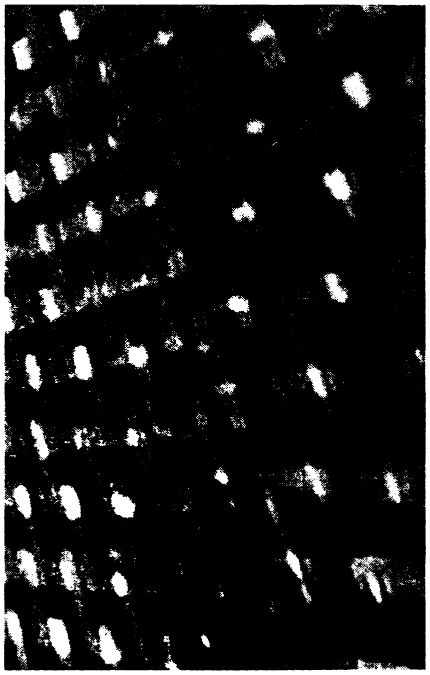
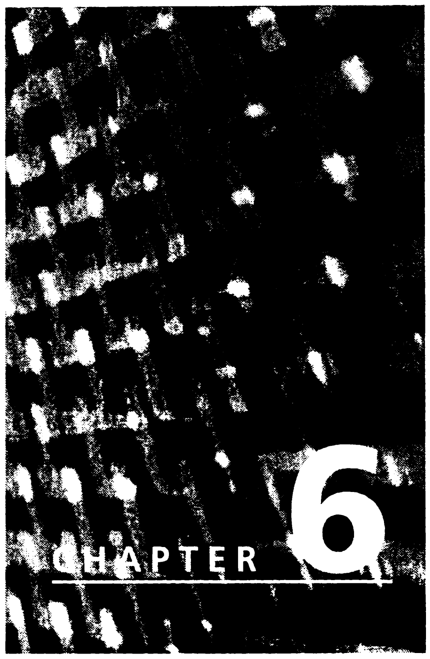


TABLE OF CONTENTS



CHAPTER 6 PROBABILITY AND AVERAGE COMPLEXITY OF ALGORITHMS

When analyzing the complexity of an algorithm we intend to use repeatedly with varying inputs, the average complexity is often as important as the worst-case complexity. In this situation, we would tend to favor one algorithm over another based on average performance. For example, although quicksort has quadratic worst-case performance, we often use it in practice because it performs well on average.

Key Fact

The utility of an algorithm is not always captured by its worst-case performance, in which case it becomes important to determine its average complexity.

The average behavior of an algorithm only makes sense in the presence of a probability distribution on the inputs of size n. The particular probability distribution used will depend on the particular environment in which the algorithm is used and is often taken to be the uniform distribution (that is, all inputs equally likely). The average behavior of an algorithm is often difficult to analyze and requires advanced techniques from probability theory. In this text, we limit ourselves to a discussion of algorithms that we can analyze using the elementary probability techniques discussed in Appendix E [formulas from this appendix are referenced as (E.*i.j*)].

6.1 Expectation and Average Complexity

The definition of the average behavior of an algorithm given in Chapter 2 informally used some concepts from elementary probability theory. In this chapter, we use the formal mathematical concepts discussed in Appendix E to formulate precisely what is meant by the average complexity of an algorithm.

DEFINITION 6.1.1 **Average Complexity** Let $\mathcal{I}_n$ denote the set of all inputs of size n to a given algorithm. For $I \in \mathcal{I}_n$, let $\tau(I)$ denote the number of basic operations performed by the algorithm on input I. Given a probability distribution on $\mathcal{I}_n$, the *average complexity* $A(n)$ of an algorithm is the expected value of τ; that is,

$$A(n) = E[\tau]. \qquad \textbf{(6.1.1)}$$

Key Fact

Average behavior depends on the probability distribution on $\mathcal{I}_n$.

Often there is a natural way to assign a probability distribution on $\mathcal{I}_n$. For example, when analyzing comparison-based sorting algorithms, we typically assume that each of the $n!$ permutations (orderings) of a list of size n is equally likely to be input to the algorithm. The average complexity of any comparison-based sorting algorithm is then the sum of the number of comparisons generated by each of the $n!$ permutations divided by $n!$. In practice, it is not feasible to examine each one of these $n!$ permutations individually, as $n!$ simply grows too fast. Fortunately, there are techniques that enable us to calculate this average without resorting to permutation-by-permutation analysis.

6.2 Techniques for Computing Average Complexity

Computing the average complexity of an algorithm frequently requires deep results from probability theory that are beyond the scope of this text. However, the relatively elementary techniques that follow suffice for our analysis of the average behavior of most of the algorithms we discuss in this text. To compute $A(n)$, we can apply one or some combination of the following formulations for $A(n)$ based on the characteristics of the algorithm we are analyzing. Each formulation assumes that $\mathcal{I}_n$ is finite.

Formulation I

$$A(n) = E[\tau] = \sum_{I \in \mathcal{I}_n} \tau(I)P(I). \qquad \textbf{(6.2.1)}$$

Formula (6.2.1) is simply Definition (E.3.5) of the expected value $E[X]$ with $X = \tau$, $s = I$, and $S = \mathcal{I}_n$. Because it is usually too cumbersome to examine each element in $\mathcal{I}_n$ individually, this formulation is rarely used directly.

Formulation II Let p_i denote the probability that the algorithm performs exactly i basic operations; that is, $p_i = P(\tau = i)$. Then

$$A(n) = E[\tau] = \sum_{i=1}^{W(n)} ip_i. \qquad \textbf{(6.2.2)}$$

Formula (6.2.2) is a special case of Formula (E.3.7), with $X = \tau$ and $x = i$. Note that Formulas (6.2.1) and (6.2.2) are the same as the formulas (2.5.3) and (2.5.4), respectively, given in Chapter 2.

Formulation III Let q_i denote the probability that the algorithm performs *at least i* basic operations; that is, $q_i = P\{\tau \geq i\}$. Then

$$A(n) = E[\tau] = \sum_{i=1}^{W(n)} q_i. \qquad \textbf{(6.2.3)}$$

Formula (6.2.3) follows easily from Formula (6.2.2) (see Exercise 6.19)

Formulation IV Given that $\tau = \sum_{i=1}^{k} \tau_i$, then

$$A(n) = E[\tau] = \sum_{i=1}^{k} E[\tau_i]. \qquad \textbf{(6.2.4)}$$

Formula (6.2.4) is Formula (E.3.12) with $X = \tau$ and $X_i = \tau_i$.

Formulation V Given that Y is a random variable defined in $\mathcal{I}_n$, then

$$A(n) = E[\tau] = \sum_{y} E[\tau \mid Y = y]P(Y = y), \qquad \textbf{(6.2.5)}$$

where the summation is taken over all y such that $P(Y = y) > 0$.

Formula (6.2.5) is Formula (E.4.6) with $X = \tau$.

Key Fact

To determine which of the formulations of *A(n)* are best suited to a given algorithm, we usually use one or more of the following three techniques:

1. **Partitioning the algorithm**
2. **Partitioning the input space**
3. **Recursion**

Partitioning the algorithm refers to breaking down the steps in the algorithm into k stages. If τ_i denotes the random variable on $\mathcal{I}_n$ mapping each input onto the number of basic operations performed in stage i, $i = 1, \ldots, k$, then $\tau = \sum_{i=1}^{k} \tau_i$, and Formulation IV applies. The partitioning is useful when each τ_i can be effectively computed when $i = 1, \ldots, k$.

Partitioning the input space is appropriate when the inputs of size n to the algorithm can be partitioned into disjoint sets by using a naturally defined random variable Y and where the quantities $E[\tau \mid Y = y]$, $P(Y = y)$ occurring in

Formula (6.2.5) are effectively computable. Typically, Y involves mapping an input to some sort of integer constraint, such as the position in a list of size n where a maximum element occurs. Then $A(n)$ is computed using (6.2.5).

Often we can find a recurrence relation expressing $A(n)$ in terms of one or more of the values $A(m)$ with $m < n$. Of course, if the algorithm itself is written recursively, then the recurrence relation is usually easy to find. In general, finding a recurrence relation for $A(n)$ involves using the techniques of partitioning the algorithm or the input space.

6.3 Average Complexity of *LinearSearch*

In Chapter 2, we informally showed that the average complexity of *LinearSearch* is $n/2 + 1/2$, under the assumptions that the search element X is in the list $L[0{:}n - 1]$ and the list consists of distinct elements, with each element equally likely to be X. We now calculate the average complexity of *LinearSearch* under these assumptions, but we assume that the search element X occurs in the list with probability p. Using the notation for conditional probability given in the Appendix E, this assumption becomes

$$P(X = L[i] \mid X \text{ is in the list } L) = 1/n, \quad i = 0, \ldots, n - 1.$$

We will calculate $A(n)$ using Formula (6.2.2). For $1 \le i \le n$, p_i is equal to the probability that X is the i^{th} list element $L[i - 1]$. Hence, from Formula (E.2.3)we have

$$\begin{aligned} p_i &= P(X = L[i-1] \mid X \text{ is in the list } L)P(X \text{ is in the list } L) \\ &= \left(\frac{1}{n}\right)p = \frac{p}{n}, \quad i = 1, \ldots, n - 1. \end{aligned}$$

LinearSearch performs n comparisons when $X = L[n - 1]$, or when X is not in the list L. Thus, $p_n = p/n + 1 - p$. Substituting these values of p_i into Formula (6.2.2) yields

$$\begin{aligned} A(n) &= 1\left(\frac{p}{n}\right) + 2\left(\frac{p}{n}\right) + \cdots + (n-1)\left(\frac{p}{n}\right) + n\left(\frac{p}{n} + 1 - p\right) \\ &= (1 + 2 + \cdots + n)\left(\frac{p}{n}\right) + n(1 - p) \qquad \textbf{(6.3.1)} \\ &= \left(\frac{n(n+1)}{2}\right)\left(\frac{p}{n}\right) + n(1-p) = \left(1 - \frac{p}{2}\right)n + \frac{p}{2}. \end{aligned}$$

REMARK

If we set $p = 1$ in Formula (6.3.1), we obtain the formula $A(n) = (n + 1)/2$ derived in Chapter 2 for the average complexity of *LinearSearch* when X is assumed to be in the list.

6.3.1 Average Complexity of *LinearSearch* with Repeated Elements

We now determine the average behavior $A(n,m)$ of *LinearSearch* for lists $L[0{:}n - 1]$ of size n having m distinct elements drawn from a given fixed set $S = \{s_1,s_2, \ldots ,s_m\}$, where the search element $X \in S$. For purposes of computing $A(n,m)$, we assume that $L[i]$ has an equal probability $1/m$ of being any element in S, $i = 0,1, \ldots ,n - 1$. Hence, the probability that X does not occur in position i is $(m - 1)/m$, $i = 0,1, \ldots$, $n - 1$. It follows that the probability that X is not in the list is $((m - 1)/m)^n$ and the probability that X is in the list is $1 - ((m - 1)/m)^n$.

Clearly, the p_i in Formula (6.2.2), $i = 1,2, \ldots ,n - 1$, is the probability that the first occurrence of the search element X is in position i, and p_n is the probability that X does not occur in the first $n - 1$ positions. Thus, we have

$$p_i = \begin{cases} ((m - 1)/m)^{i-1}(1/m) & \text{if } 1 \le i \le n - 1, \\ ((m - 1)/m)^{n-1} & \text{if } i = n. \end{cases} \qquad \textbf{(6.3.2)}$$

Substituting Formula (6.3.2) into a suitably interpreted Formula (6.2.2) gives us

$$A(n,m) = \sum_{i=1}^{W(n)} ip_i = \left(\sum_{i=1}^{n-1} i\left(\frac{m-1}{m}\right)^{i-1}\left(\frac{1}{m}\right)\right) + \left(\frac{m-1}{m}\right)^{n-1}. \qquad \textbf{(6.3.3)}$$

We now employ Formula (B.2.11) from Appendix B, which states that

$$\sum_{i=1}^{n-1} ix^{i-1} = \frac{(n - 1)x^n - nx^{n-1} + 1}{(1 - x)^2}.$$

By replacing x by $(m - 1)/m$ in this formula and substituting the result in Formula (6.3.3), we obtain

$$\begin{aligned} A(n,m) &= \left[(1/m)\frac{(n - 1)((m - 1)/m)^n - n((m - 1)/m)^{n-1} + 1}{m^{-2}}\right] + \left(\frac{m-1}{m}\right)^{n-1} \\ &= m\left(1 - \left(\frac{m-1}{m}\right)^n\right) + \left(\frac{m-1}{m}\right)^{n-1} \end{aligned}$$

If we hold m constant, then $((m-1)/m)^n$ approaches zero as n approaches ∞, so that $A(n,m) \sim m$. In particular, we have $A(n,m_{\text{fixed}}) \in \Theta(m_{\text{fixed}}) = \Theta(1)$, and thus $A(n,m_{\text{fixed}})$ has *constant* order. This behavior is very different from the linear average behavior $A(n)$ of *LinearSearch* for lists of *distinct* elements.

6.4 Average Complexity of *InsertionSort*

Because *InsertionSort* is a comparison-based algorithm, we can assume without loss of generality that inputs to *InsertionSort* are permutations of $\{1,2, \ldots ,n\}$. We also assume that each permutation is equally likely to be the input to *InsertionSort*. Unlike our analysis of *LinearSearch*, we cannot compute $A(n) = E[\tau]$ directly by applying Formula (6.2.2). Instead, we partition the algorithm *InsertionSort* into $n-1$ stages. The ith stage consists of inserting the element $L[i]$ into its proper position in the sublist $L[0{:}i-1]$, where the latter sublist has already been sorted by the algorithm. Let τ_i denote the number of comparisons performed in stage i, so that $\tau = \tau_1 + \cdots + \tau_{n-1}$ and, by Formula (6.2.4),

$$A(n) = E[\tau] = E[\tau_1] + E[\tau_2] + \cdots + E[\tau_{n-1}]. \qquad \textbf{(6.4.1)}$$

We now calculate $E[\tau_i]$, $i = 1, \ldots , n-1$, using Formula (6.2.2). We have

$$E[\tau_i] = \sum_{j=1}^{i} jP(\tau_i = j). \qquad \textbf{(6.4.2)}$$

Our assumption of a uniform distribution on the input space implies that any position in $L[0{:}i]$ is equally likely to be the correct position for $L[i]$. Thus, the probability that $L[i]$ is the jth largest of the elements in $L[0{:}i]$ is equal to $1/(i+1)$. If $L[i]$ is the jth largest, where $j \leq i$, then exactly j comparisons are performed by *InsertionSort* when placing $L[i]$ in its correct position. If $L[i]$ is the $(i+1)$st largest (that is, the smallest), then exactly i comparisons are performed by *InsertionSort* when placing $L[i]$ in its correct position. It follows that

$$\begin{aligned} P(\tau_i = j) &= \frac{1}{i+1}, \quad j = 1, \ldots , i-1, \\ P(\tau_i = i) &= \frac{2}{i+1}, \quad i = 1, \ldots , n-1. \end{aligned} \qquad \textbf{(6.4.3)}$$

Substituting Formula (6.4.3) into (6.4.2) and simplifying yields

$$E[\tau_i] = \left(\sum_{j=1}^{i} \frac{j}{i+1}\right) + \frac{i}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}, \quad i = 1, \ldots , n-1. \qquad \textbf{(6.4.4)}$$

Substituting Formula (6.4.4) into (6.4.1), we have

$$\begin{aligned} A(n) &= \sum_{i=1}^{n-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) \\ &= (n-1)\frac{n}{4} + (n-1) - \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right) \\ &= (n-1)\frac{n}{4} + n - H(n), \end{aligned}$$

where $H(n)$ is the harmonic series $1 + 1/2 + \cdots + 1/n \sim \ln n$. In particular, $A(n) \in \Theta(n^2)$. Note that the average complexity of *InsertionSort* is about half that of its worst-case complexity, since the highest-order terms in the expressions for these complexities are $n^2/4$ and $n^2/2$, respectively.

6.5 Average Complexity of *QuickSort*

As with *InsertionSort*, we assume that input lists $L[0{:}n-1]$ to *QuickSort* are all permutations of $\{1,2, \ldots n\}$, with each permutation being equally likely. We partition *QuickSort* into two stages, where the first stage is the call to *Partition* and the second stage is the two recursive calls with input lists consisting of the sublists on either side of the proper placement of the pivot element $L[0]$. Thus, $\tau = \tau_1 + \tau_2$, where τ_1 is the (constant) number $n + 1$ of comparisons performed by *Partition* and τ_2 is the number of comparisons performed by the recursive calls. Hence,

$$A(n) = E[\tau] = E[\tau_1] + E[\tau_2] = n + 1 + E[\tau_2]. \qquad \textbf{(6.5.1)}$$

We compute $E[\tau_2]$ using Formulation V by introducing the random variable Y that maps an input list $L[0{:}n-1]$ into the proper place for $L[0]$ as determined by a call to *Partition*. The uniform distribution assumption on the input space implies that

$$P(Y = i) = \frac{1}{n}, \quad i = 0, \ldots, n-1. \qquad \textbf{(6.5.2)}$$

If $Y = i$, then the recursive calls to *QuickSort* are with the two sublists $L[0{:}i-1]$ and $L[i+1{:}n-1]$. Our assumption of a uniform distribution on the input space implies that the expected number of comparisons performed by *QuickSort* on the sublists $L[0{:}i-1]$ and $L[i+1{:}n-1]$ is given by $A(i)$ and $A(n-i-1)$, respectively. Hence,

$$E[\tau_2 | Y = i] = A(i) + A(n-i-1), \quad i = 0, \ldots, n-1. \qquad \textbf{(6.5.3)}$$

Combining Formulas (6.2.5), (6.5.1), (6.5.2), and (6.5.3), we have

$$\begin{aligned} A(n) &= (n+1) + \sum_{i=0}^{n-1} E[\tau_2 | Y = i] P(Y = i) \\ &= (n+1) + \sum_{i=0}^{n-1} (A(i) + A(n-i-1))\left(\frac{1}{n}\right) \\ &= (n+1) + \frac{2}{n}(A(0) + A(1) + \cdots + A(n-1)), \end{aligned} \qquad \textbf{(6.5.4)}$$

init. cond. $A(0) = A(1) = 0$.

Recurrence relation (6.5.4) is an example of what is sometimes referred to as a *full history* recurrence relation, because it relates $A(n)$ to *all* of the previous values $A(i)$, $0 \le i \le n-1$. Fortunately, with some algebraic manipulation, we can transform (6.5.4) into a simpler recurrence relation as follows. The trick is to first observe that

$$nA(n) = n(n+1) + 2(A(0) + A(1) + \cdots + A(n-2) + A(n-1)). \qquad \textbf{(6.5.5)}$$

Substituting $n-1$ for n in Formula (6.5.5) yields

$$(n-1)A(n-1) = n(n-1) + 2(A(0) + A(1) + \cdots + A(n-2)). \qquad \textbf{(6.5.6)}$$

By subtracting Formula (6.5.6) from (6.5.5), we obtain

$$nA(n) - (n-1)A(n-1) = 2n + 2A(n-1). \qquad \textbf{(6.5.7)}$$

Rewriting Formula (6.5.7) by moving the term involving $A(n-1)$ to the right-hand side and dividing both sides by $n(n+1)$ yields

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2}{n+1}. \qquad \textbf{(6.5.8)}$$

Letting $t(n) = A(n)/(n+1)$ changes Formula (6.5.8) to

$$t(n) = t(n-1) + \frac{2}{n+1}. \qquad \textbf{(6.5.9)}$$

Recurrence relation (6.5.9) directly unwinds to yield

$$\begin{aligned} t(n) &= 2\left(\frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{(n+1)}\right) \\ &= 2H(n+1) - 3, \end{aligned} \qquad \textbf{(6.5.10)}$$

where $H(n)$ is the harmonic series. Thus, $t(n) \sim 2\ln n$, so that the average complexity $A(n)$ of *QuickSort* satisfies

$$A(n) \sim 2n\ln n. \qquad \textbf{(6.5.11)}$$

In particular, *QuickSort* exhibits $O(n\log n)$ average behavior, which is order optimal for a comparison-based sorting algorithm.

6.6 Average Complexity of *MaxMin2*

We analyze the algorithm *MaxMin2* given in Chapter 2 as our third example. For convenience, we repeat the pseudocode for *MaxMin2* here.

```
procedure MaxMin2(L[0:n − 1], MaxValue, MinValue)
Input:    L[0:n − 1] (a list of size n)
Output:   MaxValue, MinValue (maximum and minimum values occurring in L[0:n − 1])
    MaxValue ← L[0]
    MinValue ← L[0]
    for i ← 1 to n − 1 do
        if L[i] > MaxValue then
            MaxValue ← L[i]
        else
            if L[i] < MinValue then
                MinValue ← L[i]
            endif
        endif
    endfor
end MaxMin2
```

As observed in Chapter 2, the best-case complexity of *MaxMin2* is $n - 1$, and its worst-case complexity is $2(n - 1)$. Recall that the algorithm *MaxMin3* has best-case, worst-case, and average complexities all equal to $\lceil 3n/2 \rceil - 2$. Thus, to complete our comparison of *MaxMin2* and *MaxMin3*, we need to compute the average complexity of *MaxMin2*. Unfortunately, it turns out that the average complexity of *MaxMin2* is closer to its worst-case complexity than to its best-case complexity.

When analyzing the average complexity of *MaxMin2*, we again assume that the inputs are permutations of $\{1,2, \ldots ,n\}$ and that each permutation is equally likely. Observe that $n - 1$ comparisons involving *MaxValue* are performed for any input permutation. An additional comparison involving *MinValue* is performed for each iteration of the loop in which *MaxValue* is not updated. If we let D

denote the random variable that maps the input permutation to the number of times that *MaxValue* is updated, then we have

$$\tau = n - 1 + (n - 1 - D) = 2n - 2 - D. \qquad \textbf{(6.6.1)}$$

Using Formula (6.6.1), the average complexity of *MaxMin2* is given by

$$A(n) = E[\tau] = 2n - 2 - E[D]. \qquad \textbf{(6.6.2)}$$

We compute the expected number of updates $E[D]$ by partitioning the input space using the random variable M that maps an input permutation π of $\{1, \ldots ,n\}$ to $\pi(n)$. Applying formula (6.2.5)), we obtain

$$E[D] = \sum_{i=1}^{n} E[D \mid M = i]P(M = i). \qquad \textbf{(6.6.3)}$$

The assumption of a uniform distribution on the input space implies that the maximum element is equally likely to occur in any position. Thus, we have

$$P(M = i) = \frac{1}{n}, \quad i = 1, \ldots , n. \qquad \textbf{(6.6.4)}$$

Analogous with the notation $A(n)$ used for $E[\tau]$,the notation $\alpha(n)$ is used for $E[D]$ to facilitate the expression of a recurrence relation for $E[D]$. Clearly, we have

$$E[D \mid M = n] = \alpha(n - 1) + 1, \qquad \textbf{(6.6.5)}$$

because permutations of $\{1, \ldots , n\}$ with $\pi(n) = n$ are in one-to-one correspondence with permutations of $\{1, \ldots , n - 1\}$, and the maximum is updated for such permutations π exactly one more time than it was updated on $\pi(1), \ldots , \pi(n - 1)$. On the other hand, for permutations π such that $\pi(n) = i \neq n$, we have

$$E[D|M = i] = \alpha(n - 1), \quad i \in \{1, \ldots , n - 1\}, \qquad \textbf{(6.6.6)}$$

because the maximum is not updated on $\pi(n)$, and (for a fixed i) such permutations are again in one-to-one correspondence with permutations of $\{1, \ldots ,n - 1\}$. Hence, combining Formulas (6.6.3) through (6.6.6), we have

$$\begin{aligned} \alpha(n) &= \left(\frac{1}{n}\right)(\alpha(n - 1) + 1) + ((n - 1)/n)\alpha(n - 1) \\ &= \alpha(n - 1) + 1/n, \quad \textbf{init. cond. } \alpha(1) = 0. \end{aligned} \qquad \textbf{(6.6.7)}$$

Recurrence relation (6.6.7) unwinds directly to yield

$$\alpha(n) = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = H(n) - 1, \qquad \textbf{(6.6.8)}$$

where $H(n)$ is the harmonic series. By combining Formulas (6.6.2) and (6.6.8), we obtain the following formula for the average complexity $A(n)$ of *MaxMin2*:

$$A(n) = 2n - 2 - \alpha(n) = 2n - H(n) - 1. \qquad \textbf{(6.6.9)}$$

Because $H(n)$ is approximately equal to $\ln n$, $A(n)$ is approximately equal to $2n - \ln n - 1$. In particular, $A(n) \sim W(n)$ for *MaxMin2*.

6.7 Average Complexity of *BinarySearch* and *SearchBinSrchTree*

To compute the average complexity $A(n)$ of the algorithm *BinarySearch* given in Chapter 2, we choose the three-branch comparison of the **do case** statement as the basic operation. Let p denote the probability that the search element X is on the list. Given that X is on the list $L[0{:}n-1]$, we assume that it is equally likely to occur in any of the n positions. Given that the search element is not on the list, we assume that it is equally likely to occur in any of the following $n + 1$ intervals:

$$X < L[0], L[0] < X < L[1], \ldots, L[n-2] < X < L[n-1], X > L[n-1].$$

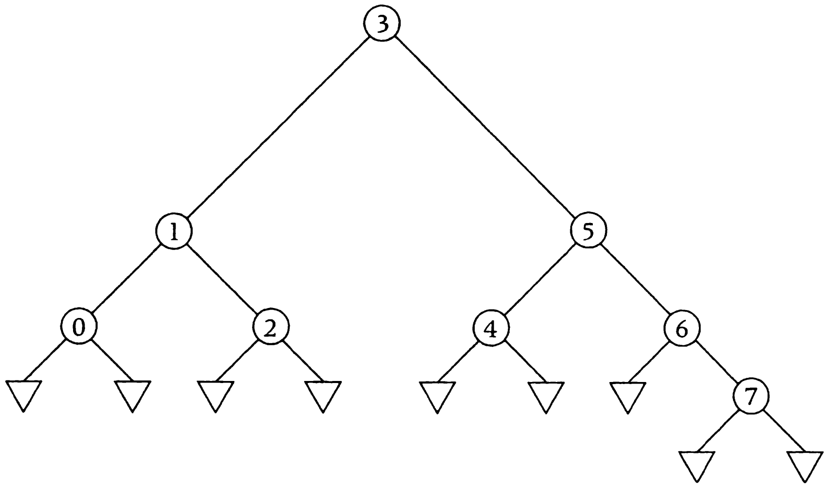
In Figure 6.1, we show the implicit search tree T for *BinarySearch* with an input list $L[0{:}7]$ of size 8. Note that T is a 2-tree having n internal nodes (squares) corresponding to the elements of $L[0{:}n-1]$ and $n + 1$ leaf nodes (triangles) corresponding to these $n + 1$ intervals. Thus, the probability that X occurs on the list and is equal to any given element $L[i]$ is p/n, $i = 0, \ldots, n-1$, and the probability that X occurs in any one of the $n + 1$ intervals is $(1-p)/(n+1)$.

Given a search element X, *BinarySearch* follows a path in the tree from the root to a node of T corresponding to X, or to a leaf node if X is not in the list. Thus, the average complexity $A(n)$ is given by

$$A(n) = \left(\frac{p}{n}\right)(IPL(T) + n) + \left(\frac{1-p}{n+1}\right)LPL(T). \qquad \textbf{(6.7.1)}$$

FIGURE 6.1

Implicit search tree for *BinarySearch* with an input list $L[0{:}7]$ of size 8. The index of the list element that is compared to the search element is shown inside each node. Leaf nodes correspond to unsuccessful searches.



Employing Proposition 4.2.7 and substituting $LPL(T) - 2n$ for $IPL(T)$ in Formula (6.7.1), we obtain

$$\begin{aligned} A(n) &= \left(\frac{p}{n}\right)(LPL(T) - n) + \left(\frac{1-p}{n+1}\right)LPL(T) \\ &= \left(\frac{p}{n} + \frac{1-p}{n+1}\right)LPL(T) - p. \end{aligned} \tag{6.7.2}$$

The implicit search tree for *BinarySearch* is a 2-tree that is full at the second-deepest level (see Exercise 6.23). Hence, by Theorem 4.2.8,

$$\begin{aligned} LPL(T) &= \lfloor L\log_2 L \rfloor + 2(L - 2^{\lfloor \log_2 L \rfloor}) \\ &\geq \lceil L\log_2 L \rceil = \lceil (n+1)\log_2(n+1) \rceil. \end{aligned} \tag{6.7.3}$$

Substituting the expression for $LPL(T)$ given in Formula (6.7.3) into (6.7.2), we obtain

$$A(n) \geq \left(\frac{p}{n} + \frac{1-p}{n+1}\right)\lceil (n+1)\log_2(n+1) \rceil - p. \tag{6.7.4}$$

The lower-bound estimate for $A(n)$ in Formula (6.7.4) looks somewhat formidable, but it is easily seen to be asymptotic to the worst-case complexity; that is, $A(n) \sim W(n) = \lceil \log_2(n+1) \rceil$. Furthermore, suppose we restrict attention to successful search, so that $p = 1$ in Formula (6.7.4). Then we have $A(n) \geq (1/n)\lceil (n+1)\log_2(n+1) \rceil - 1$. In other words, for large n, the average complexity of *BinarySearch* for successful searching is no more than 1 less than its worst-case complexity.

Another algorithm whose average behavior can be calculated using leaf path length is *SearchBinSrchTree* given in Chapter 4. The average complexity $A(n)$ of *SearchBinSrchTree* depends on the probability distribution on the set of input trees, as well as probabilities associated with the identifiers. The analysis of the average complexity $A(n)$ of *SearchBinSrchTree* over the set of all binary search trees is difficult. In any case, it is probably more interesting to analyze the average behavior $A(T,n)$ of *SearchBinSrchTree* for a given fixed tree T and a given set of probabilities associated with the identifiers.

Suppose, then, that T is a fixed input search tree (pointed at by *Root*), whose internal nodes correspond to a fixed set of n identifiers, or keys $(K_0, K_1, \ldots, K_{n-1})$. Note that T contains $n + 1$ implicit leaf nodes, corresponding to the $n + 1$ intervals

$$I_0{:}X < K_0, I_1{:}K_0 < X < K_1, \ldots, I_{n-1}{:}K_{n-2} < X < K_{n-1}, I_n{:}X > K_{n-1}.$$

Given T, the average behavior $A(T,n)$ of *SearchBinSrchTree* (either version) depends on the probabilities $p_0,p_1, \ldots ,p_{n-1}$ assigned to the internal nodes, and the probabilities $q_0,q_1, \ldots ,q_n$ assigned to the implicit leaf nodes of the search tree. Assume now that all the p_i's are equal, and all the q_i's are equal. If p denotes the probability that X is one of the n keys $K_0, \ldots ,K_{n-1}$, then $p_i = p/n$ for all $i \in \{0, \ldots, n-1\}$ and $q_j = (1-p)/(n+1)$ for all $j \in \{0, \ldots ,n\}$. Under these assumptions, we can show that

$$A(T,n) = LPL(T)\frac{(1 + p/n)}{n+1} - p. \qquad \textbf{(6.7.5)}$$

Applying Theorem 4.2.8 to Formula (6.7.5) yields the following theorem:

Theorem 6.7.1 Let T be any binary search tree (with external implicit leaf nodes added) for keys $K_0, \ldots, K_{n-1}$, where $p_i = p/n$ for each $i \in \{0, \ldots, n-1\}$, and $q_j = (1-p)/(n+1)$ for each $j \in \{0, \ldots ,n\}$. Then,

$$\begin{aligned} A(T,n) &\geq [(n+1)\lfloor \log_2(n+1)\rfloor + 2(n+1-2^{\lfloor \log_2(n+1)\rfloor})]\left(\frac{1+p/n}{n+1}\right) - p \\ &\geq \lceil \log_2(n+1)\rceil\left(1+\frac{p}{n}\right) - p \in \Omega(\log n). \end{aligned} \qquad \textbf{(6.7.6)}$$

Moreover, the first inequality in Formula (6.7.6) is an equality if and only if T is a 2-tree that is full at the second deepest level. ■

An important and natural problem is to determine an optimal search tree T in the sense that T minimizes $A(T,n)$ over all binary search trees containing n identifiers. Theorem 6.7.1 completely characterizes optimal binary search trees for the case where $p_i = p/n$ for each $i \in \{0, \dots, n-1\}$ and $q_j = (1-p)/(n+1)$ for each $j \in \{0, \dots, n\}$. That is, a binary search tree T is optimal for these probabilities if and only if it is full at the second deepest level. In Chapter 9, we will use the technique of dynamic programming to solve the more difficult problem of finding an optimal search tree for general probabilities p_i and q_j.

REMARK

A binary search tree without the implicit external leaf nodes added is full at the second deepest level if and only if the associated 2-tree with the external leaf nodes added is full at the second deepest level. Thus, Theorem 6.7.1 could be restated for binary search trees without the external leaf nodes added.

6.8 Searching a Link-Ordered List

Suppose we have a list $L[0{:}n-1]$ of records having multiple fields, and we wish to maintain a sorting of L with respect to more than one of these fields (keys). Maintaining multiple sortings is usually done using auxiliary tag arrays or link arrays. For a given key field, a tag array $Tag[0{:}n-1]$ has the property that the sorting of L is given by

$$L[Tag[0]], L[Tag[1]], \dots, L[Tag[n-1]].$$

In the linked-list implementation, we assume that a variable *Head* contains the index of the smallest element in L with respect to the given key field. Then the link array $Link[0{:}n-1]$ has the property that the sorting of L is given by

$$L[Head], L[Link[Head]], L[Link[Link[Head]]], \dots, L[Link^{n-1}[Head]],$$

where $Link^m$ denotes the m-fold composition of *Link* ($Link^0$ is the identity function). $Link[i] = -1$ indicates the end of the list. In Figure 6.2, we show a sample list $L[0{:}11]$ and its associated array $Link[0{:}11]$.

FIGURE 6.2

A list $L[0:11]$ maintained in order using an auxiliary array $Link[0:11]$

i	0	1	2	3	4	5	6	7	8	9	10	11
L	17	11	44	23	7	4	14	55	21	56	9	15
$Link$	8	6	7	2	10	4	11	9	3	−1	1	0

$Head = 5$

We now consider the problem of searching for an occurrence of a given element in L having a given key value X. The tag array $Tag[0:n-1]$ facilitates efficient searching using a binary search. Using the link array $Link[0:n-1]$, we cannot efficiently use a binary search because we have no way to directly determine the middle list element. Accessing the middle element $L[Link^{(n/2)-1}[Head]]$ requires $(n/2)-1$ accesses of the array *Link*. In fact, it can be shown that *any* algorithm for searching L requires n accesses to the array *Link* in the worst case. We now describe an algorithm *ProbeLinSrch* for searching L whose average complexity belongs to $O(\sqrt{n})$.

The algorithm *ProbeLinSrch* uses an algorithm *LinkOrdLinSrch* for searching a link-ordered list. *LinkOrdLinSrch* is a variant of the straightforward analog of a linear search for link-ordered lists. *LinkOrdLinSrch* is based on the fact that if we use the array *Link* to make a search for the occurrence of X in L in sorted order, then we can terminate the search whenever we compare X to a list element that is larger than X. *LinkOrdLinSrch* returns the index of an occurrence of X in the list L, or it returns -1 if X is not in list L. For convenience in the following discussion, we identify the record $L[i]$ with its key field.

```
function LinkOrdLinSrch(L[0:n − 1], Link[0:n − 1], Head, X)
Input:   L[0:n − 1] (array of n list elements)
         Link[0:n − 1] (array pointing to sorted order of L[0:n − 1])
         Head (index of smallest element in L)
         X (search element)
Output:  Returns index of occurrence of X in L, or −1 if X is not in L
   i ← Head
   while X > L[i] do
      i ← Link[i]
      if i = −1 then return(−1) endif
   endwhile
   if X = L[i] then
      return(i)
   else
      return (−1)
   endif
end LinkOrdLinSrch
```

To analyze *LinkOrdLinSrch*, we note that the number of updates for the index variable i in *LinkOrdLinSrch* is at most 1 less than the number of comparisons of X with a list element in L. Thus, we choose comparisons of X to list elements as our basic operation. Clearly, the best-case and worst-case complexities of *LinkOrdLinSrch* are $B(n) = 2$ and $W(n) = n + 1$, respectively.

To compute the average complexity of *LinkOrdLinSrch*, we make the usual assumption that each list element is distinct. We also assume that each element is equally likely to be X and that X falls into each of the $n + 1$ "gaps": $X < L[Head]$, $L[Head] < X < L[Link[Head]]$, ... , $L[Link^{n-2}[Head]] < X < L[Link^{n-1}[Head]]$, $X > L[Link^{n-1}[Head]]$ with equal probability. Let $A_s(n)$ and $A_u(n)$ denote the expected number of comparisons for successful and unsuccessful searches, respectively. Then, if p denotes the probability that X occurs on the list, we have

$$A(n) = pA_s(n) + (1 - p)A_u(n). \qquad \textbf{(6.8.1)}$$

We can derive Formula (6.8.1) directly from the definition of conditional expectation, and it is actually a special case of (E.4.5)).

To compute $A_s(n)$, we assume that for each i between 1 and n, the probability that X occurs in position i is $1/n$, given that X is in the list. In addition, $i + 1$ comparisons of X to a list element are performed when $X = L[Link^{i-1}[Head]]$, $1 \leq i \leq n$. Hence,

$$\begin{aligned} A_s(n) &= \frac{2 + 3 + \cdots + n + n + 1}{n} = \frac{1 + 2 + \cdots + n}{n} + 1 \\ &= \frac{n+1}{2} + 1. \end{aligned} \qquad \textbf{(6.8.2)}$$

To compute $A_u(n)$, we assume that for each i between 1 and $n + 1$, the probability that X occurs in the ith gap is $1/(n + 1)$, given that X is not in the list. In addition, notice that $i + 1$ comparisons of X to a list element are performed when X is in the ith gap, $1 \leq i \leq n$, and n comparisons are made when $X > L[Link^{n-1}[Head]]$. Hence,

$$\begin{aligned} A_u(n) &= \frac{2 + 3 + \cdots + n + 1}{n + 1} + 1 - \frac{1}{n+1} \\ &= \frac{n+2}{2} + 1 - \frac{2}{n+1}. \end{aligned} \qquad \textbf{(6.8.3)}$$

Using Formulas (6.8.2) and (6.8.3), we have $A_s(n) \sim n/2$ and $A_u(n) \sim n/2$. Thus, it follows from Formula (6.8.1) that $A(n) \sim n/2$.

ProbeLinSrch is a variant of *LinkOrdLinSrch* in which we first scan the list elements $L[0], \ldots, L[\lfloor\sqrt{n}\rfloor - 1]$ for a better starting point than *Head*. Note that the index i is a feasible starting point for our search if $X \geq L[i]$. If there is no feasible starting point in index positions 0, ... , $\lfloor\sqrt{n}\rfloor - 1$, then *ProbeLinSrch* invokes *LinkOrdLinSrch* with *Probe* = *Head*. Otherwise, *ProbeLinSrch* invokes *LinkOrdLinSrch* with *Probe* equal to the best feasible starting point found.

To illustrate, consider the lists $L[0{:}11]$ and $Link[0{:}11]$ shown in Figure 6.2. Given a search element X, we first determine the best feasible starting point *Probe* among $L[0]$, $L[1]$, $L[2] = L[\lfloor\sqrt{12}\rfloor - 1]$. For example, when X is 23, 3, or 46, then *Probe* is 0, 5, or 2, respectively.

```
function ProbeLinSrch(L[0:n − 1], Link[0:n − 1], Head, X)
Input:   L[0:n − 1] (an array of list elements)
         Link[0:n − 1] (an array pointing to sorted order of L[0:n − 1])
         Head (index of smallest element in L)
         X (search element)
Output:  returns index of occurrence of X in L, or −1 if X is not in L
   Probe ← Head
   Max ← L[Head]
         // look for the best feasible starting point (if any) between 0 and $\lfloor\sqrt{n}\rfloor - 1$
   for i = 0 to $\lfloor\sqrt{n}\rfloor - 1$ do
      Temp ← L[i]
      if Max < Temp .and. Temp ≤ X then
         Probe ← i
         Max ← Temp
      endif
   endfor
   return(LinkOrdLinSrch(L[0:n − 1], Link[0:n − 1], Probe, X)
end ProbeLinSrch
```

Clearly, the best-case and worst-case complexities of *ProbeLinSrch* are given by $B(n) = \lfloor\sqrt{n}\rfloor + 2$ and $W(n) = n + 1 + \lfloor\sqrt{n}\rfloor$, respectively. To determine the average complexity $A(n)$, it is useful to consider a generalization of *ProbeLinSrch* in which the first k list elements $L[0], \ldots, L[k-1]$ are probed for a feasible starting point. In the generalized version, k is an additional input parameter to *ProbeLinSrch*, and the ordinary version of *ProbeLinSrch* corresponds to the special case where $k = \lfloor\sqrt{n}\rfloor$. In the following discussion, we continue to refer to the generalized version as simply *ProbeLinSrch*.

We assume that the list elements are distinct and that the search element X is on the list. Further, we assume that X is equally likely to be any of the list elements. Let $x_i = L[Link^{i-1}[Head]]$ denote the i^{th}-smallest list element, $i = 1, \ldots, n$.

Let Y denote the random variable that maps the input $(L[0{:}n-1],X)$ onto m where $X = x_m$. By Formula (6.2.5) we have

$$A(n) = \sum_{m=1}^{n} E[\tau|Y=m]P(Y=m) = \frac{1}{n}\sum_{m=1}^{n} E[\tau|Y=m]. \qquad \textbf{(6.8.4)}$$

To compute $E[\tau|Y=m]$, note that *ProbeLinSrch* has three stages: Stage 1 consists of the **for** loop that determines the probe element, stage 2 consists of the **while** loop in *LinkOrdLinSrch*, and stage 3 consists of the final comparison made by *LinkOrdLinSrch*. For $m \in \{1, \dots, n\}$, let β_m denote the random variable that maps an input X such that $X = x_m$ onto the number of comparisons performed by *LinkOrdLinSrch* during stage 2 with input X. Then we have

$$E[\tau|Y=m] = k + 1 + E[\beta_m], \quad m = 1, \dots, n. \qquad \textbf{(6.8.5)}$$

Given any $m \in \{1, \dots, n\}$, we now calculate $E[\beta_m]$ using Formula (6.2.3). Let $q_i = q_i(m)$ denote the probability that *LinkOrdLinSrch* performs at least i comparisons during the second stage. Note that $q_i = 0$ when $i > m$ because the worst-case number of comparisons m occurs when a better start than *Head* was not found among $L[0], \dots, L[k-1]$ during stage 1. For $2 \le i \le m$, *LinkOrdLinSrch* performs less than i comparisons precisely when one of the elements $x_{m-i+2}, x_{m-i+3}, \dots, x_m$ belongs to $L[0], \dots, L[k-1]$. Thus, the probability q_i that at least i comparisons are performed is the probability that each of the list elements $L[0], \dots, L[k-1]$ belongs to the set of $n-i+1$ complementary elements to $x_{m-i+2}, x_{m-i+3}, \dots, x_m$. Because the total number of ways to choose such a sequence $L[0], \dots, L[k-1]$ is $(n-i+1)^{(k)}$ and the total number of ways to choose a sequence $L[0], \dots, L[k-1]$ from $x_1, x_2, \dots, x_n$ is $n^{(k)}$ (where $x^{(k)} = x(x-1)\dots(x-k+1)$), we have

$$q_i = \frac{(n-i+1)^{(k)}}{n^{(k)}} \le \frac{(n-i+1)^k}{n^k}. \qquad \textbf{(6.8.6)}$$

The inequality in (6.8.6) follows from Formula A.15 from Appendix A. Using Formula (6.2.3), we have

$$\begin{aligned} E[\beta_m] &= \sum_{i=1}^{m} q_i \le \sum_{i=1}^{m} \frac{(n-i+1)^k}{n^k} \\ &= \left(\frac{1}{n^k}\right) \sum_{i=n-m+1}^{n} i^k \\ &= \left(\frac{1}{n^k}\right)\left[\left(\sum_{i=1}^{n} i^k\right) - \left(\sum_{i=1}^{n-m} i^k\right)\right] \\ &\le \left(\frac{1}{n^k}\right)\left(\sum_{i=1}^{n} i^k\right), \quad m = 1, \dots, n. \end{aligned} \qquad \textbf{(6.8.7)}$$

As we showed in Chapter 3, $S(n,k) = \sum_{i=1}^{n} i^k$ is a polynomial in n of degree $k + 1$ with a leading coefficient of $1/(k + 1)$. Hence, Formula (6.8.7) implies

$$E[\beta_m] \leq \frac{n}{k+1} + O(1), \quad m = 1, \ldots, n. \qquad \textbf{(6.8.8)}$$

Substituting Formula (6.8.8) in (6.8.5), we have

$$E[\tau | Y = m] = k + 1 + E[\beta_m] \leq k + \frac{n}{k+1} + O(1), \quad m = 1, \ldots, n. \qquad \textbf{(6.8.9)}$$

Because the upper-bound estimate $k + n/(k + 1) + O(1)$ for $E[\tau \mid Y = m]$ given in Formula (6.8.9) is independent of m, substituting the estimate into (6.8.4) yields

$$A(n) \leq \frac{n}{k+1} + k + O(1). \qquad \textbf{(6.8.10)}$$

Using calculus, it is easy to verify that $n/(k + 1) + k$ achieves a minimum value of $2\sqrt{n} - 1$ at the point $k = \sqrt{n} - 1$, which is approximately $\lfloor\sqrt{n}\rfloor$. Hence, using $k = \lfloor\sqrt{n}\rfloor$ in *ProbeLinSrch* gives us the following from (6.8.10):

$$A(n) \leq 2\sqrt{n} + O(1). \qquad \textbf{(6.8.11)}$$

Because $A(n) \geq B(n) = \lfloor\sqrt{n}\rfloor + 1$, Formula (6.8.11) implies that $A(n) \in \Theta(\sqrt{n})$.

REMARK

Choosing a value of k that minimizes $n/(k + 1) + k$ does not automatically guarantee that this k minimizes $A(n)$, because Formula (6.8.10) is an inequality rather than an equality. However, it does give us an idea of why $\lfloor\sqrt{n}\rfloor$ is the choice for k used in the design of *ProbeLinSrch*. Moreover, our estimate of $A(n)$ is pretty sharp, since $\sqrt{n} < A(n) < 2\sqrt{n} + O(1)$.

6.9 Closing Remarks

Probability theory is an important basic tool in the analysis of algorithms. Currently, the probabilistic analysis of algorithms is a very active area of research, as is introducing probabilistic techniques into the program logic of algorithms.

Algorithms that are not completely deterministic but use random choices as part of the program logic are called *probabilistic* algorithms (see Chapter 24). These algorithms use probabilistic techniques as a design strategy as opposed to merely an analysis tool.

References and Suggestions for Further Reading

For a more advanced treatment of probabilistic algorithm analysis, see the following:

Coffman, E. G., Jr., and G. Lueker. *Probabilistic Analysis of Packing and Partitioning Algorithms.* New York: Wiley, 1991.

Hofri, M. *Probabilistic Analysis of Algorithms.* New York: Springer-Verlag, 1987.

EXERCISES

Section 6.1 Expectation and Average Complexity

For these exercises, we often refer to Appendix E for propositions and formulas.

6.1 Consider the sample space S corresponding to rolling two dice; that is, $S = \{(r_1,r_2) \mid r_1,r_2 \in \{1, \dots ,6\}\}$. Assume that the first die is fair but the second die is loaded, with probabilities 1/10, 1/10, 1/10, 1/10, 1/10, and 1/2 of rolling a 1, 2, 3, 4, 5, and 6, respectively.

a. Give a table showing the probability distribution for rolling these dice.

b. Compute the probability that at least one of the dice comes up 6.

c. Compute the conditional probability that the sum of the dice is 10 given that the loaded die does not come up 4.

6.2 Consider the random variable $X = r_1 + r_2$ defined on the sample space S given in Exercise 6.1.

a. Compute the density function $f(x) = P(X = x)$ and verify that it is a probability distribution on the sample $S_X = \{2, \dots ,12\}$.

b. Calculate the expectation $E[X]$.

6.3 Repeat Exercise 6.2 for the random variable $X = \max\{r_1, r_2\}$.

6.4 Let S be the sample space consisting of the positive integers. For a fixed p, $0 < p < 1$, show that the function $P(i) = (1 - p)^{i-1}p$ is a probability distribution on S.

6.5 Consider the sample space S corresponding to rolling three fair dice; that is, $S = (r_1,r_2,r_3) \mid r_1,r_2,r_3 \in \{1, \dots ,6\}\}$. Calculate the expectation $E[X]$ for each of the following random variables X:

a. $X = r_1 + r_2 + r_3$

b. $X = r_1 + r_2$

c. $X = \max \{r_1,r_2,r_3\}$

6.6 Give an alternative derivation of the expectation of a binomial distribution using Proposition E.3.2. (*Hint:* Let X_i be the random variable whose value is 1 if there was a success in the i^{th} trial and 0 otherwise.)

6.7 Verify that the expectation of the geometric distribution (see Appendix E) with probability p of success is $1/p$.

6.8 Verify the following formula, which was used in the derivation of the variance of the geometric distribution with probability p of success:

$$\sum_{k=1}^{\infty} k^2(1-p)^{k-1}p = \frac{2-p}{p^2}$$

6.9 Calculate the variance of the random variables given in Exercises 6.2 and 6.3.

6.10 Calculate the variance of the random variables given in Exercise 6.5.

6.11 a. Two random variables X and Y defined on a sample space S are *independent* if the events $X = x$ and $Y = y$ are independent for every x and y. Show that if $X_1,X_2, \dots ,X_n$ are pairwise independent random variables, then

$$V(X_1 + X_2 + \cdots + X_n) = V(X_1) + V(X_2) + \cdots + V(X_n).$$

b. Employing the formula given in part (a), calculate the variance of the binomial distribution with p and n. (*Hint:* Let X_i be the random variable whose value is 1 if there was a success in the i^{th} trial and 0 otherwise.)

6.12 Prove Proposition E.1.1.

6.13 Prove Propositions E.3.1 and E.3.2.

6.14 Verify that the probabilities given by Formula (E.3.3) satisfy the three axioms for a probability distribution on $\{0,1, \dots ,n\}$.

6.15 Verify that P_F defined by (E.4.1) satisfies the three axioms for a probability distribution on F.

6.16 Given a discrete random variable X, show that the probability density function $f(x) = P(X = x)$ determines a probability distribution on $S_X = \{x{:}f(x) \neq 0\}$.

6.17 Show that Formula (E.2.5) of Proposition E.2.1 reduces to Formula (E.3.7) when $X = Y$.

6.18 a. For events A and B, show that $P(A \cup B) = P(A) + P(B) - P(A \cap B)$

b. State and prove a generalization of formula in part (a) to n sets $A_1, \ldots, A_n$

Section 6.2 Techniques for Computing Average Complexity

6.19 Derive Formula (6.2.2) from Formula (6.2.1).

6.20 Derive Formula (6.2.3) from Formula (6.2.2).

Section 6.3 Average Complexity of *LinearSearch*

6.21 Calculate the average complexity $A(n)$ of *LinearSearch* assuming that *both* of the following two assumptions about the input list $L[0{:}n - 1]$ and search element X hold:

The probability that X occurs in the list is 2/3.

Given that X occurs in the list, X is twice as likely to occur in the first half of the list (positions 0 through $\lfloor n/2 \rfloor - 1$) as in the second half. Further, if X occurs in the first half of the list it is equally likely to occur in any position in the first half. A similar assumption is made about the second half.

Section 6.4 Average Complexity of *InsertionSort*

6.22 a. Give pseudocode for a recursive version *InsertionSortRec* of *InsertionSort*.

b. Obtain a recurrence relation for the average complexity $A(n)$ of *InsertionSortRec*.

c. Solve the recurrence relation obtained in (b), and compare it to the formula for $A(n)$ given in Section 6.4.

Section 6.5 Average Complexity of *QuickSort*

6.23 Suppose that we sort a list $L[0:n - 1]$ by first determining an element of maximum value, placing it at position n, and then calling *QuickSort* with the (possibly altered) list $L[0:n - 2]$. Analyze the average complexity of the resulting sorting algorithm, and compare it to *QuickSort*.

Section 6.6 Average Complexity of *MaxMin2*

6.24 Write a program to empirically test the average performance of *MaxMin2*, and compare this performance to *MaxMin3*.

Section 6.7 Average Complexity of *BinarySearch* and *SearchBinSrchTree*

6.25 Show that permutations $\pi:\{1, \ldots, n\}\rightarrow\{1, \ldots, n\}$ such that $\pi(n) = i \neq n$ are in one-to-one correspondence with permutations of $\{1, \ldots, n - 1\}$.

6.26 Show that the implicit search tree for *BinarySearch* is a 2-tree that is full at the second deepest level.

6.27 Derive Formula (6.7.5), which expresses $A(T,n)$ in terms of the leaf path length of T, and use it to prove Theorem 6.7.1

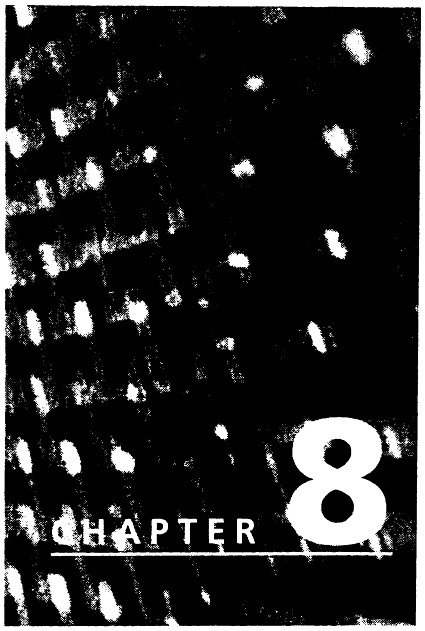
*6.28 Compute the average complexity $A(n)$ of *SearchBinSrchTree* over the set of all binary search trees T. Assume that each binary tree search T is equally likely to be the input tree to *SearchBinSrchTree*. Also assume that X is one of the keys in the search tree and that X has an equal chance of being any of the n keys.

Section 6.8 Searching a Link-Ordered List

6.29 Show that the algorithm for searching a link-ordered list of size n that simulates binary search would perform n index updates in the worst case.

6.30 Generalize Exercise 6.29 by showing that *any* comparison-based algorithm for searching a link-ordered list of size n requires n index updates in the worst case.

6.31 Show that Formula (6.8.1) can be derived directly from the definition of conditional expectation and is actually a special case of (E.4.5).



DIVIDE-AND-CONQUER

The *divide-and-conquer* paradigm is one of the most powerful design strategies available in the theory of algorithms. The paradigm can be described in general terms as follows. A problem input (instance) is divided according to some criteria into a set of smaller inputs to the same problem. The problem is then solved for each of these smaller inputs, either recursively by further division into smaller inputs or by invoking an ad hoc or a priori solution. Finally, the solution for the original input is obtained by expressing it in some form as a combination of the solution for these smaller inputs. Ad hoc solutions are often invoked when the input size is smaller than some preassigned *threshold* value. Examples of a priori solutions (solutions known in advance) include sorting single-element lists or multiplying single-digit binary numbers.

8.1 The Divide-and-Conquer Paradigm

The divide-and-conquer design strategy can be formalized as follows. Let *Known* denote the set of inputs to the problem whose solutions are known a priori or by ad hoc methods. The procedure *Divide_and_Conquer* calls two procedures, *Divide* and *Combine*. *Divide* has input parameter I and output parameters $I_1, \ldots, I_m$, where *m* may depend on *I*. The inputs $I_1, \dots, I_m$ must be smaller or simpler inputs to the problem than *I* but are not required to always be a *division* of *I* into subinputs. *Combine* has input parameters $J_1, \dots, J_m$ and output parameter *J*. *Combine* is the procedure for obtaining a solution *J* to the problem with input *I* by combining the recursively obtained solutions $J_1, \dots, J_m$ to the problem with inputs $I_1, \dots, I_m$.

```
procedure Divide_and_Conquer(I, J) recursive
Input:    I (an input to the given problem)
Output:   J (a solution to the given problem corresponding to input I)
    if $I \in$ Known then
        assign the a priori or ad hoc solution for I to J
    else
        Divide(I, $I_1, \dots, I_m$)   //m may depend on the input I
        for $i \leftarrow 1$ to m do
            Divide_and_Conquer($I_i, J_i$)
        endfor
        Combine($J_1, \dots, J_m$, J)
    endif
end Divide_and_Conquer
```

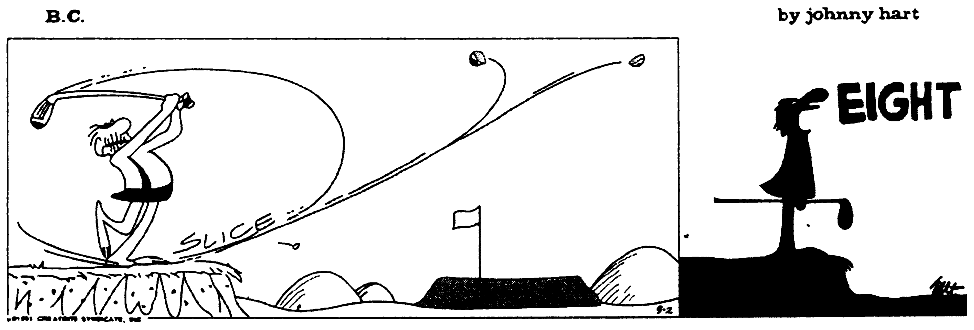
Often the work done by an algorithm based on *Divide_and_Conquer* resides solely in one of the two procedures *Divide* or *Combine*, but not both. For example, in the algorithm *QuickSort*, the divide step (a call to *Partition*) is the heart of the algorithm, and the combine step requires no work at all. On the other hand, in the algorithm *MergeSort*, the divide step is trivial and the combine step (a call to *Merge*) is the heart of the algorithm. Sometimes, both the divide and combine steps are difficult (see Figure 8.1).

For most divide-and-conquer algorithms, the number *m* of subproblems is a constant (independent of any particular input *I*). Divide-and-conquer algorithms in which *m* is a constant equal to 1 are referred to as *simplifications*. The binary search algorithm is an example of a simplification.

FIGURE 8.1

An example of a divide-and-conquer algorithm in which both the divide and combine steps are hard.

Reprinted by permission of Johnny Hart and Creators Syndicate, Inc.



One useful technique to improve the efficiency of a divide-and-conquer algorithm is to employ a known ad hoc algorithm when the input size is smaller than some threshold. Of course, the ad hoc algorithm must be more efficient than the given divide-and-conquer algorithm for sufficiently small inputs.

For example, consider the sorting algorithm *MergeSort*, which has $\Theta(n\log n)$ average complexity. The sorting algorithm *InsertionSort*, which has $\Theta(n^2)$ average complexity, is much less efficient than *MergeSort* for large values of n. However, due to the constants involved, *InsertionSort* is more efficient than *MergeSort* for small values of n. Thus, we can improve the performance of *MergeSort* by calling *InsertionSort* for any input list whose size is not larger than a suitable threshold. Finding the optimal value for a threshold is often done empirically in practice. Empirical studies are needed because the best choice of a threshold depends on the constants associated with the implementation on a particular computer. For example, empirical studies have shown that for *MergeSort*, calling *InsertionSort* with a threshold of around $n = 16$ is usually optimal.

8.2 Symbolic Algebraic Operations on Polynomials

Algebraic manipulation of polynomials is an essential tool in many applications. Therefore, the need arises to design efficient algorithms to carry out basic arithmetic operations on polynomials, such as addition and multiplication, as efficiently as possible. Given a polynomial $P(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0$, it is important to differentiate between the pointwise and the symbolic representation of $P(x)$. The *pointwise representation* of $P(x)$ is simply a function that maps an input point x to the output point $P(x)$. Thus, given two polynomials $P(x)$ and $Q(x)$, their *pointwise product* is the function *PtWiseMult*(P, Q) that maps an input point x to the output point $P(x) * Q(x)$.

The *symbolic representation* of a polynomial $P(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0$ is its coefficient array $[a_0, a_1, \ldots, a_{m-1}]$. Thus, given two polynomials $P(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0$ and $Q(x) = b_{n-1}x^{n-1} + \cdots + b_1x + b_0$, their symbolic product is the coefficient array $[c_0, c_1, \ldots, c_{m+n-2}]$ of the product polynomial $P(x)Q(x)$ given by

$$c_k = \sum_{i+j=k} a_ib_j,\ 0 \le i \le m-1,\ 0 \le j \le n-1,\ k = 0, \ldots, m+n-2. \quad \textbf{(8.2.1)}$$

For example, the symbolic product of $P(x) = 3x^2 + 2x - 5$ having coefficient array $[-5, 2, 3]$ and $Q(x) = x^3 - x + 4$ having coefficient array $[4, -1, 0, 1]$ is the polynomial $3x^5 + 2x^4 - 8x^3 + 10x^2 + 13x - 20$ having coefficient array $[-20, 13, 10, -8, 2, 3]$.

The symbolic representation of the product is particularly important for determining various properties of the product polynomial. For example, the function *PtWiseMult* does not lend itself to taking the derivative of the product polynomial, whereas it is a simple matter to compute the symbolic representation of the derivative of a polynomial that is represented symbolically. Similar comments hold for other operations on polynomials such as root finding, integration, and so forth. Throughout the remainder of this chapter, when discussing algebraic operations on polynomials, we implicitly assume that we are performing these operations symbolically.

8.2.1 Multiplication of Polynomials of the Same Input Size

An algorithm *DirectPolyMult* based on a straightforward calculation of Formula (8.2.1) has complexity $\Theta(mn)$ (where we choose multiplication of coefficients as our basic operation). We now describe a more efficient algorithm for polynomial multiplication based on the divide-and-conquer paradigm. We first assume that $m = n$. Setting $d = \lceil n/2 \rceil$, we divide the set of coefficients of the polynomials in half, with the higher-order coefficients $a_{n-1}, a_{n-2}, \ldots, a_d$ in one set and the lower-order coefficients of $a_{d-1}, a_{d-2}, \ldots, a_0$ in the other. Setting $P_1(x) = a_{d-1}x^{d-1} + a_{d-2}x^{d-2} + \cdots + a_1x + a_0$ and $P_2(x) = a_{n-1}x^{n-d-1} + a_{n-2}x^{n-d-2} + \cdots + a_{d+1}x + a_d$, we obtain

$$P(x) = x^dP_2(x) + P_1(x).$$

A similar division of the set of coefficients of $Q(x)$ yields polynomials $Q_1(x)$ and $Q_2(x)$ having input size of at most d such that

$$Q(x) = x^dQ_2(x) + Q_1(x).$$

A straightforward application of the distributive law yields

$$\begin{aligned} P(x)Q(x) = {} & x^{2d}P_2(x)Q_2(x) + x^d(P_1(x)Q_2(x) \\ & + P_2(x)Q_1(x)) + P_1(x)Q_1(x). \end{aligned} \tag{8.2.2}$$

Note that polynomials $P_1(x)$ and $Q_1(x)$ both have input size d, and polynomials $P_2(x)$ and $Q_2(x)$ both have input size either d or $d - 1$. When $P_2(x)$ and $Q_2(x)$ both have input size $d - 1$, we add a leading coefficient of zero to each so they have input size d. Thus, the problem of multiplying $P(x)$ and $Q(x)$ has been reduced to the problem of taking four products of polynomials of input size $d = \lceil n/2 \rceil$ together with two multiplications by powers of x and three additions. It turns out that the resulting divide-and-conquer algorithm based on Formula (8.2.2) still has quadratic complexity. However, there is a clever way of combining the split polynomials that uses only *three* polynomial multiplications instead of four, based on the following simple identity:

$$P(x)Q(x) = x^{2d}P_2(x)Q_2(x) + x^d((P_1(x) + P_2(x))(Q_1(x) + Q_2(x)) - P_1(x)Q_1(x) - P_2(x)Q_2(x)) + P_1(x)Q_1(x). \tag{8.2.3}$$

Formula (8.2.3) yields the divide-and-conquer algorithm *PolyMult1* for polynomial multiplication. *PolyMult1* calls a "split pea" procedure, *Split*$(P(x), P_1(x), P_2(x))$, which inputs a polynomial $P(x)$ (of input size n) and outputs the two polynomials $P_1(x)$ and $P_2(x)$. We write *PolyMult1* as a high-level recursive function whose inputs are the polynomials $P(x)$ and $Q(x)$ and whose output is the polynomial $P(x)Q(x)$. We will not be explicit about how the coefficients of the polynomials are maintained (linked lists, arrays, and so forth). We will also assume that we have well-defined procedures for multiplying a polynomial by x^i. For example, if the polynomial is maintained by an array of its coefficients, then this amounts to shifting indices by i and replacing the first i entries with zeros. We abuse the notation slightly by writing $x^iP(x)$ to mean the result of calling such a procedure. Also, for convenience we simply use the symbol + to denote the addition of two polynomials.

```
function PolyMult1(P, Q, n) recursive
Input:   $P(x) = a_{n-1}x^{n-1} + \cdots + a_1x + a_0$, $Q(x) = b_{n-1}x^{n-1} + \cdots + b_1x + b_0$
            (polynomials)
            $n$ (a positive integer)
Output:  $P(x)Q(x)$ (the product polynomial)
    if $n = 1$ then
        return($a_0b_0$)
    else
        $d \leftarrow \lceil n/2 \rceil$
        Split($P(x), P_1(x), P_2(x)$)
        Split($Q(x), Q_1(x), Q_2(x)$)
        $R(x) \leftarrow$ PolyMult1($P_2(x), Q_2(x), d$)
        $S(x) \leftarrow$ PolyMult1($P_1(x) + P_2(x), Q_1(x) + Q_2(x), d$)
        $T(x) \leftarrow$ PolyMult1($P_1(x), Q_1(x), d$)
        return($x^{2d}R(x) + x^d(S(x) - R(x) - T(x)) + T(x)$)
    endif
end PolyMult1
```

When analyzing *PolyMult1*, we choose coefficient multiplication as our basic operation and thus ignore the two multiplications by powers of x. However, the procedure referred to earlier for multiplying a polynomial by x^i has linear complexity and does not affect the order of complexity of *PolyMult1*. We also assume that n is a power of 2 because we can interpolate asymptotic behavior using Θ-scalability (see Appendix D). Since *PolyMult1* invokes itself three times with n replaced by $d = n/2$, the number of coefficient multiplications $T(n)$ performed by *PolyMult1* satisfies the following recurrence relation:

$$T(n) = 3T\left(\frac{n}{2}\right), \quad n > 1, \quad \textbf{init. cond. } T(1) = 1. \qquad \textbf{(8.2.4)}$$

It follows from a simple unwinding of Formula (8.2.4) that $T(n) \in \Theta(n^{\log_2 3})$ [see also the discussion following Formula (3.3.12) in Chapter 3]. Because $\log_2 3$ is approximately 1.59, we now have an algorithm for polynomial multiplication whose $\Theta(n^{\log_2 3})$ complexity is a significant improvement over the $\Theta(n^2)$ complexity of *DirectPolyMult*. In Chapter 22, we develop an even faster polynomial multiplication algorithm using the powerful tool known as the Fast Fourier Transform.

8.2.2 Multiplication of Polynomials of Different Input Sizes

In practice, we often encounter the problem of multiplying two polynomials $P(x)$ and $Q(x)$ of different input sizes m and n, respectively. If $m < n$, then we could merely augment $P(x)$ with $n - m$ leading zeros, but this would be quite inefficient if n is significantly larger than m. It is better to partition $Q(x)$ into blocks of size m. For convenience, we assume n is a multiple of m—that is, $n = km$ for some positive integer k. We let $Q_i(x)$ be the polynomial of degree m given by

$$Q_i(x) = b_{im-1}x^{m-1} + b_{im-2}x^{m-2} + \cdots + b_{(i-1)m+1}x + b_{(i-1)m}, \; i \in \{1, \ldots, k\}.$$

Clearly,

$$Q(x) = Q_k(x)x^{m(k-1)} + Q_{k-1}(x)x^{m(k-2)} + \cdots + Q_2(x)x^m + Q_1(x).$$

It follows immediately from the distributive law that

$$P(x)Q(x) = P(x)Q_k(x)x^{m(k-1)} + P(x)Q_{k-1}(x)x^{m(k-2)} + \cdots + P(x)Q_1(x).$$

Applying these ideas, we obtain the algorithm *PolyMult2* for multiplying two polynomials $P(x)$ and $Q(x)$. *PolyMult2* is efficient even if the degree $m - 1$ of $P(x)$ is much less than the degree $n - 1$ of $Q(x)$. If n is not a multiple of m, we compute the largest integer k such that $n > km$ and augment $P(x)$ with $n - km$ leading zeros.

```
function PolyMult2(P(x), Q(x), m, n)
Input:  $P(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0$, $Q(x) = b_{n-1}x^{n-1} + \cdots + b_1x + b_0$
            (polynomials)
            n, m (positive integers)  //$n = km$ for some integer k
Output: $P(x)Q(x)$ (the product polynomial)
    ProdPoly(x) ← 0 //initialize all coefficients of ProdPoly(x) to be 0
    for i ← 1 to k do
        $Q_i(x) = b_{im-1}x^{m-1} + b_{im-2}x^{m-2} + \cdots + B_{(i-1)m}$
    endfor
    for i ← 1 to k do
        ProdPoly(x) ← ProdPoly(x) + $x^{(i-1)m}$PolyMult1(P(x), $Q_i(x)$, m)
    endfor
    return(ProdPoly(x))
end PolyMult2
```

The complexity of *PolyMult1* for multiplying two polynomials of degree $m - 1$ is $\Theta(m^{\log_2 3})$. Since *PolyMult2* invokes *PolyMult1* a total of $k = n/m$ times, each time with input polynomials of degree $m - 1$, it follows that the complexity of *PolyMult2* is

$$\Theta(km^{\log_2 3}) = \Theta\left(\frac{nm^{\log_2 3}}{m}\right) = \Theta(nm^{\log_2(3/2)}).$$

8.3 Multiplication of Large Integers

Computers typically assign a fixed number of bits for storing integer variables. Arithmetic operations such as addition and multiplication of integers are often carried out by moving the integer operands into *fixed-length* registers and then invoking arithmetic operations built into the hardware. However, for some important applications, such as those occurring in cryptography, the number of digits is too large to be handled directly by the hardware in this way. Such applications must perform these operations by storing the integers using an appropriate data structure (such as an array or a linked list) and then using algorithms to carry out the arithmetic operations. When analyzing the complexity of such algorithms, the size n of an integer A is taken to be the number of digits of A. Any base $b \geq 2$ can be chosen for representing an integer. We denote the i^{th} least significant digit of U (base b) by u_i, $i = 0, 1, \ldots, n - 1$; that is,

$$U = \sum_{i=0}^{n-1} u_i b^i. \qquad \textbf{(8.3.1)}$$

The classic grade-school algorithm for multiplying two integers U and V of size n clearly involves n^2 multiplications of digits. Viewing the right-hand side of Formula (8.3.1) as a polynomial in b leads us to a divide-and-conquer strategy for computing UV based on a formula analogous to Formula (8.2.3). Although similar to the multiplication of polynomials, addition and multiplication of large integers include the additional task of handling carry digits. The design details of the resulting $\Theta(n^{\log_2 3})$ algorithm *MultInt* are left to the exercises.

8.4 Multiplication of Matrices

Given two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$, $0 \le i, j \le n - 1$, recall that the product AB is defined to be the $n \times n$ matrix $C = (c_{ij})$, where

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik}b_{kj}. \tag{8.4.1}$$

The straightforward algorithm based on this definition clearly performs n^3 (scalar) multiplications. In 1969 Strassen devised a divide-and-conquer algorithm for matrix multiplication of complexity $O(n^{\log_2 7})$ using certain algebraic identities for multiplying 2×2 matrices.

The classic method of multiplying 2×2 matrices performs 8 multiplications as follows:

$$AB = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}\begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{bmatrix} \tag{8.4.2}$$

Strassen discovered a way to carry out the same matrix product AB using only the following seven multiplications:

$$\begin{aligned} m_1 &= (a_{00} + a_{11})(b_{00} + b_{11}) \\ m_2 &= (a_{10} + a_{11})b_{00} \\ m_3 &= a_{00}(b_{01} - b_{11}) \\ m_4 &= a_{11}(b_{10} - b_{00}) \\ m_5 &= (a_{00} + a_{01})(b_{11}) \\ m_6 &= (a_{10} - a_{00})(b_{00} + b_{01}) \\ m_7 &= (a_{01} - a_{11})(b_{10} + b_{11}) \end{aligned} \tag{8.4.3}$$

The matrix product AB is then given by

$$AB = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix} \tag{8.4.4}$$

Consider now the case of two $n \times n$ matrices where, for convenience, we assume that $n = 2^k$. We first partition the matrices A and B into four $(n/2) \times (n/2)$ submatrices, as follows:

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}, \quad B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} \tag{8.4.5}$$

The product AB can be expressed in terms of eight matrix products as follows:

$$AB = \begin{bmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{bmatrix}. \tag{8.4.6}$$

Thus, in complete analogy with the 2×2 case, we can carry out the matrix product AB using only the following seven matrix multiplications:

$$\begin{aligned} M_1 &= (A_{00} + A_{11})(B_{00} + B_{11}) \\ M_2 &= (A_{10} + A_{11})B_{00} \\ M_3 &= A_{00}(B_{01} - B_{11}) \\ M_4 &= A_{11}(B_{10} - B_{00}) \\ M_5 &= (A_{00} + A_{01})(B_{11}) \\ M_6 &= (A_{10} - A_{00})(B_{00} + B_{01}) \\ M_7 &= (A_{01} - A_{11})(B_{10} + B_{11}) \end{aligned} \tag{8.4.7}$$

As in the case of 2×2 matrices, the matrix product AB is then given by

$$AB = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}. \tag{8.4.8}$$

Formulas (8.4.7) and (8.4.8) immediately yield Strassen's algorithm, a divide-and-conquer algorithm based on expressing the product of two $n \times n$ matrices in terms of seven products of $(n/2) \times (n/2)$ matrices. The complexity of Strassen's algorithm clearly satisfies the recurrence relation

$$T(n) = 7T\left(\frac{n}{2}\right), \quad n > 1, \quad \textbf{init. cond. } T(1) = 1. \tag{8.4.9}$$

By unwinding Formula (8.4.9), we see that $T(n) \in \Theta(n^{\log_2 7})$ [see also the discussion following Formula (3.3.12) in Chapter 3]. Because $\log_2 7$ is approximately 2.81, we now have an algorithm for matrix multiplication with complexity of $\Theta(n^{\log_2 7})$, which is a significant improvement over the $\Theta(n^3)$ complexity of the classical algorithm for matrix multiplication.

Strassen's identities (8.4.3) and (8.4.4) involve a total of 18 additions (or subtractions). Winograd discovered the following set of identities, which leads to a method of multiplying 2 × 2 matrices using only 15 additions or subtractions but still doing only seven multiplications:

$$\begin{aligned}
m_1 &= (a_{10} + a_{11} - a_{00})(b_{11} - b_{01} + b_{00}) \\
m_2 &= a_{00}b_{00} \\
m_3 &= a_{01}b_{10} \\
m_4 &= (a_{00} - a_{10})(b_{11} - b_{01}) \\
m_5 &= (a_{10} + a_{11})(b_{01} - b_{00}) \\
m_6 &= (a_{01} - a_{10} + a_{00} - a_{11})b_{11} \\
m_7 &= a_{11}(b_{00} + b_{11} - b_{01} - b_{10})
\end{aligned} \tag{8.4.10}$$

The matrix product AB is then given by

$$AB = \begin{bmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{bmatrix}. \tag{8.4.11}$$

Although Formulas (8.4.10) and (8.4.11) involve a total of 24 additions or subtractions, it can be verified that a total of only 15 distinct additions or subtractions are needed. Thus, we obtain an improvement of three less additions or subtractions over Strassen's identities while keeping the number of multiplications at seven.

8.5 Selecting the k^{th} Smallest Value in a List

Our next illustration of the divide-and-conquer paradigm is actually an instance of simplification. Given a list $L[0{:}n - 1]$ and an integer k, $1 \le k \le n$, we consider the problem of finding the k^{th} smallest value in L. When $k = 1$ or $k = n$, this problem coincides with the problem of finding the minimum or maximum value in L, respectively. When n is odd and $k = (n + 1)/2$, we are finding the median value in L, whereas for n, even the median value is obtained by averaging the two values corresponding to $k = n/2$ and $t = n/2 + 1$. Since linear algorithms exist for finding the maximum or minimum value in L, we hope to find a linear algorithm for a general value of k. In fact, we first design a linear average-behavior algorithm *Select* for the general problem by using the procedure *Partition* that formed the basis of *QuickSort*, together with a strategy analogous to a binary search. Unfortunately, as did *QuickSort*, *Select* has quadratic worst-case performance. However, by modifying the algorithm, we can obtain a rather more complicated algorithm, *Select2*, having linear worst-case complexity. Because of its linear average performance and its simplicity compared with *Select2*, *Select* is more often used in practice.

8.5.1 The Algorithm Select

The k^{th} smallest value in the list $L[0{:}n-1]$ would appear in index position $t = k - 1$ if $L[0{:}n-1]$ were sorted (in increasing order). It is convenient to use t as the input parameter to both *Select* and *Select2*. In particular, the output parameter is the value that would be in position t if $L[0{:}n-1]$ were sorted.

Given $L[0{:}n-1]$ and an index position t, suppose an initial call to the procedure *Partition* with $L[0{:}n-1]$ returns the value $j = \mathit{position}$, so that in the rearranged list (which we still denote by L), $L[i] \le L[j]$ for $0 \le i < j$, $L[i] \ge L[j]$ for $j < i \le n-1$, and the pivot element $L[j]$ has the value of the original $L[0]$. Hence, in the rearranged list, the value in index position j is the value that would appear in this position if the original L were sorted (this was the basis of *QuickSort*). So we're done if $t = j$. Because each element in $L[0{:}j-1]$ is not larger than each element in $L[j+1{:}n-1]$, it follows that if $t < j$, we are looking for the value that would appear in position t of $L[0{:}j-1]$ if the latter sublist were sorted. Using similar reasoning, if $t > j$, then we are looking for the value that would appear in position t of $L[j+1{:}n-1]$ if the latter sublist were sorted. The recursive algorithm *Select* is based on these simple observations.

```
procedure Select(L[0:n − 1], low, high, t, x) recursive
Input:   L[0:n − 1] (an array of size n), low, high (indices of L[0:n − 1])
                   t (a positive integer such that low ≤ t ≤ high)
Output:  x (the value that would appear in index position t if L[low:high] were sorted in
         increasing order)
   Partition(L[0:n − 1], low, high, position)
   case
      :t = position: x ← L[position]
      :t < position: Select(L[0:n − 1], low, position − 1, t, x)
      :t > position: Select(L[0:n − 1], position + 1, high, t, x)
   endcase
end Select
```

In Figure 8.2, we illustrate the action of *Select* for a sample list of size 7.

We now analyze *Select* for a list of size n, where we assume that $low = 0$ and $high = n - 1$. We use the list comparisons generated by calls to the procedure *Partition* as our basic operation. Let $W(n, t)$ and $A(n, t)$ denote the worst-case and average complexities of *Select* restricted to inputs where the parameter t is fixed. Hence,

$$W(n) = \max\{W(n, t) \mid 0 \le t \le n - 1\}, \qquad \textbf{(8.5.1)}$$

FIGURE 8.2 Action of *Select* with $t = 4$.

Original	index	0	1	2	3	4	5	6	7
List	list element	22	9	23	52	15	19	47	$+\infty$

call *Partition*$(L[0{:}6], position)$

Output	index	0	1	2	③	4	5	6	7
List	list element	15	9	19	22	52	23	47	$+\infty$

Output value of *position* = 3

call *Partition*$(L[4{:}6], position)$

Output	index	0	1	2	3	4	5	⑥	7
List	list element	15	9	19	22	47	23	52	$+\infty$

Output value of *position* = 6

call *Partition*$(L[4{:}5], position)$

Output	index	0	1	2	3	4	⑤	6	7
List	list element	15	9	19	22	23	47	52	$+\infty$

Output value of *position* = 5

call *Partition*$(L[4{:}5], position)$

Output	index	0	1	2	3	④	5	6	7
List	list element	15	9	19	22	23	47	52	$+\infty$

Output value of *position* = 4

$x = 23$

and assuming each t is equally likely,

$$A(n) = \frac{\sum_{t=0}^{n-1} A(n, t)}{n}. \tag{8.5.2}$$

Because *Partition* has complexity $n + 1$ for a list of size n, and because there exists an input list whose size is only narrowed by 1 with each call to *Partition*, we have

$$\begin{aligned} W(n, t) &= (n + 1) + n + (n - 1) + \cdots + 2 \\ &= \frac{(n + 1)(n + 2)}{2} - 1 \in \Theta(n^2). \end{aligned} \tag{8.5.3}$$

Thus, *Select* has the same quadratic worst-case behavior as *QuickSort*.

When analyzing the average performance of *Select*, we assume, as usual, that the inputs to *Select* are all permutations of $1, \dots, n$, and that each permutation is equally likely. We now show by the strong form of induction on n that

$$A(n, t) \le 4n, \quad \text{for all } n \text{ and all } t \in \{0, \dots, n-1\}. \tag{8.5.4}$$

Since $A(1,0) = 2$, Formula (8.5.4) is true for $n = 1$ and $t = 0$. Assuming that (8.5.4) holds for all pairs m, t with $t \in \{0, \dots, m-1\}$ and $m < n$, we show that (8.5.4) also holds for n and all $t \in \{0, \dots, n-1\}$. Let t be any integer such that $0 \le t \le n-1$. Our assumption that all permutations of $1, 2, \dots, n$ are equally likely to be input to *Select* implies that after executing the call to *Partition* with $L[0{:}n-1]$, each value $0, \dots, n-1$ is equally likely to be the output value of the parameter *position*. There are three cases to consider, depending on whether *position* is less than, equal to, or greater than t.

For convenience, let i = *position*. If $t = i$, then *Select* terminates after a single call to *Partition*, so that $n+1$ comparisons are performed. If $t < i$, then *Select* calls itself recursively looking for x in the sublist $L[0{:}i-1]$, so that the average number of comparisons done in this case is given by

$$n + 1 + A(i, t) \le n + 1 + 4i \quad \text{(by induction hypothesis)}. \tag{8.5.5}$$

Finally, if $t > i$, then *Select* calls itself recursively looking for x in the sublist $L[i+1{:}n-1]$, so that the average number of comparisons performed in this case is given by

$$n + 1 + A(n-i-1, t-i-1) \le n + 1 + 4(n-i-1) \quad \text{(by induction hypothesis)}. \tag{8.5.6}$$

Using inequalities (8.5.5) and (8.5.6), we obtain

$$\begin{aligned} A(n,t) &= \frac{1}{n}\left[n + 1 + \sum_{i=0}^{t-1} (n + 1 + A(n-i-1, t-i-1)) + \sum_{i=t+1}^{n-1} (n + 1 + A(i,t))\right] \\ &= \frac{1}{n}\left[n(n+1) + \sum_{i=0}^{t-1} A(n-i-1, t-i-1) + \sum_{i=t+1}^{n-1} A(i,t)\right] \le n + 1 + \frac{4}{n}\left[\sum_{i=0}^{t-1}(n-i-1) + \sum_{i=t+1}^{n-1} i\right]. \end{aligned} \tag{8.5.7}$$

As a function of t, the right side of the inequality in Formula (8.5.7) is maximized when $t = \lceil (n-1)/2 \rceil$. Now assume that n is odd (the proof when n is even is similar). Then using (8.5.7) and rewriting the term inside the last square bracket yields

$$\begin{aligned}A(n,t) &\le A\left(n, \frac{n-1}{2}\right) \le n + 1 + \frac{4}{n}\left[2\left(\sum_{i=1}^{n-1} i - \sum_{i=1}^{(n-1)/2} i\right)\right] \\ &= n + 1 + \frac{4}{n}\left[n(n-1) - \left(\frac{n-1}{2}\right)\left(\left(\frac{n-1}{2}\right) + 1\right)\right] \\ &\le n + 1 + \frac{4}{n}\left[n(n-1) - \frac{(n-1)^2}{4}\right] = n + 1 + \frac{4}{n}\left[(n-1)\left(n - \frac{n-1}{4}\right)\right] \\ &= n + 1 + \frac{4}{n}\left[(n-1)\left(\frac{3n+1}{4}\right)\right] = n + 1 + \frac{(n-1)(3n+1)}{n} \le 4n.\end{aligned}$$

This completes the inductive proof of (8.5.4).

It follows immediately from Formulas (8.5.2) and (8.5.4) that $A(n) \le 4n$. Because *Select* performs at least one call to *Partition* for any input, both $A(n, t)$ and $A(n)$ are at least $n + 1$, so that we have

$$n + 1 \le A(n),\ A(n,t) \le 4n, \text{ for all } n \text{ and all } t \in \{0, \dots, n-1\}. \quad \textbf{(8.5.8)}$$

Hence, $A(n), A(n, t) \in \Theta(n)$.

REMARK

We have shown $A(n, t) \le A(n, (n-1)/2) \le 4n$ and concluded that $A(n) \le 4n$. However, a similar induction argument can be used to show that $A(n, 0) \le 2(n + 1)$, which is a reduction by about half of our $4n$ upper bound. With $A(n)$ being the average of $A(n, t)$ over all $t \in \{0, \dots, n-1\}$, we might expect to obtain a better upper bound for $A(n)$. In fact, we can show that $A(n) \le 3n$, but we leave verification of this stronger inequality to an exercise.

8.5.2 A Version of *Select* with Linear Worst-Case Complexity

We now describe another version of *Select*, called *Select2*, which has linear worst-case complexity. The idea is to do a little more work choosing a pivot element before calling *Partition* so that the subsequent recursive calls (if any) are always with sublists whose size is reduced by a **fixed** ratio r. The following Proposition will then show that we should expect linear behavior in such a situation. In our analysis of *Select2*, we assume distinct list elements.

Proposition 8.5.1 Suppose r is a number such that $0 < r < 1$. Also, suppose that the worst case of a recursive algorithm satisfies the recurrence

$$W(n) = W(rn) + f(n), \text{ where } f(n) \in \Theta(n).$$

Then $W(n) \in \Theta(n)$, that is, $W(n)$ has linear complexity.

Proof

To prove Proposition 8.5.1, we unwind the recurrence as follows:

$$\begin{aligned} W(n) &= W(rn) + f(n) \\ &= W(r^2n) + f(n)(1 + r) \\ &\text{... iterate until } r^k \leq 1/n \\ &\leq W(1) + f(n)(1 + r + r^2 + \cdots + r^k + \cdots) \\ &= W(1) + f(n)/(1 - r) \in \Theta(n). \end{aligned}$$ ■

Note that in the algorithm *Select*, the first element in a list is always chosen as the pivot element in the call to *Partition*. This fixed choice of the pivot element leads to quadratic worst-case complexity because it is possible that the list is only narrowed down by 1 in our recursive calls to *Select*. What is needed is a choice of a pivot element that guarantees that the recursive calls will narrow the size of the list by a fixed ratio, which would lead to linear worst-case behavior by analogy with Proposition 8.5.1. Of course, choosing the median element in a list as a pivot element would narrow the list by about 1/2. However, finding the median element is a special case of the problem at hand. We will settle for a pivot element *pm*, called a *pseudomedian*, which will be good enough to narrow the list by about 7/10, so that the recursive call will have $W(7n/10)$ worst-case complexity. Because determining *pm* will turn out to have $W(n/5) + 6n/5$ complexity, we expect a recurrence relation of the form

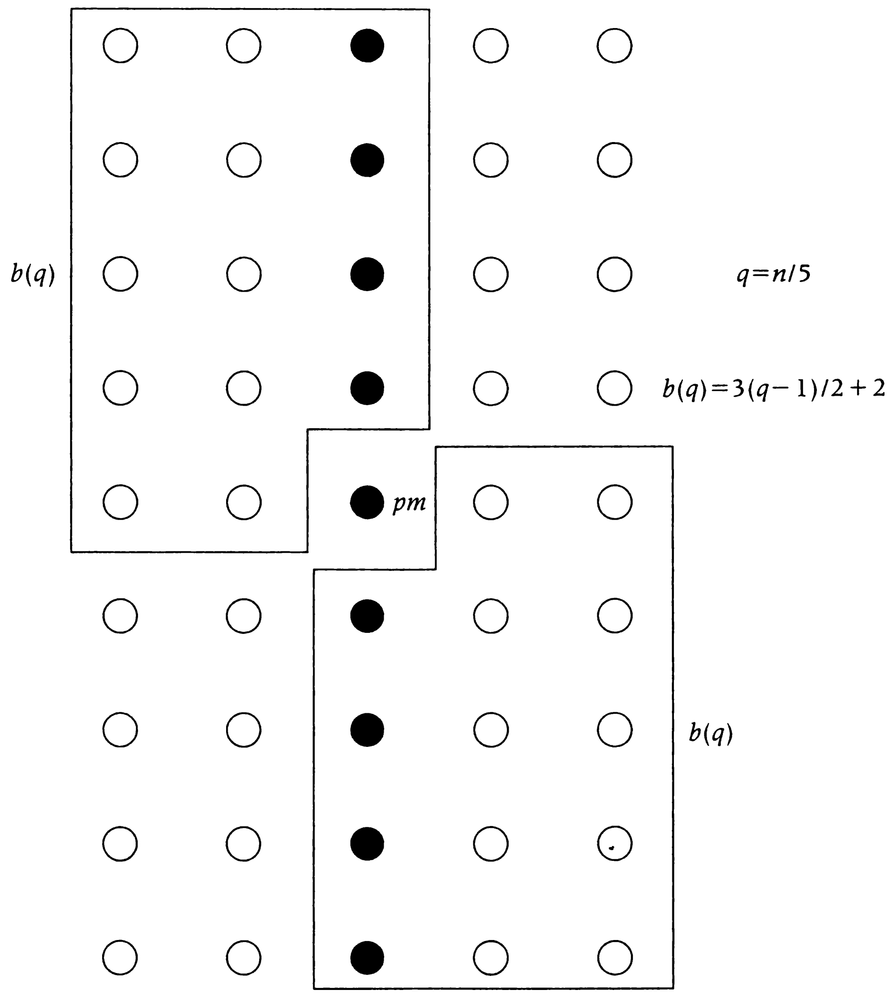
$$\begin{aligned} W(n) &= W(7n/10) + W(n/5) + 6n/5 + n + 1 \\ &= W(7n/10) + W(2n/10) + 11n/5 + 1, \end{aligned}$$

which is not exactly of the form $W(n) = W(rn) + f(n)$ stated in Proposition 8.5.1, but it is close enough to still yield linear worst-case complexity.

To compute the pseudomedian *pm*, we assume for convenience that n is an odd multiple of 5, where $n = 5q$ and q is odd. We then break up the list $L[0{:}n - 1]$ into $q = n/5$ sublists $L[0{:}4]$, $L[5{:}9]$, ... , $L[n - 5{:}n - 1]$, each of size 5, and determine the median of each sublist. The total number of comparisons required to determine these q medians $m_1, m_2, \ldots, m_q$ is $6n/5$, since the median of a five-element list can be found in six comparisons (see Exercise 8.20). The pseudomedian *pm* is the median of these q medians and is found by calling our recursive procedure *Select2* with the list of medians $m_1, m_2, \ldots, m_q$ and $k = (q + 1)/2$ (k is the median position of these medians). The worst-case complexity of finding *pm* is then $W(n/5) + 6n/5$. Now in the worst case, a recursive call to *Select2* is made with a list on one side or the other of *pm*. But as Figure 8.3 illustrates, *pm* must lie in the interval from $b(q)$ to $n - b(q) - 1$, where $b(q) = 3(q - 1)/2 + 2$ = the number of points inside each of the indicated "pseudorectangles." In Figure 8.3, the five-element sublists are arranged (in our mind's eye) so that each sublist is sorted in increasing order (from left to right) and so that the medians are also sorted in increasing order (from top to bottom). This is done for analysis purposes only and is *not* done by the algorithm *Select2*. Each list element in the upper-left pseudorectangle is strictly smaller than *pm*, and each element in the lower-right pseudorectangle is strictly larger than *pm*.

FIGURE 8.3

The pseudomedian *pm* must lie in the interval from $b(q)$ to $n - 1 - b(q)$.



Hence, the size of the sublist searched in the recursive call is at most

$$r(q) = n - b(q) - 1 = 5q - [3((q - 1)/2) + 2] - 1 = 7q/2 - 3/2 < 7n/10.$$

Thus, it follows that $W(n)$ for *Select2* satisfies the recurrence

$$W(n) \le W(7n/10) + W(2n/10) + 11n/5 + 1, \qquad \textbf{init. cond. } W(1) = 0.$$

A straightforward induction shows that $W(n) \le 24n$ (see Exercise 8.21).

The following pseudocode for the recursive procedure *Select2* follows from our discussion and is similar to that for the procedure *Select*. However, in *Select2* the procedure *Partition* is replaced by the procedure *Partition2*, which uses the input parameter *Pivot* as the pivot element (by interchanging $L[low]$ with $L[Pivot]$ and then invoking *Partition*). Also, the procedure *MedianOfFive* returns in the output parameter $M[0{:}q - 1]$ the medians of the five-element sublists of $L[low{:}high]$.

```
procedure Select2(L[0:n − 1], low, high, t, x) recursive
Input:   L[0:n − 1] (an array of distinct list elements), low, high (indices of L[0:n − 1])
                    t (a positive integer such that low ≤ t ≤ high)
Output:  x (the value that would appear in position t if L[low:high] were sorted in
            increasing order)
   if high − low + 1 ≤ 5 then
      use ad hoc method to find x
   endif
   q ← (high − low + 1)/5
   MedianOfFive(L[0:n − 1], low, high, M[0:q − 1])
   Select2(M[0:q − 1], 0, q − 1, (q − 1)/2, Pivot) //so that PseudoMedian = Pivot
   Partition2(L[0:n − 1], low, high, Pivot, position)
      case
         :t = position: x ← L[position]
         :t < position: Select2(L[0:n − 1], low, position − 1, t, x)
         :t > position: Select2(L[0:n − 1], position + 1, high, t, x)
      endcase
end Select2
```

8.6 Two Classical Problems in Computational Geometry

We now use the divide-and-conquer technique to solve two fundamental problems in computational geometry: the *closest-pair* problem and the *convex hull* problem. Although both problems can be posed for points in Euclidean

d-dimensional space for any $d \geq 1$, we limit our discussion of the solutions to these problems to the line and the plane ($d = 1$ and 2, respectively).

8.6.1 The Closest-Pair Problem

In the closest-pair problem, we are given n points $P_1, \ldots, P_n$ in Euclidean d-space, and we wish to determine a pair of points P_i and P_j such that the distance between P_i and P_j is minimized over all pairs of points drawn from $P_1, \ldots, P_n$. Of course, a brute-force quadratic complexity algorithm solving this problem is obtained by simply computing the distance between each of the $n(n-1)/2$ pairs and recording the minimum value so computed. Actually, to avoid round-off problems associated with taking square roots, it would be best to work with the square of the distance, which we assume throughout the discussion (and which, by abuse of language and notation, we still refer to as simply the distance d).

Using divide-and-conquer, we now design an $O(n\log n)$ algorithm for the closest-pair problem. This turns out to be order optimal because there is a $\Omega(n\log n)$ lower bound for the problem. We describe the algorithm informally because the actual pseudocode, though straightforward, is somewhat messy to fully describe.

To motivate the solution to the problem in the plane, we first consider the problem for points on the real line. Of course, we can solve the problem on the line by sorting the points using an $O(n\log n)$ algorithm, and then simply making a linear scan of the sorted points. However, this method does not generalize to the plane. To find a method that does generalize, let m be the median value of the n points. Using the algorithm *Select2* described in Section 8.5 we can compute m in linear time. We divide the points $x_1, \ldots, x_n$ into two subsets of equal size, X_1 and X_2, with one subset comprising points less than or equal to the median, and other subset comprising the remaining points (a linear scan determines the division). Let d_1 and d_2 be the minimum distances between pairs of points in X_1 and X_2, respectively. Either $d = \min\{d_1, d_2\}$ is also the smallest distance between any pair drawn from $x_1, \ldots, x_n$ or there is a pair $x_i \in X_1$ and $x_j \in X_2$ having a strictly smaller distance than d. However, it is clear that the interval $(m - d, m]$ contains at most one point of X_1, and the interval $(m, m + d]$ contains at most one point of X_2. In linear time, we can determine if the only possible pair having closer distance than d actually exists. Thus, for $n = 2^k$, the complexity $W(n)$ of the algorithm satisfies the recurrence.

$$W(n) = 2W(n/2) + n, \qquad \textbf{init. cond. } W(1) = 0,$$

which unwinds to yield $W(n) \in O(n\log n)$.

The divide-and-conquer solution just described for points on the real line generalizes naturally to a divide-and-conquer solution in the plane by dividing the

n points $(x_1,y_1), \dots, (x_n,y_n)$ into sets X_1 and X_2 on either side of line $x = m$, where m is the median of the x-coordinates of the points. The sets X_1 and X_2 can be determined with $O(n\log n)$ complexity by sorting the points by their x-coordinates. We then recursively find the minimum distances d_1 and d_2 between pairs of points in the sets X_1 and X_2, respectively. Again, either $d = \min\{d_1, d_2\}$ is also the smallest distance between any pair drawn from $(x_1,y_1), \dots, (x_n,y_n)$, or there is a pair $(x_i, y_i) \in X_1$ and $(x_j, y_j) \in X_2$ having a strictly smaller distance than d. Also, if there is such a pair, then (x_i, y_i) lies in the strip S_1 determined by the lines $x = m - d$ and $x = d$, whereas (x_j, y_j) lies in the strip S_2 determined by the lines $x = d$ and $x = m + d$. However, unlike the case for the line, we can no longer be sure that there is at most a single pair of points to examine. Indeed, *all* the points $(x_1,y_1), \dots, (x_n,y_n)$ might be in the strip $S = S_1 \cup S_2$ between the lines $x = m - d$ and $x = m + d$, so that we would have to examine a quadratic number of pairs, which is no better than the brute-force solution. However, the following proposition, whose proof we leave as an exercise, comes to our rescue.

Proposition 8.6.1 Consider a rectangle R in the plane of width d and height $2d$. There can only be at most six points in R such that the distance between each pair of these points is at least d. □

The following Key Fact follows from Proposition 8.6.1 and the definition of d.

Key Fact

For each point $(x, y) \in X_1 \cap S_1$, there are at most five points in $X_2 \cap S_2 \cap R$, where R is the rectangle with corner points $(x, y - d)$, $(x, y + d)$, $(x + d, y - d)$, $(x + d, y + d)$ (see Figure 8.4).

The Key Fact allows us to check less than $5n$ pairs to determine whether or not there is a pair $(x_i, y_i) \in X_1 \cap S_1$ and $(x_j, y_j) \in X_2 \cap S_2$ having distance less than d. To understand this, note that we can require our recursive calls determining X_1, d_1 and X_2, d_2 to return X_1 and X_2 sorted by their y-coordinates, which then can be merged using no more that $n - 1$ comparisons by the procedure *Merge* (discussed in Chapter 2). Thus, as we scan through the points in $X_1 \cap S_1$ in increasing order of their y-coordinates, a corresponding pointer can also scan the points in $X_2 \cap S_2$ in slightly oscillatory increasing order of their y-coordinates, checking at most $5n$ pairs. More precisely, suppose $P_1, \dots, P_m$ (respectively, $Q_1, \dots, Q_k$) are the points in $X_1 \cap S_1$ (respectively, in $X_2 \cap S_2$). We first scan $X_2 \cap S_2$ until we find a point (if any) such that d added to its y-coordinate is at least as large as P_1's y-coordinate. If we find such a point Q_i, then we leave a pointer Q at Q_i, and using Proposition 8.6.1, we need only check Q_i and the next four points $X_2 \cap S_2$ to determine if d needs updating. We then move to point P_2 and resume the scan of $X_2 \cap S_2$ starting at Q_i, this time looking for a point whose y-coordinate is at least as large as P_2's y-coordinate. We repeat this process until all of $X_1 \cap S_1$ is scanned, and less than $5n$ pairs of points will be examined for updates to the current smallest distance. Because we make at most $n - 1$ comparisons when merging X_1 and X_2, we see that the combine step in our divide-and-conquer algorithm performs less than $6n$ comparisons altogether. Hence, the worst case $W(n)$ of the algorithm satisfies

$$W(n) < 2W(n/2) + 6n, \qquad \textbf{init. cond. } W(1) = 0,$$

which shows that $W(n) \in O(n\log n)$.

When implementing this algorithm, there are various degenerate cases that have to be handled. For example, it might happen that some (or even all) of the points are on the line $x = m$. We leave the implementation details to the exercises.

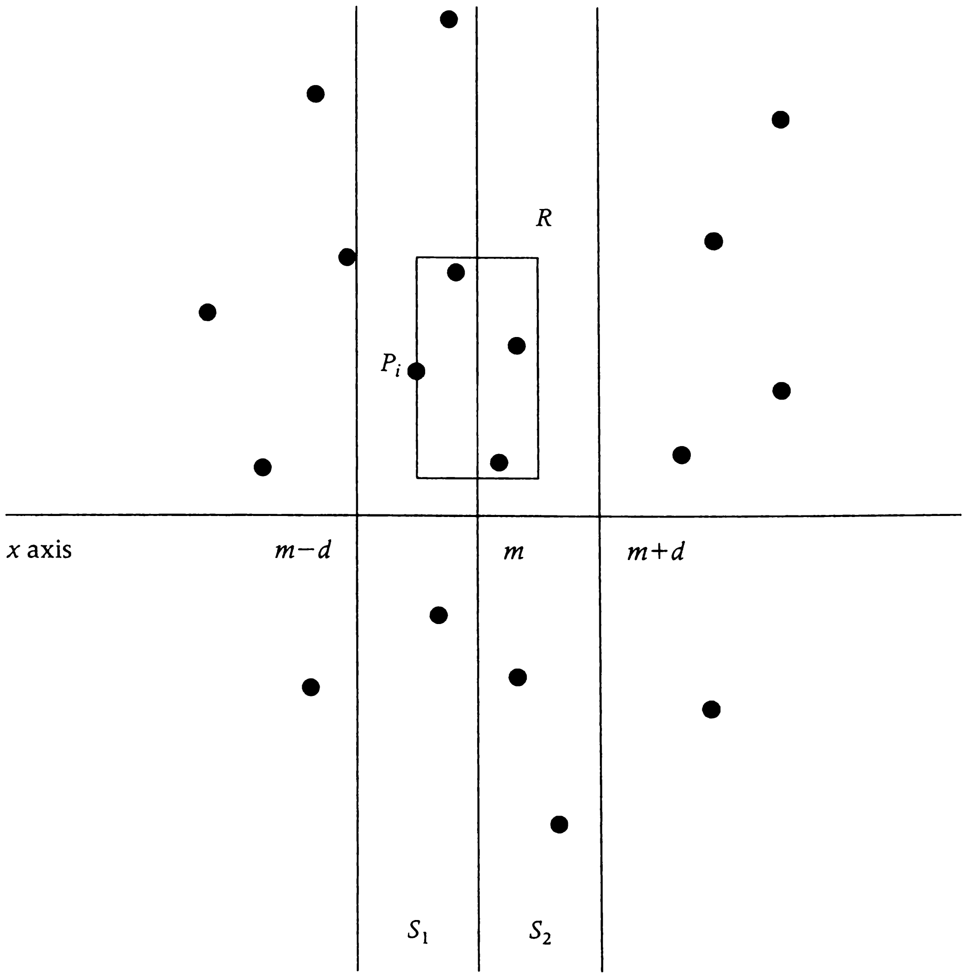


FIGURE 8.4

For $P_i \in X_1 \cap S_1$, rectangle R of width d and height $2d$ is shown where distances from P_i to points in $X_2 \cap S_2 \cap R$ need to be checked.

8.6.2 The Convex Hull Problem

Given any subset S of the Euclidean plane, S is called convex if, for each pair of points $P_1, P_2 \in S$, the line segment joining P_1 and P_2 lies entirely within the set S. Given a set of n points $(x_1, y_1), \ldots, (x_n, y_n)$ in the plane, the *convex hull* of these points is the smallest convex set containing them. In other words, the convex hull of the points is contained in any convex set containing the points. There is a nice physical interpretation of the convex hull. Consider placing pegs (or golf tees) at each of the n points $(x_1, y_1), \ldots, (x_n, y_n)$. Stretching a small rubber band around the entire set of points and releasing the band determines the boundary of the convex hull (see Figure 8.5).

Given the points $(x_1, y_1), \ldots, (x_n, y_n)$ in the plane, the convex hull problem is to determine a minimal subset of these points $P_1, \ldots, P_k, P_{k+1} = P_1$ such that the boundary of the convex hull of the points $(x_1, y_1), \ldots, (x_n, y_n)$ consists of the line segments joining P_i and P_{i+1}, $i = 1, \ldots, k$. Note that no three consecutive points in the (circular) list $P_1, \ldots, P_k, P_{k+1} = P_1$ are collinear.

The divide-and-conquer algorithm for computing the convex hull that we now describe begins in the same way as the closest-pair algorithm; namely, we divide the n points $(x_1, y_1), \ldots, (x_n, y_n)$ into sets X_1 and X_2 on either side of line $x = m$, where m is the median of the x-coordinates of the points. We then recursively determine the convex hulls $CH(X_1)$ and $CH(X_2)$ of X_1 and X_2, respectively. The initial condition for the recursion is a set of one, two, or three points, because the convex hull in these cases is a point, a line segment connecting the two points, and a triangle connecting the three points (unless they are collinear), respectively. It then becomes a question of how to merge the convex hulls $CH(X_1)$ and $CH(X_2)$ into the convex hull $CH(X_1 \cup X_2)$. The answer is to determine the upper and lower support line segments of $CH(X_1) \cup CH(X_2)$, as illustrated in Figure 8.6.

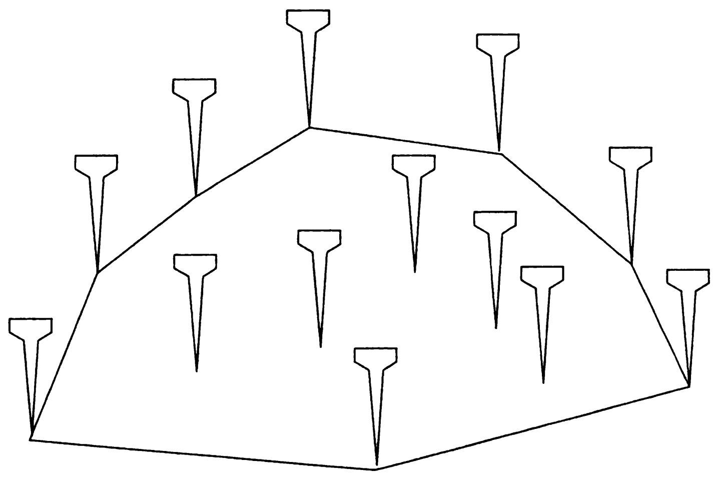
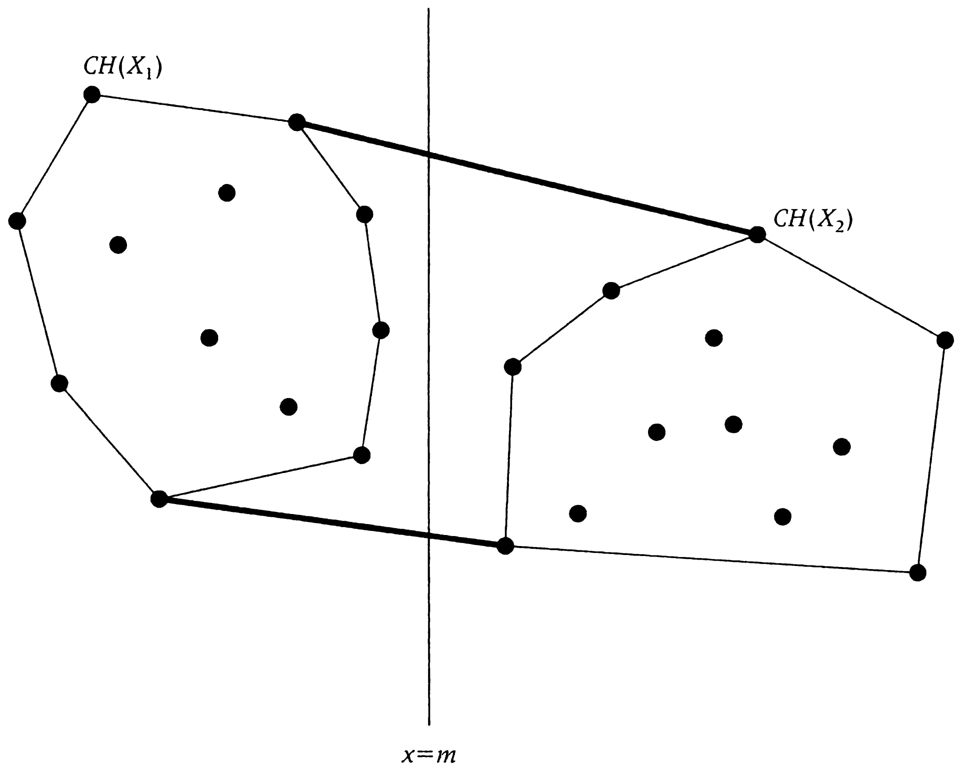


FIGURE 8.5

A small rubber band stretched and released around pegs at points in the plane determines the convex hull of these points.

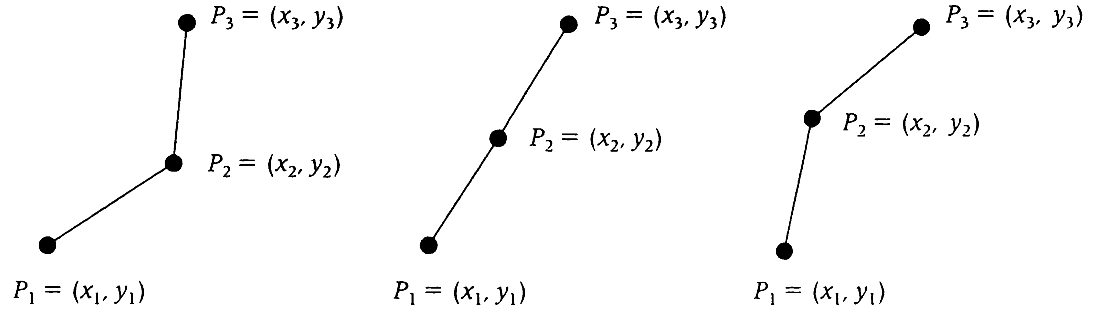
FIGURE 8.6

Upper and lower support line segments of $CH(X_1) \cup CH(X_2)$.



To determine the support line segments, we introduce the notion of a turn determined by an ordered triple of points (P_1, P_2, P_3) in the plane. We have a left turn, no turn, or right turn, respectively, as determined by the determinant shown in Figure 8.7. This determinant corresponds to twice the "signed area" determined by the triple of points (P_1, P_2, P_3).

We now use the notion of turns to determine the upper support line segment. Suppose $CH(X_1)$ is given by $P_1, \dots, P_k, P_{k+1} = P_1$ and $CH(X_2)$ is given by $Q_1, \dots, Q_m, Q_{m+1} = Q_1$, where both sequences are in clockwise order (that is, we make right turns as we pass through the sequences). Let P_i be the point in X_1 with the largest x-coordinate (if there are two such points, we take the point with the smaller y-coordinate). Consider the sequence of turns determined by $(P_i, Q_j, Q_{j+1}), j = 1, \dots, m$, where we set $Q_{m+1} = Q_1$ for convenience. Then the right-hand endpoint of the upper support line segment is the point Q_r, where the turns go from left turns or no turn to a right turn; that is, (P_i, Q_{r-1}, Q_r) is a left turn or no turn, and (P_i, Q_r, Q_{r+1}) is a right turn. Now consider the sequence of turns $(Q_r, P_j, P_{j-1}), j = 1, \dots, k$, where we set $P_0 = P_m$ for convenience (that is, we go around $CH(X_1)$ in counterclockwise order). Then the left-hand endpoint of the upper support line segment is the point P_s where the turns go from right turns or no turn to left turns. This determines the upper support segment. The lower support segment is obtained similarly.



(P_1, P_2, P_3) a left turn $\qquad$ (P_1, P_2, P_3) no turn $\qquad$ (P_1, P_2, P_3) a right turn

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} > 0 \qquad \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = 0 \qquad \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} < 0$$

FIGURE 8.7 The indicated determinant determines whether the triple (P_1, P_2, P_3) is a left turn, no turn, or right turn; that is, the turn depends on whether this determinant is positive, zero, or negative, respectively.

Given the two support line segments, it is a simple matter to eliminate the points in $CH(X_1) \cup CH(X_2)$ that are not in $CH(X_1 \cup X_2)$ and to return the sequence of points in $CH(X_1 \cup X_2)$ in clockwise order. It is also easy to verify that the algorithm has $O(n\log n)$ complexity. As with the closest-pair problem, when implementing the convex hull algorithm, various degenerate cases have to be handled. We leave the implementation details to the exercises.

Other $O(n\log n)$ algorithms for determining the convex hull are not based on divide-and-conquer. Two of the most commonly used of these are the Graham scan and the Jarvis march. The references at the end of the chapter provide a discussion of these and other convex hull algorithms. We point out there is a $\Omega(n\log n)$ lower bound for the convex hull problem in the plane because sorting can be reduced to this problem (see Exercise 8.32). Hence, the divide-and-conquer algorithm and the algorithms of Graham and Jarvis all have optimal order of complexity.

8.7 Closing Remarks

Since Strassen's result for matrix multiplication appeared in 1969 researchers have been trying to improve the basic method. They look for identities that perform fewer multiplications when multiplying matrices of a certain fixed size m and then use this reduction as the basis of a divide-and-conquer algorithm. The algorithm uses decomposition into m^2 blocks of size n/m, where, for convenience, it is assumed that $n = m^k$ for some positive integer k. It has been shown that for 2×2 matrices, at least seven multiplications are *required*. Thus, divide-and-conquer algorithms, which use as their starting point a method of multiplying two $m \times m$ matrices using less than the number of multiplications used by Strassen's method, require that m be larger than 2.

Nearly ten years after Strassen discovered his identities, Pan found a way to multiply two 70×70 matrices that involves only 143,640 multiplications (compared with more than 150,000 multiplications used by Strassen's method), yielding an algorithm that performs $O(n^{2.795})$ multiplications. Improvements over Pan's algorithm have been discovered, and the best result currently known (due to Coppersmith and Winograd) can multiply two $n \times n$ matrices using $O(n^{2.376})$ multiplications. However, all these methods require n to be quite large before improvements over Strassen's method are significant. Moreover, they are very complicated due to the large number of identities required to achieve the savings in the number of multiplications. Hence, the currently known order of complexity improvements over Strassen's algorithm are mostly of theoretical rather than practical interest.

The subject of computational geometry has a vast literature and is an active area of research. We refer you to the references at the end of this chapter for further reading on this important topic.

References and Suggestions for Further Reading

References to matrix multiplication algorithms:

Gonnet, G. H. *Handbook of Algorithms and Data Structures.* Reading, MA: Addison-Wesley, 1984.

Wilf, H. S. *Algorithms and Complexity.* Englewood Cliffs, NJ: Prentice-Hall, 1986.

Books devoted to computational geometry:

O'Rourke, J. *Computational Geometry in C.* New York: Cambridge University Press, 1994.

Preparata, F. P., and M. I. Shamos. *Computational Geometry: An Introduction.* New York: Springer-Verlag, 1985.

Handbooks containing material on computational geometry:

Goodman, J. E., and J. O'Rourke., eds. *Handbook of Discrete and Computational Geometry, 2nd edition.* Boca Raton, FLCRC Press, 2004.

Pach, J., ed. *New Trends in Discrete and Computational Geometry.* Vol. 10 of *Algorithms and Combinatorics.* New York: Springer-Verlag, 1993.

Survey articles on computational geometry:

Aurenhammer, F. "Voronoi Diagrams: A Survey of a Fundamental Geometric Data Structure." *ACM Computing Surveys* 23 (September 1991): 345-405.

Lee, D. T., and F. P. Preparata. "Computational Geometry: A Survey." *IEEE Transactions on Computers* C-33 (1984): 1072-1101.

EXERCISES

Section 8.1 The Divide-and-Conquer Paradigm

8.1 Design and analyze a divide-and-conquer algorithm for finding the maximum element in a list $L[0{:}n-1]$.

8.2 Design and analyze a divide-and-conquer algorithm for finding the maximum and minimum elements in a list $L[0{:}n-1]$.

8.3 Suppose we have a list $L[0{:}n-1]$ representing the results of an election, so that $L[i]$ is the candidate voted for by person i, $i = 0, \ldots, n-1$. Design a linear algorithm to determine whether a candidate got a majority of the votes—that is, whether there exists a list element that occurs more than $n/2$ times in the list. Your algorithm should output a majority winner if one exists. *Note:* One way to solve this problem is to determine the median value in the list, because that value is the only candidate for the majority value. This solution can be done in linear time using *Select 2* from Section 8.5. You are to design a different algorithm to solve the problem.

8.4 Design a version of *QuickSort* that uses a threshold. Do some empirical testing to determine a good threshold value.

Section 8.2 Symbolic Algebraic Operations on Polynomials

8.5 Give pseudocode and analyze the procedure *DirectPolyMult*, which is based directly on Formula (8.2.1).

8.6 Repeat Exercise 8.5 for a version of *DirectPolyMult* in which the polynomials are implemented by storing the coefficients in

a. an array

b. a linked list

c. Consider the sparse case.

8.7 Give pseudocode for a version of *PolyMult1* in which the polynomials are implemented by storing the coefficients and the associated powers of the polynomials in a linked list.

8.8 When analyzing *PolyMult1*, we choose coefficient multiplication as our basic operation. This ignores the two multiplications by powers of x. Show how the procedure for multiplying a polynomial by x^i can be implemented with linear complexity. Taking this operation into account, verify that the order of complexity of *PolyMult1* remains unchanged.

8.9 Show that the divide-and-conquer algorithm for multiplying two polynomials based on the recurrence relation (8.2.2) has quadratic complexity.

Section 8.3 Multiplication of Large Integers

8.10 Design and analyze an algorithm *MultInt* for multiplying large integers.

8.11 a. Design an algorithm for adding two large integers implemented using arrays.

b. Repeat part (a) for linked list implementations.

8.12 Give pseudocode for the algorithm *MultInt* you designed in Exercise 8.10 for the following two implementations of large integers.

a. arrays

b. linked lists

Section 8.4 Multiplication of Matrices

8.13 Verify Formula (8.4.4).

8.14 Verify Formula (8.4.11).

8.15 Verify that the evaluation of Formulas (8.4.10) and (8.4.11) together require 15 distinct additions or subtractions, which is 3 less than what is performed when using Strassen's identities.

8.16 Give pseudocode for the procedure *Strassen*, which implements Strassen's matrix multiplication algorithm. Assume that the input matrices A and B are both $n \times n$ matrices, where n is a power of 2.

8.17 Demonstrate the action of the procedure *Strassen* in Exercise 8.16 for the following input matrices:

$$A = \begin{bmatrix} 2 & 0 & 1 & 1 \\ 3 & 0 & 0 & 4 \\ 3 & -4 & 5 & 7 \\ 0 & 1 & 2 & 3 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 2 & 1 & 1 \\ 0 & 1 & 0 & -1 \\ 3 & 0 & 5 & 0 \\ 6 & 1 & 4 & 0 \end{bmatrix}$$

Section 8.5 Selecting the k^{th} Smallest Value in a List

8.18 a. Show that for a list of size n, *Select* makes at most n calls to *Partition* when determining the k^{th} smallest element.

b. For a given n and $t = k - 1$, find a list exhibiting worst-case behavior — that is, for which the size of the sublist containing the k^{th} smallest value of the original list is only reduced by one with each call to *Partition*.

8.19 Design and give pseudocode for an iterative version of the procedure *Select*.

8.20 Show that the median of five elements can be computed directly using six comparisons.

8.21 Consider the function $W(n)$ defined by the recurrence relation (8.5.9). Using induction, show that $W(n) \leq 24n$.

8.22 Design and give pseudocode for an iterative version of the procedure *Select2*.

8.23 Design a version of *Select2* that works for lists $L[0{:}n - 1]$ where the elements are not necessarily distinct

8.24 Prove that *Select* is correct.

8.25 Design an $O(\log k)$ divide-and-conquer algorithm for finding the k^{th} smallest (largest) element in a list $L[0{:}2n - 1]$, where the sublists $L[0{:}n - 1]$ and $L[n, 2n - 1]$ are in increasing order.

8.26 Design an algorithm *Partition2*(L[*low*:*high*], *Pivot*, *position*) that rearranges the input list about the element *Pivot* $\in L$[*low*:*high*] so that each element to the left of *Pivot* is no larger than *Pivot*, and each element to the right is no smaller than *Pivot*. The final position of *Pivot* in the rearranged list should be returned in the parameter *position*.

Section 8.6 Two Classical Problems in Computational Geometry

8.27 Prove that a rectangle of width d and height $2d$ can contain at most six points whose pairwise distances are at least d.

8.28 Write a computer program implementing the closest-pair algorithm.

8.29 Verify that the divide-and-conquer algorithm for the convex hull problem discussed in Section 8.6. has $O(n\log n)$ complexity.

8.30 Show that the convex hull of a set of points $P_1, P_2, \ldots, P_n$ in the plane consists of the set of all points $\lambda_1 P_1 + \lambda_2 P_2 + \cdots + \lambda_n P_n$, where $\lambda_1, \lambda_2, \ldots, \lambda_n$ are all nonnegative real numbers such that $\lambda_1 + \lambda_2 + \cdots + \lambda_n = 1$.

8.31 Write a computer program implementing the divide-and-conquer algorithm for the convex hull problem discussed in Section 8.6.

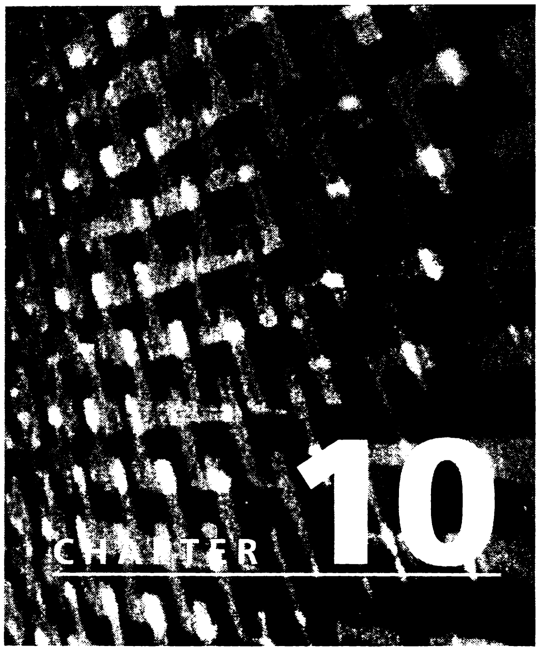
8.32 Show that a lower bound in $\Omega(n\log n)$ exists for the convex hull problem in the plane.

Section 8.7 Closing Remarks

8.33 Pan's divide-and-conquer matrix multiplication algorithm is based on a partitioning scheme that assumes n is a power of 70. The complexity $T(n)$ of Pan's divide-and-conquer algorithm satisfies the recurrence relation

$$T(n) = 143{,}640T(n/70), \quad n = 70^i, \quad i \geq 1, \qquad \textbf{init. cond. } T(1) = 1.$$

Show that this implies that $T(n)$ is approximately $n^{2.795}$.



BACKTRACKING AND BRANCH-AND-BOUND

Suppose you enter a maze of garden hedges, like that shown in Figure 10.1. To find the exit of the maze (or to determine that there is no exit), you need a guaranteed strategy. Does such a strategy exist? Yes. You simply walk along the trail with your left hand always touching the hedge. If you hit a dead end, you turn around and backtrack, still keeping your left hand touching the hedge. This left-hand backtracking strategy will always lead you to the exit, or return you to the entrance if there is no exit. It even works when you're blindfolded.

Of course, by symmetry, an algorithm for generating a path through a maze can also be based on the right-hand backtracking strategy. For the maze shown in Figure 10.1, the right-hand backtracking strategy generates a slightly shorter path. In general, however, neither backtracking strategy generates the shortest path. (In fact, Figure 10.1 provides an example.) Thus, although backtracking always gets you through the maze, some good heuristics might yield a shorter path (see Figure 10.2).

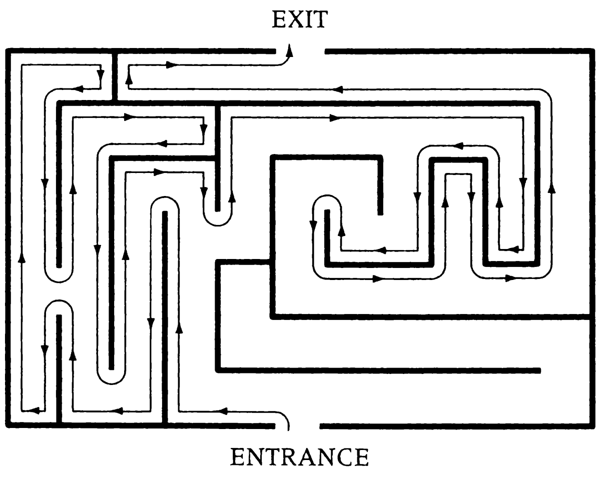


FIGURE 10.1

Garden maze and path to exit generated by the left-hand backtracking algorithm.

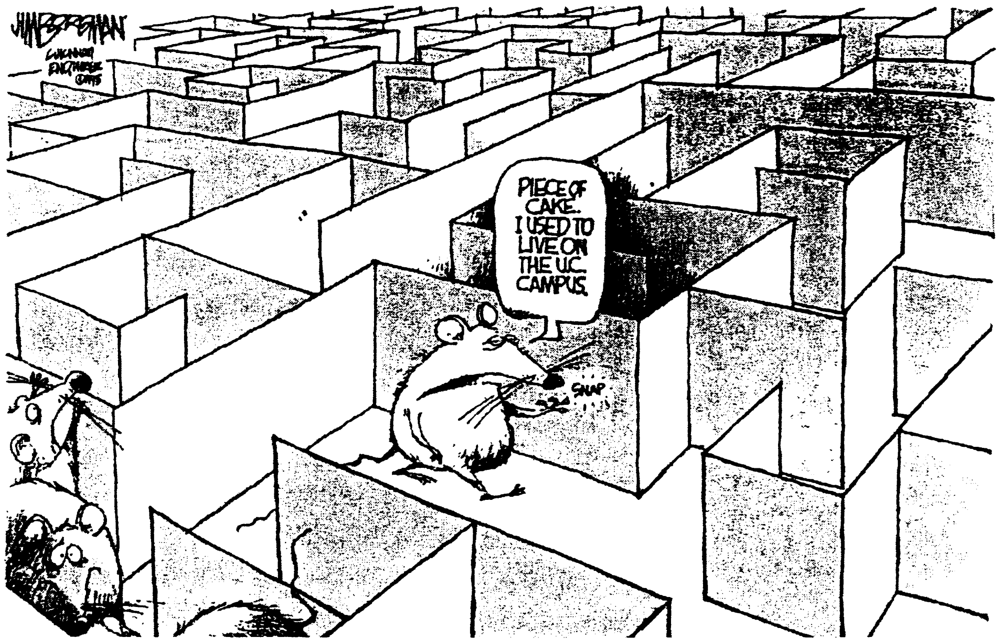


FIGURE 10.2

The left-hand backtracking algorithm versus heuristic search.

Source: Jim Borgman, © 1995 The Cincinnati Enquirer. *Reprinted with permission. All rights reserved.*

We can associate an implicit state-space *T* tree with the maze problem, where a node (or state) in *T* corresponds to a *sequence of decisions* made leading to a junction point in the maze. The children of a node correspond to the various branches from that junction point, where we restrict the choice of branches to those not leading to a junction already visited (this avoids cycles, yielding a finite tree). The left-hand backtracking algorithm is based on performing a depth-first search of the state-space tree.

10.1 State-Space Trees

The backtracking and branch-and-bound design strategies are applicable to any problem whose solution can be expressed as a sequence of decisions. Both backtracking and branch-and-bound are based on a search of an associated state-space tree that models all possible sequences of decisions. There may be several different ways to model decision sequences for a given problem, with each model leading to a different state-space tree. We assume in our model for the decision-making process that the decision x_k at stage k must be drawn from a finite set of choices. For each $k > 1$, the choices available for decision x_k may be limited by the choices that have already been made for $x_1, \dots, x_{k-1}$.

For a given problem instance, suppose n is the maximum number of decision stages that can occur. For $k \leq n$, we let P_k denote the set of all possible sequences of k decisions, represented by k-tuples $(x_1, x_2, \dots, x_k)$. Elements of P_k are called *problem states*, and problem states that correspond to solutions to the problem are called *goal states*.

Given a problem state $(x_1, \dots, x_{k-1}) \in P_{k-1}$, we let $D_k(x_1, \dots, x_{k-1})$ denote the *decision set* consisting of the set of all possible choices for decision x_k. Letting $\varnothing$ denote the null tuple ( ), note that $D_1(\varnothing)$ is the set of choices for x_1.

The decision sets $D_k(x_1, \dots, x_{k-1})$, $k = 1, \dots, n$, determine a decision tree T of depth n, called the *state-space tree*. The nodes of T at level k, $0 \leq k \leq n$, are the problem states $(x_1, \dots, x_k) \in P_k$ (P_0 consists of the null tuple). For $1 \leq k < n$, the children of $(x_1, \dots, x_{k-1})$ are the problem states $\{(x_1, \dots, x_k) \mid x_k \in D_k(x_1, \dots, x_{k-1})\}$.

A state-space tree that models a problem whose set of decision choices $D_k(x_1, \dots, x_{k-1})$ depends only on the input size is called *static*. A state-space tree in which D_k depends on not only the input size but also the particular input is called *dynamic*. For example, if we are solving the knapsack problem with objects $b_0, \dots, b_{n-1}$, a static state-space tree might be one in which the first decision is whether to include b_0, the second decision is whether to include b_1, and so forth (we are describing what we will call the static *fixed-tuple* state-space tree for the knapsack problem). On the other hand, based on some heuristic dependent on $b_0, \dots, b_{n-1}$, we might decide that our first decision is whether to include b_m, where m could be different from zero. Similar comments hold for other levels in the tree, leading to a dynamic state-space tree modeling the knapsack problem. In this chapter, we always use static state-space trees, but we do introduce a dynamic state-space tree in connection with the backtracking solution to the conjunctive normal form (CNF) satisfiability problem developed in the exercises.

10.1.1 An Example

Our first illustration of state-space trees is for the *sum of subsets problem.* An input to the sum of subsets problem is a multiset $A = \{a_0, \dots , a_{n-1}\}$ of n positive integers, together with a positive integer *Sum.* A solution to the sum of subsets problem is a subset of elements $a_{i_1}, \dots , a_{i_k}$ of A, $i_1 < \dots < i_k$, such that $a_{i_1} + \dots + a_{i_k} = Sum$.

The sum of subsets problem can be interpreted as the problem of making correct change, where a_i represents the denomination of the $(i + 1)^{st}$ coin, $i = 0, \dots , n - 1$, and *Sum* represents the desired change. This differs from the version of the coin-changing problem discussed in Chapter 7, because here a limited number of coins of each denomination are available. For example, consider the multiset $A = \{25, 1, 1, 1, 5, 10, 1, 10, 25\}$. The denominations are 1, 5, 10, 25, which occur with multiplicities 4, 1, 2, 2, respectively.

There are two natural ways to model a decision sequence leading to a solution to the sum of subsets problem. In the first model, a problem state consists of choosing k elements $a_{i_1}, \dots , a_{i_k}$ of A, $i_1 < \dots < i_k$, in succession, for some $k \in \{1, \dots , n\}$. The decision sequence can be represented by the k-tuple $(x_1, \dots , x_k) = (i_1, \dots , i_k)$, where x_j corresponds to the decision to choose element a_i at stage j, $1 \le j \le k$. For example, consider the instance $n = 5$, and suppose that we have decided to choose the second, fourth, and fifth elements. Then $x_1 = 1$, $x_2 = 3$, and $x_3 = 4$, so that the problem state associated with this decision sequence is the 3-tuple (1, 3, 4).

Given that problem state $(x_1, \dots , x_{k-1})$ has occurred (that is, the decision has been made to choose elements $a_{x_1}, \dots , a_{x_{k-1}}$), then the available choices for decision x_k are $a_{x_{k-1}+1}, \dots , a_n$, yielding

$$D_k(x_1, \dots , x_{k-1}) = \{x_{k-1} + 1, x_{k-1} + 2, \dots , n - 1\}, \quad 1 \le k \le n. \quad \textbf{(10.1.1)}$$

For example, suppose $n = 5$ and that problem state (0, 2) has occurred. The only elements available for the third decision are a_3 and a_4, so that $D_3(0, 2) = \{3, 4\}$. Figure 10.3 illustrates the state-space tree T determined by $D_k(x_1, \dots , x_{k-1})$ for the sum of subsets problem, where $n = 5$. The goal states (nodes) are not determined until a particular instance of the problem is specified. For example, for the instance $A = \{1, 4, 5, 10, 4\}$ and $Sum = 9$, the goal states are (0, 2, 4), (1, 2), and (2, 4). On the other hand, for the same set A, if $Sum = 10$, then the goal states are (0, 1, 2), (3), and (0, 2, 4). State-space trees like this, in which the size of the goal states can vary for the same input size, are called *variable-tuple* state-space trees.

The second natural way to model the sum of subsets problem is an example of a *fixed-tuple* model, in which goal states can be considered as n-tuples. In this

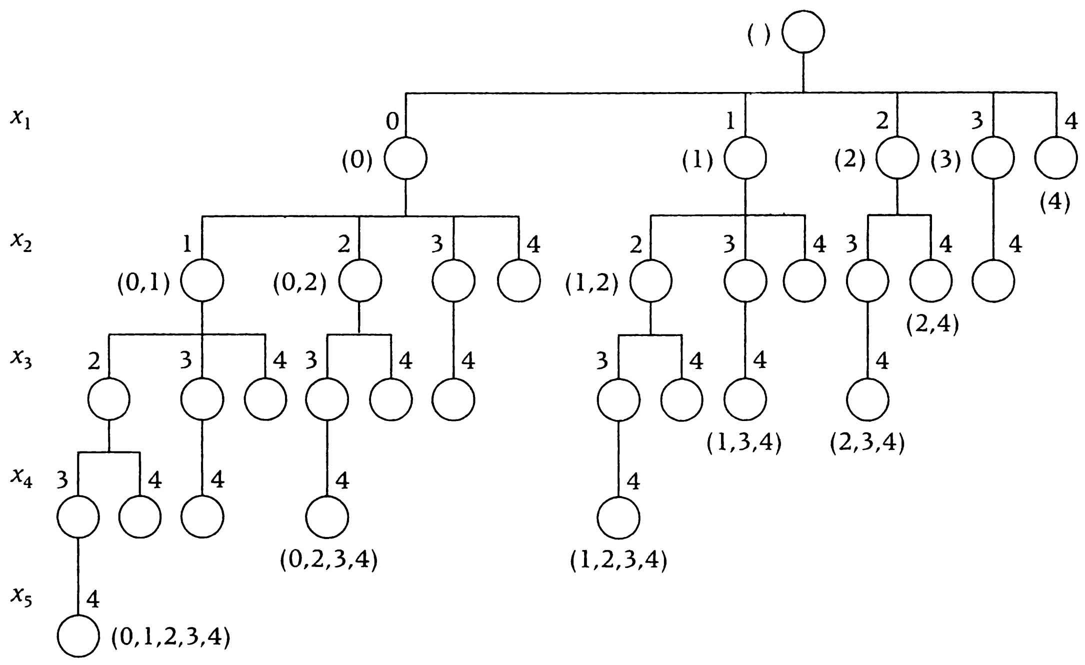


FIGURE 10.3

Variable-tuple state-space tree T modeling the decision set D_k given by Formula (10.1.1) for the sum of subsets problem with $n = 5$. Edges are labeled with the indices of the chosen elements. Index values of the problem states are shown outside some sample nodes.

model, the decision at stage k is whether to choose element a_{k-1}, $1 \le k \le n$. Thus, $D_k = \{0, 1\}$, where $x_k = 1$ if element a_{k-1} is chosen, and $x_k = 0$ otherwise. Thus, the state-space tree T associated with the decision sets $D_k(x_1, \ldots, x_{k-1})$ is the full binary tree on $2^{n+1} - 1$ nodes, with a left child of a node at level $k - 1$ corresponding to choosing a_{k-1} $(x_k = 1)$ and a right child corresponding to omitting a_{k-1} $(x_k = 0)$, so that

$$D_k(x_1, \ldots, x_{k-1}) = \{0, 1\}, \quad 1 \le k \le n. \qquad \textbf{(10.1.2)}$$

Figure 10.4 shows the fixed-tuple state-space tree for the same sum of subsets problem illustrated in Figure 10.3. For the same instance $A = \{1, 4, 5, 10, 4\}$ and $Sum = 9$ considered earlier, the goal states are now represented by the 5-tuples $(1, 1, 0, 0, 1)$, $(0, 1, 1, 0, 0)$, and $(0, 0, 1, 0, 1)$.

10.1.2 Searching State-space Trees

The state-space tree for most problems is large (exponential or worse in the input size). Thus, while a brute-force search of the entire state-space tree has the advantage of always finding a goal state if one exists, the search might not end in

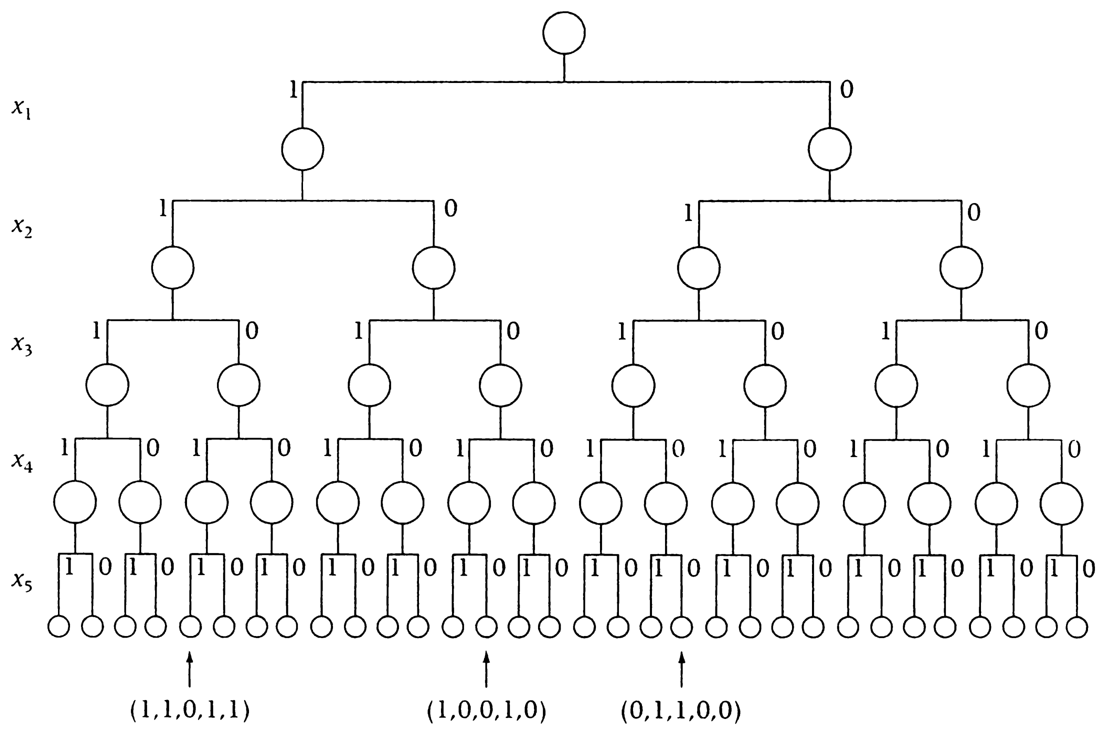


FIGURE 10.4

Fixed-tuple state-space tree T modeling the decision set D_k given by Formula (10.1.2) for the sum of subsets problem with $n = 5$. Edges ending at level k are labeled 1 or 0, depending on whether a_k was chosen or not. Labels of the path from the root to some sample leaf nodes shown.

a single lifetime, even for relatively small input sizes. However, we can often determine that there is no goal node in the subtree rooted at a given node X in the state-space tree. In this case, we say that X is *bounded*, and we can prune the state-space tree by eliminating the descendants of node X. Thus, when searching state-space trees, we look for good bounding functions. *Bounded* is a Boolean function such that if *Bounded*(X) is **.true.**, then there is no descendant of X that is a goal node. Good bounding functions can possibly limit the search to relatively small portions of the state-space tree.

Key Fact

An algorithm that performs a potentially complete search of a state-space tree modeling a given problem will always find a goal state if one exists. However, the state-space tree usually grows exponentially with the input size to the problem, so unless good bounding functions can be found to limit the search, such an algorithm will usually be too inefficient in the worst case to be practical.

In this chapter, we discuss two general-purpose design strategies based on searching the state-space tree associated with a given problem: *backtracking* and *branch-and-bound*. The state-space tree is usually implicit to backtracking algorithms, whereas branch-and-bound algorithms usually require the state-space tree to be explicitly implemented. Backtracking is based on a depth-first search of the state-space tree. When a node is accessed during a backtracking search, it becomes the current node being expanded (called the E-node), but immediately, its first child not yet visited becomes the new E-node. On the other hand, branch-and-bound algorithms are based on breadth-first searches of the state-space tree that generate all the children of the E-node when the node is first accessed. Thus, a node can be the E-node many times during a backtracking search, but a node is the E-node at most once during a branch-and-bound algorithm. There are various versions of branch-and-bound, differing only in the manner in which the next E-node is chosen. For example, the children might be placed on a queue (FIFO branch-and-bound), a stack (LIFO branch-and-bound), or a priority queue (least-cost branch-and-bound).

Key Fact

Backtracking is based on a depth-first search of the state-space tree *T*, and only needs to explicitly maintain the current path (or problem state) at any given point in the search. Branch-and-bound is based on a breadth-first search and normally needs to explicitly maintain the entire portion already reached in the search (except for nodes that are bounded).

As mentioned earlier, unless good bounding functions can be found, backtracking and branch-and-bound tend to be inefficient in the worst case. However, they can be applied in a wider variety of settings than the other major design strategies that we have discussed. Moreover, there are many practical and important problems for which the best solutions known are based on backtracking or branch-and-bound together with clever heuristics to bound the search. This is especially true for the NP-complete problems, such as the fundamental problem of determining the satisfiability of CNF Boolean expressions. The best-known solutions to CNF satisfiability are based on backtracking searches of dynamic state-space trees modeling the input as determined by clever heuristics and bounding strategies.

10.2 Backtracking

Before stating the general backtracking design strategy, we illustrate the method by applying it to the sum of subsets problem discussed in the previous section.

10.2.1 A Backtracking Algorithm for the Sum of Subsets Problem

Initially, we will not assume that the set $A = \{a_0, \ldots, a_{n-1}\}$ is ordered. (Later, we show that by sorting A in increasing order, we can obtain an improved bounding function.) We use the decision sequence formulation corresponding to the variable-tuple state-space tree, so that the decision sets $D_k(x_1, \ldots, x_{k-1})$ are given by Formula (10.1.1).

To motivate the definition of a bounding function for the problem states, we consider the instance of the sum of subsets problem where $n = 5$, $A = \{1, 4, 5, 10, 4\}$, and $Sum = 9$. The state-space tree for this instance, and the three goal states $(x_1, x_2, x_3) = (0, 1, 4)$, $(x_1, x_2) = (1, 2)$, and $(x_1, x_2) = (2, 4)$, are shown in Figure 10.5.

For example, consider the problem state (0, 1, 2) in the state-space tree in Figure 10.5 corresponding to the choice of elements a_0, a_1, a_2. Note that $a_0 + a_1 + a_2 = 10 > Sum = 9$. Further, any extension of (0, 1, 2) corresponds to a set of elements whose sum is even greater than 10. Thus, there is no path in the state-space tree from (0, 1, 2) to a goal state. In other words, *"You can't get there from here!"* (See Figure 10.6.) We *bound* node (0, 1, 2) because there is no need to examine any of its descendants.

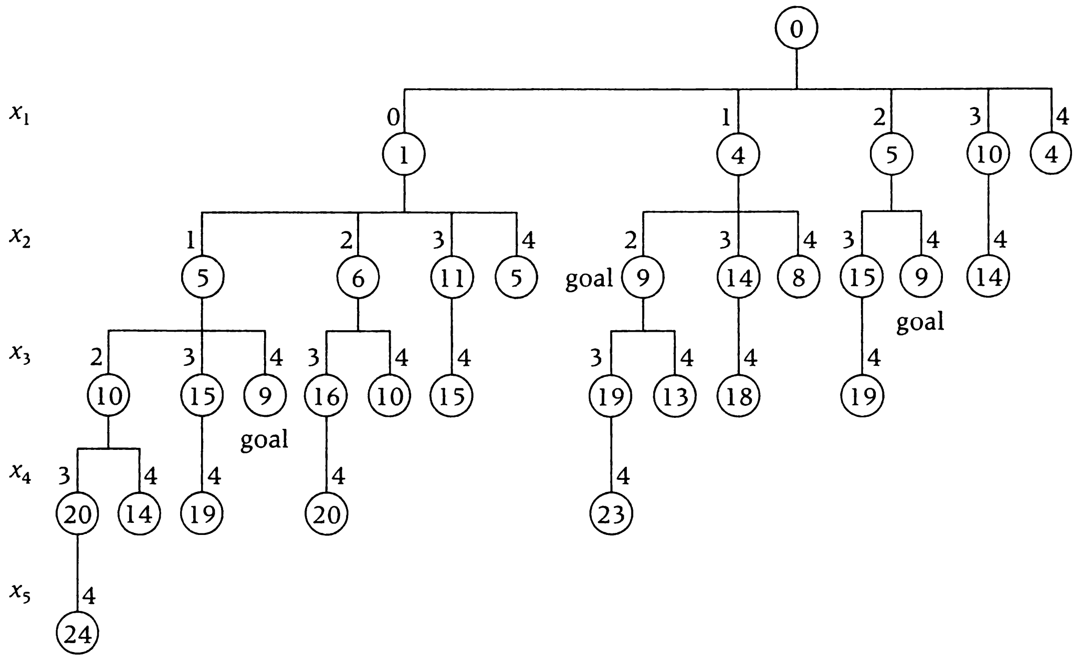


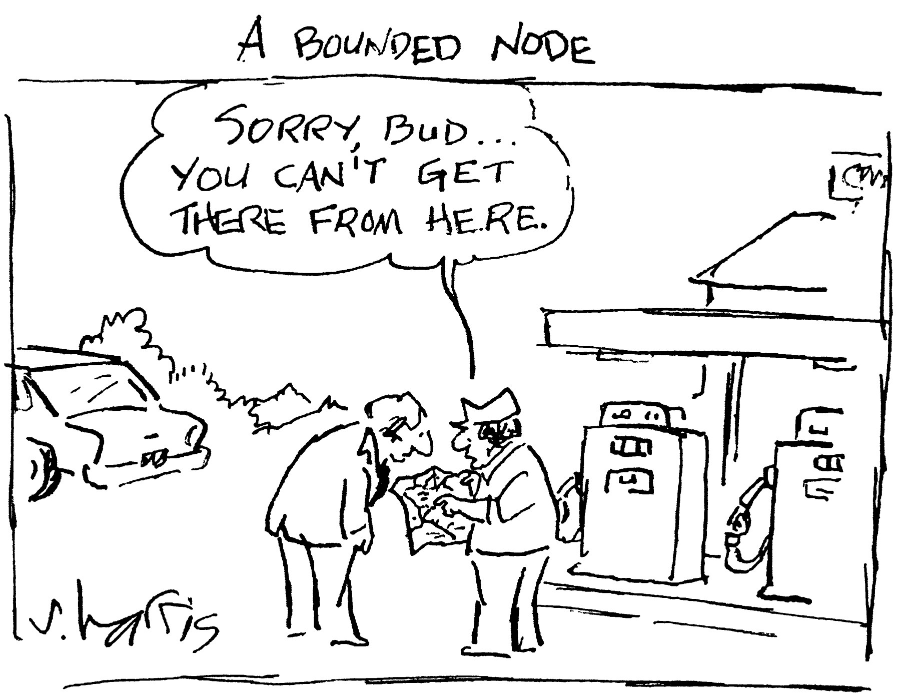
FIGURE 10.5

Variable-tuple state-space tree for sum of subsets problem with $A = \{1, 4, 5, 10, 4\}$ and $Sum = 9$. The value of the sum of the elements chosen is shown inside each node. Edges are labeled with the indices of the chosen elements.

FIGURE 10.6

A bounded node

Source: © 1996 by Sidney Harris. Altered and reprinted with permission. All rights reserved.



For the general problem state $(x_1, \dots, x_k) \in P_k$, we define the bounding function *Bounded*$(x_1, \dots, x_k)$ by

$$Bounded(x_1, \dots, x_k) = \begin{cases} \textbf{.true.} & \text{if } s_{x_1} + \cdots + s_{x_k} \geq Sum, \\ \textbf{.false.} & \text{otherwise.} \end{cases} \qquad \textbf{(10.2.1)}$$

Clearly, if the elements corresponding to $(x_1, \dots, x_k)$ have a sum greater than or equal to *Sum*, then any extension of $(x_1, \dots, x_k)$ corresponds to a set of elements whose sum is strictly greater than *Sum*. Thus, if *Bounded*$(x_1, \dots, x_k)$ = **.true.**, then no descendant of $(x_1, \dots, x_k)$ can be a goal state. For the problem instance $A = (1, 4, 5, 10, 4)$, Figure 10.7 shows the state-space tree T (from Figure 10.5) after it has been pruned at all the nodes bounded by Formula (10.2.1). The bounded nodes are labeled B, except the bounded nodes that are also goal nodes, which are so labeled. The unlabeled leaves correspond to nodes that were leaves in the original state-space T, before pruning.

The backtracking strategy performs a depth-first search of the state-space tree T, using an appropriate bounding function. By convention, when moving from an E-node to the next level of the state-space tree, we select the leftmost child not already visited. If no such child exists, or if the E-node is bounded, then we backtrack to the previous level. If only one solution to the problem is desired,

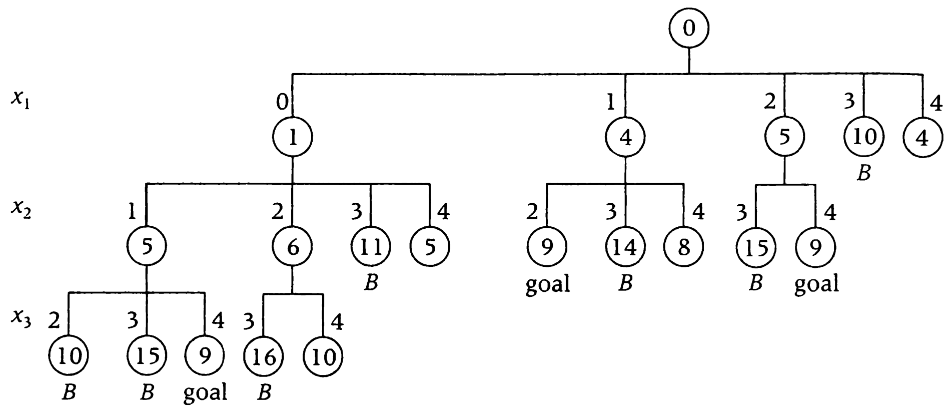


FIGURE 10.7

Pruned state-space tree for the sum of subsets problem with $A = \{1, 4, 5, 10, 4\}$ and $Sum = 9$. Edges are labeled with the indices of the chosen elements. Bounded nodes are labeled B.

then the backtracking algorithm terminates once a goal state is found. Otherwise, the algorithm continues until all the nodes have been exhausted, outputting each goal state when it is reached.

We now give pseudocode for nonrecursive and recursive backtracking procedures for solving the sum of subsets problem. These procedures perform a depth-first search of the variable-tuple state-space tree T, using the bounding function given in Formula (10.2.1). The procedures output all goal nodes but can be trivially modified to terminate once the first goal state is found.

```
procedure SumOfSubsets(A[0:n − 1], Sum, X[0:n])
Input:   A[0:n − 1] (an array of positive integers)
         Sum (a positive integer less than A[0] + ··· + A[n−1])
         X[0:n] (an array of integers where X[1:n] stores variable-tuple problem states,
         and X[0] = −1 for convenience of pseudocode)
Output:  print all goal states; that is, print all (X[1], ... , X[k]) such that
         A[X[1]] + ··· + A[X[k]] = Sum
   for i ← 0 to n do
      X[i] ← −1
   endfor
   PathSum ← 0
   k ← 1
   while k ≥ 1 do                  //E-node is (X[1], ... , X[k − 1]). Initially
                                   // E-node = ( ) corresponding to root
      ChildSearch ← .true.
      while ChildSearch do         // searching for unbounded child of E-node
         if X[k] = −1 then
            X[k] ← X[k − 1] + 1    // visit first child of E-node
         else
            PathSum ← PathSum − A[X[k]]
            X[k] ← X[k] + 1        //visit next child of E-node
            endif
            if $X[k] > n - 1$ then                // no more children of E-node
                ChildSearch ← .false.
            else
                PathSum ← PathSum + A[X[k]]
                if PathSum ≥ Sum then             //(X[1], … , X[k]) is bounded
                    if PathSum = Sum then
                        Print(X[1], … , X[k])     //print goal state
                    endif
                else                              //(X[1], … , X[k]) is not bounded
                    ChildSearch ← .false.
                endif
            endif
        endwhile
        if $X[k] > n - 1$ then
            X[k] ← - 1      //backtrack to previous level; no more children of E-node
            k ← k - 1
        else
            k ← k + 1       //go one more level deep in state-space tree
        endif
    endwhile
end SumOfSubsets
```

The following recursive backtracking algorithm *SumOfSubsetsRec*(k) for the sum of subsets problem is called initially with $k = 0$. We assume that $A[0{:}n - 1]$, *Sum*, and $X[0{:}n - 1]$ are global variables. When calling *SumOfSubsetsRec* with input parameter k, it is assumed that $X[1], \ldots, X[k]$ have already been assigned values, so that $k = 0$ on the initial call.

```
procedure SumOfSubsetsRec(k) recursive
Input:  k (a nonnegative integer, 0 on initial call)
        A[0:n - 1] (global array of positive integers)
        Sum (global positive integer less than A[0] + ⋯ + A[n-1])
        X[0:n] (an array of integers, where X[1:n] stores variable-tuple problem states,
        X[1], … , X[k] have already been assigned, and where X[0] = -1 for
        convenience of pseudocode)
        PathSum (global variable = A[X[1]] + ⋯ + A[X[k]], initialized to 0)
Output: print all descendant goal states of (X[1], ⋯ , X[k]); that is, print all (X[1], … ,
        X[k], X[k + 1], … , X[q]) such that A[X[1]] + ⋯ + A[X[k]] + A[X[k + 1]] +
        ⋯ + A[X[q]] = Sum
    k ← k + 1                                 // move on to next level
    Temp ← PathSum
```

```
        for Child ← X[k − 1] + 1 to n − 1 do
            X[k] ← Child
            PathSum ← Temp + A[X[k]]
            if PathSum ≥ Sum then                   //(X[1], ... , X[k]) is bounded
                if PathSum = Sum then
                    Print(X[1], ... , X[k])         // print goal state
                endif
            else                                    //(X[1], ... , X[k]) is not bounded
                SumOfSubsetsRec(k)
            endif
        endfor
    end SumOfSubsetsRec
```

If the elements $a_0, \dots, a_{n-1}$ are first sorted in increasing order (the reverse of the ordering used by the greedy algorithm for the coin-changing problem), then the following bounding function can be used for the problem states $(x_1, \dots, x_k)$. The bounding function (10.2.2) is stronger than that given by (10.2.1).

$$Bounded(x_1, \dots, x_k) = \begin{cases} \textbf{.true.} & \text{if } a_{x_1} + \cdots + a_{x_k} + a_{x_k+1} > Sum, \\ \textbf{.false.} & \text{otherwise} \end{cases} \qquad \textbf{(10.2.2)}$$

Because $a_0 \le a_1 \le \cdots \le a_{n-1}$, whenever the elements corresponding to problem state $(x_1, \dots, x_k, x_k + 1)$ have a sum strictly greater than *Sum*, then the elements corresponding to any problem state $(x_1, \dots, x_k, x_{k+1})$ also have a sum strictly greater than *Sum*. Thus, (10.2.2) is a valid bounding function. *SumOfSubsets* and *SumOfSubsetsRec* can be easily modified to use the bounding function given in 10.2.2

10.2.2 The General Backtracking Paradigm

The following general backtracking paradigm, *Backtrack*, follows the backtracking strategy we just described for the sum of subsets problem. *Backtrack* finds all solutions to a given problem by searching for all goal states in a state-space tree associated with the problem. *Backtrack* invokes a bounding function, *Bounded,* for the problem states. The definition of *Bounded* depends on the particular problem being solved. We assume that an implicit ordering exists for the elements of $D_k(x_1, \dots, x_{k-1})$.

```
procedure Backtrack()
Input:   T (implicit state-space tree associated with the given problem)
         $D_k$ (decision set, where $D_k = \varnothing$ for $k \geq n$)
         Bounded (bounding function)
Output:  all goal states
  $k \leftarrow 1$
  while $k \geq 1$ do                  //E-node is ($X[1], \ldots, X[k-1]$). Initially
                                         E-node = ( ) corresponding to root.
    Searching ← .true.
    while Searching do                 //searching for unbounded child
      $X[k] \leftarrow$ first of the remaining untried values from $D_k(X[1], \ldots, X[k-1])$,
            where this value is $\varnothing$ if all values in $D_k(X[1], \ldots, X[k-1])$ have
            been tried
      if $X[k] = \varnothing$ then
        Searching ← .false.
      else
        if ($X[1], \ldots, X[k]$) is a goal state then
          Print($X[1], \ldots, X[k]$)
        endif
        if .not. Bounded($X[1], \ldots, X[k]$) then
          Searching ← .false.
        endif
      endif
    endwhile
    if $X[k] = \varnothing$ then
      Arrange for all values in $D_k$ to be considered as untried
      $k \leftarrow k - 1$              //backtrack to previous level
    else
      $k \leftarrow k + 1$              //move on to next level
    endif
  endwhile
end Backtrack
```

The procedure *BacktrackRec* is the recursive version of the procedure *Backtrack*. Since we are essentially performing a depth-first search of the state-space tree T starting at the root, *BacktrackRec*(k) is initially called with $k = 0$. Note how elegantly the recursion implements the backtracking process. We assume that $D_k(X[1], \ldots, X[k])$ is empty for $k \geq n$.

```
procedure BacktrackRec(k) recursive
Input:   T (implicit state-space tree associated with the given problem)
         k (a nonnegative integer, 0 in initial call)
         $D_k$ (decision set, where $D_k = \varnothing$ for $k \geq n$)
```

```
                $X[0{:}n]$ (global array where $X[1{:}n]$ maintains the problem states of $T$, and where
                the problem state $(X[1], \dots, X[k])$ has already been generated)
                *Bounded* (bounding function)
Output: all goals that are descendants of $(X[1], \dots, X[k])$
    $k \leftarrow k + 1$
    for each $x_k \in D_k(X[1], \dots, X[k-1])$ do
        $X[k] \leftarrow x_k$
        if $(X[1], \dots, X[k])$ is a goal state then
            $Print(X[1], \dots, X[k])$
        endif
        if .not. $Bounded(X[1], \dots, X[k])$ then
            $BacktrackRec(k)$
        endif
    endfor
end BacktrackRec
```

REMARKS

1. Often, computing only one goal state is required. We can easily modify the procedures *Backtrack* and *BacktrackRec* to halt once the first goal state is reached.
2. Notice that in a slight variation from the pseudocode for procedures *Backtrack* and *BacktrackRec*, the *SumOfSubsets* and *SumOfSubsetsRec* pseudocode only checks for a goal state after determining that a problem state is bounded. Putting off this check for a goal state applies to any problem where all the goal states are bounded.

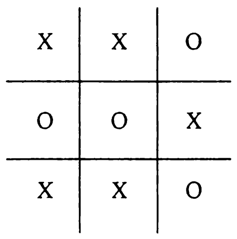
10.2.3 Tic-Tac-Toe

Consider the problem of finding a tie board (result of a "cat's game") in the familiar game of tic-tac-toe. Here we are trying to determine whether tie games are possible, not to devise a strategy for playing the game. (In Chapter 23, we consider the problem of designing strategies for various perfect-information games such as tic-tac-toe.)

The game of tic-tac-toe involves two players A and B, who alternately place Xs and Os into unoccupied positions on a board such as the one shown in Figure 10.8.

FIGURE 10.8

A 3 × 3 tic-tac-toe tie board



We assume that player A starts by placing an X in any position, and then player B places an O in any remaining position. The two players continue to alternately place Xs and Os on the board until either there are three Xs in a row (horizontally, vertically, or diagonally) so that A wins, there are three Os in a row so that B wins, or all nine positions are occupied with no three Os or three Xs in a row and the game is a tie (a cat's game).

We use backtracking to find a tie board — that is, a board corresponding to a tie game (see Figure 10.8). In fact, we solve the problem for an $n \times n$ board, where a *tie board* contains no "three in row" of either Xs or Os in any 3×3 sub-board of contiguous positions in the $n \times n$ board. In our definition of a tie board, we do not assume that the number of Xs and the number of Os differ by at most 1. However, it is interesting that all tie boards in the $n \times n$ board have this property, so we don't need to check for it in our backtracking algorithm.

We refer to the cell in row i and column j of the $n \times n$ board as cell (i, j), $i, j \in \{1, \ldots, n\}$. We also refer to cell (i, j) as the cell labeled k, where $k = n(i - 1) + j$; that is, k is a *row-major* labeling (see Figure 10.9). Row-major labeling of the cells is useful in defining the problem states P_k. However, when defining the bounding function, it is more convenient to use row-column labeling. The problem of finding a board filled with Xs and Os, containing no three in a row in either Xs or Os, can be expressed as a sequence of decisions in which the decision at stage k is whether to place an X or O in the cell labeled k. We set $x_k = 1$ if an X is placed

FIGURE 10.9

(a) Row-column labeling (i, j); (b) row-major labeling k; and (c) a 3×3 board configuration corresponding to the 7-tuple (1, 1, 0, 1, 0, 0, 1)

(1,1)	(1,2)	(1,3)
(2,1)	(2,2)	(2,3)
(3,1)	(3,2)	(3,3)

(a)

1	2	3
4	5	6
7	8	9

(b)

X	X	O
X	O	O
X		

(c)

in cell k in the k^{th} stage; otherwise, $x_k = 0$. Thus, $D_k = \{0, 1\}$, and the problem states of size k, $1 \le k \le n$, are given by

$$P_k = \{(x_1, \ldots, x_k) \mid x_1, \ldots, x_k \in \{0, 1\}\}. \qquad \textbf{(10.2.3)}$$

The (fixed-tuple) state-space tree T for tic-tac-toe is identical to the fixed-tuple state-space tree for the sum of subsets problem; namely, T is the full binary tree on $2^{n+1} - 1$ nodes. Figure 10.9*c* shows the 3 × 3 board configuration corresponding to the 7-tuple (1, 1, 0, 1, 0, 0, 1).

An obvious bounding function for tic-tac-toe is given by

$$\mathit{Bounded}(x_1, \ldots, x_k) = \begin{cases} \textbf{.true.} & \text{if board configuration corresponding} \\ & \text{to } (x_1, \ldots, x_k) \text{ contains 3 in a row,} \\ \textbf{.false.} & \text{otherwise.} \end{cases} \qquad \textbf{(10.2.4)}$$

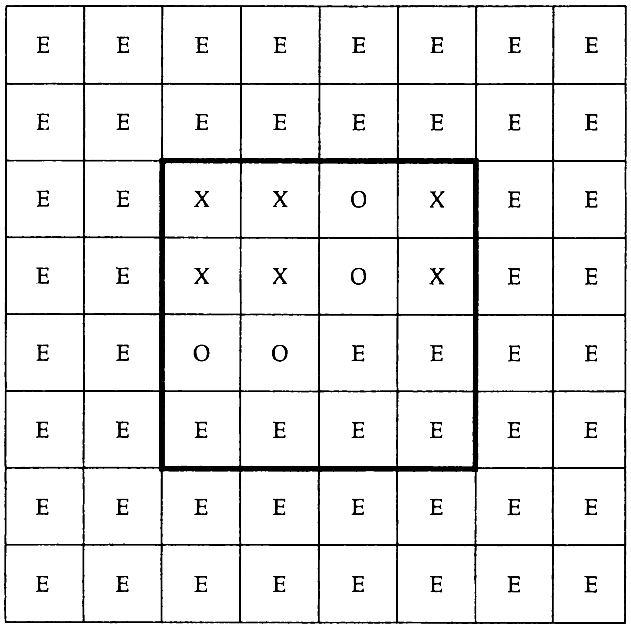
Note that *Bounded* = **.true.** for the problem state (1, 1, 0, 1, 0, 0, 1) in Figure 10.9. If the board configuration corresponding to the problem state $(x_1, \ldots, x_k)$ already contains three in a row (in either Xs or Os), then it obviously cannot be extended to a board configuration not containing three in a row. In particular, it cannot be extended to a goal state.

Before giving pseudocode for the algorithm *TicTacToe* solving the problem of finding an $n \times n$ generalized tie board configurations, we give pseudocode for the Boolean function *BoundedBoard* based on Formula (10.2.4). The board configuration is represented by the two-dimensional array $B[-1{:}n + 2, -1{:}n + 2]$, in which, for convenient implementation of *BoundedBoard* (and by abuse of notation), we assume the existence of "border" rows and columns indexed by -1, 0, $n + 1$, $n + 2$. The cells corresponding to these rows and columns are always empty; that is, $B[i, j]$ = "E" if either $i \in \{-1, 0, n + 1, n + 2\}$ or $j \in \{-1, 0, n + 1, n + 2\}$. We illustrate a bordered 4 × 4 board in Figure 10.10, together with an assignment of Xs and Os to the positions $k = 1, \ldots, 10$.

We assume that the two-dimensional array $B[-1{:}n + 2, -1{:}n + 2]$ is global to the function *BoundedBoard*. We refer to a set of three adjacent cells along a horizontal, vertical, or diagonal line as a *winning line*. Suppose the board configuration restricted to the cells labeled $1, 2, \ldots, k - 1$ in the row-major labeling of the 4 × 4 board contains no three in a row in either Xs or Os. Then, to determine whether the board configuration restricted to the cells labeled $1, 2, \ldots, k$ contains three in a row in either Xs or Os, we merely need to check all winning lines containing the cell labeled k. Four winning lines need to be checked: one horizontal, one vertical, and two diagonal. Figure 10.11 shows these lines for the cell in the board of Figure 10.10 whose row-major label is $k = 11$.

FIGURE 10.10

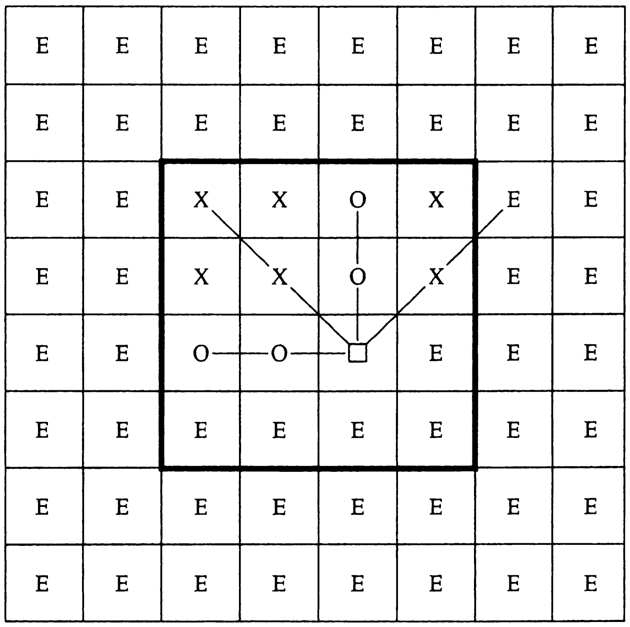
A bordered 4 × 4 tic-tac-toe board with first ten positions filled



E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E
E	E	X	X	O	X	E	E
E	E	X	X	O	X	E	E
E	E	O	O	E	E	E	E
E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E

FIGURE 10.11

Winning lines that are checked for k = 11 in a 4 × 4 × 4 tic-tac-toe board



E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E
E	E	X	X	O	X	E	E
E	E	X	X	O	X	E	E
E	E	O	O	□	E	E	E
E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E
E	E	E	E	E	E	E	E

The following Boolean function *BoundedBoard*(i, j) returns the value **.true.** if and only if the board configuration corresponding to $B[-1{:}n+2, -1{:}n+2]$ contains all *X*s or *O*s in one of the four winning lines previously described.

```
function BoundedBoard(i, j)
Input:   B[-1:n + 2, -1:n + 2]      (global array corresponding to board
                                     configuration)
         i, j                        (integers between 1 and n, inclusive)
Output:  returns .true. if the board configuration involving the cells labeled 1, ... ,
         k = n(i - 1) + j, contains three in a row in either Xs or Os along a line
         containing the cell labeled k.
   LineH ← (B[i, j] = B[i, j - 1]) .and. (B[i, j] = B[i, j - 2])
   LineV ← (B[i, j] = B[i - 1, j]) .and. (B[i, j] = B[i - 2, j])
   LineD1 ← (B[i, j] = B[i - 1, j - 1]) .and. (B[i, j] = B[i - 2, j - 2])
   LineD2 ← (B[i, j] = B[i - 1, j + 1]) .and. (B[i, j] = B[i - 2, j + 2])
   return(LineH .or. LineV .or. LineD1 .or. LineD2)
end BoundedBoard
```

We now give pseudocode for the algorithm *TicTacToe*. *TicTacToe* calls the procedures *Previous*(i, j) and *Next*(i, j), which accomplish the operations of backtracking to the previous cell ($k = k - 1$) and moving forward to the next cell ($k = k + 1$), respectively, in the row-major labeling. Thus, *Previous*(i, j) executes the statement

```
if j > 1 then
   j ← j - 1
else
   i ← i - 1
   j ← n
endif
```

and *Next*(i, j) executes the statement

```
if j < n then
   j ← j + 1
else
   i ← i + 1
   j ← 1
endif
```

```
procedure TicTacToe(n)
Input:    n (a positive integer representing size of board)
Output:   all generalized tie board configurations; that is, all board configurations not
          containing three in a row in either Xs or Os
   for i ← −1 to n + 2 do
      for j ← −1 to n + 2 do
         B[i, j] ← 'E';              //initialize all positions on board to empty
      endfor
   endfor
   i ← 1                             //k = 1
   j ← 1
   while i > 0 do
      if B[i, j] = 'O' then          //backtrack: k = k − 1
         B[i, j] ← 'E'
         Previous(i, j)
      else
         if B[i, j] = 'E' then
            B[i, j] ← 'X'            //visit left child of E-node
         else
            B[i, j] ← 'O'            //visit right child of E-node
         endif
         if .not. BoundedBoard(i, j) then
            if (i = n) .and. (j = n) then
               PrintBoard(Board[1:n, 1:n])      //print goal state
            else
               Next(i, j)
            endif
         endif
      endif
   endwhile
end TicTacToe
```

We can write a recursive version, *TicTacToeRec*(i, j), of *TicTacToe* as follows. *TicTacToeRec*(i, j) is initially called with $i = 0$ and $j = n$ $(k = 0)$. The augmented board *Board* is initialized to "E."

```
procedure TicTacToeRec(i, j) recursive
Input:    i, j  (integers between 1 and n, inclusive, called initially with i = 0 and j = n)
          B[−1:n + 2, −1:n + 2]         (global array corresponding to board
               configuration, initialized to "E," and B[1, 1], ... , B[i, j] filled with Xs and
               Os with no three in a row)
Output:   all extensions of B[1, 1], ... , B[i, j] to goal states; that is, board configurations
          not containing three in a row in either Xs or Os
```

```
        Next(i, j)                                  //k = k + 1
        for Child ← 1 to 2 do
            if Child = 1 then
                B[i, j] ← 'X'
            else
                B[i, j] ← 'O'
            endif
            if .not. BoundedBoard(i, j) then
                if (i = n) .and. (j = n) then
                    PrintBoard(Board[1:n, 1:n])     //print goal state
                else
                    TicTacToeRec(i, j)
                endif
            endif
        endfor
    end TicTacToeRec
```

10.2.5 Solving Optimization Problems Using Backtracking

Backtracking is frequently used to solve optimization problems—that is, to optimize (maximize or minimize) an objective function f over all goal states for a given problem. For example, for a sum of subsets problem interpreted as a coin-changing problem, we want to make correct change using the fewest coins (the objective function f is the number of coins). To do so, we use the following generic backtracking paradigm for solving the problem of minimizing the objective function (the paradigm is easily altered to solve maximization problems). Given an objective function f, let f^* denote the minimum of f over all solution states. A solution state X such that $f(X) = f^*$ is a goal state. Note that for *any* solution state $X = (x_1, \ldots, x_k)$, the value $f(X)$ is an upper bound for f^*. We maintain a variable *UB*, initialized to infinity. Additionally, at each stage of the backtracking algorithm, we maintain a solution state *CurrentBest* such that $UB = f(\mathit{Current Best})$ is the minimum value of f over all solution states generated so far. For many problems, we can efficiently compute a function $\mathit{LowerBound}(x_1, \ldots, x_k)$ that is not larger than the value of f on any solution state belonging to the subtree of the state-space tree rooted at $(x_1, \ldots, x_k)$. We can then dynamically bound a problem state $(x_1, \ldots, x_k)$ if $\mathit{LowerBound}(x_1, \ldots, x_k) \geq UB$. For example, in the coin-changing problem modeled on the variable-tuple state-space tree, $\mathit{LowerBound}(x_1, \ldots, x_k) = k$ if $(x_1, \ldots, x_k)$ is a goal state; otherwise, $\mathit{LowerBound}(x_1, \ldots, x_k) = k + 1$.

The generic backtracking paradigm for minimizing an objective function is based on the strategy just outlined. We describe the recursive version *Backtrack-*

MinRec of the paradigm, and leave the iterative version *BacktrackMin* as an exercise. The following high-level recursive procedure *BacktrackMinRec* is called initially with $k = 0$. During its resolution, *BacktrackMinRec* calls a function *StaticBounded*, which plays the same role as the function *Bounded* used for nonoptimization problems. For example, in the optimization version of the sum of subsets problem with input parameter *Sum*, a problem state is statically bounded if its sum was not smaller than *Sum*, whereas it is dynamically bounded if the cardinality of the subset corresponding to the problem state is at least as large as a previously generated solution state. In general, a problem state is either bounded dynamically ($LowerBound(x_1, \ldots, x_k) \geq UB$), or bounded statically ($Static\,Bounded(x_1, \ldots, x_k)$ = **.true.**).

```
procedure BacktrackMinRec(k, CurrentBest) recursive
Input:   T (implicit state-space tree associated with the given problem)
         $D_k$ (decision set, with $D_k = \varnothing$ for $k \geq n$)
         f (objective function defined on problem states)
         k (a nonnegative integer, 0 on initial call)
         X[0:n] (global array where X[1:n] maintains the problem states of T, and where
         the problem state (X[1], ... , X[k]) has already been generated)
         StaticBounded (a static bounding function on the problem states)
         LowerBound (a function defined on the problem states)
         CurrentBest (solution state extending (X[1], ... , X[k]) with the current
         minimum value of f over all descendants of (X[1], ... , X[k]) )
         UB (global variable, initialized to $\infty$)
Output:  CurrentBest (solution state extending (X[1], ... , X[k]) with the minimum value
         of f over all descendants of (X[1], ... , X[k]) )
   $k \leftarrow k + 1$
   for each $X[k] \in D_k(X[1], \ldots, X[k-1])$ do
      if (X[1], ... , X[k]) is a solution state then
         if $f(X[1], \ldots, X[k]) < UB$ then
            $UB \leftarrow f(X[1], \ldots, X[k])$
            $CurrentBest \leftarrow (X[1], \ldots, X[k])$
         endif
      endif
      if $LowerBound(X[1], \ldots, X[k]) < UB$ .and.
         .not. StaticBounded(X[1], ... , X[k]) then
         BacktrackMinRec(k, CurrentBest)
      endif
   endfor
end BacktrackMinRec
```

REMARK

The paradigm *BacktrackMinRec* ends up returning the first optimal goal state that was encountered. (Of course, unlike nonoptimization problems solved using backtracking, the algorithm has no way of checking that it was an optimal goal state until all goal states have been examined or eliminated.) If all optimal goal states are desired, then *CurrentBest* must maintain *all* the goal states that have the current minimum value of *f*. For example, the sum of subsets problem illustrated in Figure 10.12 has two optimal goal states, represented by the tuples (1, 2) and (2, 4). Procedure *SumOfSubsetsMinRec* only outputs the goal state (1, 2).

The algorithm *BacktrackMinRec* is easily altered to apply to optimization problems where we wish to maximize the objective function f. For such problems, LB is the current maximum value of f, and $UpperBound(x_1, \dots, x_k)$ is an upper bound for the maximum value of f over all solution states in the subtree of the state-space tree rooted at $(x_1, \dots, x_k)$. Alternatively, a problem involving maximizing an objective function f can be canonically transformed into an equivalent problem of minimizing the associated objective function $g = M - f$, where M is a suitable constant. By replacing f by $M - f$ in a maximization problem, *BacktrackMinRec* can be applied directly to solve both minimization and maximization problems. Of course, M can be taken as zero, but for a given problem a nonzero value of M might yield a natural interpretation for g. For example, for the 0/1 knapsack problem, if M is taken as the sum of the values of all of the input objects, then $M - f$ is the sum of the values of the objects left out of the knapsack.

As our first illustration of the use of *BacktrackMinRec*, we show how it can be directly translated into a solution for the optimization version of the sum of subsets problem, where goal states are solution states having minimum cardinality. Thus, UB is the smallest cardinality of a solution state currently generated. Note that a problem state $(x_1, \dots, x_k)$ corresponding to a subset of cardinality k can be dynamically bounded if k is not smaller than the current value of UB. Interpreting dynamic bounding in terms of the generic paradigm *BacktrackMinRec*, we see that $LowerBound(x_1, \dots, x_k) = k + 1$ if $x_1 + \cdots + x_k < Sum$; otherwise, $LowerBound(x_1, \dots, x_k) = k$.

```
procedure SumOfSubsetsMinRec(k, CurrentBest) recursive
Input:  k (a nonnegative integer, 0 on initial call)
        A[0:n − 1] (global array of positive integers)
        Sum (global positive integer less than A[0] + ··· + A[n − 1])
        X[0:n] (global array initialized to −1s. It is assumed that X[1], ... , X[k]
        representing a partial solution is already defined) CurrentBest (solution state
        (X[1], ... , X[m]) extending (X[1], ... , X[m]) such that m is minimum over all
        currently examined solution states that are descendants of (X[1], ... , X[k]) )
```

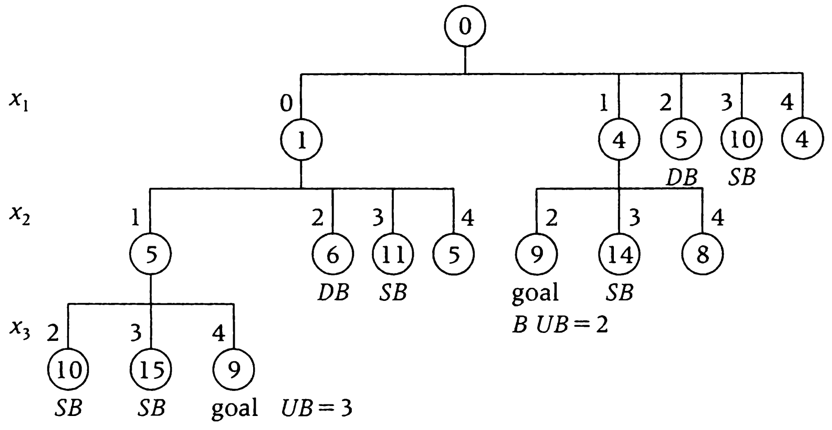
```
          PathSum (global variable = $A[X[1]] + \cdots + A[X[k]]$)
          UB (global variable, initialized to $\infty$)
Output:   CurrentBest (solution state $(X[1], \ldots, X[m])$ such that $m$ is a minimum over all
          solution states that are descendants of $(X[1], \ldots, X[k])$)
  $k \leftarrow k + 1$                          //go one level deeper in state-space tree
  $Temp \leftarrow PathSum$
  for $Child \leftarrow X[k-1] + 1$ to $n - 1$ do
     $X[k] \leftarrow Child$
     $PathSum \leftarrow Temp + A[X[k]]$
     if $Temp \geq Sum$ then                    //$(X[1], \ldots, X[k])$ is statically bounded
        if $Temp = Sum$ then                    //$(X[1], \ldots, X[k])$ is a solution state
           if $k < UB$ then
              $UB \leftarrow k$
              $CurrentBest \leftarrow (X[1], \ldots, X[k])$
           endif
        endif
     else                                       //$(X[1], \ldots, X[k])$ is not statically bounded
        if $k < UB$ then                        //$(X[1], \ldots, X[k])$ is not dynamically bounded
           SumOfSubsetsMinRec(k, CurrentBest)
        endif
     endif
  endfor
end SumOfSubsetsMinRec
```

The portion of the state-space tree generated by procedure *SumOfSubsets MinRec* is illustrated in Figure 10.12 for a sample set $A = \{1, 4, 5, 10, 4\}$ and $Sum = 9$.

SumOfSubsetsMinRec was written as a direct translation of *BacktrackMinRec*. However, since *BacktrackMinRec* is a generic paradigm, it is often the case that additional efficiencies might be possible when adapting it to a specific problem. Indeed, in the variable-tuple implementation of the optimization version of the sum of subsets problem, where a single optimal goal is to be output, when a goal state X is generated, there is no need to generate the remaining siblings of X because the subsets corresponding to these siblings cannot have smaller cardinality than the subset corresponding to X. To implement this improvement in the procedure *CoinChangingRec*, we need only break out of the **for** loop whenever a goal is generated (by simply adding a **break** statement after the assignment updating *CurrentBest*). For example, with this alteration, we would not generate the two siblings of the goal $(X[1] = 1, X[2] = 2)$ in Figure 10.12.

FIGURE 10.12

Portion of the state-space tree generated by procedure *SumOfSubsetsMinRec* for the set $A = \{1, 4, 5, 10, 4\}$ and $Sum = 9$. Statically bounded nodes are labeled *SB*. Nodes not statically bounded but dynamically bounded are labeled *DB*. Leaf nodes in the entire state-space tree are unlabeled, unless they are goals. Solution states resulting in updates to *UB* and *CurrentBest* are labeled with the updated value of *UB*. At termination, *SumOfSubsetsMinRec* outputs the final value *CurrentBest* = (1, 2).



Our next example, the 0/1 knapsack problem, illustrates the transformation of maximization problems into minimization problems. In Chapter 7, an efficient greedy algorithm was given for solving the knapsack problem. The greedy method does not necessarily yield an optimal solution to the 0/1 knapsack problem. In fact, there is no known worst-case polynomial algorithm for solving the 0/1 knapsack problem. However, the greedy algorithm for the knapsack problem helps us to define a useful function *LowerBound* for dynamic bounding in the transformed minimization problem.

Since the 0/1 knapsack problem involves looking at subsets of a set of size n, we may solve the problem using backtracking by searching the same state-space tree as in the sum of subsets problem. Consider the variable-tuple state-space tree determined by (10.1.1) (see Figure 10.3). An obvious static bounding function for the problem state $(x_1, \ldots, x_k)$ is given by

$$StaticBounded(x_1, \ldots, x_k) = \begin{cases} \textbf{.true.} & \text{if } w_{x_1} + \cdots + w_{x_k} \geq C, \\ \textbf{.false.} & \text{otherwise.} \end{cases} \qquad \textbf{(10.2.5)}$$

For $(x_1, \ldots, x_k)$ a problem state, let

$$\begin{aligned} Value(x_1, \ldots, x_k) &= \sum_{i=1}^{k} v_{x_i}. \\ Weight(x_1, \ldots, x_k) &= \sum_{i=1}^{k} w_{x_i}. \end{aligned} \qquad \textbf{(10.2.6)}$$

For the 0/1 knapsack problem, the solution states consist of all tuples not statically bounded — that is, all tuples $(x_1, \ldots, x_k)$ such that $Weight(x_1, \ldots, x_k) \leq C$

goal states maximize the objective function *Value* over all solution states. The maximization problem is transformed into a minimization problem by letting

$$M = \sum_{i=0}^{n-1} v_i, \qquad LeftOutVal(x_1, \dots, x_k) = M - Value(x_1, \dots, x_k) \tag{10.2.7}$$

In other words, $LeftOutVal(x_1, \dots, x_k)$ is the total value of all the objects left out of the knapsack corresponding to solution state $(x_1, \dots, x_k)$. In the transformed problem, which we now consider, the objective is to minimize the objective function *LeftOutVal*. To dynamically bound problem states, we maintain a variable *UB*, which at each stage of the backtracking algorithm keeps track of the minimum value of $LeftOutVal(x_1, \dots, x_k)$ over all the solution states generated so far.

Consider a solution state $(x_1, \dots, x_k)$ corresponding to the partial filling of the knapsack with the subset of objects $B_k = \{b_{x_1}, \dots, b_{x_k}\}$. Suppose $(x_1, \dots, x_k)$ is not bounded by Formula (10.2.6), and let $C' = C - Weight(x_1, \dots, x_k)$ denote the remaining capacity of the knapsack. Let $LeftOutVal^*(x_1, \dots, x_k)$ denote the minimum value of *LeftOutVal* over all solution states in the state-space subtree rooted at $(x_1, \dots, x_k)$. In other words, $LeftOutVal^*(x_1, \dots, x_k)$ is the smallest value of *LeftOutVal* that can be achieved by placing additional objects in the knapsack from the remaining set of objects $B' = \{b_{x_k+1}, \dots, b_{n-1}\}$. Clearly, if $LeftOutVal^*(x_1, \dots, x_k) \geq UB$, then $(x_1, \dots, x_k)$ can be dynamically bounded. Unfortunately, there is no known efficient method to compute $LeftOutVal^*(x_1, \dots, x_k)$. In fact, the general problem of computing $LeftOutVal^*(x_1, \dots, x_k)$ is equivalent to the original 0/1 knapsack problem.

Fortunately, we can efficiently compute a useful lower bound for $LeftOutVal^*(x_1, \dots, x_k)$ by applying the greedy algorithm *Knapsack*. For *B*, a given set of objects (with associated values and weights), and *C*, a given capacity for the knapsack, we let $Greedy(C, B)$ denote the value of the optimal placement of objects in the knapsack, where fractions of objects are permitted (that is, the value of the knapsack generated by *Knapsack*). We define $LowerBound(x_1, \dots, x_k)$ by

$$LowerBound(x_1, \dots, x_k) = LeftOutVal(x_1, \dots, x_k) - Greedy(C', B'). \tag{10.2.8}$$

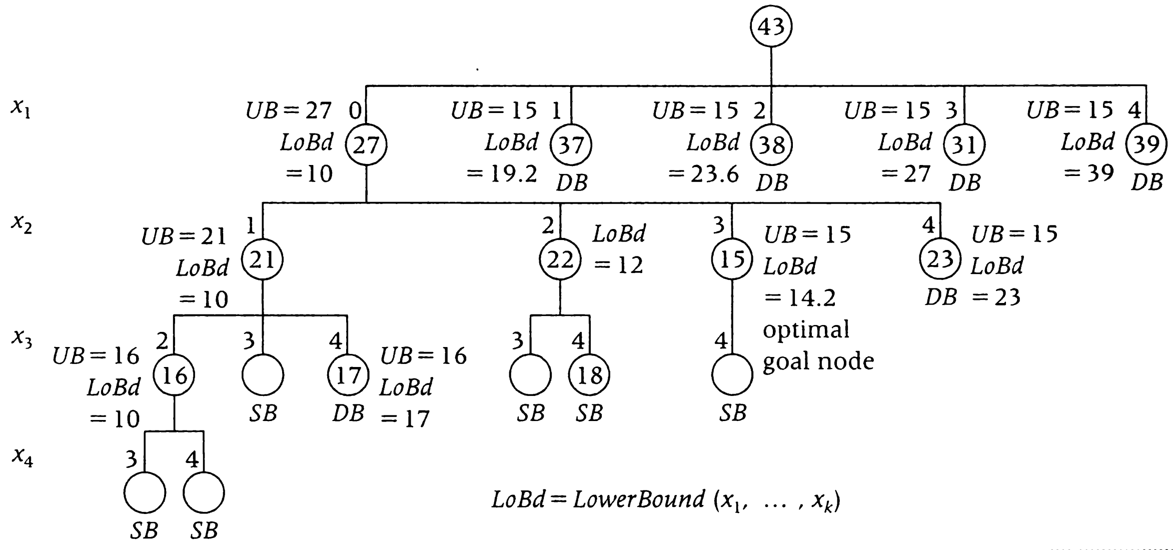
Clearly, we have

$$LeftOutVal^*(x_1, \dots, x_k) \geq LowerBound(x_1, \dots, x_k).$$

Thus, we can dynamically bound a problem state $(x_1, \dots, x_k)$ if $LowerBound(x_1, \dots, x_k) \geq UB$. Figure 10.13 shows the portion of the state-space tree *T* generated by backtracking for a sample instance of the 0/1knapsack problem.

i	0	1	2	3	4
v_i	16	6	5	12	4
w_i	4	2	2	6	5
v_i/w_i	4	3	2.5	2	.8

, $C = 11$
$M = 43$



43
x_1
UB = 27 0 LoBd = 10 27
UB = 15 1 LoBd = 19.2 37 DB
UB = 15 2 LoBd = 23.6 38 DB
UB = 15 3 LoBd = 27 31 DB
UB = 15 4 LoBd = 39 39 DB
x_2
UB = 21 1 LoBd = 10 21
2 22 LoBd = 12
3 15 UB = 15 LoBd = 14.2
4 23 DB UB = 15 LoBd = 23
x_3
UB = 16 2 LoBd = 10 16
3 SB
4 17 DB UB = 16 LoBd = 17
3 SB
4 18 SB
4 SB optimal goal node
x_4
3 SB
4 SB
LoBd = LowerBound (x_1, … , x_k)

FIGURE 10.13

Portion of the variable-tuple state-space tree T generated by the procedure *BacktrackMin* for a sample input to the 0/1 knapsack problem. Problem states that are statically bounded (*Weight*(x_1, … , x_k) ≥ *C*) are labeled *SB*, whereas problem states not statically bounded but dynamically bounded (*LowerBound* (x_1, … , x_k) ≥*UB*) are labeled *DB*. *LeftOutVal*(x_1, … , x_k) is shown inside each solution state. *LowerBound*(x_1, … , x_k) and the current value of *UB* are shown outside each problem state where *UB* is updated, or where the problem state is dynamically bounded.

10.3 Branch-and-Bound

Cowards die many times before their deaths;
The valiant never taste of death but once.
—Shakespeare, *Julius Caesar,* Act II, Scene II

As with backtracking algorithms, branch-and-bound algorithms are based on searches of an associated state-space tree for goal states. However, in a branch-and-bound algorithm, *all* the children of the *E*-node (the node currently being expanded) are generated before the next *E*-node is chosen. When the children are generated, they become *live* nodes and are stored in a suitable data structure, *LiveNodes*. *LiveNodes* is typically a queue, a stack, or a priority queue. Branch-and-bound algorithms using the latter three data structures are called *FIFO* (*first in, first out*) *branch-and-bound*, *LIFO* (*last in, first out*) *branch-and-bound*, and *least cost branch-and-bound*, respectively.

Immediately upon expansion, the current E-node becomes a *dead* node and a new E-node is selected from *LiveNodes*. Thus, branch-and-bound is quite different from backtracking, where we might backtrack to a given node many times, making it the E-node each time until all its children have finally been generated or the algorithm terminates. The nodes of the state-space tree at any given point in a branch-and-bound algorithm are therefore in one of the following four states: *E-node, live node, dead node,* or *not yet generated.*

As with backtracking, the efficiency of branch-and-bound depends on the utilization of good bounding functions. Such functions are used in attempting to determine solutions by restricting attention to small portions of the entire state-space tree. When expanding a given E-node, a child can be bounded if it can be shown that it cannot lead to a goal node.

We illustrate branch-and-bound by revisiting the sum of subsets problem, where the data structure *LiveNodes* is a queue. Such a branch-and-bound, called FIFO branch-and-bound, involves performing a breadth-first search of the state-space tree. Initially the queue of live nodes is empty. The algorithm begins by generating the root node of the state-space tree and enqueuing it in the queue *LiveNodes*. At each stage of the algorithm, a node is dequeued from *LiveNodes* to become the new E-node. All the children of the E-node are then generated. The children that are not bounded are enqueued (as they are generated from left to right). If only one goal state is desired, then the algorithm terminates after the first goal state is found. Otherwise, the algorithm terminates when *LiveNodes* is empty. Because of the nature of FIFO branch-and-bound, the first goal state found for the sum of subsets problems automatically has the smallest cardinality; that is, it solves the coin-changing problem.

Figure 10.14 illustrates FIFO branch-and-bound for the sum of subsets problem for the instance $A = (1, 11, 6, 2, 6, 8, 5)$ and $Sum = 10$. The action of the queue *LiveNodes* and the portion of the state-space tree generated in reaching the first goal state are given. For this instance of the sum of subsets problem, FIFO branch-and-bound generates fewer nodes of the state-space tree before reaching a goal state than are generated by backtracking.

Figure 10.15 illustrates LIFO branch-and-bound, where *LiveNodes* is a stack, for the same instance of the sum of subsets problem given in Figure 10.14. LIFO branch-and-bound is similar to backtracking, except that a move is made to the rightmost child of a node first instead of the leftmost. However, unlike backtracking, all the children of a node are generated before moving on.

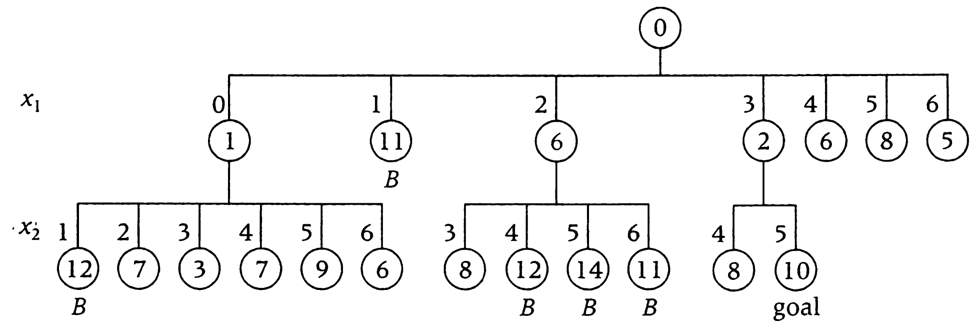
LIFO branch-and-bound generates fewer nodes than does FIFO branch-and-bound for the input considered in Figure 10.14, but for other inputs, the opposite can be true. In general, since FIFO branch-and-bound is based on a breadth-first search of the state-space tree, it is more efficient than LIFO branch-and-bound when goal nodes are not very deep in the state-space tree.

FIGURE 10.14

Action of queue *LiveNodes* and a portion of the variable-tuple state-space tree generated by FIFO branch-and-bound for the sum of subsets problem with $A = \{1, 11, 6, 2, 6, 8, 5\}$ and $Sum = 10$. The sum of the elements chosen is shown inside each node.

queue *LiveNodes*

generate ()	enqueue ()
dequeue	*E*-node = ()
generate (0)	enqueue (0)
generate (1)	bounded
generate (2)	enqueue (2)
generate (3)	enqueue (3)
generate (4)	enqueue (4)
generate (5)	enqueue (5)
generate (6)	enqueue (6)
dequeue	*E*-node = (0)
generate (0,1)	bounded
generate (0,2)	enqueue (0,2)
generate (0,3)	enqueue (0,3)
generate (0,4)	enqueue (0,4)
generate (0,5)	enqueue (0,5)
generate (0,6)	enqueue (0,6)
dequeue	*E*-node = (2)
generate (2,3)	enqueue (2,3)
generate (2,4)	bounded
generate (2,5)	bounded
generate (2,6)	bounded
dequeue	*E*-node = (3)
generate (3,4)	enqueue (3,4)
generate (3,5)	goal node



10.3.1 General Branch-and-Bound Paradigm

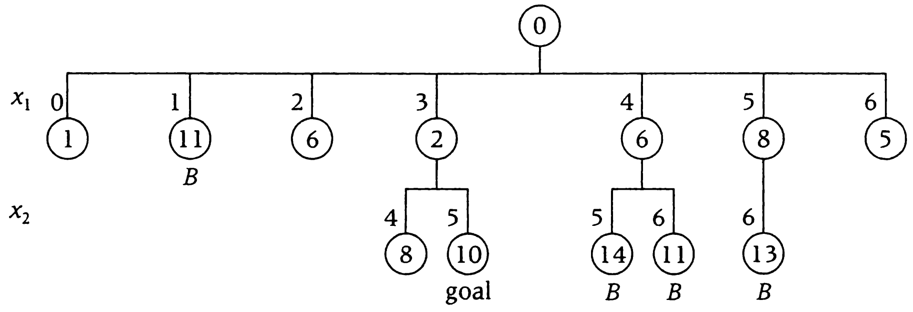
When we use backtracking, we do not explicitly implement the state-space tree. However, in branch-and-bound algorithms, we must explicitly implement the state-space tree and maintain the data structure storing the live nodes. In the general branch-and-bound paradigm *BranchAndBound*, the state-space tree *T* is implemented using the parent representation.

FIGURE 10.15

Action of queue *LiveNodes* and a portion of the variable-tuple state-space tree generated by LIFO branch-and-bound for the sum of subsets problem with $A = \{1, 11, 6, 2, 6, 8, 5\}$ and $Sum = 10$. The sum of the elements chosen is shown inside each node.

stack *LiveNodes*

generate ()	push ()
pop	E-node := ()
generate (0)	push (0)
generate (1)	bounded
generate (2)	push (2)
generate (3)	push (3)
generate (4)	push (4)
generate (5)	push (5)
generate (6)	push (6)
pop	E-node := (6)
pop	E-node := (5)
generate (5,6)	bounded
pop	E-node := 4
generate (4,5)	bounded
generate (4,6)	bounded
pop	E-node := (3)
generate (3,4)	push (3,4)
generate (3,5)	goal node



The nodes of T that are generated by paradigm *BranchAndBound* are represented as follows:

```
TreeNode = record
  Info:      InfoType
  Parent:    →TreeNode
end TreeNode
```

Only the value x_k need be stored in the information field *Info* of the node N corresponding to problem state $(x_1, \dots, x_k)$. Given a pointer *PtrNode* to N, the entire tuple $(x_1, \dots, x_k)$ is recovered by following the path in T from N to the root. For convenience, we denote $D_k(x_1, \dots, x_{k-1})$ by $D_k(PtrNode)$, where *PtrNode* is a pointer to the problem state $(x_1, \dots, x_{k-1})$.

The next E-node is chosen from the elements of *LiveNodes* by calling the procedure *Select*(*LiveNodes*, *E-node*, k), where *E-node* is a pointer to the E-node and k is the size of the E-node. The definition of procedure *Select* is dependent on the type of branch-and-bound being implemented. For example, *Select* may choose the next *E*-node from a queue *LiveNodes* (FIFO branch-and-bound), a stack *LiveNodes* (LIFO branch-and-bound), a priority queue *LiveNodes* (least cost branch-and-bound), and so forth.

Paradigm *BranchAndBound* adds a node to *LiveNodes*, by calling the procedure *Add*(*LiveNodes*, *PtrNode*). *BranchAndBound* also invokes the Boolean functions *Answer*(*PtrNode*) and *Bound*(*PtrNode*). *Answer*(*PtrNode*) assumes the value **.true.** if the node pointed to by *PtrNode* is a goal state. *Bound*(*PtrNode*) returns the value **.true**. if the node pointed to by *PtrNode* is bounded. Similar to backtracking, the definition of the bounding function depends on the particular problem being solved.

```
procedure BranchAndBound
Input:   function $D_k(x_1, \ldots, x_{k-1})$ determining state-space tree $T$ associated with the
         given problem)
         Bounding function Bounded
Output:  All goal states to the given problem
   LiveNodes is initialized to be empty
   AllocateTreeNode(Root)
   Root→Parent ← null
   Add(LiveNodes, Root)               //add root to list of live nodes
   while LiveNodes is not empty do
      Select(LiveNodes, E-node, k)    //select next E-node from live nodes
      for each $X[k] \in D_k$(E-node) do  //for each child of the E-node do
         AllocateTreeNode(Child)
         Child→Info ← X[k]
         Child→Parent ← E-node
         if Answer(Child) then        //if child is a goal node then
            Path(Child)               //output path from child to root
         endif
         if .not. Bounded(Child) then
            Add(LiveNodes, Child)     //add child to list of live nodes
         endif
      endfor
   endwhile
end BranchAndBound
```

As with the general paradigm *Backtrack* there is a corresponding version of the general paradigm *BranchAndBound* for problems involving minimizing or maximizing objective functions. For example, the paradigm *BranchAndBoundMin* maintains in *CurrentBest* the solution state with the current minimum value of the objective function *f*, and the value *f*(*CurrentBest*) is used to dynamically bound nodes. Again, this dynamic bounding is done using a suitable function, *LowerBound*, whose value at a given node *X* is a lower-bound estimate of the value of the objective function at all goal states in the subtree rooted at *X*. A node *X* can be (dynamically) bounded if $f(CurrentBest) \le LowerBound(X)$. As with the 0/1 knapsack problem discussed earlier, the function *LowerBound*(*X*) is often expressed in the form $f(X) + h(X)$, where $h(X)$ is a lower-bound estimate of the smallest incremental increase in *f* incurred in going from *X* to a descendant goal state. Sometimes $h(X)$ is a heuristic estimate that might not be provably a lower bound, but which nevertheless has been shown to work well in practice. *LiveNodes* is often maintained as a priority queue, where *LowerBound*(*X*) is taken as the priority of a node *X*. The next *E*-node chosen by *Select* is the node in *LiveNodes* with the least value of *LowerBound*, and the strategy is called *least cost* branch-and-bound. A more detailed discussion of least cost branch-and-bound will be given in the Chapter 23, where it is shown to be a special case of a more general search strategy.

10.4 Closing Remarks

Both LIFO and FIFO branch-and-bound are blind searches of the state-space tree *T* in the sense that they search the nodes of *T* in the same order regardless of the input to the algorithm. Thus, they tend to be inefficient for searching the large state-space trees that often arise in practice. Using heuristics can help narrow the scope of otherwise blind searches. The least cost branch-and-bound strategy discussed above uses a heuristic cost function associated with the nodes of the state-space tree *T*, where the set of live nodes is maintained as a priority queue with respect to this cost function. In this way, the next node to become the *E*-node is the one that is the most promising to lead quickly to a goal.

Least cost branch-and-bound is closely related to the general heuristic search strategy called A*-search. A*-search can be applied to state-space digraphs (digraphs are discussed in Chapter 11), rather than just state-space trees. A*-search is one of the most commonly used search strategies in artificial intelligence. Both A*-search and least cost branch-and-bound are discussed in Chapter 23.

The backtracking and branch-and-bound strategies are well suited to parallelization because different portions of the state-space tree can be assigned to different processors for searching. In Chapter 18, we discuss a general parallel backtracking paradigm in the context of message-passing distributed computing; and in Appendix F, we give code for an MPI implementation for the optimization version of the sum of subsets problem.

References and Suggestions for Further Reading

Golumb, S., and L. Baumert. "Backtracking Programming." *Journal of the ACM* 12 (1965): 516–524. An early general description, and applications, of the backtracking method.

Walker, R. J. "An Enumerative Technique for a Class of Combinatorial Prob lems." *Proceedings of Symposia in Applied Mathematics.* Vol. X. Providence, RI: American Mathematical Society, 1960. The backtracking and branch-and-bound design strategies have been studied for a long time. (The name backtrack was coined by D. H. Lehmer in the 1950s.) Walker's article is one of the first accounts of the backtracking method.

For two early survey articles on the branch-and-bound paradigm, see:

Lawler, E. L., and D. W. Wood. "Branch-and-Bound Methods: A Survey." *Operations Research* 14 (1966): 699–719.

Mitten, L. "Branch-and-Bound Methods: General Formulation and Properties." *Operations Research* 18 (1970): 24–34.

EXERCISES

Section 10.1 State-Space Trees

10.1 Consider the backtracking solution to the following instance of the 0/1 knapsack problem. The capacity of knapsack = $C = 15$.

i	0	1	2	3	4	5	6
v_i	25	45	12	7	6	10	5
w_i	5	11	3	2	2	7	4

a. Give P_k, and $D_k(x_1, x_2, \dots, x_{k-1})$.

b. Draw the variable-tuple state-space tree (first three levels).

10.2 Repeat Exercise 10.1 for the fixed-tuple state-space tree.

10.3 Show that the number of nodes of both the fixed-tuple and variable-tuple state-space trees for the sum of subsets problem are exponential in n.

Section 10.2 Backtracking

10.4 Modify the pseudocode for the procedure *SumOfSubsets* to use the bounding function given in 10.2.2

10.5 a. Give pseudocode for a nonrecursive backtracking procedure for solving the sum of subsets problem, based on the state-space tree given in Figure 10.4.

b. Repeat part (a) for a recursive backtracking procedure.

10.6 a. Reformulate procedure *Backtrack* so it halts once the first goal is reached.

b. Repeat part (a) for the procedure *BacktrackRec*.

10.7 Give pseudocode for versions of the procedures *Backtrack* and *BacktrackRec* that only check for a goal state after determining that a problem state is bounded. These modified algorithms only apply to problems in which all the goal states are bounded.

10.8 Give pseudocode for a nonrecursive version of the paradigm *BacktrackMin* for minimizing an objective function.

10.9 Show the portion of the state-space tree generated during backtracking for the instance of the 0/1 knapsack problem given in Exercise 10.1 using the bounding functions (10.2.5) and (10.2.8).

10.10 a. Give pseudocode for a backtracking algorithm that solves the 0/1 knapsack problem using the bounding functions (10.2.5) and (10.2.8).

b. Write a program implementing your algorithm in part (a), and run the program for various inputs.

10.11 a. Write a program using backtracking that proves there are no tie boards for the $3 \times 3 \times 3$ tic-tac-toe game, even if we relax the condition that the number of Xs and the number of Os differ by one.

b. Write a program using backtracking that outputs all tie boards for the $3 \times 3 \times 3$ board minus the center position (when playing the game of tic-tac-toe, no X or O is placed in the center position), where we relax the condition that the number of Xs is equal to the number of Os. Is there a tie board where the number of Xs equals the number of Os?

10.12 Two queens in the ordinary chessboard are *nonattacking* if they are not in the same row, column, or diagonal. A classical problem known as the *8-queens problem* is to place eight queens on the board so that each pair of queens is nonattacking. One solution to the 8-queens problem is shown in Figure 10.16.

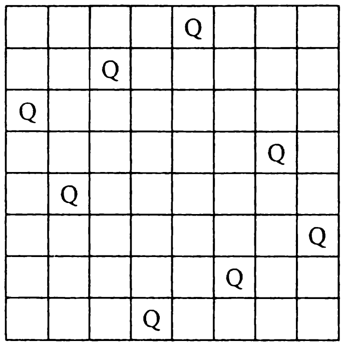


FIGURE 10.16
A solution to the 8-queens problem

The *n-queens problem* is to place n queens on the $n \times n$ chessboard so that each pair of queens is nonattacking.

a. Design a backtracking algorithm that generates all solutions to the n-queens problem.

b. Write a program implementing your algorithm in part (a), and run your program with various values of n.

10.13 Another classical problem associated with chess is the *knight's tour* problem. A knight can make up to eight moves, as shown in Figure 10.17. Starting at an arbitrary position in the $n \times n$ board, a knight's tour is a sequence of $n^2 - 1$ moves such that every square of the board is visited once.

a. Design a backtracking algorithm that either produces a knight's tour or determines that no such tour exits.

b. Write a program implementing your algorithm in part (a), and run your program with various values of n.

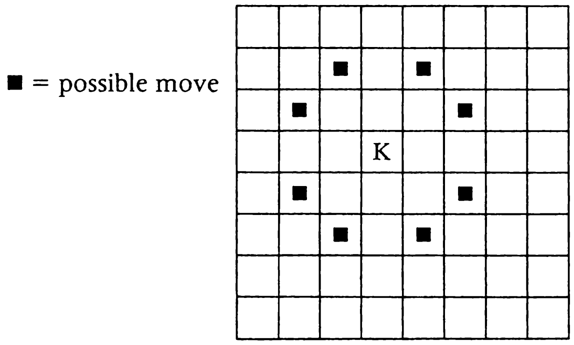


FIGURE 10.17
The eight possible moves for a knight in the given position

10.14 Let $Maze[0{:}n-1, 0{:}n-1]$ be a 0/1 two-dimensional array.

a. Design a backtracking algorithm that either finds a path from $Maze[0, 0]$ to $Maze[n-1, n-1]$ or determines that no such path exists. Adjacent vertices in the path correspond to adjacent cells in the matrix. You are not allowed to move to a cell that contains a 1.

b. Write a program implementing your algorithm in part (a), and run your program with various values of n.

10.15 Write a program that uses backtracking to solve the game of Hi-Q. Hi-Q is a popular game that can be found in many toy stores. Thirty-two pieces are arranged on a board as shown in Figure 10.18, with the center position left empty. The goal is to remove all the pieces but one by jumping and have the last piece end up in the middle position. A piece is allowed to jump a neighbor in either a horizontal or vertical direction (diagonal jumps are not permitted). When a piece is jumped, it is removed from the board. Output the 32 board configurations showing the solution: the initial board configuration and the board configuration after each jump is performed.

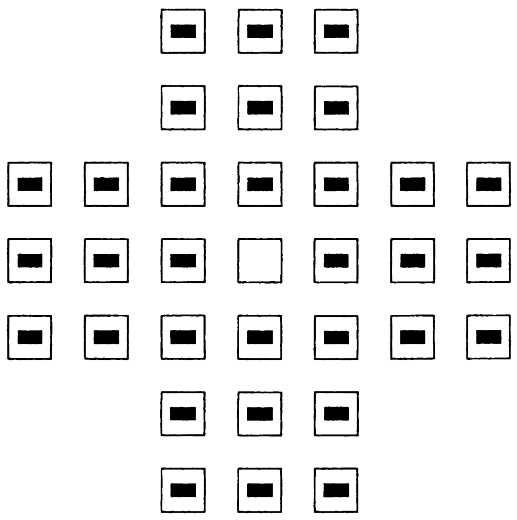


FIGURE 10.18

Initial board configuration for Hi-Q.

Section 10.3 Branch-and-Bound

10.16 Consider the optimization version of the sum of subsets problem for instance $\{a_0, \dots, a_6\} = \{1, 11, 6, 2, 6, 8, 5\}$ and $Sum = 10$. Show that for this instance of the sum of subsets problem, FIFO branch-and-bound generates fewer nodes of the state-space tree before reaching a goal state than backtracking does.

10.17 Give pseudocode for a version of the general procedure *BranchAndBound* that terminates as soon as a goal is found.

10.18 Give pseudocode for the procedure *Path*(*PtrNode*).

10.19 Draw that portion of the variable-tuple state-space tree generated by FIFO branch-and-bound for the 0/1 knapsack problem given by the following chart. Label the nodes with appropriate values of *UB*, *LoBd*, *SB*, *DB*, and indicate the optimal goal node, as in Figure 10.13. Trace the action of the queue *LiveNodes,* as illustrated in Figure 10.14.

i	0	1	2	3	4	Capacity $C = 12$
v_i	28	15	18	5.5	1	
w_i	8	5	6	4	1	

10.20 Repeat Exercise 10.19 for least cost branch-and-bound.

10.21 Repeat Exercise 10.19 using the fixed-tuple state-space tree.

10.22 Repeat Exercise 10.20 using the fixed-tuple state-space tree.

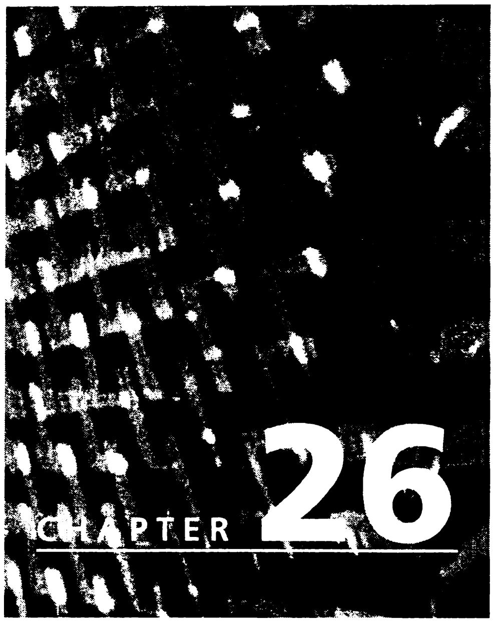
10.23 Given a set of Boolean variables $y_1, y_2, \ldots, y_m$, a CNF expression involving these variables is a conjunction of the form $C_1 \wedge C_2 \wedge \ldots \wedge C_n$, where each C_i is a disjunction of clauses of the form $z_{i,1} \vee z_{i,2} \vee \cdots \vee z_{i,n(i)}$, and where each $z_{i,j}$ (called a *literal*) is one of the Boolean variables y_1, $y_2, \ldots, y_m$ or its negation. The CNF SAT problem is to determine for a given CNF expression whether or not there is a truth assignment to the Boolean variables for which the CNF expression evaluates to **.true.** (that is, is *satisfied*). For a positive integer k, a k-CNF expression has the property that each clause contains exactly k literals. It turns out that the 2-CNF SAT problem has a polynomial solution (see Chapter 26), whereas the 3-CNF problem is already NP-complete. The best-known solutions to the CNF SAT problem involve fixed-tuple dynamic state-space trees and backtracking, together with such things as using clever heuristics to bound the search. In this exercise, we assume that each clause in a CNF expression is input as a string of integers, with positive integer i meaning that x_i occurs in the clause, and negative integer i meaning that the negative of x_i occurs in the clause. For example, 2, -5, 10 would represent the clause $y_2 \vee \overline{y_5} \vee y_{10}$.

a. Write a program that accepts a CNF expression as input and uses backtracking on a fixed-tuple static state-space tree to determine whether or not the CNF expression is satisfiable. The left (right) child of a node at level $k - 1$ corresponds to assigning **.true.** (**.false.**) to y_k, $k = 1, \ldots, n$.

b. Repeat part (a), but now use a dynamic state-space tree, where the k^{th} decision (level) in the tree is to give a truth assignment to the Boolean variable that occurs (either positively or negatively) k^{th} most often in the CNF expression (ties are decided using subscript ordering).

c. Repeat part (b), but now the decision at a given node in the tree is to assign a truth value to the variable that occurs most often in the clauses that have not already been satisfied by the previous assignments.

d. Run the programs written in parts (a), (b), and (c) with various input CNF expressions, and compare the results.



NP-COMPLETE PROBLEMS

It is a curious phenomenon that the worst-case complexities of known sequential algorithms for most of the commonly encountered problems in computer science fall into the following two categories:

1. Bounded above by low degree polynomials.
2. Bounded below by super-polynomial functions. (A function f is *super-polynomial* if $f \in \Omega(n^k)$ for *all* integers $k \geq 1$.)

Problems in graph theory illustrate this situation well. For example, we have seen $O(n^2)$ and $O(n^3)$ algorithms, respectively, for the problems of finding minimum-cost spanning trees and finding the shortest paths between every pair of vertices in weighted graphs, whereas the best-known algorithms for the Hamiltonian cycle problem and the graph-coloring problem have super-polynomial complexity. On the other hand, no one has been able to show that super-polynomial lower bounds for these problems exist. Problems such as the Hamiltonian cycle and graph-coloring problems (or more precisely, their decision versions, as defined in the next section) belong to a large class of fundamental decision problems called *NP-complete problems*. It turns out that if any NP-complete problem is solvable by a polynomial (time) algorithm, then they all are so solvable. It has long been conjectured that no polynomial algorithm exists for any of the NP-complete problems. This conjecture is arguably the most important open question in theoretical computer science.

In this chapter, we give a brief introduction to the class of NP-complete problems. Our treatment is somewhat intuitive and informal. In particular, we do not establish the formal machinery necessary to prove the famous result of Cook that NP-complete problems exist; rather, we refer the interested student to the references for a completely rigorous treatment.

We also expand our earlier discussion of the class NC and introduce the notion of P-completeness. P-completeness can be viewed as the analog for parallel computation of NP-completeness for sequential computation.

26.1 The Classes P and NP

We consider a problem to be *tractable* if it can be solved by a worst-case polynomial (time) algorithm. Problems having super-polynomial worst-case complexity are considered *intractable*. Throughout this chapter, when we use the term *computing time* (or *complexity*) we always mean the worst-case computing time (complexity).

Super-polynomial algorithms are computationally infeasible to implement in the worst case. Even though we have called a problem tractable if it can be solved by a polynomial algorithm, such an algorithm may still be computationally infeasible if it has complexity $\Omega(n^k)$ where k is large. For example, an algorithm having complexity n^{64} will not finish in our lifetime even for $n = 2$. Nevertheless, it is standard in the theory of algorithms to regard problems solved by polynomial algorithms as tractable, and it becomes important to identify those problems. Besides, if a polynomial algorithm has been shown to exist for a problem (even one of high degree), then there may be hope that a more practical polynomial algorithm of relatively small degree can be found for the problem.

In the theory of complexity, it is convenient and useful to restrict attention to decision problems—that is, problems whose solutions simply output "yes" or "no" (0 or 1). Let P denote the class of decision problems that are solvable by algorithms having polynomial (worst-case) complexity.

26.1.1 Input Size

When considering membership in P, we must be very careful about how input size is measured (recall our discussion of primality testing in Chapter 24). For example, when considering a decision problem having input parameters that are numeric quantities, the size of these numeric quantities is taken into account when measuring input size. We usually use the number of digits in the binary representation of the sum of the appropriate sizes of these numeric quantities. For example, consider an input $(w_0, w_1, \dots, w_{n-1})$, $(v_0, v_1, \dots, v_{n-1})$ and C to the

0/1 knapsack problem. If we let $m = \Sigma_{i=0}^{n-1}(w_i + v_i)$ and d denote the number of digits of m, then the size of this input is $n + d \in \Theta(n + \log m)$.

26.1.2 Optimization Problems and Their Decision Versions

Decision problems occur naturally in many contexts. For example, an important decision problem in graph theory is to determine whether a clique of size k occurs in a given input graph. In mathematical logic, a fundamental decision problem is whether the Boolean variables in a given Boolean formula can be assigned truth values that make the entire formula true. In addition to such decision problems that are of interest in their own right, optimization problems usually give rise to associated decision problems. Moreover, restricting attention to the decision version of an optimization problem sometimes can be done without loss of generality (up to polynomial factors). In other words, the decision version of an optimization problem may have the property that a polynomial solution to the optimization problem exists if and only if a polynomial solution to the associated decision problem exists (the "only if" part is always true).

To illustrate, consider the optimization problem of determining the chromatic number $\chi = \chi(G)$ of a graph G on n vertices—that is, finding the minimum number of colors necessary to properly color the vertices of a graph G such that no two adjacent vertices get the same color. Given an integer k, the associated decision problem asks whether a proper k-coloring of G exists. Clearly, a solution to the optimization problem immediately implies a solution to the decision problem: A proper k-coloring exists if and only if $k \geq \chi(G)$. On the other hand, a polynomial solution to the decision problem also implies a polynomial solution to the original problem: Simply call the decision problem for $k = 1, 2, \ldots$ until a "yes" is returned (which will happen after at most n calls).

As another example, consider the 0/1 knapsack problem with input weights $w_0, w_1, \ldots, w_{n-1}$, values $v_0, v_1, \ldots, v_{n-1}$, and capacity C. The decision version of this problem adds an integer input parameter k and asks whether a collection of the objects can be placed in the knapsack whose total value is at least k. Here again, it is obvious that the ordinary version of the 0/1 knapsack problem yields a solution of the decision version with no extra work. However, in the case of noninteger (rational) weights and values, there is no obvious polynomial number of decision problems whose answers yield the optimal solution to the 0/1 knapsack problem. The situation improves if we restrict attention to the case of *integer* weights and values. Then each possible solution has an integer value, with the largest possible value equal to $m = v_0 + v_1 + \ldots + v_{n-1}$. Unlike the graph-coloring problem, we cannot simply call the decision problem for $k = m, m - 1, \ldots$ until a "yes" is returned, since this will take an exponential number (with respect to input size) of calls in general. However, using a binary search strategy, by making $\log_2 m$ calls to the decision problem for suitable values of k, we can determine the optimal solution in polynomial time (see Exercise 26.1).

It is not known in general whether a polynomial solution to the 0/1 knapsack decision problem implies a polynomial solution to the 0/1 knapsack optimization problem.

26.1.3 The Class NP

The class NP (nondeterministic polynomial) consists of decision problems for which yes instances can be solved in polynomial time by a nondeterministic algorithm. Formal treatments of the class NP and nondeterministic algorithms are usually given in terms of Turing machines and formal languages. We proceed less formally and base our discussion on the intuitive notion of "guessing and verifying." In this context, we give the following high-level pseudocode for a generic nondeterministic polynomial algorithm *NPAlgorithm*. Step 1 in *NPAlgorithm* is the nondeterministic step, and step 2 is the deterministic step.

```
function NPAlgorithm(A, I)
Input:    A (a decision problem), I (an instance of problem A)
Output:   "yes" or "don't know"
              1. In polynomial time, guess a candidate certificate C for the problem A.
              2. In polynomial time, use C to deterministically verify that I is a yes instance.
                 if a yes instance is verified in step 2 then
                    return ("yes")
                 else
                    return("don't know")
                 endif
end NPAlgorithm
```

NPAlgorithm is considered correct if, whenever I is a yes instance, then a certificate can be produced in step 1 (by "perfect" guessing) that verifies that I is a yes instance, whereas if I is a no instance, then "don't know" is always returned. Be careful to note that a given running of *NPAlgorithm* might return "don't know" even on a yes instance. The correctness requirement says that if I is a yes instance, then a verification certificate *can be* produced in step 2, assuming the appropriate guess is made.

For example, if the decision problem is whether a graph can be properly colored using at most k colors for a given input graph G, a nondeterministic algorithm would simply guess a color from $\{1, 2, \dots, k\}$ for each vertex. Perfect guessing would produce a proper k-coloring for a graph G if it exists, and this coloring would then be a certificate that can be used to deterministically verify that G is a yes instance.

A decision problem is in the class NP if it can be solved by a nondeterministic algorithm. In the k-coloring problem, clearly a certificate coloring can be verified to be a proper coloring in polynomial time.

Note that $P \subseteq NP$, because if A is a polynomial algorithm for a decision problem, then a nondeterministic algorithm for the problem is obtained by guessing any succinct (that is, computable in polynomial time) certificate S for a given input I and then using A to solve the problem instance I, completely ignoring S.

26.1.4 Some Sample Problems in NP

We illustrate the definition of *NP* with some sample problems in NP. In each example, it is obvious that a certificate can be produced and verified for a yes instance in polynomial time. A *clique* Q of size k in a graph G is a subset of k vertices that are pair-wise adjacent.

Clique. Given the graph $G = (V, E)$ and the integer k, does there exist a clique of size k in G?

A nondeterministic algorithm guesses a candidate solution by choosing a set of k vertices. The verification that the set of vertices is distinct and forms a clique can be done in polynomial time.

Sum of Subsets. Given the input integers $\{a_0, a_1, \dots, a_{n-1}\}$, and the target sum C, is there a subset of the integers whose sum equals C?

The nondeterministic algorithm guesses a candidate solution by choosing a subset of the integers $\{a_{i_1}, a_{i_2}, \dots, a_{i_m}\}$. The verification that $a_{i_1} + a_{i_2} + \cdots + a_{i_m} = C$ can be done in polynomial time.

Hamiltonian Cycle. Given the graph $G = (V, E)$, does G have a Hamiltonian cycle?

A nondeterministic algorithm guesses a Hamiltonian cycle by choosing a sequence of vertices $v_0, v_1, \dots, v_{n-1}, v_0$. Verification that the sequence of vertices is a Hamiltonian cycle can be done in polynomial time.

Showing that the problems in this section were in NP was relatively easy. However, for some problems, membership in NP is quite difficult to establish. A notable example is prime testing, which asks whether a given integer n is prime. Note that a yes instance means that the answer is no to the question of whether n has factors, and the nonexistence of factors has no obvious certificate. However, it was shown by Pratt that a succinct certificate can be produced by a nondeterministic algorithm that can be verified in polynomial time (see Exercise 26.2). In 2002, it was shown that primality testing is actually in P.

The question of whether P = NP can be viewed as asking whether adding nondeterminism allows us to solve a class of problems in polynomial time that may not otherwise be solvable in polynomial time. It seems reasonable that adding the luxury of making good guesses should really help, and most computer scientists believe that P ≠ NP. For example, the question of whether or not a search element occurs in a list of size n can be solved in constant time by a nondeterministic algorithm, whereas any deterministic algorithm solving this problem has linear worst-case complexity. Reinforcement of the belief that P ≠ NP can be found in the fact that there are thousands of varied problems in NP that have resisted all attempts to find polynomial solutions.

26.2 Reducibility

One reason that polynomial complexity is such a convenient measure is that polynomials have nice closure properties. For example, the composition $f(x) = g(h(x))$ of two polynomials g and h is yet another polynomial; in other words, a polynomial of a polynomial is a polynomial. Another example is the property that the sum (or difference or product) of a polynomial and a polynomial is a polynomial. These closure properties of polynomials allow a convenient description of when a given decision problem A is not harder to solve than another decision problem B; namely, the solution to B should yield the solution to A with at most a polynomial factor of additional work. We make this precise in the following definition.

EFINITION 26.2.1 Given two decision problems A and B, we say that A is polynomially *reducible* to B, denoted $A \propto B$, if there is a mapping f from the inputs to problem A to the inputs to problem B, such that the following conditions apply:

1. The mapping f can be computed in polynomial time (that is, a polynomial algorithm exists that, for input I to problem A, outputs the input $f(I)$ to problem B).
2. The answer to a given input I to problem A is yes if, and only if, the answer to the input $f(I)$ to problem B is yes.

In particular, condition (1) of Definition 26.2.1 implies that there exists a polynomial $p(n)$ such that $|f(I)| \in O(p(|I|))$, for any input I to algorithm A, where $|I|$ denotes the size of I.

The following two propositions are easy consequences of the closure properties of polynomials.

Proposition 26.2.1 The relation $\propto$ is transitive; that is, if $A \propto B$ and $B \propto C$ then $A \propto C$. □

Proposition 26.2.2 If $A \propto B$ and B has polynomial complexity, then so does A. □

We now illustrate the notion of reducibility by several examples. Further examples are given in Section 26.4.

Example 26.2.1: Hamiltonian Cycles $\propto$ Traveling Salesman. Given the graph $G = (V, E)$, $|V| = n$, define the following weighting w on K_n on the complete graph on n vertices: $w(e) = 1$ if $e \in E$; otherwise, $w(e) = 2$. Clearly, w can be computed in polynomial time. We now map G to the instance $f(G) = (K_n, w, k = n)$ of Traveling Salesman (so that $p(n) = n^2$). Thus, G has a Hamiltonian cycle if and only if K_n with weighting w has a tour of cost no more than n.

Example 26.2.2: Clique $\propto$ Vertex Cover $\propto$ Independent Set $\propto$ Clique. A *vertex cover* C in a graph G is a set of vertices with the property that every edge in G is incident to at least one vertex in C. The vertex cover problem asks whether a graph G has a vertex cover of size k. An *independent set* of vertices in a graph G is a set of vertices no two of which are adjacent. The independent set problem asks whether a graph G has a vertex cover of size k. Since the relation $\propto$ is transitive, Example 26.2.2 says that the three problems, clique, vertex cover, and independent set, are each polynomially reducible to one another. Given the graph $G = (V, E)$, recall that the complement $\overline{G} = (V, \overline{E})$ is the graph with the same vertex set V, such that $e \in \overline{E}$ if and only if $e \notin E$. Given G, clearly the complement $\overline{G}$ can be computed in polynomial time. The polynomial reductions are consequences of the following proposition, whose proof is left as an exercise.

Proposition 26.2.3 Given the graph $G = (V, E)$, then the following three conditions are mutually equivalent conditions on a subset of vertices $U \subseteq V$.

1. U is a clique.
2. U is an independent set of vertices in the complement $\overline{G} = (V, \overline{E})$.
3. The set $V - U$ is a vertex cover in the complement $\overline{G} = (V, \overline{E})$. □

The reducibility illustrated in the previous examples was easy to show and not very surprising because the problems involved were highly related. In Section 26.4, we give examples of disparate problems A and B where $A \propto B$.

The following definition makes precise the notion of two problems being polynomially equivalent to one another.

:FINITION 26.2.2 Two decision problems A and B are *polynomially equivalent* to one another if $A \propto B$ and $B \propto A$.

Since the relation $\propto$ is clearly reflexive, it follows from Proposition 26.2.1 that the relation of polynomial equivalence is an equivalence relation on the set of decision problems.

26.3 NP-Complete Problems: Cook's Theorem

S. A. Cook raised a fundamental question: Is there a problem in NP that is as hard (up to polynomial factors) as every other problem in NP? In other words, is there a problem C in NP such that if A is *any* problem in NP, then $A \propto C$? If such a problem exists, then it is called *NP-complete.* (Note that all NP-complete problems are automatically polynomially equivalent to one another.) In one of the most celebrated results in theoretical computer science, Cook showed in 1971 that the satisfiability problem for Boolean expressions (as defined presently) is NP-complete. About the same time, Levin showed that a certain tiling problem was NP-complete. The following proposition follows easily from the transitivity of the relation $\propto$.

›roposition 26.3.1 Suppose A is in NP, and B is NP-complete. If $B \propto A$, then A is NP-complete. □

In the time since Cook's result appeared, hundreds of NP-complete problems have been identified. Despite considerable effort, no polynomial algorithm has been found for any NP-complete problem. This lack of success reinforces the belief that $P \neq NP$. The following corollary of Proposition 26.3.1 shows that it is unlikely that a polynomial algorithm for any NP-complete problem will ever be found.

Corollary 26.3.2 If any NP-complete problem A also belongs to P, then $P \doteq NP$. □

A problem A (not necessarily a decision problem) is *NP-hard* if it is as hard to solve as any problem in NP. More precisely, A is NP-hard if a polynomial-time solution for A would imply P = NP. For example, if the decision version of an optimization problem is NP-complete, then the optimization problem itself is NP-hard. Note that the NP-complete problems are precisely those problems in NP that are NP-hard.

To describe Cook's result, we need to recall some terms from first-order logic (see also Exercise 10.23 in Chapter 10). As with our pseudocode conventions, a Boolean (logical) variable is a variable that can only take on two values, true (T) or false (F). Given a Boolean variable x, we denote by $\bar{x}$ the variable that has the value T if and only if x has the value F. Boolean variables can be combined using logical operators *and* (denoted by $\wedge$ and called *conjunction*), *or* (denoted by $\vee$ and called *disjunction*), *not* (denoted by $\neg$ and called *negation*), and parenthesization, to form Boolean expressions (formulas). A *literal* in a Boolean formula is of the form x or $\bar{x}$, where x is a Boolean variable. Note that $\neg x$ has the same truth value as $\bar{x}$. A Boolean formula is said to be in *conjunctive normal form* (CNF) if it has the form $C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where each clause C_i is a disjunction of $n(i)$ literals, $i = 1, \ldots, m$.

A Boolean formula is called *satisfiable* if there is an assignment of truth values to the literals occurring in the formula that makes the formula evaluate to true. For example, the CNF formula

$$(x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2) \wedge (x_1 \vee x_3) \wedge (\bar{x}_1 \vee x_3)$$

is satisfiable (by setting $x_1 = x_2 = x_3 = T$), whereas the CNF formula

$$(x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge (x_1 \vee \bar{x}_2) \wedge (x_1 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$$

is not satisfiable.

The *CNF satisfiability* problem (CNF SAT) is the problem of determining whether a given CNF formula is satisfiable. CNF SAT is a fundamental problem in mathematical logic and computer science, with numerous applications.

The input size of a CNF formula can be defined as the total number of literals (counting repetitions) occurring in the formula. Clearly, CNF SAT $\in$ NP, because a linear scan of the formula determines whether a candidate assignment to the literals occurring in the formula results in the formula evaluating to true. CNF SAT was the first example shown to be NP-complete.

Theorem 26.3.3 **COOK'S THEOREM**

CNF SAT is NP-complete. □

Given any decision problem A in NP, there is a nondeterministic algorithm that produces a candidate solution for A that can be checked in polynomial time. Cook proved Theorem 26.3.3 by constructing a CNF formula that, roughly speaking, modeled the action of the nondeterministic algorithm. The length of the formula constructed by Cook is polynomial in the size of an instance I of A and is satisfiable if and only if the problem A is true for I.

26.4 Some Sample NP-Complete Problems

As soon as it was established that CNF SAT is NP-complete, the door was opened to show that a host of other NP-complete problems exist. Proposition 26.3.1 gives a recipe for proving that a problem A is NP-complete, which we state as a key fact.

Two steps are required to show that a problem A is NP complete:

1. **Show that A is in NP.**
2. **Find any NP-complete problem B such that B $\propto$ A.**

Condition 1 is usually easy to show, so that condition 2 becomes the issue. However, as we have already seen, there are cases where condition 1 is difficult to verify.

Thousands of classical and practical problems have been shown to be NP-complete. If we are working on a problem that we think might be NP-complete, we can check the vast list of NP-complete problems for one that might resemble our problem. The following subsections provide a short sample list of NP-complete problems. We have already shown that the clique problem and the sum of subsets problems are in NP. For each of the other examples, we leave it as an exercise to show the problem is in NP. For each of the sample problems in this section, we give a reduction that shows the problem is NP-complete.

Example 26.4.1: 3-CNF SAT. An instance of the 3-CNF SAT problem is a CNF Boolean formula in which every clause contains *exactly* three literals. Because general CNF SAT is in NP, 3-CNF SAT is also in NP. Surprisingly, 3-CNF SAT is NP-complete, even though 2-CNF SAT (in which each clause consists of two literals) is in P (see Exercise 26.9).

We show that CNF SAT $\propto$ 3-CNF SAT by showing how any CNF formula $I = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ can be mapped onto a 3-CNF formula $f(I)$ with no more than $12\,|I|$ literals, such that I is satisfiable if and only if $f(I)$ is satisfiable. More precisely, we construct $f(I)$ by replacing each clause $C_i = i = 1, \ldots, m$, by conjunctions of three-literal clauses according to the following three cases.

1. C_i has exactly three literals.
2. C_i has more than three literals.
3. C_i has less than three literals.

In case 1, we leave C_i unaltered. In case 2, suppose C_i contains $k > 3$ literals $x_1, x_2, \ldots, x_k$, so that $C_i = x_1 \vee x_2 \vee \cdots \vee x_k$. We then replace C_i by the $k - 2$ clauses

$$(x_1 \vee x_2 \vee y_1) \wedge (\bar{y}_1 \vee x_3 \vee y_2) \wedge (\bar{y}_2 \vee x_4 \vee y_3) \wedge \cdots \wedge (\bar{y}_{k-3} \vee x_{k-1} \vee x_k),$$

where $y_1, y_2, \ldots, y_{k-3}$ are new Boolean variables. We leave it as an exercise to show that C_i is satisfiable if, and only if, the conjunction of the $k - 2$ clauses replacing C_i is satisfiable.

Now consider case 3. If C_i consists of a single literal x, then we can replace x by $(x \vee y \vee z) \wedge (x \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee z) \wedge (x \vee \bar{y} \vee \bar{z})$, where y and z are new Boolean variables. If C_i consists of the two-literal clause $x_1 \vee x_2$, then we replace C_i by $(x_1 \vee x_2 \vee y) \wedge (x_1 \vee x_2 \vee \bar{y})$, where y is a new Boolean variable. In both cases, it follows easily that C_i is satisfiable if and only if the conjunction of the clauses replacing it is satisfiable.

Example 26.4.2: Clique. This example establishes a connection between Boolean formulas and graph theoretic problems, thereby extending the realm of NP-complete problems into the latter domain. We show that CNF SAT $\propto$ Clique. Consider any CNF formula $I = C_1 \wedge C_2 \wedge \cdots \wedge C_m$. Writing clause C_i as $y_1 \vee \cdots \vee y_j \vee \overline{y_{j+1}} \vee \cdots \vee \overline{y_k}$ (the j and k depend on i), where $y_1, \ldots, y_j \in \{x_1, \ldots, x_n\}$ and $\overline{y_{j+1}}, \ldots, \overline{y_k} \in \{\overline{x_1}, \ldots, \overline{x_n}\}$, let $V_i = \{(y_1, i), \ldots, (y_j, i), (\overline{y_{j+1}}, i), \ldots, (\overline{y_k}, i)\}$, $i = 1, \ldots, m$. We map I to the graph $G = f(I)$ as follows. The vertex set V of G is the union of the V_i's, $i = 1, \ldots m$. We define two vertices (a, i) and (b, j) to be adjacent if and only if $i \neq j$ and the literal a is not the negative of the literal b. For example, if $(x_1, 3)$, $(x_5, 7)$, $(\overline{x_8}, 2)$, $(x_2, 4)$, $(\overline{x_3}, 1)$, $(\overline{x_8}, 1)$, $(x_6, 2)$, and $(\overline{x_6}, 5)$ are vertices of G, then $\{(x_1, 3), (x_5, 7)\}$ and $\{(\overline{x_8}, 2), (x_2, 4)\}$ are edges of G but $\{(\overline{x_3}, 1), (\overline{x_8}, 1)\}$ and $\{(x_6, 2), (\overline{x_6}, 5)\}$ are not. Note that G can be constructed in polynomial time.

CLAIM

I is satisfiable if and only if G has a clique of size m.

PROOF

Suppose that G has a clique W of size m. By definition of adjacency in G, it follows that, for each clause C_i in I, there is exactly one vertex in W of the form (a, i). If a is the positive literal x_q, then assign x_q the value T. If a is the negative literal $\overline{x_q}$, then assign $\overline{x_q}$ the value T (that is, x_q is F). These assignments are well defined because, by definition of adjacency in G, none of the literals (in the second component) of a vertex is the negative of another. Moreover, these assignments make each of the clauses C_i evaluate to T, so that I evaluates as true.

Now suppose that I is satisfiable. Let $Z = \{z_1, \dots, z_n\}$ denote the n literals from $\{x_1, \dots, x_n\} \cup \{\overline{x_1}, \dots, \overline{x_n}\}$ that have the value T. Since each clause C_i must have the value T, $i = 1, \dots, m$, C_i contains at least one literal z_i from Z. However, by the definition of G, (z_i, i) is a vertex of G. Further, the set of vertices $\{(z_i, i) \mid i \in \{1, \dots, m\}\}$ forms a clique of size m. ■

Example 26.4.3: Vertex Cover and Independent Set. That these problems are both NP-complete follows from Example 26.4.2 and Proposition 26.2.3.

Example 26.4.4: Three-Dimensional Matching. Recall that the perfect-matching problem in a bipartite graph interprets the classical marriage problem, which asks whether marriages can be arranged between a set of n boys and a set of n girls so that each boy marries a girl that he knows, and no girl marries more than one boy. The three-dimensional matching problem generalizes the marriage problem by adding n houses to the mix. Not only do we want to arrange marriages, but we also wish to match each married couple with a house from a specified subcollection of the houses available to the couple, and we want to ensure that no two couples share the same house. Adding houses turns a problem solved by an $O(n^3)$ algorithm into an NP-complete problem.

More formally, given three sets H, B, and G (houses, boys, and girls, respectively), each having the same cardinality $|H|$, and a subset $M \subseteq H \times B \times G$, the three-dimensional matching problem asks whether there is a subset (matching) $M' \subseteq M$ of cardinality $|M'| = |H|$ such that none of the triples in M' agree in any coordinate. We refer to the triples in M' as *households*. The three-dimensional matching problem is in NP because a nondeterministic algorithm guesses a set of $|H|$ triples, and the verification that no two triples agree in any coordinate can be done in polynomial time. We now show that the three-dimensional matching problem is NP-complete by showing that CNF SAT reduces to it.

Suppose $I = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ is any instance of CNF SAT, and suppose I contains the Boolean variables $x_1, x_2, \dots, x_n$. We will construct an instance of the three-dimensional matching problem $H, B, G, M \subseteq H \times B \times G$, where $|H| = |B| = |G| = 2mn$, such that a matching $M' \subseteq M$ exists if and only if I is satisfiable. H consists of m copies of the literals $\{x_1, x_2, \dots, x_n\}$, $\{\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\}$. We use subscripts and superscripts to denote the elements in H, where subscripts reference literals, and superscripts reference clauses in I; that is

$$H = \{x_i^{(j)}, \bar{x}_i^{(j)} \mid 1 \le i \le n,\ 1 \le j \le m\}.$$

We now create sets B and G of $2mn$ boys and $2mn$ girls, respectively. The sets B and G will be divided into three classes, and marriages are restricted to taking place only between boys and girls in the same class. Thus, the division of the boys and girls induces a division of M into three classes of households, which we call truth-setting, satisfaction-testing, and remaining households, respectively. As the names imply, truth-setting households force an assignment of truth values to the Boolean variables $x_1, x_2, \dots, x_n$, whereas satisfaction-testing households determine whether this truth assignment makes the formula I true. The remaining households in M are made up of a set of additional couples created merely to occupy the remaining houses not covered by the households in the first two classes. We now describe the division of the boys and girls into three classes together with the resulting three classes of households.

Truth-Setting Households. The truth-setting households are divided into n components T_i, $i = 1, \dots, n$, where each component holds m households. For each i, $1 \le i \le n$, we create a set of m boys and m girls

$$B_{T_i} = \{b_i^{(1)}, b_i^{(2)}, \dots, b_i^{(m)}\},\ G_{T_i} = \{g_i^{(1)}, g_i^{(2)}, \dots, g_i^{(m)}\},\ i = 1, \dots, n.$$

Marriages are restricted, so that a boy in the i^{th} component B_{T_i} must marry a girl in the i^{th} component G_{T_i}. The marriages are further restricted within each component T_i to take place between people who are adjacent when arranged in a circle, as in Figure 26.1.

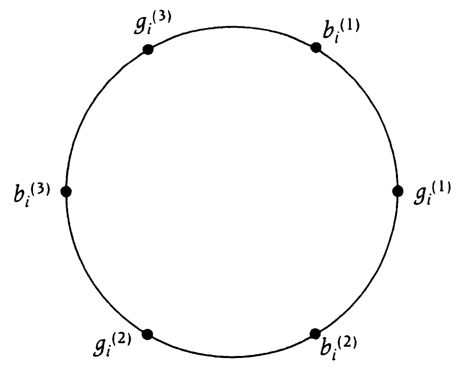


FIGURE 26.1

In each component, marriages must take place between adjacent people in the given circle arrangement, as illustrated for $m = 3$.

In particular, there are only two marriage schemes possible in T_i: a clockwise scheme, $\{b_i^{(1)}, g_i^{(1)}\}, \{b_i^{(2)}, g_i^{(2)}\}, \dots, \{b_i^{(m)}, g_i^{(m)}\}$, and a counterclockwise scheme, $\{b_i^{(1)}, g_i^{(m)}\}, \{b_i^{(m)}, g_i^{(m-1)}\}, \{b_i^{(m-1)}, g_i^{(m-2)}\}, \dots, \{b_i^{(2)}, g_i^{(1)}\}$.

In each component T_i, only one house is made available to a given couple. If girl $g_i^{(j)}$ marries boy $b_i^{(j)}$, then she must live in house $\bar{x}_i^{(j)}$. Otherwise (girl $g_i^{(j)}$ marries boy $b_i^{(j+1)}$, where we set $b_i^{(m+1)} = b_i^{(1)}$), she must live in house $x_i^{(j)}$. The housing restrictions are illustrated in Figure 26.2, which helps explain the "truth-setting" nomenclature. We see from Figure 26.2 that any matching M' must intersect the households in the ith component T_i in precisely one of two ways: Either $M' \cap T_i$ consists of the set $T_i(t)$ of shaded households, or $M' \cap T_i$ consists of the set $T_i(f)$ of unshaded households. Thus, the component T_i forces a matching M' to set a truth value for x_i, $i = 1, \dots, n$, where $M' \cap T_i = T_i(t)$ is considered as setting x_i to T, whereas $M' \cap T_i = T_i(f)$ is considered as setting x_i to F.

In summary, the truth-setting households in M consist of n components T_i, $i = 1, \dots, n$, where each T_i is the union of two sets of triples:

$$T_i(t) = \{(\bar{x}_i^{(j)}, b_i^{(j)}, g_i^{(j)}) \mid 1 \le i \le n, 1 \le j \le m\},$$
$$T_i(f) = \{(x_i^{(j)}, b_i^{(j+1)}, g_i^{(j)}) \mid 1 \le i \le n, 1 \le j < m\} \cup (x_i^{(m)}, b_i^{(1)}, g_i^{(m)}).$$

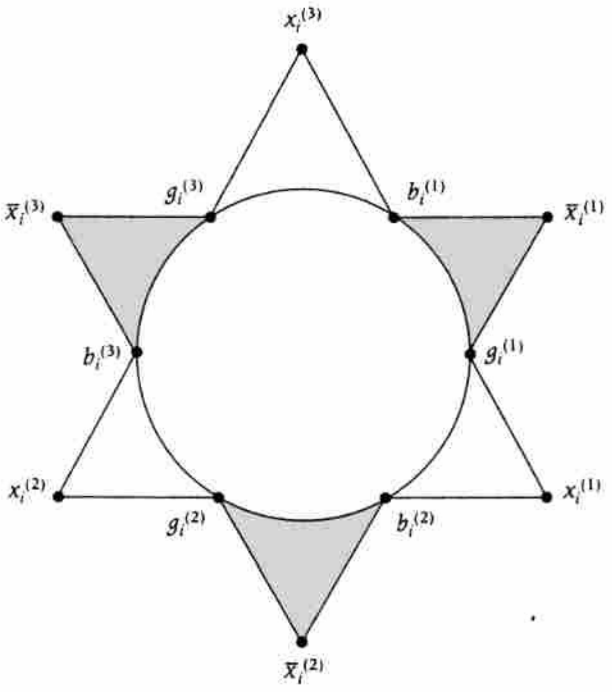


FIGURE 26.2

The household of T_i corresponding to $m = 3$. Any matching must intersect T_i precisely in either the shaded households $T_i(t)$ or the unshaded households $T_i(f)$.

Moreover, no household in M other than T_i contains a boy $b_i^{(j)}$ or a girl $g_i^{(j)}$, $1 \le i \le n$, $1 \le j \le m$.

Satisfaction-Testing Households. We now create m boys $s_b^{(j)}$ and m girls $s_g^{(j)}$, respectively, $j = 1, \ldots, m$, and match them by prearranged marriages $\{\{s_b^{(j)}, s_g^{(j)}\} \mid j = 1, \ldots, m\}$. Moreover, the only houses available to couple $\{s_b^{(j)}, s_g^{(j)}\}$ are literals (x_i or $\bar{x}_i$) that appear in clause C_j. Thus, the satisfaction-testing households in M consist of the triples:

$$S_j = \{(x_i^{(j)}, s_b^{(j)}, s_g^{(j)}) \mid x_i \in C_j, i = 1, \ldots, n\} \cup \{(\bar{x}_i^{(j)}, s_b^{(j)}, s_g^{(j)}) \mid \bar{x}_i \in C_j, i = 1, \ldots, n\}.$$

No other households in M contain a boy $s_b^{(j)}$ or a girl $s_g^{(j)}$, $j = 1, \ldots, m$. Hence, any matching $M' \subseteq M$ must contain exactly one household from each S_j, $j = 1, \ldots, m$. Moreover, if the household chosen in S_j uses a house corresponding to the variable x_i, then that house must not be included in $M' \cap T_i$. In other words, we must choose a house whose truth setting for x_i as determined by M' satisfies clause C_j. *Thus, a matching $M' \subseteq M$ cannot exist unless I is satisfiable.*

Remaining Households. Any matching M' uses nm houses to accommodate the truth-setting households and m houses to accommodate the satisfaction-testing households. Thus, there are $m(n-1)$ houses not covered by the truth-setting or satisfaction-testing households, making up a *partial-matching* M'. We create a set of $m(n-1)$ boys $r_b^{(j)}$ and $m(n-1)$ girls $r_g^{(j)}$, respectively, $j = 1, \ldots, m(n-1)$, so that they can occupy the remaining $m(n-1)$ houses not covered by such a partial-matching M' and thereby extend M' to a matching. These boys and girls are matched by prearranged marriages consisting of couples $\{\{r_b^{(j)}, r_g^{(j)}\} \mid j = 1, \ldots, m(n-1)\}$. While these couples had no choice in their marriages, the set M places no housing restrictions on them. Thus, the remaining households R in M consist of the following triples:

$$\begin{aligned} R = {} & \{(x_i^{(j)}, r_b^{(k)}, r_g^{(k)}) \mid i = 1, \ldots, n, j = 1, \ldots, m, k = 1, \ldots, m(n-1)\} \\ & \cup \{(\bar{x}_i^{(j)}, r_b^{(k)}, r_g^{(k)}) \mid i = 1, \ldots, n, j = 1, \ldots, m, k = 1, \ldots, m(n-1)\}. \end{aligned}$$

Given the instance I of CNF SAT, the set $H \times B \times G$ and the subset of households $M \subseteq H \times B \times G$ can clearly be constructed in polynomial time. We have already noted that a matching $M' \subseteq M$ cannot exist unless I is satisfiable. We complete the proof that CNF SAT $\propto$ three-dimensional matching by showing that if I is satisfiable, then a matching $M' \subseteq M$ exists. Consider a truth assignment to the variables $x_1, x_2, \ldots, x_n$ that makes I true. For each clause C_j, we choose a literal l_j occurring in C_j having the value T. Such a choice must be possible since I is assumed true with the given truth assignment. We then set

$$M' = \left(\bigcup_{x_i = T} T_i(t)\right) \cup \left(\bigcup_{\bar{x}_i = T} T_i(f)\right) \cup \left(\bigcup_{j=1}^{m} \{(l_j, s_b^{(j)}, s_g^{(j)})\}\right) \cup R',$$

where R' is a suitably constructed collection of remaining households that includes all the couples $\{r_b^{(k)}, r_g^{(k)}\}$, $k = 1, \dots, m(n-1)$ and all the remaining houses not covered by the truth-setting or satisfaction-testing households in M'. We leave it as an exercise to verify that R' can be constructed and that the resulting set M' is a matching.

Example 26.4.5: 3-Exact Cover. Given a family $F = \{S_1, S_2, \dots, S_n\}$ of n subsets of $S = \{x_1, x_2, \dots, x_{3m}\}$, each of cardinality three, the *3-exact cover* problem asks if there is a subfamily $\{S_{i_1}, S_{i_2}, \dots, S_{i_m}\}$ of m subsets of F that covers S; that is, $S = \bigcup_{j=1}^{m} S_{i_j}$. The 3-exact cover problem is in NP because a nondeterministic algorithm guesses a subfamily and the verification that the subfamily covers S can be done in linear time. (We can use characteristic bit vectors of length $3m$ to represent subsets of F.)

Note that Three-Dimensional Matching $\propto$ 3-Exact Cover. In fact, Three-Dimensional Matching is actually just a special case of 3-Exact Cover where we ignore the ordering of the subsets. More precisely, we simply associate the instance $M \subseteq H \times B \times G$ of the three-dimensional matching problem with the instance $S = H \cup B \cup G$ and $F = \{\{h, b, g\} \mid (h, b, g) \in M\}$ of the 3-exact cover problem.

Example 26.4.6: Sum of Subsets. We now show that 3-Exact Cover $\propto$ Sum of Subsets. Given an instance $F = \{S_1, S_2, \dots, S_n\}$, $S = \{x_1, x_2, \dots, x_{3m}\}$ of 3-Exact Cover, as noted previously we can consider the sets in F as represented by their characteristic bit vectors of length $3m$. For example, if $m = 3$ and $S_1 = \{x_2, x_3, x_8\}$, then S_1 is represented by $(0, 1, 1, 0, 0, 0, 0, 1, 0)$. We map an instance of 3-Exact Cover to an instance of Sum of Subsets by interpreting each characteristic vector as an integer in the base-$(n+1)$ system. Thus, for each $S_j \in F$, we associate with S_j the integer $a_{j-1} = \Sigma_{x_i \in S_j} (n+1)^{i-1}$. We let C be the integer corresponding to the bit vector of all 1s; that is, $C = \Sigma_{i=0}^{3m-1} (n+1)^i$. We leave it as an exercise to show that F, S is a yes instance of 3-Exact Cover if and only if $a_0, a_1, \dots, a_{n-1}$ and target sum C is a yes instance of Sum of Subsets.

Example 26.4.7: Graph Coloring. We now show that 3-CNF SAT $\propto$ Graph Coloring. Suppose $I = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ is an instance of 3-CNF SAT involving the

Boolean variables $\{x_1, x_2, \ldots, x_n\}$. We can assume that $n \geq 4$, because otherwise the satisfiability of I can be checked in polynomial time. We construct an instance $G = (V, E)$, $k = n + 1$, of Graph Coloring as follows.

$$V = \{x_1, x_2, \ldots, x_n\} \cup \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\} \cup \{y_1, y_2, \ldots, y_n\} \cup \{c_1, c_2, \ldots, c_m\},$$

$$\begin{aligned} E = {} & \{x_i\bar{x}_i \mid 1 \leq i \leq n\} \cup \{y_iy_j \mid 1 \leq i < j \leq n\} \cup \{x_iy_j \mid 1 \leq i \neq j \leq n\} \\ & \cup \{\bar{x}_iy_j \mid 1 \leq i \neq j \leq n\} \cup \{x_ic_j \mid x_i \notin C_j,\ 1 \leq i \leq n,\ 1 \leq j \leq m\} \\ & \cup \{\bar{x}_ic_j \mid \bar{x}_i \notin C_j,\ 1 \leq i \leq n,\ 1 \leq j \leq m\}. \end{aligned}$$

Clearly, G can be constructed from I in polynomial time. We leave it as an exercise to show that I is satisfiable if and only if the graph G is $(n + 1)$-colorable.

26.5 The Class co-NP

Given a decision problem Π, the complementary problem $\overline{\Pi}$ is the decision problem with the same instances, but an instance I of the problem Π is true if and only if I is false for $\overline{\Pi}$. For example, suppose Π is the decision problem, "Does a graph G contain a Hamiltonian cycle?" The complementary decision problem $\overline{\Pi}$ asks, "Is there *no* Hamiltonian cycle in G?" A certificate for a Hamiltonian cycle is easy to verify, but what would a certificate be for the complementary problem? One would be an enumeration of all possible Hamiltonian cycles, but there are far too many candidates to check in polynomial time. In fact, it is believed that the non-Hamiltonian problem does not belong to NP.

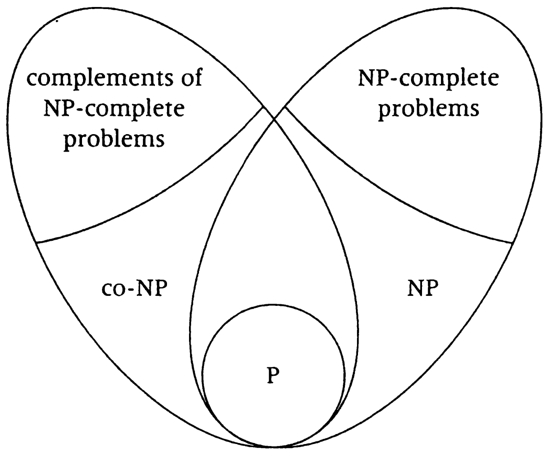
Clearly, the complement $\overline{\Pi}$ of a problem in P also belongs to P. Indeed, a polynomial algorithm for Π also serves as a polynomial algorithm for $\overline{\Pi}$ by outputting "yes" for an instance I of $\overline{\Pi}$ if and only if the algorithm outputs "no" for instance I of problem Π. However, the same argument does not work for problems in NP. Given a problem $\Pi \in$ NP, a yes instance for Π corresponds to a no instance for $\overline{\Pi}$, and we cannot be assured of the existence of certificates that can be verified in polynomial time for no instances of problems in NP.

The class *co-NP* comprises problems that are complements of problems in NP. As an example, consider again the problem of prime testing, which asks whether a given integer n is prime. Prime testing is in co-NP because its complement belongs to NP. A nondeterministic algorithm merely guesses a factor, and it can be checked in polynomial time whether the factor divides n evenly. Recall that it has recently been shown that prime testing is actually in P.

A natural question is whether co-NP $\neq$ NP. Most computer scientists believe the answer is yes and conjecture that any NP-complete problem provides the answer. The following theorem motivates this belief.

FIGURE 26.3

Conjectured relationships between P, NP, and co-NP.



Theorem 26.5.1 Let Π be any NP-complete problem. If the complement $\overline{\Pi}$ belongs to NP, then NP = co-NP. □

We leave the proof of Theorem 26.5.1 as an exercise.

The Venn diagram in Figure 26.3 shows the conjectured relationships between P, NP, and co-NP. However, there still is the (very unlikely) possibility that P = NP, in which case the whole diagram would collapse to P.

26.6 The Classes NC and P-Complete

We considered a sequential algorithm to be good (tractable) if it has polynomial worst-case complexity. Analogously, we consider a parallel algorithm on a PRAM to be good if it uses a polynomial number of processors and has polylogarithmic worst-case complexity. The class of problems solvable by such an algorithm is called the class NC. (Because any algorithm using P processors on an EREW PRAM or CREW PRAM can be simulated on the CRCW PRAM using p processors with at most logarithmic slowdown, the definition of the class NC is independent of the model of PRAM chosen.) In this section, we introduce the notion of P-completeness with respect to NC, which can be viewed as the analog for parallel computation of NP-completeness.

DEFINITION 26.6.1 Given two decision problems A and B, we say that A is *NC-reducible* to B if there is a mapping f from instances of A to instances of B such that the following two conditions apply:

1. The mapping f can be computed in polylogarithmic time using a polynomial number of processors on a PRAM.
2. The answer to a given instance I of problem A is yes if and only if the answer to instance $f(I)$ of B is yes.

Using the fact that the composition of two polynomials is a polynomial, it follows that if B is in NC then A is also in NC. We say that a problem B is *P-hard* if for any problem A in P, A is NC-reducible to B. We say that a problem B is *P-complete* if it belong to P and is P-hard. The P-complete problems can be viewed as the "hardest" problems in P with respect to the class NC because they have the following property: If a P-complete problem B is in NC, then every problem in P is in NC. The question of whether there exists a P-complete problem that is in NC—that is, whether NC = P—is an important open problem, comparable to the problem of whether P = NP for sequential computation.

It is not obvious that P-complete problems exist, but it turns out that many natural problems are P-complete. In this section, we give several examples of such problems.

26.6.1 Straight-Line Code Problem for Boolean Expressions

Just as the satisfiability of CNF Boolean expressions is a fundamental NP-complete problem, the satisfiability of a certain form of a set of Boolean equations is a fundamental P-complete problem. Consider the following set of Boolean equations:

$$\begin{aligned}
x_1 &= B_0\\
x_2 &= B_1(x_1)\\
x_3 &= B_2(x_1, x_2)\\
&\;\;\vdots\\
x_n &= B_{n-1}(x_1, \dots, x_{n-1}),
\end{aligned}$$

where B_0 is a constant and $B_i(x_1, \dots, x_{i-1})$ is a Boolean expression involving only the Boolean variables $x_1, \dots, x_{i-1}$, $i = 2, \dots, n$. Regarding = as the assignment operator, we call these equations *straight-line code*. The problem of computing the value of x_i, $i = 1, \dots, n$, is called the *straight-line code (SLC) problem*.

Clearly, the straightforward sequential algorithm for solving the SLC problem, which successively evaluates x_i, $i = 1, \dots, n$, has linear computing time, where the input size is the total number m of symbols over all n equations (a symbol is counted k times if it occurs in k equations). Thus, the SLC problem is in P.

The SLC problem can be formulated as a decision problem, in which we ask the question, "Given $i \in \{1, \dots, n\}$, does x_i have the value T?" We can obtain a solution to the SLC problem by solving this decision problem concurrently for each x_i, $i = 1, \dots, n$. The SLC problem is our first example of a P-complete problem. The proof that the SLC problem is P-complete is beyond the scope of this book. In the remainder of this section, we illustrate NC reductions from the SLC problem to several other problems in P, thereby showing that they too are P-complete.

26.6.2 Straight-Line Code Without Disjunction

Straight-line code without disjunction (SLCWD) is a straight-line code in which the operation of disjunction is not used. We now describe an NC reduction of the SLC problem to the SLCWD problem, thereby showing that the SLCWD problem is P-complete. We employ the following useful identities, known as DeMorgan's laws, where $\neg$ represents the unary operation of negation.

roposition 26.6.1 DeMorgan's Laws

For x, $y \in \{T, F\}$, the following laws apply:

1. $\neg (x \vee y) = \neg x \wedge \neg y$,
2. $\neg(x \wedge y) = \neg x \vee \neg y$. □

Using the fact that $\neg(\neg x) = x$ and DeMorgan's law 1 we have

$$x \vee y = \neg(\neg x \wedge \neg y).$$

We now consider the transformation f that maps an instance I of the SLC problem to the instance $f(I)$ of the SLCWD problem, where $f(I)$ is constructed from I by concurrently replacing each occurrence of $x \vee y$, where x and y are Boolean expressions, with $\neg (\neg x \wedge \neg y)$. Each such replacement involves changing $\vee$ to $\wedge$ and inserting three NOT operations and a left and right parenthesis in the appropriate place. We leave it as an exercise to design.an NC algorithm for computing $f(I)$ from I. Figure 26.4 illustrates this transformation for a sample instance of the SLC problem. Since $f(I)$ can be computed in polylogarithmic time using a polynomial number of processors, the SLC problem is NC-reducible to the SLCWD problem, thereby proving that the SLCWD problem is P-complete.

FIGURE 26.4

The NC reduction transformation f from the general SLC problem to the SLCWD problem shown for a sample instance I of the SLC problem

$$\begin{aligned}
x_1 &= T\\
x_2 &= \neg x_1\\
x_3 &= \neg x_1 \wedge (x_1 \vee x_2)\\
x_4 &= (x_3 \wedge \neg(x_1 \vee x_2)) \vee (x_1 \wedge \neg x_3)
\end{aligned}$$

Instance I of the general SLC problem

$$\begin{aligned}
x_1 &= T\\
x_2 &= \neg x_1\\
x_3 &= \neg x_1 \wedge (\neg(\neg x_1 \wedge x_2))\\
x_4 &= \neg(\neg(x_3 \wedge \neg(\neg(\neg x_1 \wedge \neg x_2))) \wedge \neg(x_1 \wedge \neg x_3))
\end{aligned}$$

Instance $f(I)$ of the SLCWD problem

26.6.3 NOR Straight-Line Code

The operation NOR is defined by

$$x \text{ NOR } y = \neg(x_j \vee x_k)$$

NOR straight-line code is straight-line code where all assignments are of the following two forms:

$$\begin{aligned}
x_i &= T,\\
x_i &= x_j \text{ NOR } x_k, \quad j, k < i.
\end{aligned} \qquad \textbf{(26.6.1)}$$

We now show that the problem of solving NOR straight-line code (the NOR SLC problem) is P-complete. We first describe a transformation g from an instance I of the general SLC problem to a set of equations involving only NOR operations and assignments to T, but not necessarily of the form given in Formula (26.6.1). We construct $g(I)$ from I by concurrently performing the following replacement operations:

1. F is replaced with T NOR T.
2. $\neg x$ is replaced with x NOR x. **(26.6.2)**
3. $x \wedge y$ is replaced with (x NOR x) NOR (y NOR y).

In these operations, x and y are Boolean expressions. We leave it as an exercise to describe an NC transformation h that converts $g(I)$ into an instance of the NOR SLC problem by replacing each equation with a set of equations of the form given in Formula (26.6.1). The transformation $f(I) = h(g(I))$ yields an NC reduction from the general SLC problem to the NOR SLC problem, showing that the NOR SLC problem is P-complete.

26.6.4 Lexicographically First Maximal Independent Set

Let $G = (V, E)$ be a graph with vertex set $V = \{1, \ldots, n\}$ and edge set E of size m. A *maximal independent set* is an independent set U (no two vertices of U are adjacent) such that $U \cup \{v\}$ is not an independent set for any vertex $v \in V$. The *lexicographical ordering* of 2^V (the *power set* of V, consisting of all subsets of V) is a linear ordering such that a subset $A < B$ if either $A \subset B$ or the minimum element in $A \backslash B$ is smaller than the minimum element in $B \backslash A$. For example,

$$\{2, 4, 5\} < \{2, 4, 5, 9\} < \{3\} < \{5, 7\}.$$

The *lexicographically first maximal independent set* (LFMIS) is the maximal independent set U whose lexicographical order is smallest among all the maximal independent sets of G. The lexicographically first maximal independent set is illustrated for a sample graph in Figure 26.5.

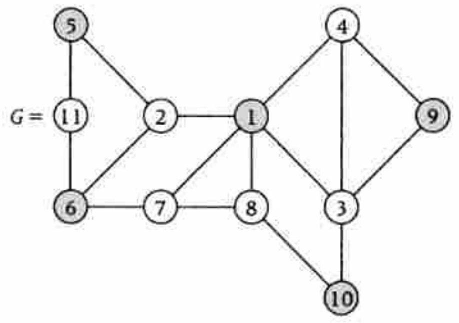


FIGURE 26.5 Lexicographically first maximal independent set of G is $\{1, 5, 6, 9, 10\}$.

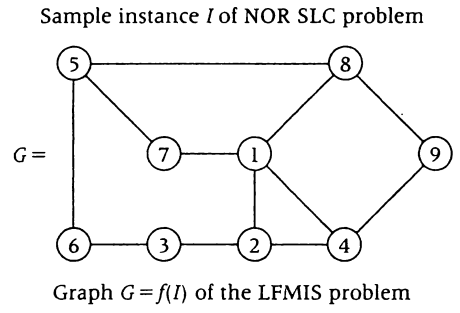
In this subsection, we show that the LFMIS problem is P-complete. (As with the SLC problem, the LFMIS problem can be formulated as a decision problem, where we ask the question, "Given a vertex $i \in V$, does i belong to the lexicographically first maximal independent set?")

First, we show that the LFMIS problem is in P by designing a polynomial-time procedure *SequentialLFMIS* for computing the lexicographically first maximal independent set U. Procedure *SequentialLFMIS* maintains a set R of remaining vertices where R is initially the set V of all vertices. At each stage, procedure *SequentialLFMIS* (1) chooses the smallest vertex u from R, (2) adds u to U, and (3) removes vertex u and all vertices adjacent to u from R. Procedure *SequentialLFMIS* terminates when R is empty. We leave the pseudocode for procedure *SequentialLFMIS* as an exercise. It is easily verified that *SequentialLFMIS* has complexity $\Theta(m + n)$, where m denotes the number of edges of G.

We now show how to NC-reduce an instance I of the NOR SLC problem to an instance $f(I)$ of the LFMIS problem. Instance $f(I)$ is a graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$, where vertex i corresponds to variable x_i of I, $i = 1, 2, \ldots, n$. For each assignment $x_i = x_j$ NOR x_k of I there is an edge $\{i, j\}$ and an edge $\{i, k\}$. It is easily verified that G can be computed in polylogarithmic time on a PRAM using a polynomial number of processors. It can also be verified that a node i of G belongs to the lexicographically first maximal independent set if and only if the value of x_i is T in I. In Figure 26.6, the NC transformation f is shown for a sample instance I of the NOR SLC problem.

FIGURE 26.6

The transformation from a sample instance I of the NOR SLC problem to the instance $G = f(I)$ of the LFMIS problem.


Sample instance I of NOR SLC problem
$G=$
5
8
7
1
9
6
3
2
4
Graph $G=f(I)$ of the LFMIS problem

26.7 Closing Remarks

Cobham introduced the class P in 1964. Edmonds, who independently proposed the class P in 1965, also introduced the class NP and conjectured that P $\neq$ NP. Cook, who introduced the notion of NP-completeness in 1971, showed that CNF SAT and 3-CNF SAT are NP-complete. Shortly thereafter, Levin independently discovered the notion and also proved the existence of NP-complete problems. Karp popularized the notion of NP-completeness by establishing that a wide variety of important and practical problems are NP-complete.

Our discussions of the notions of NP, NP-completeness, and P-completeness were somewhat informal in the sense that we did not establish a formal model of computation such as a Turing machine. The interested student can find formal treatments in the references.

References and Suggestions for Further Reading

Garey, M., and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: W. H. Freeman, 1979. This classic reference to the theory of NP-completeness contains a list of hundreds of important NP-complete problems.

Books containing extensive discussions of NP-completeness:

Horowitz, E., and S. Sahni. *Fundamentals of Computer Algorithms.* Potomac, MD: Computer Science Press, 1978.

Papadimitriou, C., and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Englewood Cliffs, NJ: Prentice-Hall, 1982.

Greenlaw, R., H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* New York: Oxford University Press, 1995. A recent text devoted to P-completeness.

EXERCISES

Sections 26.1 and 26.2 The Classes P and NP and Reducibility

26.1 Suppose that the weights and values of the objects in the 0/1 knapsack problem are all integers. Show that the 0/1 knapsack problem and its associated decision version are polynomially equivalent. (*Hint*: Use a binary search technique.)

26.2 Given an integer n, it was shown by Pratt that n is prime if and only if there exists an integer a such that

i. $a^{n-1} \equiv 1 \pmod{n}$.

ii. $a^{(n-1)/p} \not\equiv 1 \pmod{n}$ for all prime divisors p of $n - 1$

Based on this result, show that the prime-testing problem is in NP.

26.3 Prove a result for the traveling salesman problem analogous to that of Exercise 26.1.

26.4 Prove Proposition 26.2.1.

26.5 Prove Proposition 26.2.2.

26.6 Prove Proposition 26.2.3.

Sections 26.3 and 26.4 NP-Complete Problems

26.7 Verify that each of the NP-complete problems in Section 26.4 belong to NP.

26.8 Prove Proposition 26.3.1

26.9 Show that 2-CNF SAT is in P.

26.10 This exercise refers to the mapping f from CNF SAT to 3-CNF SAT described in Example 26.4.1.

a. Show that the input size of $f(I)$ is at most 12 times the input size of I for any instance I of CNF SAT.

b. Show in cases 2 and 3, each clause C_i of I is satisfiable if and only if the conjunction of the clauses replacing C_i is satisfiable.

26.11 In Example 26.4.4, verify that the set R' of remaining households can be constructed and that the resulting set M' is a matching.

26.12 Show that the instance F, S of the 3-exact cover problem is a yes instance if and only if the instance $a_1, a_2, \ldots, a_n$ and C constructed in Example 26.4.6 is a yes instance of the sum of subsets problem.

26.13 Show that the problem of determining whether a graph G is bipartite (has a proper 2-coloring) is in P.

26.14 In Example 26.4.7, show that the graph G constructed from the instance I of 3-CNF SAT is $(n + 1)$-colorable if and only if I is satisfiable.

26.15 Given a multiset $S = \{s_1, s_2, \ldots, s_n\}$ of positive integers, the *partition problem* asks whether S can be partitioned into two subsets having the same sum. Show that the partition problem is NP-complete. (*Hint*: Show that sum of subsets $\propto$ partition.)

26.16 Show that the 0/1 knapsack problem is NP-complete. (*Hint*: Show that partition $\propto$ 0/1 knapsack [partition is defined in Exercise 26.15].)

26.17 Suppose we have a set of n objects $\{x_1, x_2, \ldots, x_n\}$ of sizes $\{s_1, s_2, \ldots, s_n\}$, where $0 < s_i < 1$, $i = 1, \ldots, n$. The *bin-packing* optimization problem is to place the objects into bins of unit size using the minimum number of bins. Each bin can contain any subset of objects whose total size does not exceed one. The decision version asks for a given integer k whether we can pack the objects using no more than k bins. Show that the bin problem is NP-complete. (*Hint*: Show that sum of subsets $\propto$ bin packing.)

Section 26.5 The Class co-NP

26.18 Prove Theorem 26.5.1.

Section 26.6 The Classes NC and P-Complete

26.19 Prove Proposition 26.6.1 (DeMorgan's Laws).

26.20 Use induction to prove the following generalization of DeMorgan's laws. For $x_1, x_2, \dots, x_k \in \{T, F\}$,

a. $\neg(x_1 \wedge x_2 \wedge \cdots \wedge x_k) = \neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_k$.

b. $\neg(x_1 \vee x_2 \vee \cdots \vee x_k) = \neg x_1 \wedge \neg x_2 \wedge \cdots \wedge \neg x_k$.

26.21 Let f be the transformation that maps an instance I of the SLC problem to the instance $f(I)$ of the SLCWD problem, where $f(I)$ is constructed from I by concurrently replacing each occurrence of $x \vee y$, where x and y are Boolean expressions, with $\neg(\neg x \wedge \neg y)$. Design an NC algorithm for computing $f(I)$ from I.

26.22 a. Design an NC algorithm for performing the replacement operations given in Formula (26.6.2) to obtain $g(I)$ from an instance I of the SLCWD problem.

b. Design an NC algorithm for converting $g(I)$ to an instance I of the NOR SLC problem.

26.23 a. Give pseudocode for the procedure *SequentialLFMIS* for computing the lexicographically first maximal independent set of a graph G.

b. Analyze the complexity of procedure *SequentialLFMIS*.

26.24 a. Design an NC algorithm for performing the reduction from an instance I of the NOR SLC problem to the instance $G = f(I)$ of the LFMIS problem shown in Figure 26.5.

b. Show that x_i has the value T in instance I of the NOR SLC problem if and only if vertex i is in the lexicographically maximal independent set in $G = f(I)$.