CHAPTER 7

Induction and Recursion

Introduction

Suppose $\mathcal{A}(n)$ is an assertion that depends on n. We use *induction* to prove that $\mathcal{A}(n)$ is true when we show that

- it's true for the smallest value of n and
- if it's true for everything less than n, then it's true for n.

Closely related to proof by induction is the notion of a recursion. A *recursion* describes how to calculate a value from previously calculated values. For example, $n!$ can be calculated by

$$n! = \begin{cases} 1, & \text{if } n = 0; \\ n \cdot (n-1)!, & \text{otherwise.} \end{cases}$$

We discussed recursions briefly in Section 1.4.

Notice the similarity between the two ideas: There is something to get us started and then each new thing depends on similar previous things. Because of this similarity, recursions often appear in inductively proved theorems as either the theorem itself or a step in the proof. We'll study inductive proofs and recursive equations in the next section.

Inductive proofs and recursive equations are special cases of the general concept of a recursive approach to a problem. Thinking recursively is often fairly easy when one has mastered it. Unfortunately, people are sometimes defeated before reaching this level. We've devoted Section 2 to helping you avoid some of the pitfalls of recursive thinking.

In Section 3 we look at some concepts related to recursive algorithms including proving correctness, recursions for running time, local descriptions and computer implementation.

Not only can recursive methods provide more natural solutions to problems, they can also lead to faster algorithms. This approach, which is often referred to as "divide and conquer," is discussed in Section 4. The best sorting algorithms are of the divide and conquer type, so we'll see a bit more of this in Chapter 8.

7.1 Inductive Proofs and Recursive Equations

The concept of proof by induction is discussed in Appendix A (p. 361). We strongly recommend that you review it at this time. In this section, we'll quickly refresh your memory and give some examples of combinatorial applications of induction. Other examples can be found among the proofs in previous chapters. (See the index under "induction" for a listing of the pages.)

We recall the theorem on induction and some related definitions:

Theorem 7.1 Induction *Let $\mathcal{A}(m)$ be an assertion, the nature of which is dependent on the integer m. Suppose that we have proved $\mathcal{A}(n)$ for $n_0 \le n \le n_1$ and the statement*

"If $n > n_1$ and $\mathcal{A}(k)$ is true for all k such that $n_0 \le k < n$, then $\mathcal{A}(n)$ is true."

Then $\mathcal{A}(m)$ is true for all $m \ge n_0$.

Definition 7.1 *The statement "$\mathcal{A}(k)$ is true for all k such that $n_0 \le k < n$" is called the* **induction assumption** *or* **induction hypothesis** *and proving that this implies $\mathcal{A}(n)$ is called the* **inductive step**. *The cases $n_0 \le n \le n_1$ are called the* **base cases**.

Proof: We now prove the theorem. Suppose that $\mathcal{A}(n)$ is false for some $n \ge n_0$. Let m be the least such n. We cannot have $m \le n_0$ because one of our hypotheses is that $\mathcal{A}(n)$ has been proved for $n_0 \le n \le n_1$. On the other hand, since m is as small as possible, $\mathcal{A}(k)$ is true for $n_0 \le k < m$. By the inductive step, $\mathcal{A}(m)$ is also true, a contradiction. Hence our assumption that $\mathcal{A}(n)$ is false for some n is itself false; in other words, $\mathcal{A}(n)$ is never false. □

Example 7.1 The parity of binary trees The numbers b_n, $n \ge 1$, given by

$$b_1 = 1 \quad \text{and} \quad b_n = b_1 b_{n-1} + b_2 b_{n-2} + \cdots + b_{n-1} b_1 \text{ for } n > 1 \qquad 7.1$$

count the number of "unlabeled full binary RP-trees." We prove this recursion in Example 7.10 and study these trees more in Section 9.3 (p. 259). For now, all that matters is (7.1), not what the b_n count.

Using the definitions, we compute the first few values:

$$b_1 = 1 \quad b_2 = 1 \quad b_3 = 2 \quad b_4 = 5 \quad b_5 = 14 \quad b_6 = 42 \quad b_7 = 132.$$

Most values appear to be even. If you compute b_8, you will discover that it is odd. Since b_1, b_2, b_4 and b_8 are the only odd values with $n \le 8$, we conjecture that b_n is odd if and only if n is a power of 2. Call the conjecture $\mathcal{A}(n)$. How should we choose n_0 and n_1? Since the recursion in (7.1) is only valid for $n > 1$, the case $n = 1$ appears special. Thus we try letting $n_0 = n_1 = 1$ and using the recursion for $n > 1$.

Since $b_1 = 1$ is odd, $\mathcal{A}(1)$ is true. We now consider $n > 1$. If n is odd, let $k = (n-1)/2$ and note that we can write the recursion as

$$b_n \;=\; 2(b_1 b_{n-1} + b_2 b_{n-2} + \cdots + b_k b_{k+1}).$$

Hence b_n is even and no induction was needed. Now suppose n is even and let $k = n/2$. Now our recursion becomes

$$b_n \;=\; 2(b_1 b_{n-1} + b_2 b_{n-2} + \cdots + b_{k-1} b_{k+1}) + b_k^2.$$

Hence b_n is odd if and only if $b_k = b_{n/2}$ is odd. By the induction assumption, $b_{n/2}$ is odd if and only if $n/2$ is a power of 2. Since $n/2$ is a power of 2 if and only if n is a power of 2, we are done. □

Example 7.2 The Fibonacci numbers One definition of the Fibonacci numbers is

$$F_0 = 0, \quad F_1 = 1, \quad \text{and} \quad F_{n+1} = F_n + F_{n-1} \ \text{ for } n > 0. \tag{7.2}$$

We want to prove that

$$F_n \;=\; \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \quad \text{for } n \geq 0. \tag{7.3}$$

Let that be $\mathcal{A}(n)$. Since (7.2) is our only information, we'll use it to prove (7.3). We must either think of our induction in terms of proving $\mathcal{A}(n+1)$ or rewrite the recursion as $F_n = F_{n-1} + F_{n-2}$. We'll use the latter approach Since the recursion starts at $n+1 = 2$, we'll have to prove $\mathcal{A}(0)$ and $\mathcal{A}(1)$ separately. Hence $n_0 = 0$ and $n_1 = 1$ in Theorem 7.1. Since

$$\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^0 - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^0 \;=\; 1-1 \;=\; 0,$$

$\mathcal{A}(0)$ is true. Since

$$\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^1 - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^1 \;=\; 1,$$

$\mathcal{A}(1)$ is true.

Now for the induction. We want to prove (7.3) for $n \geq 1$. By the recursion, $F_n \;=\; F_{n-1} + F_{n-2}$. Now use $\mathcal{A}(n-1)$ and $\mathcal{A}(n-2)$ to replace F_{n-1} and F_{n-2}. Thus

$$F_n \;=\; \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n-1} + \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n-2},$$

and so we want to prove that

$$\begin{aligned}\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \;&=\; \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\\ &\quad + \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n-2}.\end{aligned}$$

Consider the three terms that involve $1+\sqrt{5}$. Divide by $\left(\frac{1-\sqrt{5}}{2}\right)^{n-2}$ and multiply by $\sqrt{5}$ to see that they combine correctly if

$$\left(\frac{1+\sqrt{5}}{2}\right)^2 \;=\; \frac{1+\sqrt{5}}{2} + 1,$$

which is true by simple algebra. The three terms with $1-\sqrt{5}$ are handled similarly. □

Example 7.3 Disjunctive form for Boolean functions We will consider functions with domain $\{0,1\}^n$ and range $\{0,1\}$. A typical function is written $f(x_1,\ldots,x_n)$. These functions are called *Boolean functions* on n variables. With 0 interpreted as "false" and 1 as "true," we can think of $x_1,\ldots,x_n$ as statements which are either true or false. In this case, f can be thought of as a complicated statement built from $x_1,\ldots,x_n$ which is true or false depending on the truth and falsity of the x_i's.

For $y_1,\ldots,y_k \in \{0,1\}$, $y_1y_2\cdots y_k$ is normal multiplication; that is,

$$y_1y_2\cdots y_k \;=\; \begin{cases} 1, & \text{if } y_1=y_2=\cdots=y_k=1;\\ 0, & \text{otherwise.}\end{cases}$$

Define

$$y_1+y_2+\cdots+y_m \;=\; \begin{cases} 0, & \text{if } y_1=y_2=\cdots=y_k=0;\\ 1, & \text{otherwise.}\end{cases}$$

With the true-false interpretation, multiplication corresponds to "and" and + corresponds to "or." Define $x' = 1-x$, the *complement* of x.

A function f is said to be written in *disjunctive form* if

$$f(x_1,\ldots,x_n) \;=\; A_1+\cdots+A_k, \tag{7.4}$$

where each A_j is the product of terms, each of which is either an x_i or an x_i'. For example, let $g(x_1,x_2,x_3)$ be 1 if exactly two of x_1, x_2 and x_3 are 1, and 0 otherwise. Then

$$g(x_1,x_2,x_3) \;=\; x_1x_2x_3' + x_1x_2'x_3 + x_1'x_2x_3$$

and

$$g(x_1,x_2,x_3)' \;=\; x_1'x_2' + x_1'x_3' + x_2'x_3' + x_1x_2x_3$$

If $k=0$ in (7.4) (i.e., no terms present), then it is interpreted to be 0 for all $x_1,\ldots,x_n$.

We will prove

Theorem 7.2 *Every Boolean function can be written in disjunctive form.*

Let $\mathcal{A}(n)$ be the theorem for Boolean functions on n variables. There are $2^2 = 4$ Boolean functions on 1 variable. Here are the functions and disjunctive forms for them:

$(f(0), f(1))$	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
form	x_1x_1'	x_1	x_1'	x_1+x_1'

This proves $\mathcal{A}(1)$.

For $n>1$ we have

$$f(x_1,\ldots,x_n) \;=\; \bigl(g_0(x_1,\ldots,x_{n-1})\,x_n'\bigr) + \bigl(g_1(x_1,\ldots,x_{n-1})\,x_n\bigr), \tag{7.5}$$

where $g_k(x_1,\ldots,x_{n-1}) = f(x_1,\ldots,x_{n-1},k)$. To see this, note that when $x_n=0$ the right side of (7.5) is $(g_0\cdot 1)+(g_1\cdot 0) = g_0 = f$ and when $x_n = 1$ it is $(g_0\cdot 0)+(g_0\cdot 1) = g_1 = f$.

By the induction assumption, both g_0 and g_1 can be written in disjunctive form, say

$$g_0 \;=\; A_1+\cdots+A_a \qquad\text{and}\qquad g_1 \;=\; B_1+\cdots+B_b. \tag{7.6}$$

We claim that

$$(C_1+\cdots+C_c)y \;=\; C_1y+\cdots+C_cy. \tag{7.7}$$

If this is true, then it can be used in connection with (7.6) in (7.5) to complete the inductive step.

To prove (7.7), notice that

$$\text{(the left side of (7.7) equals 1)} \quad \text{if and only if} \quad (y=1 \text{ and some } C_i=1).$$

This is equivalent to

$$\text{(the left side of (7.7) equals 1)} \quad \text{if and only if} \quad (\text{some } C_i y = 1).$$

however,

$$\text{(the right side of (7.7) equals 1)} \quad \text{if and only if} \quad (\text{some } C_i y = 1).$$

This proves that the left side of (7.7) equals 1 if and only if the right side equals 1. Thus (7.7) is true. □

Suppose you have a result that you are trying to prove. If you are unable to do so, you might try to prove a bit less because proving less should be easier. That is not always true for proofs by induction. Sometimes it is easier to prove more! How can this be? The statement $\mathcal{A}(n)$ is not just the thing you want to prove, it is also the assumption that you have to help you prove $\mathcal{A}(m)$ for $m > n$. Thus, a stronger inductive hypothesis gives you more to prove *and* more to prove it with. This should be emphasized:

Principle More may be better *If the induction hypothesis seems to weak to carry out an inductive proof, consider trying to prove a stronger theorem.*

We've already encountered this in proving Theorem 6.2 (p. 153). We had wanted to prove that every connected graph has a lineal spanning tree. We might have used

$\mathcal{A}_1(n)$: "If G is an n-vertex connected graph, it has a lineal spanning tree."

Instead we used the stronger statement

$\mathcal{A}_2(n)$: "If G is an n-vertex connected graph containing the vertex r, it has a lineal spanning tree with root r."

If you try to prove $\mathcal{A}_1(n)$ by induction, you'll soon run into problems. Try it. The following example illustrates the usefulness of generalizing the hypothesis for some inductive proofs.

Example 7.4 Graphs and Ramsey Theory Let k be a positive integer and let $G = (V, E)$ be an arbitrary simple graph. Can we find a subset $S \subseteq V$ such that $|S| = k$ and either

- for all $x, y \in S$, we have $\{x, y\} \in E$ or
- for all $x, y \in S$, we have $\{x, y\} \notin E$?

If $|V|$ is too small, e.g., $|V| < k$, the answer is obviously "No." Instead, we might ask, "Is there an $N(k)$ such that there exists an S with the above properties whenever $|V| \geq N(k)$?" You should be able to see that, if we find some value which works for $N(k)$, then any larger value will also work.

It's easy to see that we can choose $N(2) = 2$: Pick any two $x, y \in V$, let $S = \{x, y\}$. Since $\{x, y\}$ is either an edge in G or is not, we are done.

Let's try to show that $N(3)$ exists and find the smallest possible value we can choose for it.

You should find a simple graph G with $|V| = 5$ for which the result is false when $k = 3$; that is, for any set of three vertices in G there is at least one pair that are joined by an edge and at least one pair that are not joined by an edge. Having done this, you've shown that, if $N(3)$ exists it must be greater than 5.

We now prove that we may take $N(3) = 6$. Select any $v \in V$. Of the remaining five or more vertices in V there must be at least three that are joined to v or at least three that are not joined to v. We do the first case and leave the latter for you. Let x_1, x_2 and x_3 be three vertices joined to v. If $\{x_i, x_j\} \in E$, then all pairs of vertices in $\{v, x_i, x_j\}$ are joined by edges and we are done. If $\{x_i, x_j\} \notin E$ for all i and j, then none of the pairs of vertices in $\{x_1, x_2, x_3\}$ are joined by edges and, again, we are done. We have shown that we may take $N(3) = 6$.

Since the proof that $N(3)$ exists involved reduction to a smaller situation, it suggests that we might be able to prove the existence of $N(k)$ by induction on k. How would this work? Here's a brief sketch. We'd select $v \in V$ and note the existence of a large enough set all of whose vertices were either joined to v by edges or not joined to v. As above, we could assume the former case. We now want to know that there exist either $k-1$ vertices all joined to v or k vertices not joined to v. This requires the induction assumption, but we are stuck because we are looking for either a set of size $k-1$ or one of size k, which are two different sizes. We can get around this problem by strengthening the statement of the theorem to allow two different sizes. Here's the theorem we'll prove.

Theorem 7.3 A special case of Ramsey's Theorem *There exists a function $N(k_1, k_2)$, defined for all positive integers k_1 and k_2, such that, for all simple graphs $G = (V, E)$ with at least $N(k_1, k_2)$ vertices, there is a set $S \subseteq V$ such that either*

- *$|S| = k_1$ and $\{x, y\} \in E$ for all $x \neq y$ both in S, or*
- *$|S| = k_2$ and $\{x, y\} \notin E$ for all $x \neq y$ both in S.*

In fact, we may define an acceptable $N(k_1, k_2)$ recursively by

$$N(k_1,k_2) = \begin{cases} 1, & \text{if } k_1 = 1; \\ 1, & \text{if } k_2 = 1; \\ N(k_1-1,k_2)+N(k_1,k_2-1)+1, & \text{otherwise.} \end{cases} \qquad 7.8$$

If you have mistakenly assumed that $N(k_1, k_2)$ is uniquely defined—an easy error to make—the phrase "an acceptable $N(k_1, k_2)$" in the theorem probably bothers you. Look back over our earlier discussion of $N(k)$, which is the case $k_1 = k_2$. We said that $N(k)$ was any number such that if we had at least $N(k)$ vertices something was true, and we observed that if some value worked for $N(k)$, any larger value would also work. Of course we could look for the smallest possible choice for $N(k)$. We found that this is 2 when $k = 2$ and is 6 when $k = 3$. The theorem does not claim that the recursion (7.8) gives the smallest possible choice for $N(k_1, k_2)$. In fact, it tends to give numbers that are much too big. Since we showed earlier that the smallest possible value for $N(3, 3)$ is 6, you might mistakenly think that finding the smallest is easy. In fact, the smallest possible value of $N(k_1, k_2)$ is unknown for almost all (k_1, k_2).

Proof: We'll use induction on $n = k_1 + k_2$.

Before starting the induction step, we'll do the case in which $k_1 = 1$ or $k_2 = 1$ (or both). If $k_1 = 1$, choose $s \in V$ and set $S = \{s\}$. The theorem is trivially true because there are no $x \neq y$ in S. Similarly, it is trivially true if $k_2 = 1$.

We now carry out the inductive step. By the previous paragraph, we can assume that $k_1 > 1$ and $k_2 > 1$. Choose $v \in V$ and define

$$\begin{aligned} V_1 &= \Big\{x \in V - \{v\} \,\Big|\, \{x, v\} \in E\Big\} \\ V_2 &= \Big\{x \in V - \{v\} \,\Big|\, \{x, v\} \notin E\Big\}. \end{aligned}$$

It follows by (7.8) that either $|V_1| \geq N(k_1 - 1, k_2)$ or $|V_2| \geq N(k_1, k_2 - 1)$. We assume the former. (The argument for the latter case would be very similar to the one we are about to give.)

Look at the graph $(V_1, E \cap \mathcal{P}_2(V_1))$. Since $|V_1| \geq N(k_1 - 1, k_2)$, it follows from the inductive hypothesis that there is a set $S' \subseteq V_1$ such that either

- $|S'| = k_1 - 1$ and $\{x, y\} \in E$ for all $x \neq y$ both in S', or
- $|S'| = k_2$ and $\{x, y\} \notin E$ for all $x \neq y$ both in S'.

If the former is true, let $S = S' \cup \{v\}$; otherwise, let $S = S'$. This completes the proof.

A more general form of Ramsey's Theorem asserts that there exists a function $N_r(k_1, \ldots, k_d)$ such that for all V with $|V| \geq N_r(k_1, \ldots, k_d)$ and all f from $\mathcal{P}_r(V)$ to $\underline{d}$, there exists an $i \in \underline{d}$ and a set $S \subseteq V$ such that $|S| = k_i$ and $f(e) = i$ for all $e \in \mathcal{P}_r(S)$. The theorem we proved is the special case $N_2(k_1, k_2)$ and $f(e)$ is 1 or 2 according as $e \in E$ or $e \notin E$. Although the more general statement looks fairly complicated, its no harder to prove than the special case—provided you don't get lost in all the notation. You might like to try proving it. $\square$

Exercises

In these exercises, indicate clearly

(i) what $\mathcal{A}(n)$ is,

(ii) what the inductive step is and

(iii) where the inductive hypothesis is used.

7.1.1. Indicate (i)–(iii) in the proof of the rank formula (Theorem 3.1 (p. 76)).

7.1.2. Indicate (i)–(iii) in the proof of the greedy algorithm for unranking in Section 3.2 (p. 80).

7.1.3. Do Exercise 1.3.11.

7.1.4. For $n \geq 0$, let D_n be the number of derangements (permutations with no fixed points) of an n-set. By convention, $D_0 = 1$. The next few values are $D_1 = 0$, $D_2 = 1$, $D_3 = 2$ and $D_4 = 9$. Here are some statements about D_n.

$$\begin{aligned}
&\text{(i)}\ D_n = nD_{n-1} + (-1)^n \quad \text{for } n \geq 1.\\
&\text{(ii)}\ D_n = (n-1)(D_{n-1} + D_{n-2}) \quad \text{for } n \geq 2.\\
&\text{(iii)}\ D_n = n! \sum_{k=0}^{n} \frac{(-1)^k}{k!}.
\end{aligned}$$

(a) Use (i) to prove (ii). (Induction is not needed.)

(b) Use (ii) to prove (i).

(c) Use (i) to prove (iii).

(d) Use (iii) to prove (i). (Induction is not needed)

7.1.5. Write the following Boolean functions in disjunctive form. The functions are given in two-line form.

(a) $\begin{pmatrix} 0,0 & 0,1 & 1,0 & 1,1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$.

(b) $\begin{pmatrix} 0,0 & 0,1 & 1,0 & 1,1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$.

(c) $\begin{pmatrix} 0,0,0 & 0,0,1 & 0,1,0 & 0,1,1 & 1,0,0 & 1,0,1 & 1,1,0 & 1,1,1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$.

(d) $\begin{pmatrix} 0,0,0 & 0,0,1 & 0,1,0 & 0,1,1 & 1,0,0 & 1,0,1 & 1,1,0 & 1,1,1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$.

7.1.6. Write the following Boolean functions in disjunctive form.

(a) $(x_1 + x_3)(x_2 + x_4)$.

(b) $(x_1 + x_2)(x_1 + x_3)(x_2 + x_3)$.

7.1.7. A Boolean function f is written in *conjunctive form* if $f = A_1A_2\cdots$, where A_i is the "or" of terms each of which is either an x_i or an x_i'. Prove that every Boolean function can be written in conjunctive form.
Hint. The proof parallels that in Example 7.3. The hardest part is probably finding the equation that replaces (7.5).

7.1.8. Part II contains a variety of results that are proved by induction. Some appear in the text and some in the exercises. Write careful inductive proofs for each of the following.

(a) Every connected graph has a lineal spanning tree.

(b) The number of ways to color a simple graph G with x colors is a polynomial in x. (Do this by deletion and contraction.)

(c) Euler's relation: $v - e + f = 2$.

(d) Every planar graph can be colored with 5 colors.

(e) Using the fact that every tree has a leaf, prove that an n-vertex tree has exactly $n-1$ edges.

(f) Every n-vertex connected graph has at least $n-1$ edges.

*7.1.9. Using the definition of the Fibonacci numbers in Example 7.2, prove that

$$F_{n+k+1} = F_{n+1}F_{k+1} + F_nF_k \quad \text{for} \quad k \geq 0 \text{ and } n \geq 0.$$

Do *not* use formula (7.3).
Hint. You may find it useful to note that $n + k + 1 = (n - 1) + (k + 1) + 1$.

7.2 Thinking Recursively

A *recursive formula* tells us how to compute the value we are interested in terms of earlier ones. (The "earliest" values are specified separately.) How many can you recall from previous chapters? A *recursive definition* describes more complicated instances of a concept in terms of simpler ones. (The "simplest" instances are specified separately.) These are examples of the recursive approach, which we defined at the beginning of this part:

Definition 7.2 Recursive approach *A* **recursive approach** *to a problem consists of two parts:*

1. *The problem is reduced to one or more problems of the same kind which are simpler in some sense.*
2. *There is a set of simplest problems to which all others are reduced after one or more steps. Solutions to these simplest problems are given.*

This definition focuses on tearing down (reduction to simpler cases). Sometimes it may be easier or better to think in terms of building up (construction of bigger cases). We can simply turn Definition 7.2 on its head:

Definition 7.3 Recursive solution *We have a* **recursive solution** *to the problem (proof, algorithm, data structure, etc.) if the following two conditions hold.*

1. *The set of simplest problems can be dealt with (proved, calculated, sorted, etc.).*
2. *The solution to any other problem can be built from solutions to simpler problems, and this process eventually leads back to the simplest problems.*

Let's look briefly at some examples where recursion can be used. Suppose that we are given a collection of things and a problem associated with them. Examples of things and problems are

- assertions $\mathcal{A}(n)$ that we want to prove;
- the binomial coefficients $C(n,k)$ that we want to compute using $C(n,k) = C(n-1,k) + C(n-1,k-1)$ from Section 1.4;
- the recursion $D_n = (n-1)(D_{n-1} + D_{n-2})$ for derangements that we want to prove by a direct combinatorial argument;
- lists to sort;
- RP-trees that we want to define.

Suppose we have some binary relation between these things which we'll denote by "simpler than." If there is nothing simpler than a thing X, we call X "simplest." There may be several simplest things. In the examples just given, we'll soon see that the following notions are appropriate.

- $\mathcal{A}(n)$ is simpler than $\mathcal{A}(m)$ if $n < m$ and $\mathcal{A}(1)$ is the simplest thing.
- $C(n,k)$ is simpler than $C(m,j)$ if $n < m$ and the $C(0,k)$'s are the simplest things.

- D_n is simpler that D_m if $n > m$ and the simplest things are D_0 and D_1.
- One list is simpler than another if it contains fewer items and the lists with one item are the simplest things.
- One tree is simpler than another if it contains less vertices and the one vertex tree is the simplest.

Example 7.5 The induction theorem The induction theorem in the previous section solves the problem of proving $\mathcal{A}(n)$ recursively. There is only one simplest problem: $\mathcal{A}(1)$. We are usually taking a reduction viewpoint when we prove something by induction. $\square$

Example 7.6 Calculating binomial coefficients Find a method for calculating the binomial coefficients $C(n,k)$. As indicated above, we let the simplest values be those with $n = 0$. From Chapter 1 we have

- $C(0,k) \;=\; \begin{cases} 1, & \text{if } k=0; \\ 0, & \text{otherwise.} \end{cases}$
- $C(n,k) \;=\; C(n-1,k-1) + C(n-1,k)$.

This solves the problem recursively. $\square$

Is the derivation of the binomial coefficient recursion done by reduction or construction? We can derive it by dividing the k-subsets of $\underline{n}$ into those that contain n and those that do not. This can be regarded as reduction or construction. Such ambiguities are common because the two concepts are simply different facets of the same thing. Nevertheless, it is useful to explore reduction versus construction in problems so as to gain facility with solving problems recursively. We do this now for derangements.

Example 7.7 A recursion for derangements A derangement is a permutation without fixed points and D_n is the number of derangements of an n-set. In Exercise 7.1.4 the recursion

$$D_n \;=\; (n-1)(D_{n-1} + D_{n-2}) \quad \text{for} \quad n \ge 2 \qquad 7.9$$

with initial conditions $D_0 = 1$ and $D_1 = 0$ was stated without proof. We now give a derivation using reduction and construction arguments.

Look at a derangement of $\underline{n}$ in cycle form. Since a derangement has no fixed points, no cycles have length 1. We look at the cycle of the derangement that contains n.

(a) If this cycle has length 2, throw out the cycle.

(b) If this cycle has length greater than 2, remove n from the cycle.

In case (a), suppose k is in the cycle with n. We obtain every derangement of $\underline{n} - \{k,n\}$ exactly once. Since there are $n-1$ possibilities for k, (a) contributes $(n-1)D_{n-2}$ to the count. This is a reduction point of view.

In case (b), we obtain derangements of $\underline{n-1}$. To find the contribution of (b), it may be easier to take a construction view: Given a derangement of $\underline{n-1}$ in cycle form, we choose which of the $n-1$ elements to insert n after. This gives a contribution of $(n-1)D_{n-1}$

The initial conditions and the range of n for which (7.9) is valid can be found by examining our argument. There are two approaches:

- We could take the view that derangements only make sense for $n \ge 1$ and so (7.9) is used when $n > 3$, with initial conditions $D_1 = 0$ and $D_2 = 1$.
- We could look at the argument used to derive the recursion and ask how we should define D_n for $n < 1$ so that the argument makes sense. Note that for $n = 1$, the values of D_0 and D_{-1} don't matter since the recursion gives

  $$D_1 \;=\; (1-1)(D_0 + D_{-1}) \;=\; 0(D_0 + D_{-1}) \;=\; 0,$$

  which is correct. What about $n = 2$? We look at (a) and (b) separately.

(a) We want to get the derangement $(1,2)$, so we need $D_0 = 1$.

(b) This should give zero since there is no derangement of $\underline{2}$ containing a cycle of length exceeding 2. Thus we need $D_1 = 0$, which we have.

To summarize, we can use (7.9) for $n \geq 1$ with the initial conditions $D_0 = 1$. $\square$

Example 7.8 Merge sorting Merge sorting can be described as follows.

1. The lists containing just one item are the simplest and they are already sorted.
2. Given a list of $n > 1$ items, choose k with $1 \leq k < n$, sort the first k items, sort the last $n-k$ items and merge the two sorted lists.

This algorithm builds up a way to sort an n-list out of procedures for sorting shorter lists. Note that we have not specified how the first k or last $n-k$ items are to be sorted, we simply assume that it has been done. Of course, an obvious way to do this is to simply apply our merge sorting algorithm to each of these sublists.

Let's implement the algorithm using people rather than a computer. Imagine training a large number of obedient people to carry out two tasks: splitting a list for other people to sort and merging two lists. We give one person the unsorted list and tell him to sort it using the algorithm and return the result to us.

What happens? Anyone who has a list with only one item returns it unchanged to the person he received it from. This is Case 1 in Definition 7.3 (p. 204) (and also in the algorithm). Anyone with a list having more than one item splits it and gives each piece to a person who has not received a list, telling each person to sort it and return the result. When the results have been returned, this person merges the two lists and returns the result to whoever gave him the list. If there are enough obedient people around, we'll eventually get our answer back.

Notice that no one needs to pay any attention to what anyone else is doing to a list. $\square$

We now look at one of the most important recursive definitions in computer science.

Example 7.9 Defining rooted plane trees recursively Rooted plane trees (RP-trees) were defined in Section 5.4 (p. 136). Here is a recursive constructive definition of RP-trees.

- A single vertex, which we call the root, is an RP-tree.
- If $T_1, \ldots, T_k$ is an ordered list of RP-trees with roots $r_1, \ldots, r_k$ and no vertices in common, then an RP-tree can be constructed by choosing an unused vertex r to be the root, letting its ith child be r_i and forgetting that $r_1, \ldots, r_k$ were called roots.

This is a more compact definition than the nonconstructive one given in Section 5.4. This approach to RP-trees is very important for computer science. We'll come back to it in the next section.

We should, and will, prove that this definition is equivalent to that in Section 5.4. In other words, our new "definition" should not be regarded as a definition but, rather, as a theorem—you can only define something once!

Define an *edge* to be any set of two vertices in which one vertex is the child of the other. Note that the recursive definition insures that the graph is connected and the use of distinct vertices eliminates the possibility of cycles. Thus, the "definition" given here leads to a rooted, connected, simple graph without loops. Furthermore, the edges leading to a vertex's sons are ordered. Thus we have an RP-tree. To actually prove this carefully, one must use induction on the number of vertices. This is left as an exercise.

It remains to show that every RP-tree, as defined in Section 5.4, can be built by the method described in the recursive "definition" given above. One can use induction on the number of vertices. It is obvious for one vertex. Remove the root vertex and note that each child of the root now becomes the root of an RP-tree. By the induction hypothesis, each of these can be built by our recursive process. The recursive process allows us to add a new root whose children are the roots of these trees, and this reconstructs the original RP-tree.

Here is another definition of an RP-tree.

- A single vertex, which we call the root, is an RP-tree.
- If T_1 and T_2 are RP-trees with roots r_1 and r_2 and no vertices in common, then an RP-tree can be constructed by connecting r_1 to r_2 with an edge, making r_2 the root of the new tree and making r_1 the leftmost child of r_2.

We leave it to you to prove that this is equivalent to the previous definition $\square$

Example 7.10 Recursions for rooted plane trees Rooted trees in which each non-leaf vertex has exactly two edges leading away from the root is called a full binary tree. By replacing k in the previous example with 2, we have a recursive definition of them: A full binary RP-tree is either a single vertex or a new root vertex joined to two full binary RP-trees.

As noted in Section 1.4 (p. 32), recursive constructions lead to recursions. Let's use the previous recursive definition to get a recursion for full binary trees. Suppose we require that any node that is not a leaf have exactly two children. Let b_n be the number of such trees that have n leaves. From the recursive definition, we have $b_1 = 1$ for the single vertex tree. Since the recursive construction gives us each tree *exactly once*, we have

$$b_n \;=\; b_1 b_{n-1} + b_2 b_{n-2} + \cdots + b_{n-1} b_1 \;=\; \sum_{j=1}^{n-1} b_j b_{n-j} \qquad \text{for } n > 1.$$

To see why this is so, apply the Rules of Sum and Product: First, partition the problem according to the number of leaves in T_1, which is the j in our formula. Second, for each case choose T_1 and then choose T_2, which gives us the term $b_j b_{n-j}$.

If we try the same approach for general rooted plane trees, we have two problems. First, we had better not count by leaves since there are an infinite number of trees with just one leaf, namely trees with of the form $\bullet\!-\!\bullet\!-\!\bullet\cdots\!-\!\bullet$. Second, the fact the the definition involves $T_1, \ldots, T_k$ where k can be any positive integer makes the recursion messy: we'd have to sum over all such k and for each k we'd have a product $t_{j_1} \cdots t_{j_k}$ to sum over all j's such that $j_1 + \cdots + j_k$ has the appropriate value.

The first problem is easy to fix: let t_n be the number of rooted plane trees with n vertices. The second problem requires a new recursive construction, which means we have to be clever. We use the construction in the last paragraph of the previous example. We then have $t_1 = 1$ and, for $n > 1$, $t_n = \sum_{j=1}^{n-1} t_j t_{n-j}$, because if the constructed tree has n vertices and T_1 has j vertices, then T_2 has $n - j$ vertices. Notice that t_n and b_n satisfy the same recursion with the same initial conditions. Since the recursion lets us compute all values recursively, it follows that $t_n = b_n$. (Alternatively, you could prove $b_n = t_n$ using the recursion and induction on n.) Actually, there is a slight gap here: we didn't prove that the new recursive definition gives all rooted plane trees and gives them exactly once. We leave it to you to convince yourself of this. $\square$

A *recursive algorithm* is an algorithm that refers to itself when it is executing. As with any recursive situation, when an algorithm refers to itself, it must be with "simpler" parameters so that it eventually reaches one of the "simplest" cases, which is then done without recursion. Our recursion for $C(n, k)$ can be viewed as a recursive algorithm. Our description of merge sorting in Examples 7.8 gives a recursive algorithm if the sorting required in Step 2 is done by using the algorithm itself. Let's look at one more example illustrating the recursive algorithm idea.

Example 7.11 A recursive algorithm Suppose you are interested in listing all sequences of length eight, consisting of four zeroes and four ones. Suppose that you have a friend who does this sort of thing, but will only make such lists if the length of the sequence is seven or less. "Nope," he says, "I can't do it—the sequence is too long." There is a way to trick your friend into doing it. First give him the problem of listing all sequences of length seven with three ones. He doesn't mind, and gives you the list 1110000, 1011000, 0101100, etc. that he has made. You thank him politely, sneak off, and put a "1" in front of every sequence in the list he has given you to obtain 11110000, 11011000, 10101100, etc. Now, you return to him with the problem of listing all strings of length seven with four ones. He returns with the list 1111000, 0110110, 0011101, etc. Now you thank him and sneak off and put a "0" in front of every sequence in the list he has given you to obtain 01111000, 00110110, 00011101, etc. Putting these two lists together, you have obtained the list you originally wanted.

How did your friend produce these lists that he gave you? Perhaps he had a friend that would only do lists of length 6 or less, and he tricked this friend in the same way you tricked him! Perhaps the "6 or less" friend had a "5 or less friend" that he tricked, etc. If you are sure that your friend gave you a correct list, it doesn't really matter how he got it. □

These examples are rather easy to follow, but what happens if we look into them more deeply? We might ask just how $C(15,7)$ is calculated in terms of the simplest values $C(0,k)$ without specifying any of the intermediate values. We might ask just what all of our trained sorters are doing. We might ask how your friend got his list of sequences.

This kind of analysis is often tempting to do when we are debugging recursive algorithms. It is almost always the wrong thing to do. Asking about such details usually leads to confusion and gets one so off the track that it is even harder to convince oneself that the algorithm is correct.

Why is it unnecessary to "unwind" the recursion in this fashion? If Case 2 of our recursive solution as given by Definition 7.3 (p. 204) correctly describes what to do, *assuming that the simpler problems have been done correctly*, then our recursive solution works! This can be demonstrated the way induction was proved: If the solution fails, there must be a problem for which it fails such that it succeeds for all simpler problems. If this problem is simplest, it contradicts Case 1 in Definition 7.3. If this problem is not simplest, it contradicts Case 2 in Definition 7.3 since all simpler problems have been dealt with. Thus our assumption that the solution fails has led to a contradiction. It is important to understand this proof since it is the theoretical basis for recursive methods. To summarize:

Principle Thinking recursively *Carefully verify the two parts of Definition 7.2 or of Definition 7.3. Avoid studying the results of iterating the recursive solution.*

If you are just learning about recursion, you may find it difficult to believe that this general strategy will work without seeing particular solutions where the reduction to the simplest cases is laid out in full detail. In even a simple recursive solution, it is likely that you'll become confused by the details, even if you're accustomed to thinking recursively. If you agree that the proof in the previous paragraph is correct, then such detail is not needed to see that the algorithm is correct. It is *very important* to realize this and to avoid working through the stages of the recursive solution back to the simplest things.

If for some reason you must work backwards through the recursive stages, do it gingerly and carefully. When must you work backwards like this?

- For some reason you may be skipping over an error in your algorithm and so are unable to correct it. The unwinding process can help, probably not because it will help you find the error directly but because it will force you to examine the algorithm more closely.
- You may wish to replace the recursive algorithm with a nonrecursive one. That may require a much deeper understanding of what happens as the recursion is iterated.

The approach of focusing on the two steps of an inductive solution is usually difficult for beginners to maintain. Resist the temptation to abandon it! This does not mean that you should avoid details, but the details you should concern yourself with are different:

- "Is every solution built from simplest solutions, and have I handled the simplest solutions properly?" If not, then the foundation of your recursively built edifice is rotten and the entire structure will collapse.
- "Is my description of how to use simpler solutions to build up more complicated ones correct?"
- "If this is an algorithm, have I specified all the recursive parameters?"

This last point will be dealt with in the next section where we'll discuss implementation.

This does not mean one should never look at the details of a recursion. There are at least two situations in which one does so. First, one may wish to develop a nonrecursive algorithm. Understanding the details of how the recursive algorithm works may be useful. Second, one may need to reduce the amount of storage a recursive algorithm requires.

Exercises

7.2.1. We will prove that all positive integers are equal. Let $\mathcal{A}(n)$ be the statement "All positive integers that do not exceed n are equal." In other words, "If p and q are integers between 1 and n inclusive, then $p = q$." Since 1 is the only positive integer not exceeding 1, $\mathcal{A}(1)$ is true. For $n > 1$, we now assume $\mathcal{A}(n-1)$ and prove $\mathcal{A}(n)$. If p and q are positive integers not exceeding n, let $p' = p - 1$ and $q' = q - 1$. Since p' and q' do not exceed $n - 1$, we have $p' = q'$ by $\mathcal{A}(n-1)$. Thus $p = q$. This proves $\mathcal{A}(n)$. Where is the error?

7.2.2. What is wrong with the following proof that every graph can be drawn in the plane in such a way that no edges intersect? Let $\mathcal{A}(n)$ be the statement for all graphs with n vertices. Clearly $\mathcal{A}(1)$ is true. Let G be a graph with vertices $v_1, \ldots, v_n$. Let G_1 be the subgraph induced by $v_2, \ldots, v_n$ and let G_n be the subgraph induced by $v_1, \ldots, v_{n-1}$. By the induction assumption, we can draw both G_1 and G_n in the plane. After drawing G_n, add the vertex v_n near v_1 and use the drawing of G_1 to see how to connect v_n to the other vertices.

7.2.3. What is wrong with the following proof that all positive integers are interesting? Suppose the claim is false and let n be the smallest positive integer which is not interesting. That is an interesting fact about n, so n is interesting!

7.2.4. What is wrong with the following method for doing this exercise? Ask someone else in the class who will tell you the answer if he/she knows it. If that person knows it, you are done; otherwise that person can use this method to find the answer and so you are done anyway.
Remark: Of course it could be wrong morally because it may be cheating. For this exercise, you should find another reason.

7.2.5. This relates to Example 7.9. Fill in the details of the proof of the equivalence of the two definitions of RP-trees.

7.3 Recursive Algorithms

We'll begin this section by using merge sort to illustrate how to obtain information about a recursive algorithm. In this case we'll look at proof of correctness and a recursion for running time. Next we'll turn our attention to the local description of a recursive procedure. What are the advantages of thinking locally?

- Simplicity: By thinking locally, we can avoid the quagmire that often arises in attempting to unravel the details of the recursion. To avoid the quagmire: *Think locally*, but remember to deal with initial conditions.
- Implementation: A local description lays out in graphical form a plan for coding up a recursive algorithm.
- Counting: One can easily develop a recursion for counting structures, operations, etc.
- Proofs: A local description lays out the plan for an inductive proof.

Finally, we'll turn our attention to the problem of how recursive algorithms are actually implemented on a computer. If you are not programming recursive algorithms at present, you may think of the implementation discussion as an extended programming note and file it away for future reference after skimming it.

Obtaining Information: Merge Sorting

Here's an algorithm for "merge sorting" the sequence s and storing the answer in the sequence t.

```
Procedure SORT($s_1,\ldots,s_n$ into $t_1,\ldots,t_n$)
    If ($n=1$)
        $t_1=s_1$
        Return
    End if
    Let $m$ be $n/2$ with remainder discarded
    SORT($s_1,\ldots,s_m$ into $u_1,\ldots,u_m$)
    SORT($s_{m+1},\ldots,s_n$ into $v_1,\ldots,v_{n-m}$)
    MERGE(sequences $u$ and $v$ into $t$)
    Return
End
```

How do we know it doesn't run forever—an "infinite loop"? How do we know it's correct? How long does it take to run? As we'll see, we can answer such questions by making modifications to the algorithm.

The infinite loop question can be dealt with by verifying the conditions of Definition 7.2 (p. 204). For the present algorithm, the complexity of the problem is the length of the sequence and the simplest case is a 1-long sequence. The algorithm deals directly with the simplest case. Other cases are reduced to simpler ones because a list is divided into shorter lists.

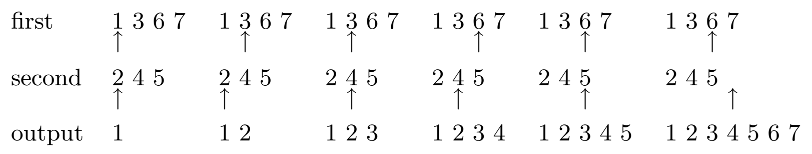


Figure 7.1 Merging two sorted lists. The first and second lists are shown with their pointers at each step.

Example 7.12 The merge sort algorithm is correct One way to prove program correctness is to insert claims into the code and then prove that the claims are correct. For recursive algorithms, this requires induction. We'll assume that the MERGE algorithm is known to be correct. (Proving that is an another problem.) Here's our code with comments added for proving correctness.

```
Procedure SORT($s_1,\ldots,s_n$ into $t_1,\ldots,t_n$)
    If ($n = 1$)
        $t_1 = s_1$
        Return                                        /* $t$ is sorted */
    End if
    Let $m$ be $n/2$ with remainder discarded
    SORT($s_1,\ldots,s_m$ into $u_1,\ldots,u_m$)          /* $u$ is sorted */
    SORT($s_{m+1},\ldots,s_n$ into $v_1,\ldots,v_{n-m}$)   /* $v$ is sorted */
    MERGE(sequences $u$ and $v$ into $t$
    Return                                        /* $t$ is sorted */
End
```

We now use induction on n to prove

$$\mathcal{A}(n) \quad = \quad \text{“When a comment is reached in sorting an } n\text{-list, it is true.”}$$

For $n = 1$, only the first comment is reached and it is clearly true since there is only one item in the list. For $n > 1$, the claims about u and v are true by $\mathcal{A}(m)$ and $\mathcal{A}(n-m)$. Also, the claim about t is true by the assumption that MERGE runs correctly. □

Example 7.13 The running time for a merge sort How long does the merge sort algorithm take to run? Let's ignore the overhead in computing m, subroutine calling and so forth and focus on the part that takes the most time: merging.

Suppose that $u_1 \le \ldots \le u_i$ and $v_1 \le \ldots \le v_j$ are two sorted lists. We can merge these two ordered lists into one ordered list very simply by moving pointers along the two lists and comparing the elements being pointed to. The smaller of the two elements is output and the pointer in its list is advanced. (The decision as to which to output at a tie is arbitrary.) When one list is used up, simply output the remainder of the other list. The sequence of operations for the lists 1,3,6 and 2,4,5 is shown in Figure 7.1. Since each comparison results in at least one output and the last output is free, we require at most $i + j - 1$ comparisons to merge the lists $u_1 \le \ldots \le u_i$ and $v_1 \le \ldots \le v_j$. On the other hand, if one list is output before any of the other list, we might use only $\min(i,j)$ comparisons.

Let $\mathtt{C}(n)$ be an upper bound on the number of comparisons needed to merge sort a list of n things. Clearly $\mathtt{C}(1) = 0$. The number of comparisons needed to merge two lists with a total of n items is at most $n - 1$. We can convert our sorting procedure into one for computing $\mathtt{C}(n)$. All we need to do is replace SORT with C and add up the various counts. Here's the result.

```
Procedure C($n$)
    $C = 0$
    If ($n = 1$), then Return $C$
    Let $m$ be $n/2$ with remainder discarded
    $C = C + $C($m$)
    $C = C + $C($n - m$)
    $C = C + (n - 1)$
    Return $C$
End
```

To make things easier for ourselves, let's just look at lengths which are powers of two so that the division comes out even. Then $\mathtt{C}(1) = 0$ and $\mathtt{C}(2^k) = 2\mathtt{C}(2^{k-1}) + 2^k - 1$. By applying the recursion we get

$$\begin{array}{rclclcrclcl} \mathtt{C}(2) &=& 2\cdot 0 + 1 &=& 1, & \quad & \mathtt{C}(4) &=& 2\cdot 1 + 3 &=& 5,\\ \mathtt{C}(8) &=& 2\cdot 5 + 7 &=& 17, & & \mathtt{C}(16) &=& 2\cdot 17 + 15 &=& 49. \end{array}$$

What's the pattern?

* * * Stop and think about this! * * *

It appears that $\mathtt{C}(2^k) = (k-1)2^k + 1$. We leave it to you to prove this by induction. This suggests that the number of comparisons needed to merge sort an n long list is bounded by about $n \log_2(n)$. We've only proved this for n a power of 2 and will not give a general proof.

There are a couple of points to notice here. First, we haven't concerned ourselves with how the algorithm will actually be implemented. In particular, we've paid no attention to how storage will be managed. Such a cavalier attitude won't work with a computer so we'll discuss implementation problems in the next section. Second, the recursive algorithm led naturally to a recursive estimate for the speed of the algorithm. This is often true. □

Local Descriptions

We begin with the local description for two ideas we've seen before when discussing decision trees. Then we look at the "Tower of Hanoi" puzzle, using the local description to illustrate the claims for thinking locally made at the beginning of this section.

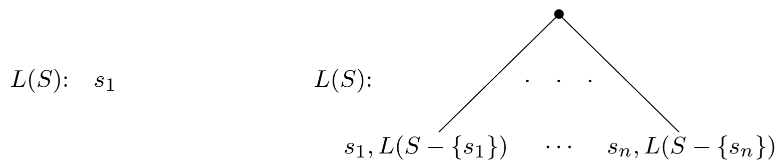


Figure 7.2 The two cases for the local description of $L(S)$, the lex order permutation tree for $S = \{s_1, \ldots, s_n\}$. Left: the initial case $n = 1$. Right: the recursive case $n > 1$.

Example 7.14 The local description of lex order permutations

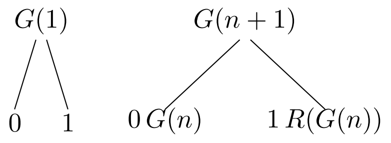
Suppose that S is an n element set with elements $s_1 < \ldots < s_n$. In Section 3.1 we discussed how to create the decision tree for generating the permutations of S in lex order. (See page 70.) Now we'll give a recursive description that follows the pattern in Definition 7.3 (p. 204).

Let $L(S)$ stand for the decision tree whose leaves are labeled with the permutations of S in lex order and whose root is labeled $L(S)$. If x is some string of symbols, let $x, L(S)$ stand for the $L(S)$ with the string of symbols "x," appended to the front of each label of $L(s)$. For Case 1 in Definition 7.3, $n = 1$. Then $L(S)$ is simply one leaf labeled s_1. See Figure 7.2 for $n > 1$,. What we have just given is called the *local description* of the lex order permutation tree because it looks only at what happens from one step of the inductive definition to the next. In other words, a local description is nothing more that the statement of Definition 7.3 for a specific problem.

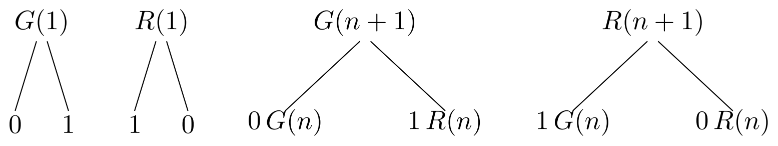
We'll use induction to prove that this is the correct tree. When $n = 1$, it is clear. Suppose it is true for all S with cardinality less than n. The permutations of S in lex order are those beginning with s_1 followed by those beginning with s_2 and so on. If s_k is removed from those permutations of S beginning with s_k, what remains is the permutations of $S - \{s_k\}$ in lex order. By the induction hypothesis, these are given by $L(S - \{s_k\})$. Note that the validity of our proof does not depend on how they are given by $L(S - \{s_k\})$. $\square$

Example 7.15 Local description of Gray code for subsets

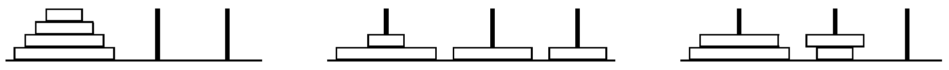
We studied Gray codes for subsets in Examples 3.12 (p. 82) and 3.13 (p. 86). We can give a local description of the algorithm as



where $n > 0$, $R(T)$ is T with the order of the leaves reversed, and $1T$ is T with 1 prepended to each leaf. Alternatively, we could describe two interrelated trees, where now $R(n)$ is the tree for the Gray code listed in reverse order:



The last tree may appear incorrect, but it is not. When we reverse $G(n + 1)$, we must move $1\,R(n)$ to the left child and reverse it. Since the reversal of $R(n)$ is $G(n)$, this gives us the left child of $G(n + 1)$. The right child is explained similarly. $\square$



(a) Starting (b) Intermediate (c) Illegal!

Figure 7.3 Three positions in the Tower of Hanoi puzzle for $n = 4$.

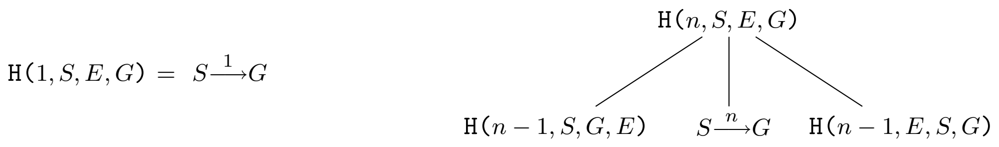

$\mathtt{H}(1, S, E, G) = S \xrightarrow{1} G$
$\mathtt{H}(n, S, E, G)$
$\mathtt{H}(n-1, S, G, E)$
$S \xrightarrow{n} G$
$\mathtt{H}(n-1, E, S, G)$

Figure 7.4 The local description of the solution to the Tower of Hanoi puzzle. The left hand figure describes the initial case $n = 1$ and the right hand describes the recursive case $n > 1$. Instead of labeling the tree, we've identified the root vertex with the label. This is convenient if we expand the tree as in the next figure.

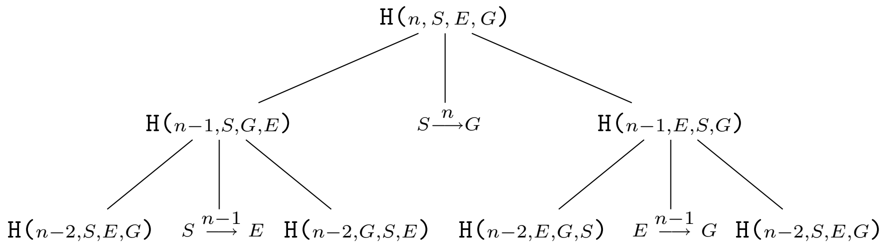


Figure 7.5 The first expansion of the Tower of Hanoi tree for $n > 2$. This was obtained by applying Figure 7.4 to itself.

Example 7.16 The Tower of Hanoi puzzle

The *Tower of Hanoi* puzzle consists of n different sized washers (i.e., discs with holes in their centers) and three poles. Initially the washers are stacked on one pole as shown in Figure 7.3(a). The object is to switch all of the washers from the left hand pole to the right hand pole. The center pole is extra, to assist with the transfers. A legal move consists of taking the top washer from a pole and placing on top of the pile on another pole, provided it is not placed on a smaller washer.

How can we solve the puzzle?

To move the largest washer, we must move the other $n-1$ to the spare peg. After moving the largest, we can then move the other $n-1$ on top of it. Let the washers be numbered 1 to n from smallest to largest. When we are moving any of the washers 1 through k, we can ignore the presence of all larger washers beneath them. Thus, moving washers 1 through $n-1$ from one peg to another when washer n is present uses the same moves as moving them when washer n is not present. Since the problem of moving washers 1 through $n-1$ is simpler, we practically have a recursive description of a solution. All that's missing is the observation that the simplest case, $n = 1$, is trivial. The local description of the algorithm is shown in Figure 7.4 where $X \xrightarrow{k} Y$ indicates that washer k is to be moved from peg X to peg Y.

If we want to think globally, we need to expand this tree until all the $H(\cdots)$ are replaced by moves. In other words, we continue until reaching $\mathtt{H}(1, X, Y, Z)$, which is simply $X \xrightarrow{1} Z$. How much expansion is required to reach this state depends on n. The first step in expanding this tree is shown in Figure 7.5.

To get the sequence of moves, expand the tree as far as possible, extend the edges leading to leaves so that they are all on the same level, and then list the leaves as they are encountered reading from left to right. When $n = 3$, we can use Figure 7.5 and the left side of Figure 7.4. The resulting sequence of moves is

$$S \xrightarrow{1} G \quad S \xrightarrow{2} E \quad G \xrightarrow{1} E \quad S \xrightarrow{3} G \quad E \xrightarrow{1} S \quad E \xrightarrow{2} G \quad S \xrightarrow{1} G.$$

As you can see, this is getting rather complex.

Here's some pseudocode that implements the local description. It's constructed directly from Figure 7.4. We execute it by running `H(`n`,start,extra,goal)`.

```
Procedure H($n, S, E, G$)
    If ($n = 1$)
        Print: Move washer 1 from $S$ to $G$
        Return
    End if
    H($n-1, S, G, E$)
    Print: Move washer $n$ from $S$ to $G$
    H($n-1, E, S, G$)
End
```

How many moves are required to solve the puzzle? Let the number be h_n. From the local description we have $h_1 = 1$ and $h_n = h_{n-1} + 1 + h_{n-1} = 2h_{n-1} + 1$ for $n > 1$. Using this one can prove that $h_n = 2^n - 1$.

We can prove by induction that the algorithm works. It clearly works for $n = 1$. Suppose $n = 1$. By induction, the left child in the local description (Figure 7.4) moves the $n - 1$ smallest washers from S to E. Thus the move in the middle child is valid. Finally, the right child moves the $n - 1$ smallest washers from E to G (again, by induction).

We can prove other things as well. For example, washer 1 moves on the kth move if and only if k is odd. Again, this is done by induction. It is true for $n = 1$. For $n > 1$, it true for the left child of local description by induction. Similarly, it is true for the right child because the left child and the middle child involve a total of $h_{n-1} + 1 = 2^{n-1}$ moves, which is an even number. If you enjoy challenges, you may wish to pursue this further and try to determine for all k when washer k moves as well as what each move is. Determining "when" is not too difficult. Determining "what" is tricky. $\square$

*Computer Implementation

Computer implementation of recursive procedures involves the use of stacks to store information for different levels of the recursion. A *stack* is a list of data in which data is added to the end of the list and is also removed from the same end. The end of the list is called the top of the stack, adding something is called *pushing* and removing something is called *popping*.

Example 7.17 Implementing the Tower of Hanoi solution Let's return to the Tower of Hanoi procedure H(n, S, E, G), which is described in Figure 7.4. To begin, we push n, S, E and G on the stack and call the program H. The stack entries, from the top, may be referred to as the first, second and so on items. If $n = 1$, H simply carries out the action in the left side of Figure 7.4. If $n > 1$, it carries out actions corresponding to each of the three sons on the right side of Figure 7.4 in turn: The left son causes it

- to push $n-1$ and the second, fourth and third items on the stack, in that order,
- to call the program H

and, when H finishes,

- to pop four items off the stack.

The middle son is similar to $n = 1$ and the right son is similar to the left son.

You may find it helpful to see what this process leads to when $n = 3$. How does it relate to Figure 7.5? □

Example 7.18 Computing a bound on comparisons in merge sorting Let's look at the pseudocode for computing C(n), the upper bound on the number of comparisons in our merge sort (Example 7.13 (p. 211)). Here it is

```
Procedure C($n$)
    $C = 0$
    If ($n = 1$),
        Return $C$
    End if
    Let $m$ be $n/2$ with remainder discarded
    $C = C + $C($m$)
    $C = C + $C($n - m$)
    $C = C + (n - 1)$
    Return $C$
End
```

This has a new feature not present in H: The procedure contains the variable C which we must save if we call C recursively. This can be done as follows. When a procedure is called, space is allocated on (pushed onto) the stack to store all of its "local variables" (that is, variables that exist only in the procedure). When the procedure is done the space is deallocated (popped off the stack). Thus, in a programming language that permits recursion, each call of a procedure Proc uses space for

- the address in the calling procedure to return to when Proc is done,
- the values of the variables passed to Proc and
- the values of the variables that are local to Proc

until the procedure Proc is done. Since a recursive procedure may call itself, which calls itself, which calls itself, . . . ; it may use a considerable amount of storage. □

Example 7.19 Implementing merge sorting Look at example Example 7.13 (p. 211). It requires a tremendous amount of extra storage since we need space for the s, t, u and v arrays every time the procedure calls itself. If we want to implement this algorithm on a real computer, it will have to be rewritten to avoid creating arrays recursively. This can be done by placing the sorted array in the original array. Here's the new version

```
Procedure SORT($a[lo]$ through $a[hi]$)
    If ($lo = hi$), then Return
    Let $m$ be $(lo + hi)/2$ with remainder discarded
    SORT($a[lo]$ through $a[m]$)
    SORT($a[m+1]$ through $a[hi]$)
    MERGE($a[lo]$ through $a[m]$ with $a[m+1]$ through $a[hi]$)
End
```

This requires much less storage. A simple implementation of `MERGE` requires a temporary array, but multiple copies of that array will not be created through recursive calls because `MERGE` is not recursive. The additional array problem can be alleviated. We won't discuss that. □

Exercises

7.3.1. In Example 7.13 we computed an upper bound $C(n)$ on the number of comparisons required in a merge sort. The purpose of this exercise is to compute a lower bound. Call this bound $c(n)$.

(a) Explain why merging two sorted lists of lengths k_1 and k_2 requires at least $\min(k_1, k_2)$ comparisons, where "min" denotes minimum. Give an example of when this is achieved for all values of k_1 and k_2.

(b) Write code like Procedure C(n) in Example 7.13 to compute $c(n)$.

(c) State and prove a formula for $c(n)$ when $n = 2^k$, a power of 2. Compare $c(n)$ with $C(n)$ when n is a large power of 2.

7.3.2. Give a local description of listing the strictly decreasing functions from $\underline{k}$ to $\underline{n}$ in lex order. (These are the k-subsets of $\underline{n}$.) Call the list $D(n, k)$ and use the notation $i, D(j, k)$ to mean the list obtained by prepending i to each of the functions in $D(j, k)$ written in one-line form. For example

$$D(3,2) = (2,1;\ \ 3,1;\ \ 3,2) \qquad \text{and} \qquad 5, D(3,2) = (5,2,1;\ \ 5,3,1;\ \ 5,3,2).$$

7.3.3. Merging two lists in a single list stored elsewhere requires that each item be moved once. Dividing a list approximately in two requires no moves. State and prove a formula for the number of moves required by a merge sort of n items when $n = 2^k$, a power of 2.

7.3.4. We have a pile of n coins, all of which are identical except for a single counterfeit coin which is lighter than the other coins. We have a "beam balance," a device which compares two piles of coins and tells which pile is heavier. Here is a recursive algorithm for finding the counterfeit coin in a set of $n \geq 2$ coins.

```
Procedure Find($n$,Coins)
    If ($n = 2$) Put one coin in each pile
    and report the result.
    Else
        Select a coin C in Coins.
        Find($n-1$,Coins−C)
        If a counterfeit is reported, report it.
        Else report C.
        Endif
    Endif
End
```

Since `Find` only uses the beam balance if $n = 2$, this recursive algorithm finds the counterfeit coin by using the beam balance only once regardless of the value of $n \geq 2$. What is wrong? How can it be corrected?

7.3.5. Suppose we have a way to print out the characters 0–9 and −, but do not have a way to print out integers such as −360. We want a procedure OUT(m) to print out integers m, both positive, negative, and zero, as strings of digits. If $n \geq 0$ is a positive integer, let q and r be the quotient and remainder when n is divided by 10.

(a) Using the fact the digits of n are the digits of q followed by the digit r to write a *recursive* procedure OUT(m) that prints out m for any integer (positive, negative, or zero).

(b) What are the simplest objects in your recursive solution?

(c) Explain why your procedure never runs forever.

7.3.6. Let $n \geq 0$ be an integer and let q and r be the quotient and remainder when n is divided by 10. We want a procedure DSUM(n) to sum the digits of n.

(a) Using the fact that the sum of the digits of n equals r plus the sum of the digits of q, write a recursive procedure DSUM(n).

(b) What are the simplest objects in your recursive solution?

(c) Explain why your procedure never runs forever.

7.3.7. What is the local description for the tree that generates the decreasing functions in $\underline{n}^{\underline{k}}$? Decreasing functions were discussed in Example 3.8.

7.3.8. Expand the local description of the Tower of Hanoi to the full tree for $n = 2$ and for $n = 4$. Using the expanded trees, write down the sequence of moves for $n = 2$ and for $n = 4$.

7.3.9. Let $S(n)$ be the number of moves required to solve the Tower of Hanoi puzzle.

(a) Prove by induction that `Procedure H` takes the least number of moves.

(b) Convert `Procedure H` into a procedure that computes $S(n)$ recursively as was done for sorting in Example 7.13. Translate the code you have just written into a recursion for $S(n)$.

(c) Construct a table of $S(n)$ for $1 \leq n \leq 7$.

(d) Find a simple formula (*not* a recursion) for $S(n)$ and prove that it is correct by using the result in (b) and induction.

(e) Assuming the starting move is called move one, what washer is moved on move k?
Hint. There is a simple description in terms of the binary representation of k.

*(f) What are the source and destination poles on move k?

7.3.10. We have discovered a simpler procedure for the Tower of Hanoi: it only involves one recursive call. To move washers k to n we use $\mathtt{H}(k, n, S, E, G)$. Here's the procedure.

```
Procedure H($k, n, S, E, G$)
    If ($k = n$)
        Move washer $n$ from $S$ to $G$
        Return
    End if
    Move washer $k$ from $S$ to $E$
    H($k + 1, n, S, E, G$)
    Move washer $k$ from $E$ to $G$
    Return
End
```

To get the solution, run $\mathtt{H}(1, n, S, E, G)$. This is an incorrect solution to the Tower of Hanoi problem. Which of the two conditions for a recursive solution fails and how does it fail? Why does the algorithm in the text not fail in the same way?

7.3.11. We consider a modification of the Tower of Hanoi. All the old rules apply, but the moves are more limited: You can think of the poles as being in a row with the extra pole in the middle. The new rule then says that a washer can only move to an adjacent pole. In other words, a washer can never be moved directly from the original starting pole to the original destination pole. Thus, when $n = 1$ we require two moves: $S \xrightarrow{1} E$ and $E \xrightarrow{1} G$.

Let $H^*(n, P_1, P_2, P_3)$ be the tree that moves n washers from P_1 to P_3 while using P_2 as the extra pole. The middle pole is P_2.

(a) At the start of the problem, we described the moves for $n = 1$. For $n > 1$, washer n must first move to the extra post and then to the goal. The other $n - 1$ washers must first be stacked on the goal and then on the start to allow these moves. Draw the local description of H^* for $n > 1$.

(b) Let h_n^* be the number of washers moved by $H^*(n, S, E, G)$. Write down a recursion for h_n^*, including initial conditions.

(c) Compute the first few values of h_n^*, guess the general solution, and prove it.

7.3.12. The number of partitions of the set $\underline{n}$ into k blocks was defined in Example 1.27 to be $S(n, k)$, the Stirling numbers of the second kind. We developed the recursion $S(n, k) = S(n-1, k-1) + kS(n-1, k)$ by considering whether n was in a block by itself or in one of the k blocks of $S(n - 1, k)$. By using the actual partitions instead of just counting them, we can interpret the recursion as a means of producing all partitions of $\underline{n}$ with k blocks.

(a) Write pseudocode to do this.

(b) Draw the local description for the algorithm.

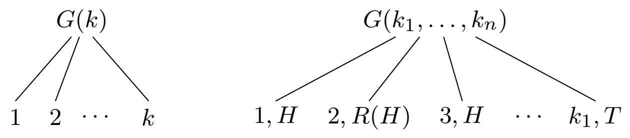
7.3.13. We want to produce all sequences $\alpha = a_1, \ldots, a_n$ where $1 \le a_i \le k_i$. This is to be done so that if β is produced immediately after α, then all but one of the entries in β is the same as in α and that entry differs from the α entry by one. Such a list of sequences is called a *Gray code.* If T is a tree whose leaves are labeled by sequences, let a, T be the same tree with each leaf label α replaced by a, α. Let $R(T)$ be the tree obtained by taking the mirror image of T. (The sequences labeling the leaves are moved but are not reversed.) For example, if the leaves of T are labeled, from left to right,

$$1,2 \qquad 1,3 \qquad 2,4 \qquad 1,4 \qquad 2,3,$$

then the leaves of $R(T)$ are labeled, from left to right,

$$2,3 \qquad 1,4 \qquad 2,4 \qquad 1,3 \qquad 1,2.$$

Let $G(k_1, \ldots, k_n)$ be the decision tree with the local description shown below. Here $n > 1$, $H = G(k_2, \ldots, k_n)$ and T is either H or $R(H)$ according as k_1 is odd or even.



(a) Draw the full tree for $G(3, 2, 3)$ and the full tree for $G(2, 3, 3)$.

(b) Prove that $G(k_1, \ldots, k_n)$ contains all sequences $a_1, \ldots, a_n$ where $1 \le a_i \le k_i$.

(c) Prove that adjacent leaves of $G(k_1, \ldots, k_n)$ differ in exactly one entry and that entry changes by one from one leaf to the next.

*(d) Suppose that $k_1 = \cdots = k_n = 2$. Describe RANK(α).

*(e) Suppose that $k_1 = \cdots = k_n = 2$. Tell how to find the sequence that follows α without using RANK and UNRANK.

7.3.14. For each of the previous exercises that requested pseudocode, tell what is placed on the stack as a result of the recursive call.

7.4 Divide and Conquer

In its narrowest sense, divide and conquer refers to the division of a problem into a few smaller problems that are of the same kind as the original problem and so can be handled by a recursive method. We've seen binary insertion, Quicksort and merge sorting as examples of this. In a broader sense, divide and conquer refers to any method that divides a problem into a few simpler problems. Heapsort illustrates this broader definition.

The broad divide and conquer technique is important for dealing with most complex situations. Delegation of responsibility is an application in everyday life. Scientific investigation often employs divide and conquer. In computer science it appears in both the design and implementation of algorithms, where it is referred to by such terms as "top-down programming," "structured programming," "object oriented programming" and "modularity." Properly used, these are techniques for efficiently creating and implementing understandable, correct and flexible programs.

What tools are available for applying divide and conquer to smaller problems? For example, how might one discover the algorithms we've discussed in this text? An algorithm that has a nonrecursive nature doesn't seem to fit any general rules; for example, all we can recommend for discovering something like Heapsort is inspiration. You can cultivate inspiration by being familiar with a variety of ideas and by trying to look at problems in novel ways.

We can say a bit more about trying to discover recursive algorithms; that is, algorithms that we use divide and conquer in its narrowest sense. Suppose the data is such that it is possible to split it into a few large blocks. You can ask yourself if anything is accomplished by solving the original problem on the separate blocks and then exploiting it. We'll see how this works by first reviewing earlier material and then move on to some new problems.

Example 7.20 Locating items in lists If we are trying to find an item in an ordered list, we can divide the list in half. What does this accomplish?

Suppose the item actually lies in the first half. When we look at the start of the second half list, we can immediately solve the problem for that sublist: the item we're looking for is not in that sublist because it precedes the first item. Thus we've divided the original problem into two problems, one of which is trivial. Binary insertion exploits this observation. Analysis of the resulting algorithm shows that it is considerably faster than simply looking through the list item by item. $\square$

Example 7.21 Sorting If we are trying to sort a list, we could divide it into two parts and sort each part separately. What does this accomplish? That depends on how we divided the list.

Suppose that we divided it in half arbitrarily. If each half is sorted, then we must merge two sorted lists. Some thought reveals that this is a fairly easy process. Exploiting this idea leads to merge sorting. Analysis of the algorithm shows that it is fast.

Suppose that we can arrange the division so that all the items in the first part should precede all the items in the other part. When the two parts are sorted, the list will be sorted. How can we divide the list this way? A bit of thought and a *clever idea* may lead to the method used by Quicksort. Analysis of the algorithm shows that it is usually fast. $\square$

Example 7.22 Calculating powers Suppose that we want to calculate x^n when n is a large positive integer. A simple way to do this is to multiply x by itself the appropriate number of times. This requires $n-1$ multiplications.

We can do better with divide and conquer. Suppose that $n = mk$. We can compute $y = x^m$ and then compute x^n by noting that it equals y^k. Using the method in the previous paragraph to compute y and then to compute y^k means that we require only $(m-1)+(k-1) = m+k-2$ multiplications. This is much less than $n-1 = mk-1$. We'll call this the "factoring method."

As is usual with divide and conquer, recursive application of the idea is even better.

In other words, we regard the computation of x^m and y^k as new problems and solve them by first factoring m and k and so forth. For example, computing x^{2^t} requires only t multiplications.

There is a serious drawback with the factoring method: n may not have many factors; in fact, it might even be a prime. What can we do about this?

If $n > 3$ is a prime, then $n-1$ is not a prime since it is even. Thus we can use the factoring method to compute x^{n-1} and then multiply it by x. We still have to deal with the factorization problem. This is getting complicated, so perhaps we should look for a simpler method. Try to think of something.

* * * Stop and think about this! * * *

The request that you try to think of something was quite vague, so it is quite likely that different people would have come up with different ideas. Here's a fairly simple method that is based on the observations $x^{2m} = (x^m)^2$ and $x^{2m+1} = (x^m)^2x$ applied recursively. Let the binary representation of n be $b_kb_{k-1}\ldots b_0$; that is

$$n \;=\; \sum_{i=0}^{k} b_i 2^i \;=\; \Big(\cdots\big((b_k)2+b_{k-1}\big)2+\cdots b_1\Big)2+b_0. \qquad 7.10$$

It follows that

$$x^n \;=\; \Big(\cdots\big((x^{b_k})^2x^{b_{k-1}}\big)^2\cdots x^{b_1}\Big)^2 x^{b_0},$$

where x^{b_i} is either x or 1 and so requires no multiplications to compute. Since multiplication by 1 is trivial, the total number of multiplications is k (from squarings) plus the number of b_0 through b_{k-1}

which are nonzero. Thus the number of multiplications is between k and $2k$. Since $2^{k+1} > n \geq 2^k$ by (7.10), this method always requires $\Theta(\ln n)$ multiplications.[1] In contrast, our previous methods always required at least $\Theta(\ln n)$ multiplications and sometimes required as many as $\Theta(n)$.

Does our latest method require the least number of multiplications? Not always. There is no known good way to calculate the minimum number of multiplications required to compute x^n. $\square$

Example 7.23 Finding a maximum subsequence sum Suppose we are given a sequence of n arbitrary real numbers $a_1, a_2, \ldots, a_n$. We want to find i and $j \geq i$ such that $\sum_{k=i}^{j} a_k$ is a large as possible.

Here's a simple approach: For each $1 \leq i \leq j \leq n$ compute $A_{i,j} = \sum_{k=i}^{j} a_k$, then find the maximum of these numbers. Since it takes $j - i$ additions to compute $A_{i,j}$, the number of additions required is

$$\sum_{j=1}^{n} \sum_{i=1}^{j} (j-i),$$

which turns out to be approximately $n^3/6$. The simple approach to find the maximum of the approximately $n^2/2$ numbers $A_{i,j}$ requires about $n^2/2$ comparisons. Thus the total work is $\Theta(n^3)$.

Can we do better? Yes, there are ways to compute the $A_{i,j}$ in $\Theta(n^2)$. The work will be $\Theta(n^2)$.

Can we do better? Yes, there is a divide and conquer approach. The idea is

- split $a_1, \ldots, a_n$ into two sequences $a_1, \ldots, a_k$ and $a_{k+1}, \ldots, a_n$, where $k \approx n/2$,
- compute the information for the two halves recursively,
- put the two halves together to get the information for the original sequence $a_1, \ldots, a_n$.

There is a problem with this. Consider the sequence $3, -4, 2, 2, -4, 3$ the two half sequences each have a maximum of 3, but the maximum of the entire sequence is $2 + 2 = 4$. The problem arises because the maximum sum is split between the two half-sequences $3, -4, 2$ and $2, -4, 3$. We get around this by keeping track of more information. In fact we keep track of the maximum sum, the maximum sum that starts at the left end of the sequence, the maximum sum that ends at the right end of the sequence, and the total sum. Here's an algorithm.

```
Procedure MaxSum($M, L, R, T, (a_1, \ldots, a_n)$)
    If $n = 1$
        Set $M = L = R = T = a_1$
    Else
        $k = \lfloor n/2 \rfloor$
        MaxSum($M_\ell, L_\ell, R_el, T_\ell, (a_1, \ldots, a_k)$)
        MaxSum($M_r, L_r, R_r, T_r, (a_{k+1}, \ldots, a_n)$)
        $M$ = max($M_\ell, M_r, R_\ell + L_r$)
        $L$ = max($L_\ell, T_\ell + L_r$)
        $R$ = max($R_r, T_r + R_\ell$)
        $T$ = $T_\ell + T_r$
    End if
    Return
End
```

[1] The notation Θ indicates same rate of growth to within a constant factor. For more details, see page 368.

Why does this work?

- It should be clear that the calculation of T is correct.
- You should be able to see that M, the maximum sum, is either the maximum in the left half (M_ℓ), the maximum in the right half (M_r), or a combination from each half ($R_\ell + L_r$). This is just what the procedure computes.
- L, the maximum that starts on the left, either ends in the left half (L_ℓ) or extends into the right half ($T_\ell + L_r$), which is what the procedure computes.
- The reasoning for R is the mirror image of that for L.

How long does this algorithm take to run? Ignoring the recursive part, there is a constant amount of work in the code, with one constant for $n = 1$ and another for $n > 1$. Hence

$$\text{(total time)} \quad = \quad \Theta(1) \times \text{(number of calls of MaxSum)}.$$

Every call of MaxSum when $n > 1$ divides the sequence. We must insert $n-1$ divisions between the elements of $a_1, \ldots, a_n$ to get sequences of length 1. Hence there are $n-1$ calls of this type. `MaxSum` also calls itself for each of the n elements of the sequence. Thus there are a total of $2n-1$ calls and so the running time is $\Theta(n)$. □

There is an important principle that surfaced in the previous example which didn't arise in our simpler examples of finding a recursive algorithm:

Principle *In order to find a recursive algorithm for a problem, it may be helpful, even necessary, to ask for more—either a stronger result or the calculation of more information.*

In the example, we introduced new variables in our algorithm to keep track of other sums. Without such variables, the algorithm would not have worked. As we remarked in Section 7.1, this principle also applies to inductive proofs. We have seen some examples of this:

- When we proved the existence of lineal spanning trees in Theorem 6.2 (p. 153), it was necessary to prove that we could find one with any vertex of the graph as the root of the tree.
- When we studied Ramsey problems in Example 7.4 (p. 201), we had to replace $N(k)$ with the more general $N(k_1, k_2)$ in order to carry out our inductive proof.

Exercises

7.4.1. What is the least number of multiplications you can find to compute each of the following: x^{15}, y^{21}, z^{47} and w^{49}.

7.4.2. This problem concerns the Fibonacci numbers. They satisfy the recursion $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. People use various values for F_0 and F_1. We will use $F_0 = 0$ and $F_1 = 1$. (Elsewhere in the text we may find it convenient to choose different initial values.)

(a) Compute F_n for $n \leq 7$.

(b) Recall that A^{t} means the transpose of the matrix A. Let $\vec{v}_n$ be the column vector $(F_n, F_{n+1})^{\mathrm{t}}$. Show that $\vec{v}_{n+1} = M\vec{v}_n$ where $M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Conclude that $\vec{v}_n = M^n \vec{v}_0$. Suggest a rapid method for calculating F_n.

(c) Show that

$$M^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix}.$$

(d) Use $M^{2n} = (M^n)^2$ to prove that $F_{2n} = F_n(F_{n+1} + F_{n-1}) = F_{n+1}^2 - F_{n-1}^2$ and $F_{2n+1} = F_{n+1}^2 + F_n^2$.

7.4.3. Suppose that $a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k}$ for $n \geq k$. Extend the idea in the previous exercise to describe how to rapidly compute a_n for a large value of n.

7.4.4. In this problem you are given a bag of n coins. All the coins have the same weight, except for one, which is counterfeit. You are given a balance type scales. This means that given two piles of coins, you can use the scales to determine whether the piles have the same weight or, if they differ, which is heavier. The goal is to devise a strategy for locating the counterfeit coin with the least possible number of weighings. (See Exercise 7.3.4 (p. 218).)

(a) Given that the counterfeit coin is lighter, devise a recursive divide and conquer strategy for finding it.

Hint. If you strategy is a good one, it will find the coin after k weighings when $n = 3^k$.

(b) Modify the proof of the lower bound for the number of comparisons needed for sorting to show that the average number of weighings needed in any algorithm is at least $\log_3 n$.

(c) Devise an algorithm when it is not known whether the counterfeit coin is lighter or heavier.

(d) Suppose that there are two counterfeit coins, both of the same weight and both lighter than the real coins. Devise a strategy.

*7.4.5. A full binary RP-tree is a rooted plane tree in which each vertex is either a leaf or has exactly two children. An example is a decision tree in which each decision is either "yes" or "no".

Suppose that we are given a full binary RP-tree T with root r and a function f from vertices to the real numbers. If v is a vertex of T, let $F(v)$ be the sum of $f(x)$ over all vertices x in the subtree rooted at v. We want to find

$$F(T) = \max_{v \in T} F(v).$$

One way to do this is compute $F(v)$ for each vertex $v \in T$ and then find the maximum. This takes $\Theta(n \ln n)$ work, where n is the number of vertices in T. Find a better method.

*7.4.6. If you found Example 7.23 easy, here's a challenge: Extend the problem to a doubly indexed array $a_{k,m}$. In this case we want i_1, i_2, j_1 and j_2 so that $\sum_{k=i_1}^{j_1} \sum_{m=i_2}^{j_2} a_{k,m}$ is as large as possible. We warn you that you must keep track of quite a bit more information.

7.4.7. Here's another approach to the identities of Exercise 7.4.2. Let $a_n = a_{n-1} + a_{n-2}$, where a_0 and a_1 are given.

(a) Prove that $a_n = a_0 F_{n-1} + a_1 F_n$ for $n > 0$.

(b) Show that, if $a_0 = F_k$ and $a_1 = F_{k+1}$, then $a_n = F_{n+k}$. Conclude that $F_{n+k} = F_k F_{n-1} + F_{k+1} F_{n-2}$.

Notes and References

Most introductory combinatorics texts discuss induction, but discussions of recursive algorithms are harder to find. The subject of recursion in treated beautifully by Roberts [4]. Williamson [6; Ch.6] discusses the basic ideas behind recursive algorithms with applications to graphs.

Further discussion of Ramsey's Theorem as well as some of its consequences appears in Cohen's text [2; Ch. 5]. A more advanced treatment of the theorem and its generalizations has been given in the monograph by Graham, Rothschild and Spencer [3].

People have studied the Tower of Hanoi puzzle with the same rules but with more than one extra pole for a total of $k > 3$ poles. The optimal strategy is not known; however, it is known that if $H_n(k)$ is the least number of moves required, then $\log_2 H_n(k) \sim (n(k-2)!)^{1/(k-2)}$ [1].

Divide and conquer methods are discussed in books that teach algorithm *design*; however, little has been written on general design strategies. Some of our discussion is based on the article by Smith [5].

1. Xiao Chen and Jian Shen, On the Frame-Stewart conjecture about the Towers of Hanoi, *SIAM J. Comput.* **33** (2004) 584–589.
2. Daniel I.A. Cohen, *Basic Techniques of Combinatorial Theory*, John Wiley (1978).
3. Ronald L. Graham, Bruce L. Rothschild and Joel H. Spencer, *Ramsey Theory*, 2nded., John Wiley (1990).
4. Eric Roberts, *Thinking Recursively*, John Wiley (1986).
5. Douglas R. Smith, Applications of a strategy for designing divide-and-conquer algorithms, *Science of Computer Programming* **8** (1987), 213–229.
6. S. Gill Williamson, *Combinatorics for Computer Science*, Dover (2002).