Next, we recursively solve the maxima-set problem for the set of points on the left of this line and also for the points on the right. Given these solutions, the maxima set of points on the right are also maxima points for S. But some of the maxima points for the left set might be dominated by a point from the right, namely the point, q, that is leftmost. So then we do a scan of the left set of maxima, removing any points that are dominated by q, until reaching the point where q's dominance extends. (See Figure 11.3.) The union of remaining set of maxima from the left and the maxima set from the right is the set of maxima for S. We give the details in Algorithm 11.4.

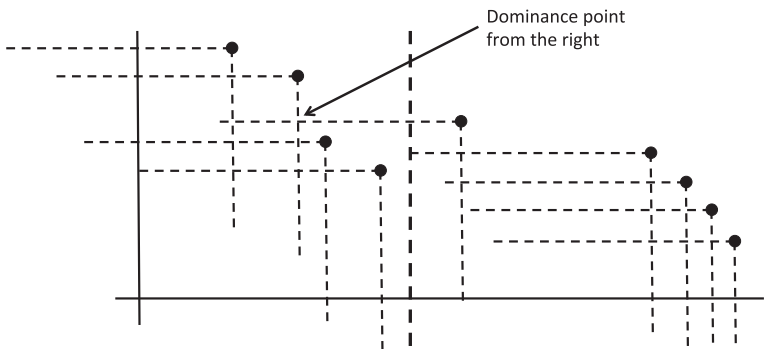


Figure 11.3: The combine step for the divide-and-conquer maxima-set algorithm.

Algorithm MaximaSet(S):

Input: A set, S, of n points in the plane
Output: The set, M, of maxima points in S

if $n \leq 1$ **then**
  return S
Let p be the median point in S, by lexicographic (x, y)-coordinates
Let L be the set of points lexicographically less than p in S
Let G be the set of points lexicographically greater than or equal to p in S
$M_1 \leftarrow$ MaximaSet(L)
$M_2 \leftarrow$ MaximaSet(G)
Let q be the lexicographically smallest point in M_2
for each point, r, in M_1 **do**
  if $x(r) \leq x(q)$ **and** $y(r) \leq y(q)$ **then**
    Remove r from M_1
return $M_1 \cup M_2$

Algorithm 11.4: A divide-and-conquer algorithm for the maxima-set problem.

Analysis of the Divide-and-Conquer Maxima-Set Algorithm

To analyze the divide-and-conquer maxima-set algorithm, there is a minor implementation detail in Algorithm 11.4 that we need to work out. Namely, there is the issue of how to efficiently find the point, p, that is the median point in a lexicographical ordering of the points in S according to their (x, y)-coordinates. There are two immediate possibilities. One choice is to use a linear-time median-finding algorithm, such as that given in Section 9.2. This achieves a good asymptotic running time, but adds some implementation complexity. Another choice is to sort the points in S lexicographically by their (x, y)-coordinates as a preprocessing step, prior to calling the MaxmaSet algorithm on S. Given this preprocessing step, the median point is simply the point in the middle of the list. Moreover, each time we perform a recursive call, we can be passing a sorted subset of S, which maintains the ability to easily find the median point each time. In either case, the rest of the nonrecursive steps can be performed in $O(n)$ time, so this implies that, ignoring floor and ceiling functions (as allowed by the analysis of Exercise C-11.5), the running time for the divide-and-conquer maxima-set algorithm can be specified as follows:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2, \end{cases}$$

for some constant $b \geq 1$. As has now been observed several times in this chapter, this implies that the running time for the divide-and-conquer maxima-set algorithm is $O(n \log n)$.

11.5 Exercises

Reinforcement

R-11.1 Characterize each of the following recurrence equations using the master theorem (assuming that $T(n) = c$ for $n < d$, for constants $c > 0$ and $d \geq 1$).

a. $T(n) = 2T(n/2) + \log n$
b. $T(n) = 8T(n/2) + n^2$
c. $T(n) = 16T(n/2) + (n \log n)^4$
d. $T(n) = 7T(n/3) + n$
e. $T(n) = 9T(n/3) + n^3 \log n$

R-11.2 Use the divide-and-conquer algorithm, from Section 11.2, to compute $10110011 \cdot 10111010$ in binary. Show your work.

R-11.3 Use Strassen's matrix multiplication algorithm to multiply the matrices

$$X = \begin{pmatrix} 3 & 2 \\ 4 & 8 \end{pmatrix} \qquad \text{and} \qquad Y = \begin{pmatrix} 1 & 5 \\ 9 & 6 \end{pmatrix}.$$

R-11.4 A complex number $a + b\mathbf{i}$, where $\mathbf{i} = \sqrt{-1}$, can be represented by the pair (a, b). Describe a method performing only three real-number multiplications to compute the pair (e, f) representing the product of $a + b\mathbf{i}$ and $c + d\mathbf{i}$.

R-11.5 What is the maxima set from the following set of points:

$$\{(7,2), (3,1), (9,3), (4,5), (1,4), (6,9), (2,6), (5,7), (8,6)\}?$$

R-11.6 Give an example of a set of n points in the plane such that every point is a maximum point, that is, no point in this set is dominated by any other point in this set.

Creativity

C-11.1 Consider the recurrence equation,

$$T(n) = 2T(n-1) + 1,$$

for $n > 1$, where $T(n) = 1$ for $n = 1$. Prove that $T(n)$ is $O(2^n)$.

C-11.2 There are several cases of divide-and-conquer recurrence relations that are not covered in the master theorem. Nevertheless, the intuition for the master theorem can still give us some guidance. Derive and prove a general solution, like that given in the master theorem, to the following recurrence equation (assuming $T(n)$ is a constant for n less than or equal to a given constant, $a \geq 2$):

$$T(n) = a\,T(n/a) + n \log \log n.$$

C-11.3 There is a sorting algorithm, "Stooge-sort," which is named after the comedy team, "The Three Stooges." if the input size, n, is 1 or 2, then the algorithm sorts the input immediately. Otherwise, it recursively sorts the first $2n/3$ elements, then the last $2n/3$ elements, and then the first $2n/3$ elements again. The details are shown in Algorithm 11.5. Show that Stooge-sort is correct and characterize the running time, $T(n)$, for Stooge-sort, using a recurrence equation, and use the master theorem to determine an asymptotic bound for $T(n)$.

Algorithm StoogeSort(A, i, j):
Input: An array, A, and two indices, i and j, such that $1 \leq i \leq j \leq n$
Output: Subarray, $A[i..j]$, sorted in nondecreasing order
$n \leftarrow j - i + 1$ // The size of the subarray we are sorting
if $n = 2$ **then**
  if $A[i] > A[j]$ **then**
    Swap $A[i]$ and $A[j]$
else if $n > 2$ **then**
  $m \leftarrow \lfloor n/3 \rfloor$
  StoogeSort$(A, i, j - m)$ // Sort the first part
  StoogeSort$(A, i + m, j)$ // Sort the last part
  StoogeSort$(A, i, j - m)$ // Sort the first part again
return A

Algorithm 11.5: Stooge-sort.

C-11.4 Consider the Stooge-sort algorithm, shown in Algorithm 11.5, and suppose we change the assignment statement for m (on line 6) to the following:

$$m \leftarrow \max\{1, \lfloor n/4 \rfloor\}$$

Characterize the running time, $T(n)$, in this case, using a recurrence equation, and use the master theorem to determine an asymptotic bound for $T(n)$.

C-11.5 Consider a version of the divide-and-conquer recurrence equation based on use of the ceiling function, as follows:

$$T(n) = a\,T(\lceil n/b \rceil) + f(n),$$

where $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, for some integer constant, $k \geq 0$. Show that, by using the inequality,

$$T(n) \leq a\,T(n/b + 1) + f(n),$$

for sufficiently large n, $T(n)$ is $O(n^{\log_b a} \log^{k+1} n)$.

Applications

A-11.1 A very common problem in computer graphics is to approximate a complex shape with a ***bounding box***. For a set, S, of n points in 2-dimensional space, the idea is to find the smallest rectangle, R, with sides parallel to the coordinate axes that

contains all the points in S. Once S is approximated by such a bounding box, we can often speed up lots of computations that involve S. For example, if R is completely obscured some object in the foreground, then we don't need to render any of S. Likewise, if we shoot a virtual ray and it completely misses R, then it is guarantee to completely miss S. So doing comparisons with R instead of S can often save time. But this savings is wasted if we spend a lot of time constructing R; hence, it would be ideal to have a fast way of computing a bounding box, R, for a set, S, of n points in the plane. Note that the construction of R can be reduced to two instances of the problem of simultaneously finding the minimum and the maximum in a set of n numbers; namely, we need only do this for the x-coordinates in S and then for the y-coordinates in S. Therefore, design a divide-and-conquer algorithm for finding both the minimum and the maximum element of n numbers using no more than $3n/2$ comparisons.

A-11.2 Given a set, P, of n teams in some sport, a ***round-robin tournament*** is a collection of games in which each team plays each other team exactly once. Such round-robin tournaments are often used as the first round for establishing the order of teams (and their seedings) for later single- or double-elimination tournaments. Design an efficient algorithm for constructing a round-robin tournament for a set, P, of n teams assuming n is a power of 2.

A-11.3 In some multi-objective optimization problems (such as that exemplified by the choosing of hotels based on the sizes of their pools and quality scores of their restaurants), we may have different kinds of constraints involving the variables instead of the desire to avoid points dominated by others. Suppose, for example, that we have a two-variable optimization problem where we have a set, C, of constraints of the form,

$$y \geq m_i x + b_i,$$

for real numbers, m_i and b_i, for $i = 1, 2, \ldots, n$. In such a case, we would like to restrict our attention to points that satisfy all the linear inequality constraints in C. One way to address this desire is to construct the ***upper envelope*** for C, which is a representation of the function, $f(x)$, defined as

$$f(x) = \max_{1 \leq i \leq n} \{m_i x + b_i\},$$

where the (m_i, b_i) pairs are from the inequalities in C. Equivalently, the upper envelope is a representation of the part of the plane determined by the intersection of all the halfplanes determined by the inequalities in C. If we consider how this function behaves as x goes from $-\infty$ to $+\infty$, we note that each inequality in C can appear at most once during this process. Thus, we can represent f in terms of a set, $S = \{(a_1, b_1, i_1), (a_2, b_2, i_2), \ldots, (a_k, b_k, i_k)\}$, such that each triple, (a_j, b_j, i_j) in S, represents the fact that interval $[a_j, b_j]$ is a maximal interval such that $f(x) = m_{i_j} x + b_{i_j}$. Describe an $O(n \log n)$-time algorithm for computing such a representation, S, of the upper envelope of the linear inequalities in C.

A-11.4 Selecting out a maxima set of points from among a set of points in the plane seems is intended to filter away a lot of points from consideration in a multi-objective optimization problem. For example, if we grade hotels by the sizes of their pools and the quality of their restaurants, then we would hope to have a small set of hotels to choose from that are not dominated by any other hotel along these

axes. So let us analyze the expected size of a maxima set of randomly chosen points in the plane with distinct x- and y-coordinates. We can model such as set by defining a random set of n points in the plane by first randomly permuting the sequence of numbers, $(1, 2, \ldots, n)$, giving $(y_1, y_2, \ldots, y_n)$. Let us then define a set of two dimensional points as the set,

$$R = \{(1, y_1), (2, y_2), \ldots, (n, y_n)\}.$$

Since it is the relative order of points that determines whether a point is a maximum point or not, this set of points with integer coordinates fully captures all the possibilities, with respect to the maxima-set problem, for any random set of points in the plane with distinct x- and y-coordinates. So as to show just how effective it is to select out the maxima set of points from such a set as a filtering step (hence, it should be a good filter in practice), show that the expected size of the maxima set for R is $O(\log n)$.

A-11.5 Suppose you have a high-performance computer, which has a CPU containing a dedicated bank of $k > 2$ ***optimization*** registers. This bank of registers supports the insertion of key-value pair, (x, y), into the bank of registers in $O(1)$ time, assuming that there are fewer than k numbers already stored in the bank. In addition, this bank of registers allows you to remove and return a key-value pair, (x, y), with smallest key, x, from the bank of registers in $O(1)$ time. Show that you can use this high-performance computer to sort n numbers in $O(n \log n / \log k)$ time.

A-11.6 Suppose you have a geometric description of the buildings of Manhattan and you would like to build a representation of the New York skyline. That is, suppose you are given a description of a set of rectangles, all of which have one of their sides on the x-axis, and you would like to build a representation of the union of all these rectangles. Formally, since each rectangle has a side on the x-axis, you can assume that you are given a set, $S = \{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]\}$ of subintervals in the interval $[0, 1]$, with $0 \leq a_i < b_i \leq 1$, for $i = 1, 2, \ldots, n$, such that there is an associated height, h_i, for each interval $[a_i, b_i]$ in S. The ***skyline*** of S is defined to be a list of pairs $[(x_0, c_0), (x_1, c_1), (x_2, c_2), \ldots, (x_m, c_m), (x_{m+1}, 0)]$, with $x_0 = 0$ and $x_{m+1} = 1$, and ordered by x_i values, such that, each subinterval, $[x_i, x_{i+1}]$, is the maximal subinterval that has a single highest interval, which is at height c_i, in S, containing $[x_i, x_{i+1}]$, for $i = 0, 1, \ldots, m$. Design an $O(n \log n)$-time algorithm for computing the skyline of S.

Chapter Notes

The master theorem for solving divide-and-conquer recurrences traces its origins to a paper by Bentley, Haken, and Saxe [30]. The divide-and-conquer algorithm for multiplying two large integers in $O(n^{1.585})$ time is generally attributed to the Russians Karatsuba and Ofman [117]. The matrix multiplication algorithm we present is due to Strassen [202]. Kung, Luccio, and Preparata [138] present a divide-and-conquer algorithm for finding maxima sets. The Stooge-sort algorithm was first given by Cormen *et al.* (e.g., see [51]).