It is not necessary to design the steps required to solve the problem from scratch; it is sufficient to guarantee that (1) it is possible to solve a small instance of the problem (the base case), and (2) a solution to every problem can be constructed from solutions of smaller problems (the inductive step).

With this principle in mind, we should concentrate on reducing the problem to a smaller problem (or to a set of smaller problems). The trouble is that it is usually not easy to find a way to reduce the problem. In this chapter, we present several techniques to facilitate this process. The examples in this chapter were chosen not because of their importance (some of them have limited applicability), but because they are simple and yet they illustrate the principles we want to emphasize. We will present numerous other examples of this approach throughout the book.

5.2 Evaluating Polynomials

We start with a simple algebraic problem — evaluating a given polynomial at a given point.

The Problem Given a sequence of real numbers $a_n, a_{n-1}, \ldots, a_1, a_0$, and a real number x, compute the value of the polynomial $P_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$.

This problem may not seem to be a natural candidate for an inductive approach. Nevertheless, we will show that induction can lead directly to a very good solution to the problem. We start with the most simple (almost trivial) approach, then find variations of it that lead to better solutions.

The problem involves $n+2$ numbers. The inductive approach is to solve this problem in terms of a solution to a smaller problem. In other words, we try to reduce the problem to one with smaller size, which we then solve recursively, or, as we call it, *by induction.* The first natural attempt is to reduce the problem by removing a_n. We are left with the problem of evaluating the polynomial

$$P_{n-1}(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0.$$

This is the same problem, except that it has one less parameter. Therefore, we can solve it by induction.

Induction hypothesis: *We know how to evaluate a polynomial represented by the input $a_{n-1}, \ldots, a_1, a_0$, at the point x (i.e., we know how to compute $P_{n-1}(x)$).*

We can now use the hypothesis to solve the problem by induction. First, we have to solve the base case, which is computing a_0; this is trivial. Then, we must show how to solve the original problem (computing $P_n(x)$) with the aid of the solution to the smaller problem (which is the value of $P_{n-1}(x)$). This step is straightforward in this case; simply compute x^n, multiply it by a_n, and add the result to $P_{n-1}(x)$:

$$P_n(x) = P_{n-1}(x) + a_n x^n.$$

At this point it may seem that the use of induction in this problem is frivolous — it just complicates a very simple solution. The algorithm implied by the preceding discussion is merely evaluating the polynomial from right to left as it is written. In a moment, however, we will see the power of our approach.

Although the algorithm is correct, it is not efficient. It requires $n + n-1 + n-2 + \cdots + 1 = n(n+1)/2$ multiplications and n additions. We now use induction a little differently to obtain a better solution.

We make the first improvement by observing that there is a great deal of redundant computation: The powers of x are computed from scratch. We can save many multiplications by using the value of x^{n-1} when we compute x^n. We make this change by including the computation of x^k in the induction hypothesis.

Stronger induction hypothesis: *We know how to compute the value of the polynomial* $P_{n-1}(x)$, and *we know how to compute* x^{n-1}.

This induction hypothesis is stronger, since it requires computing x^{n-1}, but it is easier to extend (since it is now easier to compute x^n). We need to perform only one multiplication to compute x^n, then one more multiplication to get $a_n x^n$, then one addition to complete the computation. (The induction hypothesis is not too strong, since we need to compute x^{n-1} anyway.) Overall, there are $2n$ multiplications and n additions. It is interesting to note that, even though the induction hypothesis requires more computation, it leads to less work overall. We will return to this point later. This algorithm looks good by all measures. It is efficient, simple, and easy to implement. However, a better algorithm exists. We discover it by using induction in yet another different way.

Reducing the problem by removing the last coefficient, a_n, is the straightforward step, but it is not the only possible reduction. We can also remove the first coefficient, a_0. The smaller problem becomes the evaluation of the polynomial represented by the coefficients $a_n, a_{n-1}, \ldots, a_1$, which is

$$P'_{n-1}(x) = a_n x^{n-1} + a_{n-1} x^{n-2} + \cdots + a_1.$$

(Notice that a_n is now the $(n-1)$th coefficient, a_{n-1} is the $(n-2)$th coefficient, and so on.) So we have a new induction hypothesis.

Induction hypothesis (reversed order): *We know how to evaluate the polynomial represented by the coefficients $a_n, a_{n-1}, \ldots, a_1$ at the point x (i.e., we know how to compute $P'_{n-1}(x)$).*

This hypothesis is more suited to our purposes, because it is easier to extend. Clearly, $P_n(x) = x \cdot P'_{n-1}(x) + a_0$. Therefore, only one multiplication and one addition are required to compute $P_n(x)$ from $P'_{n-1}(x)$. The complete algorithm can be described by the following expression:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = ((\cdots((a_n x + a_{n-1})x + a_{n-2})\cdots)x + a_1)x + a_0.$$

This algorithm is known as **Horner's rule** after the English mathematician W.G. Horner. (It was also mentioned by Newton, see [Knuth 1981], page 467.) The program to evaluate the polynomial is given in Fig. 5.1.

```
Algorithm Polynomial_Evaluation ($\bar{a}$, $x$) ;
Input: $\bar{a} = a_0, a_1, a_2, \ldots, a_n$ (coefficients of a polynomial), and $x$ (a real
    number).
Output: $P$ (the value of the polynomial at $x$).

begin
    $P := a_n$;
    for $i := 1$ to $n$ do
        $P := x * P + a_{n-i}$
end
```

Figure 5.1 Algorithm *Polynomial_Evaluation.*

Complexity The algorithm requires only n multiplications, n additions, and one extra memory location. Even though the previous solutions seemed very simple and very efficient, we have found it worthwhile to pursue a better algorithm. Not only is this algorithm faster than those described previously, but also its corresponding program is simpler.

Comments Induction allows us to concentrate on extending solutions of smaller subproblems to those of larger problems. Suppose that we want to solve $P(n)$, which is a problem P that depends on a parameter n (usually its size). We start with an arbitrary instance of $P(n)$, and try to solve it by using the assumption that $P(n-1)$ has already been solved. There are many possible ways to define the induction hypothesis and many possible ways to use it. We will survey several of these methods, and will show their power in designing algorithms.

This simple example illustrates the flexibility we have when we use induction. The trick that led to Horner's rule was merely considering the input from left to right, instead of the intuitive right to left. Another common possibility is comparing top down versus bottom up (when a tree structure is involved). It is also possible to go in increments of 2 (or more) rather than 1, and there are numerous other possibilities. Moreover, sometimes the best induction sequence is not the same for all inputs. It may be worthwhile to design an algorithm just to find the best way to perform the reduction. We will see examples of all these possibilities.

5.3 Maximal Induced Subgraph

Consider the following problem. You are arranging a conference of scientists from different disciplines and you have a list of people you want to invite. You assume that everyone on the list will agree to come under the condition that there will be ample opportunity to exchange ideas. For each scientist, you write down the names of all other scientists on the list with whom interaction is likely. You would like to invite as many people on the list as possible, but you want to guarantee that each one will have at least k other people with whom to interact (k is a fixed number, independent of the number of invitees). You do not have to arrange the interactions; in particular, you do not have to make sure that there is enough time for them to occur. You just want to lure everyone to the conference. How do you decide whom to invite? This problem corresponds to the following graph-theoretic problem. Let $G=(V, E)$ be an undirected graph. An **induced subgraph** of G is a graph $H=(U, F)$ such that $U \subseteq V$ and F includes all edges in E both of whose incident vertices are in U. A **degree** of a vertex is the number of vertices adjacent to that vertex. The vertices of the graph correspond to the scientists, and two vertices are connected if there is a potential for the two corresponding scientists to exchange ideas. An induced subgraph corresponds to a subset of the scientists.

The Problem Given an undirected graph $G=(V, E)$ and an integer k, find an induced subgraph $H=(U, F)$ of G of maximum size such that all vertices of H have degree $\geq k$ (in H), or conclude that no such induced subgraph exists.

A direct approach to solving this problem is to remove vertices whose degree is $<k$. As vertices are removed with their adjacent edges, the degrees of other vertices may be reduced. When the degree of a vertex becomes less than k, that vertex should be removed. The order of removals, however, is not clear. Should we remove all the vertices of degree $<k$ first, then deal with vertices whose degrees were reduced? Should we remove first one vertex of degree $<k$, then continue with affected vertices? (These two approaches correspond to breadth-first search versus depth-first search, which are discussed in detail in Section 7.3.) Will both approaches lead to the same result? Will the resulting graph be of maximum size? All these questions are easy to answer; the approach we will describe makes answering them even easier.

Instead of thinking about our algorithm as a sequence of steps that a computer has to take to calculate a result, think of *proving a theorem* that the algorithm exists. We do not suggest attempting a formal proof (at least not at this first stage). The idea is to imitate the steps we take in proving a theorem, in order to gain insight into the problem. We need to find the maximum induced subgraph that satisfies the given conditions. Here is a "proof" by induction.

Induction hypothesis: *We know how to find maximum induced subgraphs all of whose vertices have degrees $\geq k$, provided that the number of vertices is $< n$.*

We need to prove that this "theorem" is true for a base case, and that its truth for $n-1$ implies its truth for n. The first nontrivial base case occurs when $n = k+1$, because if $n \leq k$, then all the degrees are less than k. If $n = k+1$, then the only way to have all the degrees equal to k is to have a **complete graph** (namely, all vertices are connected), which we can detect. So, assume now that $G = (V, E)$ is a graph with $n > k+1$ vertices. If all the vertices have degrees $\geq k$, then the whole graph satisfies the conditions and we are done. Otherwise, there exists a vertex v with degree $< k$. It is obvious that the degree of v remains $< k$ in any induced subgraph of G; hence, v does not belong to any subgraph that satisfies the conditions of the problem. Therefore, we can remove v and its adjacent edges without affecting the conditions of the theorem. After v is removed, the graph has $n-1$ vertices — and, by the induction hypothesis, we know how to solve the problem.

We are now done. The algorithm and the answers to the questions we raised earlier are now clear. Any vertex of degree $< k$ can be removed. The order of removals is immaterial. The graph remaining after all these removals must be of maximum size because these removals are *mandatory*. It is also clear that the algorithm is correct, because we designed it by proving its correctness!

Comments The best way to reduce a problem is to eliminate some of its elements. In this example, the application of induction was straightforward, mainly because it was clear which vertices we should eliminate and how we should eliminate them. The reduction follows immediately. In general, however, the elimination process may not be straightforward. We will see examples of combining two elements into one, causing the number of elements to be reduced (Section 6.6); of eliminating restrictions on the problem rather than eliminating parts of the input (Section 7.7); and of designing a special algorithm to find which elements can be eliminated (Section 5.5). Another example of eliminating the right elements is presented next. It is interesting to note that, if we replace "$\geq$" with "$\leq$" in the statement of the problem (that is, if we look for a maximal induced subgraph all of whose degrees are *at most* k), the problem becomes much more difficult (see Exercise 11.12).

5.4 Finding One-to-One Mappings

Let f be a function that maps a finite set A into itself (i.e., every element of A is mapped to another element of A). For simplicity, we denote the elements of A by the integers 1 to n. We assume that the function f is represented by an array $f[1..n]$ such that $f[i]$ holds the value of $f(i)$ (which is an integer between 1 and n). We call f a *one-to-one* function if, for every element j, there is at most one element i that is mapped to j. The function f can be represented by a diagram, as shown in Fig. 5.2, where both sides correspond to the same set and the edges indicate the mapping. The function in Fig. 5.2 is clearly not a one-to-one function.

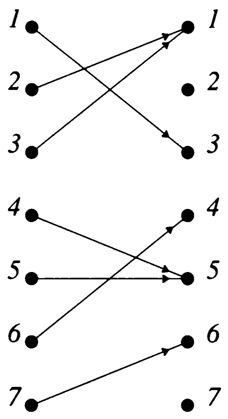

Figure 5.2 A mapping from a set into itself (both sides represent the same set).

The Problem Given a finite set A and a mapping f from A to itself, find a subset $S \subseteq A$ with the maximum number of elements, such that (1) the function f maps every element of S to another element of S (i.e., f maps S into itself), and (2) no two elements of S are mapped to the same element (i.e., f is one-to-one when restricted to S).

If f is originally one-to-one, then the whole set A satisfies the conditions of the problem, and A is definitely maximal. If, on the other hand, $f(i)=f(j)$ for some $i \neq j$, then S cannot contain both i and j. For example, the set S that solves the problem given in Fig. 5.2 cannot contain both 2 and 3 since $f(2)=f(3)=1$. The choice of which one of them to eliminate cannot be arbitrary. Suppose, for example, that we decide to eliminate 3. Since 1 is mapped to 3, we must eliminate 1 as well (the mapping must be into S and 3 is no longer in S). But if 1 is eliminated, then 2 must be eliminated as well (for the same reason). But, this subset is not maximal, since it is easy to see that we could have eliminated 2 alone. (The solution for Fig. 5.2 is the subset $\{1,3,5\}$.) The problem is to find a general method to decide which elements to include.

Fortunately, we have some flexibility in deciding how to reduce the problem to a smaller one. We can reduce the size of the problem by finding either an element that belongs to S or an element that does not belong to S. We will do the latter. We use the straightforward induction hypothesis.

Induction hypothesis: *We know how to solve the problem for sets of* $n-1$ *elements.*

The base case is trivial: If there is only one element in the set, then it must be mapped to itself, which is a one-to-one mapping. Assume now that we have a set A of n elements and we are looking for a subset S that satisfies the conditions of the problem. We claim that any element i that has no other element mapped to it cannot belong to S. (In other words, an element i in the right side of the diagram, which is not connected to any edge, cannot be in S.) Otherwise, if $i \in S$ and S has, say, k elements, then those k elements are mapped into at most $k-1$ elements; therefore, the mapping cannot be one-to-one. If there is such an i, then we simply remove it from the set. We now have a set $A'=A-\{i\}$ with $n-1$ elements, which f maps into itself; by the induction hypothesis, we know how to solve the problem for A'. If no such i exists, then the mapping is one-to-one, and we are done.

The essence of this solution is that we *must* remove i. We proved that i cannot belong to S. This is the strength of induction: Once we remove an element and reduce the size of the problem, we are done. We have to be careful, however, that the reduced problem is exactly the same (except for size) as the original problem. The only condition on the set A and the function f was that f maps A into itself. This condition is still maintained for the set $A-\{i\}$, since there was nothing that was mapped to i. The algorithm terminates when no more elements can be removed.

Implementation We described the algorithm as a recursive procedure. In each step, we found an element such that no other element is mapped to it, removed it, and continued recursively. The implementation, however, need not be recursive. We can maintain a counter $c[i]$ with each element i. Initially, $c[i]$ should be equal to the number of elements that are mapped to i. We can compute $c[i]$, for all i, in n steps by scanning the array and incrementing the appropriate counters. We then put all the elements that have a zero counter in a queue. In each step, we remove an element j from the queue (and the set), decrement $c[f(j)]$, and, if $c[f(j)]=0$, we put $f(j)$ in the queue. The algorithm terminates when the queue is empty. The algorithm is given in Fig. 5.3.

Complexity The initialization part requires $O(n)$ operations. Every element can be put on the queue at most once, and the steps involved in removing an element from the queue take constant time. The total number of steps is thus $O(n)$.

Comments In this example, we reduced the size of the problem by eliminating elements from a set. Therefore, we tried to find the easiest way to remove an element without changing the conditions of the problem. Because the only requirement we made was that the function maps A into itself, the choice of an element to which no other element is mapped is natural.

5.5 The Celebrity Problem

The next example is a popular exercise in algorithm design. It is a nice example of a problem that has a solution that does not require scanning all the data (or even a significant part of them). Among n persons, a *celebrity* is defined as someone who is known by everyone but does not know anyone. The problem is to identify the celebrity, if one exists, by asking questions only of the form, "Excuse me, do you know the person over there?" (The assumption is that all the answers are correct, and that even the celebrity will answer.) The goal is to minimize the number of questions. Since there are

```
Algorithm Mapping ( f, n ) ;
Input: f (an array of integers whose values are between 1 and n).
Output: S (a subset of the integers from 1 to n, such that f is one-to-one on S).

begin
    S := A; { A is the set of numbers from 1 to n }
    for j := 1 to n do  c[j] := 0;
    for j := 1 to n do  increment c[f[j]];
    for j := 1 to n do
        if c[j] = 0 then put j in Queue;
    while Queue is not empty do
        remove i from the top of the queue;
        S := S - {i};
        decrement c[f[i]];
        if c[f[i]] = 0 then put f[i] in Queue
end
```

Figure 5.3 Algorithm *Mapping*.

$n(n-1)/2$ pairs of persons, there is potentially a need to ask $n(n-1)$ questions, in the worst case, if the questions are asked arbitrarily. It is not clear that we can do better in the worst case.

We can use a graph-theoretical formulation. We can build a directed graph with the vertices corresponding to the persons and an edge from person A to person B if A knows B. A celebrity corresponds to a **sink** of the graph (no pun intended). A sink is a vertex with indegree $n-1$ and outdegree 0. Notice that a graph can have at most one sink. The input to the problem corresponds to an $n \times n$ adjacency matrix (whose ij entry is 1 if the ith person knows the jth person, and 0 otherwise).

The Problem Given an $n \times n$ adjacency matrix, determine whether there exists an i such that all the entries in the ith column (except for the iith entry) are 1, and all the entries in the ith row (except for the iith entry) are 0.

The base case of two persons is simple. Consider as usual the difference between the problem with $n-1$ persons and that with n persons. We assume that we can find the celebrity among the first $n-1$ persons by induction. Since there is at most one celebrity, there are three possibilities: (1) the celebrity is among the first $n-1$, (2) the celebrity is the nth person, and (3) there is no celebrity. The first case is the easiest to handle. We need only to check that the nth person knows the celebrity, and that the celebrity does not know the nth person. The other two cases are more difficult because, to determine whether the nth person is the celebrity, we may need to ask $2(n-1)$ questions. If we ask $2(n-1)$ questions in the nth step, then the total number of questions will be $n(n-1)$ (which is what we tried to avoid). We need another approach.

The trick here is to consider the problem "backward." It may be hard to identify a celebrity, but it is probably easier to identify someone as a noncelebrity. After all, there are definitely more noncelebrities than celebrities. If we eliminate someone from consideration, then we reduce the size of the problem from n to $n-1$. Moreover, we do not need to eliminate someone specific; anyone will do. Suppose that we ask Alice whether she knows Bob. If she does, then she cannot be a celebrity; if she does not, then Bob cannot be a celebrity. We can eliminate one of them with one question.

We now consider again the three cases with which we started. We do not just take an arbitrary person as the nth person. We use the idea in the last paragraph to eliminate either Alice or Bob, then solve the problem for the other $n-1$ persons. We are guaranteed that case 2 will not occur, since the person eliminated cannot be the celebrity. Furthermore, if case 3 occurs — namely, there is no celebrity among the $n-1$ persons — then there is no celebrity among the n persons. Only case 1 remains, but this case is easy. If there is a celebrity among the $n-1$ persons, it takes two more questions to verify that this is a celebrity for the whole set. Otherwise, there is no celebrity.

The algorithm proceeds as follows. We ask A whether she knows B, and eliminate either A or B according to the answer. Let's assume that we eliminate A. We then find (by induction) a celebrity among the remaining $n-1$ persons. If there is no celebrity, the algorithm terminates; otherwise, we check that A knows the celebrity and that the celebrity does not know A.

Implementation As was the case with the algorithm in the previous section, it is more efficient to implement the celebrity algorithm iteratively, rather than recursively. The algorithm is divided into two phases. In the first phase, we eliminate all but one candidate, and in the second phase we check whether this candidate is indeed the celebrity. We start with n candidates, and, for the purpose of this discussion, let's assume that they are stored in a stack. For each pair of candidates, we can eliminate one candidate by asking one question — whether one of them knows the other. We start by taking the first two candidates from the stack, and eliminating one of them. Then, in each step, we have one remaining candidate, and, as long as the stack is nonempty, we take one additional candidate from the stack, and eliminate one of these two candidates. When the stack becomes empty, one candidate remains. We then check that this candidate is indeed the celebrity. The algorithm is presented in Fig. 5.4 (notice that the stack is implemented explicitly by the use of the indices i, j, and *next*).

Complexity At most $3(n-1)$ questions will be asked: $n-1$ questions in the first phase to eliminate $n-1$ persons, and then at most $2(n-1)$ questions to verify that the candidate is indeed a celebrity. Notice that the size of the input is not n, but rather $n(n-1)$ (the number of entries of the matrix). This solution shows that it is possible to identify a celebrity by looking at only $O(n)$ entries in the adjacency matrix, even though a priori the solution may be sensitive to each of the $n(n-1)$ entries.

```
Algorithm Celebrity (Know) ;
Input: Know (an $n \times n$ Boolean matrix ).
Output: celebrity.

begin
    i := 1 ;
    j := 2 ;
    next := 3 ;
    { in the first phase we eliminate all but one candidate }
    while next $\le n + 1$ do
        if Know[i, j] then i := next
        else j := next ;
        next := next + 1 ;
        { one of either i or j is eliminated }
    if i = $n + 1$ then
        candidate := j
    else
        candidate := i ;
    { Now we check that the candidate is indeed the celebrity }
    wrong := false ;
    k := 1 ;
    Know[candidate, candidate] := false ;
        { a dummy variable to pass the test }
    while not wrong and $k \le n$ do
        if Know[candidate, k] then wrong := true ;
        if not Know[k, candidate] then
            if candidate $\ne$ k then wrong := true ;
        k := k + 1 ;
    if not wrong then celebrity := candidate
    else celebrity := 0 { no celebrity }
end
```

Figure 5.4 Algorithm *Celebrity.*

Comments The key idea in this elegant solution is to reduce the size of the problem from n to $n-1$ in a clever way. This example shows that it sometimes pays to expend some effort (in this case — one question) to perform the reduction more effectively. Do not start by simply considering an *arbitrary* input of size $n-1$ and attempting to extend it. Select a *particular* input of size $n-1$. We will see more examples where we spend substantial time just constructing the right order of induction — and that time is well spent.

5.6 A Divide-and-Conquer Algorithm: The Skyline Problem

So far, we have seen examples from graph theory and numerical computation. This example deals with a problem of drawing shapes.

> **The Problem** Given the exact locations and shapes of several rectangular buildings in a city, draw the skyline (in two dimensions) of these buildings, eliminating hidden lines.

An example of an input is given in Fig. 5.5(a); the corresponding output is given in Fig. 5.5(b). We are interested in only two-dimensional pictures. We assume that the bottoms of all the buildings lie on a fixed line (i.e., they share a common horizon). Building B_i is represented by a triple (L_i, H_i, R_i). L_i and R_i denote the left and right x coordinates of the building, respectively, and H_i denotes the building's height. A **skyline** is a list of x coordinates and the heights connecting them arranged in order from left to right. For example, the buildings in Fig. 5.5(a) correspond to the following input:

(1,**11**,5), (2,**6**,7), (3,**13**,9), (12,**7**,16), (14,**3**,25), (19,**18**,22), (23,**13**,29), and (24,**4**,28).

(The numbers in boldface type are the heights.) The skyline in Fig. 5.5(b) is represented as follows:

(1,**11**,3,**13**,9,**0**,12,**7**,16,**3**,19,**18**,22,**3**,23,**13**,29,**0**).

(Again, the numbers in boldface type are heights.)

The straightforward algorithm for this problem is based on adding one building at a time to the skyline. The induction hypothesis is the simple one. We assume that we know how to solve the problem for $n-1$ buildings, and then we add the nth building.

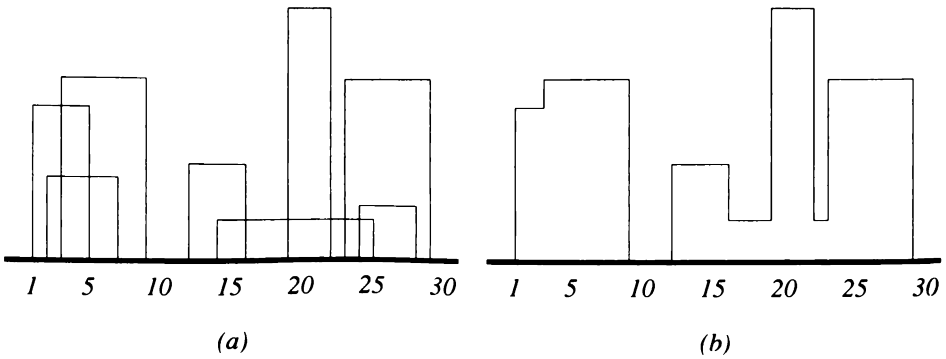

Figure 5.5 The skyline problem: (a) The input. (b) The skyline.

The problem is trivial for one building. To add a building B_n to the skyline, we need to intersect it with the existing skyline (see Fig. 5.6). Let B_n be (5,**9**,26). We first scan the skyline from left to right to find where the left side of B_n fits (i.e., we search for the appropriate x coordinate — 5 in this example). In this case, the horizontal line that "covers" 5 is the one from 3 to 9, and its height is 13. We can now scan the skyline, looking at one horizontal line after another, and adjusting whenever the height of B_n is higher than the existing height. We stop when we reach an x coordinate that is greater than the right side of B_n. For this example, we do not adjust the height from 3 to 9, but we do adjust it all the way from 9 to 19, then adjust it once more from 22 to 23. The new skyline is represented by

(1,**11**,3,**13**,9,**9**,19,**18**,22,**9**,23,**13**,29,**0**).

This algorithm is clearly correct, but it is not necessarily efficient. In the worst case, the scan for B_n requires $O(n)$ steps. Hence, the total number of steps will be $O(n)+O(n-1)+ \cdots +O(1)=O(n^2)$.

To improve the performance of this algorithm, we use a well-known technique called **divide and conquer**. Instead of using the simple induction principle of extending the solution for $n-1$ to a solution for n, we extend a solution for $n/2$ to a solution for n. (Again, the base case of one building is trivial.) Divide-and-conquer algorithms divide the inputs into smaller subsets, solve (conquer) each subset recursively, and merge the solutions together. Generally, it is more efficient to divide the problem into subproblems of about equal size. As we saw in Chapter 3, the solution of the recurrence relation $T(n)=T(n-1)+O(n)$ is $T(n)=O(n^2)$, whereas that of $T(n)=2T(n/2)+O(n)$ is $T(n)=O(n \log n)$. Therefore, if we divide the problem into two equal-sized subproblems, then combine the solutions in linear time, the algorithm runs in time $O(n \log n)$. The divide-and-conquer technique is very useful, and we will see many examples of it.

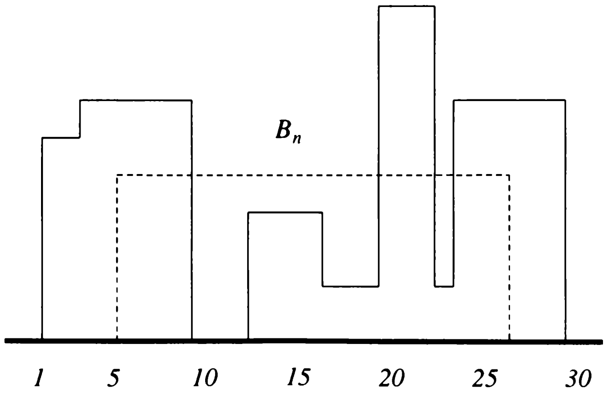

Figure 5.6 Addition of a building (dotted line) to the skyline of Figure 5.5(b) (solid lines).

The key idea behind the divide-and-conquer algorithm in this example is the observation that, in the worst case, it takes linear time to merge one building with the skyline, and also linear time to merge two different skylines. In about the same time, we achieve more using the latter approach. Two skylines can be merged with basically the same algorithm that merges one building into a skyline (Fig. 5.7). We scan the two skylines together from left to right, match x coordinates, and adjust heights when necessary. The merge can be achieved in linear time, and therefore the complete algorithm runs in time $O(n \log n)$ in the worst case. This algorithm is similar to mergesort, which is discussed in detail in Section 6.4.3. Therefore, we do not give the precise algorithm for the skyline algorithm here.

Comments Always try to get more for your money. There is nothing mysterious or technical about this principle. If the algorithm includes a step that is more general than required, consider applying this step to a more complicated part of the problem. The reason the divide-and-conquer approach is so useful is that it uses the combine step to its fullest. The recurrence relations given in Section 3.5.2 cover the most common divide-and-conquer algorithms. You should memorize these recurrence relations.

5.7 Computing Balance Factors in Binary Trees

Let T be a binary tree with root r. The **height** of a node v is the distance between v and the farthest leaf down the tree. The **balance factor** of a node v is defined as the difference between the height of the node's left subtree and the height of the node's right subtree (we assume that the children of a node are labeled by left or right). In Chapter 4, we discussed AVL trees, in which all nodes have balance factors of -1, 0, or 1. In this section, we consider arbitrary binary trees. Figure 5.8 shows a tree in which each node is labeled with numbers representing h/b, where h is the node's height and b is its balance factor.

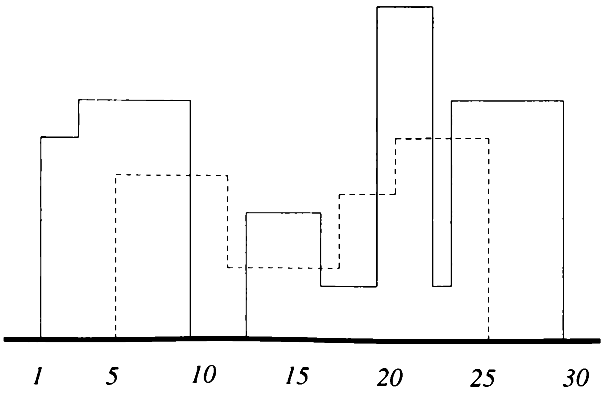

Figure 5.7 Merging two skylines.

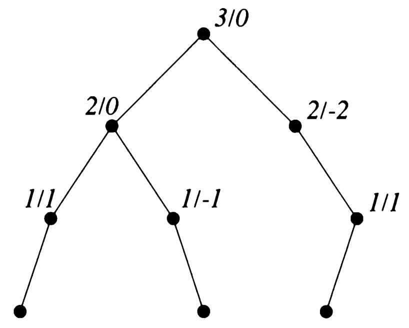

Figure 5.8 A binary tree. The numbers represent h/b, where h is the height and b is the balance factor.

The Problem Given a binary tree T with n nodes, compute the balance factors of all the nodes.

We use the regular inductive approach with the straightforward induction hypothesis.

> **Induction hypothesis:** *We know how to compute balance factors of all nodes in trees that have* $< n$ *nodes.*

The base case of $n = 1$ is trivial. Given a tree with $n > 1$ nodes, we remove the root, then solve the problem (by induction) for the two subtrees that remain. We chose to remove the root because the balance factor of a node depends on only the nodes below that node. We now know the balance factors of all the nodes, except for the root. The root's balance factor, however, depends not on the balance factors of the root's children, but rather on their height. Hence, simple induction does not work in this case. We need to know the heights of the children of the root. The idea is to include the height-finding problem within the original problem:

> **Stronger induction hypothesis:** *We know how to compute balance factors* ***and*** *heights of all nodes in trees that have* $< n$ *nodes.*

Again, the base case is trivial. Now, when we consider the root, we can determine its balance factor easily by calculating the difference between the heights of its children. Furthermore, we can also determine the height of the root — it is the maximal height of the two children plus 1.

The key to the algorithm is that it solves a slightly extended problem. Instead of computing only balance factors, we also compute heights. The extended problem turns out to be an easier one to solve, because the heights are easy to compute. In many cases, solving a stronger problem is easier. With induction, we need only to extend a solution of a small problem to a solution of a larger problem. If the solution is broader (because the problem is extended), then the induction step may be easier, since we have more with which to work. It is a common error to forget that there are two different parameters in this problem, and that each one should be computed separately. We will present several examples of such errors later in the book.

5.8 Finding the Maximum Consecutive Subsequence

The following problem is from Bentley [1986] (it also appeared in Bates and Constable [1985]).

The Problem Given a sequence $x_1, x_2, \ldots, x_n$ of real numbers (not necessarily positive) find a subsequence $x_i, x_{i+1}, \ldots, x_j$ (of consecutive elements) such that the sum of the numbers in it is maximum over all subsequences of consecutive elements.

We call such a subsequence a **maximum subsequence**. For example, in the sequence (2, −3, 1.5, −1, 3, −2, −3, 3), the maximum subsequence is (1.5, −1, 3); its sum is 3.5. There may be several maximum subsequences in a given sequence. If all the numbers are negative, then the maximum subsequence is empty (by definition, the sum of the empty subsequence is 0). We would like to have an algorithm that solves the problem and reads the sequence in order only once.

The straightforward induction hypothesis is as follows:

Induction hypothesis: *We know how to find the maximum subsequence in sequences of size* $<n$.

If $n=1$, then the maximum subsequence consists of the single number if that number is nonnegative, or the empty subsequence otherwise. Consider a sequence $S=(x_1, x_2, \ldots, x_n)$ of size $n>1$. By induction, we know how to find a maximum subsequence in $S'=(x_1, x_2, \ldots, x_{n-1})$. If that maximum subsequence is empty, then all the numbers in S' are negative, and we need to consider only x_n. Assume that the maximum subsequence found by induction in S' is $S'_M=(x_i, x_{i+1}, \ldots, x_j)$, for certain i and j such that $1 \le i \le j \le n-1$. If $j=n-1$ (namely, the maximum subsequence is a *suffix*), then it is easy to extend the solution to S: If x_n is positive, then it extends S'_M; otherwise, S'_M remains maximum. However, if $j<n-1$, then there are two possibilities. Either S'_M remains maximum, or there is another subsequence, which is not maximum in S', but is maximum in S when x_n is added to it.

The key idea here is to **strengthen the induction hypothesis**. We first illustrate the technique by using it to solve the maximum-subsequence problem, then discuss it in more generality in the next section. The problem we had with the straightforward induction hypothesis was that x_n may extend a subsequence that is not maximum in S', and thus may create a new maximum subsequence. Knowing only the maximum subsequence in S' is thus not sufficient. However, x_n can extend only a subsequence that ends at $n-1$ — that is, a suffix of S'. Suppose that we strengthen the induction hypothesis to include the knowledge of the maximum suffix, denoted by $S'_E = (x_k, x_{k+1}, \ldots, x_{n-1})$.

> **Stronger induction hypothesis:** *We know how to find, in sequences of size $< n$, a maximum subsequence overall, and the maximum subsequence that is a suffix.*

If we know both subsequences, the algorithm becomes clear. We add x_n to the maximum suffix. If the sum is more than the global maximum subsequence, then we have a new maximum subsequence (as well as a new suffix). Otherwise, we retain the previous maximum subsequence. We are not done yet. We also need to find the new maximum suffix. It is not true that we always simply add x_n to the previous maximum suffix. It could be that the maximum suffix ending at x_n is negative. In that case, it is better to take the empty set as the maximum suffix (such that later x_{n+1} will be considered by itself). The algorithm for finding the sum of the maximum subsequence is given in Fig. 5.9.

```
Algorithm Maximum_Consecutive_Subsequence (X, n) ;
Input: X (an array of size n).
Output: Global_Max (the sum of the maximum subsequence).

begin
   Global_Max := 0 ;
   Suffix_Max := 0 ;
   for i := 1 to n do
      if x[i] + Suffix_Max > Global_Max then
         Suffix_Max := Suffix_Max + x[i] ;
         Global_Max := Suffix_Max
      else if x[i] + Suffix_Max > 0 then
            Suffix_Max := x[i] + Suffix_Max
         else Suffix_Max := 0
end
```

Figure 5.9 Algorithm *Maximum_Consecutive_Subsequence.*

5.9 Strengthening the Induction Hypothesis

Strengthening the induction hypothesis is one of the most important techniques for proving mathematical theorems with induction. When attempting an inductive proof, we often encounter the following scenario. Denote the theorem by P. The induction hypothesis can be denoted by $P(<n)$, and the proof must conclude that $P(<n) \Rightarrow P(n)$. In many cases, we can add another assumption, call it Q, under which the proof becomes easier. That is, it is easier to prove $[P \text{ and } Q](<n) \Rightarrow P(n)$ than it is to prove

$P(<n) \Rightarrow P(n)$. The assumption seems correct, but it is not clear how we can prove it. The trick is to include Q in the induction hypothesis. We now have to prove that $[P$ and $Q](<n) \Rightarrow [P$ and $Q](n)$. P and Q is a stronger theorem than just P, but often stronger theorems are easier to prove. This process can be repeated and, with the right added assumptions, the proof becomes tractable. The maximum-subsequence problem is a good example of how this principle is used to improve algorithms.

A nice analogy to this principle is a well-known phenomenon: It is easier to add $1 million to profits that are based on $100 million of sales, than it is to add $1 thousand to profits that are based on $10 of sales.

The most common error people make while using this technique is to ignore the fact that an additional assumption was added and to forget to adjust the proof. In other words, they prove that $[P$ and $Q](<n) \Rightarrow P(n)$, without even noticing that Q was assumed. This oversight corresponds to forgetting to compute the new maximum suffix in the maximum-subsequence example. In the balance factors example, it corresponds to forgetting to compute the heights separately — which, unfortunately, is a common error. We cannot overemphasize this fact:

It is crucial to follow the induction hypothesis precisely.

We will present more complicated examples of strengthening the induction hypothesis in Sections 6.11.3, 6.13.1, 7.5, 8.3, and 12.3.1 (among others).

5.10 Dynamic Programming: The Knapsack Problem

Suppose that we are given a *knapsack* and we want to pack it fully with items. There may be many different items of different shapes and sizes, and our only goal is to pack the knapsack as full as possible. The knapsack may correspond to a truck, a ship, or a silicon chip, and the problem is to package items. There are many variations of this problem; we consider only a simple one dealing with one-dimensional items. Other variations of the knapsack problem are presented in the exercises, and in Chapter 11.

The Problem Given an integer K and n items of different sizes such that the ith item has an integer size k_i, find a subset of the items whose sizes sum to exactly K, or determine that no such subset exists.

We denote the problem by $P(n, K)$, such that n denotes the number of items and K denotes the size of the knapsack. We will implicitly assume that the n items are those that are given as the input to the problem, and we will not include their sizes in the notation of the problem. Thus, $P(i, k)$ denotes the problem with the first i items and a knapsack of size k. For simplicity, we first concentrate on only the decision problem, which is to determine whether a solution exists. We start with the straightforward induction approach.

Induction hypothesis (first attempt): *We know how to solve $P(n-1, K)$.*

The base case is easy; there is a solution only if the single element is of size K. If there is a solution to $P(n-1, K)$ — that is, if there is a way to pack some of the $n-1$ items into the knapsack — then we are done; we will simply not use the nth item. Suppose, however, that there is no solution for $P(n-1, K)$. Can we use this negative result? Yes — it means that the nth item must be included. In this case, the rest of the items must fit into a smaller knapsack of size $K-k_n$. We have reduced the problem to two smaller subproblems: $P(n-1, K)$ and $P(n-1, K-k_n)$. To complete the solution, we have to strengthen the hypothesis. We need to solve the problem not only for knapsacks of size K, but also for knapsacks of all sizes at most K.

Induction hypothesis (second attempt): *We know how to solve $P(n-1, k)$ for all $0 \le k \le K$.*

The previous reduction did not depend on a particular value of K; it will work for any k. We can use this hypothesis to solve $P(n, k)$ for all $0 \le k \le K$. The base case $P(1, k)$ can be easily solved: If $k=0$, then there is always a (trivial) solution. Otherwise, there is a solution only if the first item is of size k. We now reduce $P(n, k)$ to the two problems $P(n-1, k)$ and $P(n-1, k-k_n)$. If $k-k_n<0$, then we ignore the second problem. Both problems can be solved by induction. This is a valid reduction, and we now have an algorithm; however, the algorithm may be inefficient. We reduced a problem of size n to *two* subproblems of size $n-1$! (We also reduced the value of k in one subproblem.) Each of these two subproblems may be reduced to two other subproblems, leading to an exponential algorithm.

Fortunately, it is possible in many cases to improve the running time for these kinds of problems. The main observation is that the total number of possible problems may not be too high. In fact, we introduced the notation of $P(i, k)$ especially to demonstrate this observation. There are n possibilities for the first parameter and K possibilities for the second one. Overall, there are only nK different possible problems! The exponential running time resulted from doubling the number of problems after every reduction, but if there are only nK different problems, then we must have generated the same problem many many times. The solution is to remember all the solutions and never solve the same problem twice. This approach is a combination of strengthening the induction hypothesis and using strong induction (which is using the assumption that all solutions to smaller cases, and not only that for $n-1$, are known). Let's see now how to implement this approach.

We store all the known results in an $n \times K$ matrix. The (i, k)th entry in the matrix contains the information about the solution of $P(i, k)$. The reduction from the second-attempt hypothesis basically computes the nth row of the matrix. Each entry in the nth row is computed from two of the entries above it. If we are interested in finding the actual subset, then we can add to each entry a flag that indicates whether the corresponding item was selected in that step. This flag can then be traced back from the (n, K)th entry, and the subset can be recovered. The algorithm is given in Fig. 5.10. Figure 5.11 shows the complete matrix for a given input.

```
Algorithm Knapsack (S, K) ;
Input: S (an array of size n storing the sizes of the items),
    and K (the size of the knapsack).
Output: P (a two-dimensional array such that P [i, k ].exist = true if there
    exists a solution to the knapsack problem with the first i elements and a
    knapsack of size k, and P [i, k ].belong = true if the ith element belongs
    to that solution).
    { See Exercise 5.15 for suggestions about improving this program. }

begin
    P [0, 0].exist := true ;
    for k := 1 to K do
        P [0, k ].exist := false ;
    { there is no need to initialize P [i, 0] for $i \geq 1$, because it will
        be computed from P [0, 0] }
    for i := 1 to n do
        for k := 0 to K do
            P [i, k ].exist := false ; { the default value }
            if P [i – 1, k ].exist then
                P [i, k ].exist := true ;
                P [i, k ].belong := false
            else if $k - S[i] \geq 0$ then
                if P [i – 1, k – S [i ]].exist then
                    P [i, k ].exist := true ;
                    P [i, k ].belong := true
end
```

Figure 5.10 Algorithm *Knapsack*.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k_1=2	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-	-	-	-	-	-	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I	-	I	-	-	-	-	-	-
k_4=6	O	-	O	O	-	O	I	O	O	I	O	I	-	I	I	-	I

Figure 5.11 An example of the table constructed for the knapsack problem. The input consists of four items of sizes 2, 3, 5, and 6. The symbols in the table are the following: "I": a solution containing this item has been found; "O": a solution without this item has been found; "-": no solution has not yet been found. (If the symbol "-" appears in the last line, then there is no solution for a knapsack of this size.)

The method we just used is an instance of a general technique called **dynamic programming**. The essence of dynamic programming is to build large tables with all known previous results. The tables are constructed iteratively. Each entry is computed from a combination of other entries above it or to the left of it in the matrix. The main problem is to organize the construction of the matrix in the most efficient way. Another example of dynamic programming is presented in Section 6.8.

Complexity There are nK entries in the table, and each one is computed in constant time from two other entries. Hence, the total running time is $O(nK)$. If the sizes of the items are not too large, then K cannot be too large and nK is much better than an exponential expression in n. (If K is very large or if the sizes are real numbers, then this approach will not work; we discuss this issue in Chapter 11.) If we are interested only in determining whether a solution exists, then the answer is in $P[n, K]$. If we are interested in finding the actual subset, then we can trace back from the (n, K)th entry, using, for example, the *belong* flag in the knapsack program, and recover the subset in $O(n)$ time.

Comments The dynamic programming approach is effective when the problem can be reduced to several smaller, but not small enough, subproblems. All possible subproblems are computed. We do this computation by maintaining a large matrix. Hence, dynamic programming can work only if the total number of possible subproblems is not too large. Even then, dynamic programming requires building large matrices, and thus it usually requires a large space. (In some cases, as in the program in Fig. 5.10, it is possible to use less space by storing only a small part of the matrix at any moment.) The running times are usually at least quadratic.

5.11 Common Errors

In this section, we briefly mention some common errors in the use of induction to design algorithms. We have already discussed common errors in induction proofs in Section 2.13. All those errors have analogous errors here. For example, forgetting the base case is common. In the case of a recursive procedure, a base case is essential to terminate the recursion. Another common error is to extend a solution for n to a solution of a *special instance* of the problem for $n + 1$, instead of an arbitrary instance.

Changing the hypothesis unintentionally is another common mistake. Here is a typical example of it. A graph $G = (V, E)$ is called **bipartite** if its set of vertices can be partitioned into two subsets such that there is no edge connecting two vertices from the same subset. If the graph is connected and bipartite, then the partition is unique (we omit the proof of this fact).

The Problem Given a connected undirected graph $G = (V, E)$, determine whether it is bipartite and, if it is, partition the vertices accordingly.

A wrong solution: Remove a vertex v and partition the rest of the graph, if possible, by induction. We call the first subset *red*, and the second subset *blue*. If v is connected to only red vertices, add it to the blue subset. If v is connected to only blue vertices, add it to the red subset. If v is connected to vertices from both subsets, then the graph is not bipartite (since the partition is unique).

The main error in this attempted solution, and the one we want to illustrate, is that after we have removed a vertex the graph may not be connected. Hence, the smaller instance of the problem is not the same as the original instance, and induction cannot be used. Had we removed a vertex that does not disconnect the graph, this solution would have been valid. This problem has a better solution, which does not depend on the graph being connected; we leave that solution to the reader (Exercise 7.32). For a similar example and further discussion of this common error, see Section 7.5. A result related to this incorrect algorithm is included in Exercise 5.24.

Changing the hypothesis is sometimes very tempting. If the hypothesis is something of the form "we know how to find such and such," then we are tempted to think that we can find other simple things with the same effort. But we cannot use any such assumption unless it is included specifically in the induction hypothesis. One way to avoid changing the hypothesis unintentionally is to think of it as a black box. Do not make any changes to that black box, unless you are ready to open it (namely, to redefine it explicitly).

5.12 Summary

Several techniques for designing algorithms, all of which are variations of the same approach, were introduced in this chapter. These are by no means all the known methods for designing algorithms. Additional techniques and numerous examples are presented in the following chapters. The best way to learn these techniques is to use them to solve problems. The rest of this book is devoted to precisely that purpose. The principles presented in this chapter are as follows:

- We can use the principle of induction to design algorithms by reducing an instance of a problem to one or more of smaller size. If the reduction can always be achieved, and the base case can be solved, then the algorithm follows by induction. The main idea is to concentrate on reducing a problem, rather than on solving it directly.
- One of the easiest ways to reduce the size of a problem is to eliminate some of its elements. That technique should be the first line of attack. The elimination can take many forms. In addition to simply eliminating elements that clearly do not contribute (as in Section 5.3), it is possible to merge two elements into one, to find elements that can be handled by special (easy) cases, or to introduce a new element that takes on the role of two or more original elements (Section 6.6).
- We can reduce the size of the problem in many ways. Not all reductions, however, lead to the same efficiency. As a result, all possibilities for reductions should be considered. In particular, it is worthwhile to consider different orders for the

induction sequence. We have seen examples where it is better to take the largest element first. Sometimes, it is better to take the smallest element first. We will see examples of starting from the middle (Section 6.2). We also will see examples of induction on trees in which the root is removed first (top down), and examples in which the leaves are removed first (bottom up) (Section 6.4.4).

- One of the most efficient ways to reduce the size of a problem is to divide it into two (or more) equal parts. Divide and conquer works effectively if the problem can be divided such that the output of the subproblems can easily generate the output for the whole problem. Divide-and-conquer algorithms are given in Sections 6.4, 6.5, 8.2, 8.4, 9.4, and 9.5.
- Since a reduction can change only the size of the problem, but not the problem itself, we should look for smaller subproblems that are as independent as possible. For example, the problem of finding some ordering among several items can be reduced to finding (and removing) the item that is first in the order; the relative order of the rest of the items is independent of the first item (see Sections 6.4 and 7.5).
- There is one way, however, to overcome the limitation that the reduced problem must be identical to the original problem: Change the problem statement. This is a very important method that we will use often. Sometimes, it is better to weaken the hypothesis and to arrive at a weaker algorithm, which can be used as a step in the complete algorithm (see Section 6.10).
- Finally, we can use all these techniques together, or in various combinations. For example, we can use the divide-and-conquer approach with strengthening the induction hypothesis, so that the different subproblems become easier to combine (see Section 8.4).

Bibliographic Notes and Further Reading

The method presented in this chapter was developed by the author (Manber [1988]). It is by no means new. The use of induction, and in general mathematical proof techniques, in the algorithms area has its origin in the flowcharts of Goldstine and von Neumann (see von Neumann [1963]), but was first fully developed by Floyd [1967]. Dijkstra [1976], Manna [1980], Gries [1981], and Dershowitz [1983] present methodologies similar to ours to develop programs together with their proof of correctness. Their approach addresses program design in a much more rigorous and detailed fashion than the presentation in this chapter. The use of loop invariants, described in Section 2.12, can be considered, in some sense, to be equivalent to the use of induction in this chapter. Recursion, of course, has been used extensively in algorithm design (see, for example, Burge [1975] and Paull [1988]).

The celebrity problem was first suggested by Aanderaa (see Rosenberg [1973]). It is possible to save an additional $\lfloor \log_2 n \rfloor$ questions by being careful not to repeat, in the verification phase, questions asked during the elimination phase (King and Smith-Thomas [1982]). Strengthening the induction hypothesis is probably a very old trick. Polya [1957] calls this technique **the inventor's paradox** (because it is easier to invent, or prove, something that is stronger). It is also sometimes called **generalization.** Dynamic programming was introduced and formalized by Bellman [1957]. It has numerous applications, and many variations. For a detailed description of dynamic programming see, for example, Dreyfus and Law [1977], or Denardo [1982]. The observation leading to Exercise 5.24 was pointed out to us by Tom Trotter.

Drill Exercises

5.1 Design a divide-and-conquer algorithm for polynomial evaluation. How many additions and multiplications does your algorithm require? Can you think of an advantage this algorithm has over Horner's rule?

5.2 Try to follow the steps of inductive reasoning that were used in Section 5.3 to solve the following maximal induced subgraph problem: Given a graph $G=(V, E)$, we are looking for the maximal induced subgraph G' such that all the degrees in G' are *at most* k (as opposed to "at least" in the problem in Section 5.3). This version is much more difficult than the original version, and the approach taken for the original version does not work here. Discuss why it does not work. (See Chapter 11 for a discussion of this problem for the simple case of $k=0$.)

5.3 Consider algorithm *Mapping* (Fig. 5.3). Is it possible that the set S will become empty at the end of the algorithm? Show an example, or prove that it cannot happen.

5.4 Write the appropriate loop invariant for the first while loop in algorithm *Celebrity* (Fig. 5.4).

5.5 You are given a binary tree T. T is called an **AVL tree** (see also Section 4.3.4) if the balance factors of all its nodes are 0, 1, or -1. Assume that the nodes do not have enough space to store the balance factor. Design an efficient algorithm to solve the following decision problem. Given a tree T, the algorithm should determine whether or not T is an AVL tree. The answer should be only yes or no.

5.6 Modify algorithm *Maximum_Consecutive_Subsequence* (Fig. 5.9) such that it finds the actual subsequence and not only the sum.

5.7 Write a program to recover the solution to a knapsack problem using the *belong* flag.

5.8 In algorithm *Knapsack*, we first checked whether the ith item is unnecessary (by checking $P[i-1, j]$). If there is a solution with the $i-1$ items, we take this solution. We can also make the opposite choice, which is to take the solution with the ith item if it exists (i.e., check $P[i, j-k_i]$ first). Which version do you think will have a better performance? Redraw Fig. 5.11 to reflect this choice.

5.9 A given knapsack problem may have many different solutions. What are the special characteristics of the solution obtained from algorithm *Knapsack*? What separates this solution from all the rest? How does your answer change if the choice is made according to the policy of Exercise 5.7?