1.3 Designing algorithms

1.2-2
Consider linear search again (see Exercise 1.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in \( \Theta \)-notation? Justify your answers.

1.2-3
Consider the problem of determining whether an arbitrary sequence \( \langle x_1, x_2, \ldots, x_n \rangle \) of \( n \) numbers contains repeated occurrences of some number. Show that this can be done in \( \Theta(n \log n) \) time, where \( \log n \) stands for \( \log_2 n \).

1.2-4
Consider the problem of evaluating a polynomial at a point. Given \( n \) coefficients \( a_0, a_1, \ldots, a_{n-1} \) and a real number \( x \), we wish to compute \( \sum_{i=0}^{n-1} a_i x^i \). Describe a straightforward \( \Theta(n^2) \)-time algorithm for this problem. Describe a \( \Theta(n) \)-time algorithm that uses the following method (called Horner's rule) for rewriting the polynomial:

\[
\sum_{i=0}^{n-1} a_i x^i = (\cdots ((a_{n-1}x + a_{n-2})x + \cdots + a_1)x + a_0).
\]

1.2-5
Express the function \( n^3/1000 - 100n^2 - 190n + 3 \) in terms of \( \Theta \)-notation.

1.2-6
How can we modify almost any algorithm to have a good best-case running time?

1.3 Designing algorithms

There are many ways to design algorithms. Insertion sort uses an incremental approach: having sorted the subarray \( A[1 \ldots j-1] \), we insert the single element \( A[j] \) into its proper place, yielding the sorted subarray \( A[1 \ldots j] \).

In this section, we examine an alternative design approach, known as "divide-and-conquer." We shall use divide-and-conquer to design a sorting algorithm whose worst-case running time is much less than that of insertion sort. One advantage of divide-and-conquer algorithms is that their running times are often easily determined using techniques that will be introduced in Chapter 4.
1.3.1 The divide-and-conquer approach

Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

**Divide** the problem into a number of subproblems.

**Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

**Combine** the solutions to the subproblems into the solution for the original problem.

The **merge sort** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

**Divide:** Divide the $n$-element sequence to be sorted into two subsequences of $n/2$ elements each.

**Conquer:** Sort the two subsequences recursively using merge sort.

**Combine:** Merge the two sorted subsequences to produce the sorted answer.

We note that the recursion “bottoms out” when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the “combine” step. To perform the merging, we use an auxiliary procedure $\text{MERGE}(A, p, q, r)$, where $A$ is an array and $p$, $q$, and $r$ are indices numbering elements of the array such that $p \leq q < r$. The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are in sorted order. It **merges** them to form a single sorted subarray that replaces the current subarray $A[p..r]$.

Although we leave the pseudocode as an exercise (see Exercise 1.3-2), it is easy to imagine a **MERGE** procedure that takes time $O(n)$, where $n = r - p + 1$ is the number of elements being merged. Returning to our card-playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining
input pile and place it face down onto the output pile. Computationally, each basic step takes constant time, since we are checking just two top cards. Since we perform at most \( n \) basic steps, merging takes \( \Theta(n) \) time.

We can now use the \textsc{Merge} procedure as a subroutine in the merge sort algorithm. The procedure \textsc{Merge-Sort}(\( A, p, r \)) sorts the elements in the subarray \( A[p..r] \). If \( p \geq r \), the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index \( q \) that partitions \( A[p..r] \) into two subarrays: \( A[p..q] \), containing \( \lceil n/2 \rceil \) elements, and \( A[q+1..r] \), containing \( \lfloor n/2 \rfloor \) elements.  

\textsc{Merge-Sort}(\( A, p, r \))
\[
\begin{align*}
1 & \quad \text{if } p < r \\
2 & \quad \text{then } q \leftarrow \lfloor (p + r)/2 \rfloor \\
3 & \quad \textsc{Merge-Sort}(A, p, q) \\
4 & \quad \textsc{Merge-Sort}(A, q + 1, r) \\
5 & \quad \textsc{Merge}(A, p, q, r)
\end{align*}
\]

To sort the entire sequence \( A = \langle A[1], A[2], \ldots, A[n] \rangle \), we call \textsc{Merge-Sort}(\( A, 1, \text{length}[A] \)), where once again \( \text{length}[A] = n \). If we look at the operation of the procedure bottom-up when \( n \) is a power of two, the algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length \( n/2 \) are merged to form the final sorted sequence of length \( n \). Figure 1.3 illustrates this process.

1.3.2 Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call to itself, its running time can often be described by a \textit{recurrence equation} or \textit{recurrence}, which describes the overall running time on a problem of size \( n \) in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

A recurrence for the running time of a divide-and-conquer algorithm is based on the three steps of the basic paradigm. As before, we let \( T(n) \) be the running time on a problem of size \( n \). If the problem size is small enough, say \( n \leq c \) for some constant \( c \), the straightforward solution takes constant time, which we write as \( \Theta(1) \). Suppose we divide the problem into \( a \) subproblems, each of which is \( 1/b \) the size of the original. If we take \( D(n) \) time to divide the problem into subproblems and \( C(n) \) time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

\[ \text{(The expression } \lfloor x \rfloor \text{ denotes the least integer greater than or equal to } x, \text{ and } \lceil x \rceil \text{ denotes the greatest integer less than or equal to } x. \text{ These notations are defined in Chapter 2.)} \]
Chapter 1  Introduction

sorted sequence

\[ \begin{array}{ccccccc}
1 & 2 & 2 & 3 & 4 & 5 & 6 \\
\end{array} \]

merge

\[ \begin{array}{cccc}
2 & 4 & 5 & 6 \\
\end{array} \]

merge

\[ \begin{array}{cccc}
2 & 5 & 4 & 6 \\
\end{array} \]

merge

\[ \begin{array}{cccc}
5 & 2 & 4 & 6 \\
\end{array} \]

merge

\[ \begin{array}{cccc}
5 & 2 & 4 & 6 \\
\end{array} \]

merge

\[ \begin{array}{cccc}
1 & 2 & 3 & 6 \\
\end{array} \]

merge

\[ \begin{array}{cccc}
1 & 3 & 2 & 6 \\
\end{array} \]

initial sequence

Figure 1.3  The operation of merge sort on the array \( A = \{5, 2, 4, 6, 1, 3, 2, 6\} \). The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq c, \\
\frac{aT(n/b) + D(n) + C(n)}{} & \text{otherwise}. 
\end{cases} \]

In Chapter 4, we shall see how to solve common recurrences of this form.

Analysis of merge sort

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of two. Each divide step then yields two subsequence of size exactly \( n/2 \). In Chapter 4, we shall see that this assumption does not affect the order of growth of the solution to the recurrence.

We reason as follows to set up the recurrence for \( T(n) \), the worst-case running time of merge sort on \( n \) numbers. Merge sort on just one element takes constant time. When we have \( n > 1 \) elements, we break down the running time as follows.

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, \( D(n) = \Theta(1) \).

Conquer: We recursively solve two subproblems, each of size \( n/2 \), which contributes \( 2T(n/2) \) to the running time.

Combine: We have already noted that the MERGE procedure on an \( n \)-element subarray takes time \( \Theta(n) \), so \( C(n) = \Theta(n) \).
1.3 Designing algorithms

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of $n$, that is, $\Theta(n)$. Adding it to the $2T(n/2)$ term from the “conquer” step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
2T(n/2) + \Theta(n) & \text{if } n > 1.
\end{cases}$$

In Chapter 4, we shall show that $T(n)$ is $\Theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$. For large enough inputs, merge sort, with its $\Theta(n \lg n)$ running time, outperforms insertion sort, whose running time is $\Theta(n^2)$, in the worst case.

Exercises

1.3-1
Using Figure 1.3 as a model, illustrate the operation of merge sort on the array $A = (3, 41, 52, 26, 38, 57, 9, 49)$.

1.3-2
Write pseudocode for $\text{Merge}(A, p, q, r)$.

1.3-3
Use mathematical induction to show that the solution of the recurrence

$$T(n) = \begin{cases} 
2 & \text{if } n = 2, \\
2T(n/2) + n & \text{if } n = 2^k, k > 1
\end{cases}$$

is $T(n) = n \lg n$.

1.3-4
Insertion sort can be expressed as a recursive procedure as follows. In order to sort $A[1..n]$, we recursively sort $A[1..n-1]$ and then insert $A[n]$ into the sorted array $A[1..n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

1.3-5
Referring back to the searching problem (see Exercise 1.1-3), observe that if the sequence $A$ is sorted, we can check the midpoint of the sequence against $v$ and eliminate half of the sequence from further consideration. Binary search is an algorithm that repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

1.3-6
Observe that the while loop of lines 5–7 of the $\text{INSERTION-SORT}$ procedure in Section 1.1 uses a linear search to scan (backward) through the sorted subarray $A[1..j-1]$. Can we use a binary search (see Exercise 1.3-5)
where we take $\Pr \{B_1\} = 1$ as an initial condition. In other words, the probability that $b_1, b_2, \ldots, b_k$ are distinct birthdays is the probability that $b_1, b_2, \ldots, b_{k-1}$ are distinct birthdays times the probability that $b_k \neq b_i$ for $i = 1, 2, \ldots, k - 1$, given that $b_1, b_2, \ldots, b_{k-1}$ are distinct.

If $b_1, b_2, \ldots, b_{k-1}$ are distinct, the conditional probability that $b_k \neq b_i$ for $i = 1, 2, \ldots, k - 1$ is $(n - k + 1)/n$, since out of the $n$ days, there are $n - (k - 1)$ that are not taken. By iterating the recurrence (6.46), we obtain

$$
\Pr \{B_k\} = \Pr \{B_1\} \Pr \{A_1 \mid B_1\} \Pr \{A_2 \mid B_2\} \cdots \Pr \{A_{k-1} \mid B_{k-1}\}
= 1 \cdot \left(1 - \frac{1}{n}\right) \left(1 - \frac{2}{n}\right) \cdots \left(1 - \frac{k-1}{n}\right).
$$

The inequality (2.7), $1 + x \leq e^x$, gives us

$$
\Pr \{B_k\} \leq e^{-1/n} e^{-2/n} \cdots e^{-(k-1)/n}
= e^{-\sum_{i=1}^{k-1} i/n}
= e^{-k(k-1)/2n}
\leq \frac{1}{2}
$$

when $-k(k-1)/2n \leq \ln(1/2)$. The probability that all $k$ birthdays are distinct is at most $1/2$ when $k(k-1) \geq 2n \ln 2$ or, solving the quadratic equation, when $k \geq \left(1 + \sqrt{1 + (8 \ln 2)/n}\right)/2$. For $n = 365$, we must have $k \geq 23$. Thus, if at least 23 people are in a room, the probability is at least $1/2$ that at least two people have the same birthday. On Mars, a year is 669 Martian days long; it therefore takes 31 Martians to get the same effect.

**Another method of analysis**

We can use the linearity of expectation (equation (6.26)) to provide a simpler but approximate analysis of the birthday paradox. For each pair $(i, j)$ of the $k$ people in the room, let us define the random variable $X_{ij}$, for $1 \leq i < j \leq k$, by

$$
X_{ij} = \begin{cases} 
1 & \text{if person } i \text{ and person } j \text{ have the same birthday,} \\
0 & \text{otherwise.}
\end{cases}
$$

The probability that two people have matching birthdays is $1/n$, and thus by the definition of expectation (6.23),

$$
E[X_{ij}] = 1 \cdot (1/n) + 0 \cdot (1 - 1/n)
= 1/n.
$$

The expected number of pairs of individuals having the same birthday is, by equation (6.24), just the sum of the individual expectations of the
pairs, which is
\[
\sum_{i=2}^{k} \sum_{j=1}^{i-1} E[X_{ij}] = \binom{k}{2} \frac{1}{n} = \frac{k(k-1)}{2n}.
\]
When \(k(k - 1) \geq 2n\), therefore, the expected number of pairs of birthdays is at least 1. Thus, if we have at least \(\sqrt{2n}\) individuals in a room, we can expect at least two to have the same birthday. For \(n = 365\), if \(k = 28\), the expected number of pairs with the same birthday is \((28 \cdot 27)/(2 \cdot 365) \approx 1.0356\). Thus, with at least 28 people, we expect to find at least one matching pair of birthdays. On Mars, where a year is 669 Martian days long, we need at least 38 Martians.

The first analysis determined the number of people required for the probability to exceed 1/2 that a matching pair of birthdays exists, and the second analysis determined the number such that the expected number of matching birthdays is 1. Although the numbers of people differ for the two situations, they are the same asymptotically: \(\Theta(\sqrt{n})\).

6.6.2 Balls and bins

Consider the process of randomly tossing identical balls into \(b\) bins, numbered \(1, 2, \ldots, b\). The tosses are independent, and on each toss the ball is equally likely to end up in any bin. The probability that a tossed ball lands in any given bin is \(1/b\). Thus, the ball-tossing process is a sequence of Bernoulli trials with a probability \(1/b\) of success, where success means that the ball falls in the given bin. A variety of interesting questions can be asked about the ball-tossing process.

**How many balls fall in a given bin?** The number of balls that fall in a given bin follows the binomial distribution \(b(k; n, 1/b)\). If \(n\) balls are tossed, the expected number of balls that fall in the given bin is \(n/b\).

**How many balls must one toss, on the average, until a given bin contains a ball?** The number of tosses until the given bin receives a ball follows the geometric distribution with probability \(1/b\), and thus the expected number of tosses until success is \(1/(1/b) = b\).

**How many balls must one toss until every bin contains at least one ball?** Let us call a toss in which a ball falls into an empty bin a “hit.” We want to know the average number \(n\) of tosses required to get \(b\) hits.

The hits can be used to partition the \(n\) tosses into stages. The \(i\)th stage consists of the tosses after the \((i - 1)\)st hit until the \(i\)th hit. The first stage consists of the first toss, since we are guaranteed to have a hit when all bins are empty. For each toss during the \(i\)th stage, there are \(b - i + 1\) empty bins. Thus, for all tosses in the \(i\)th stage, the probability of obtaining a hit is \((b - i + 1)/b\).
13.1 What is a binary search tree?

Figure 13.1 Binary search trees. For any node $x$, the keys in the left subtree of $x$ are at most $key[x]$, and the keys in the right subtree of $x$ are at least $key[x]$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. (a) A binary search tree on 6 nodes with height 2. (b) A less efficient binary search tree with height 4 that contains the same keys.

The keys in a binary search tree are always stored in such a way as to satisfy the **binary-search-tree property**: Let $x$ be a node in a binary search tree. If $y$ is a node in the left subtree of $x$, then $key[y] \leq key[x]$. If $y$ is a node in the right subtree of $x$, then $key[x] \leq key[y]$.

Thus, in Figure 13.1(a), the key of the root is 5, the keys 2, 3, and 5 in its left subtree are no larger than 5, and the keys 7 and 8 in its right subtree are no smaller than 5. The same property holds for every node in the tree. For example, the key 3 in Figure 13.1(a) is no smaller than the key 2 in its left subtree and no larger than the key 5 in its right subtree.

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an **inorder tree walk**. This algorithm derives its name from the fact that the key of the root of a subtree is printed between the values in its left subtree and those in its right subtree. (Similarly, a **preorder tree walk** prints the root before the values in either subtree, and a **postorder tree walk** prints the root after the values in its subtrees.) To use the following procedure to print all the elements in a binary search tree $T$, we call \texttt{INORDER-TREE-WALK(root[T])}.

**INORDER-TREE-WALK($x$)**
\begin{enumerate}
  \item If $x \neq \texttt{NIL}$
  \item \hspace{1em} then \texttt{INORDER-TREE-WALK(left[$x$])}
  \item \hspace{1em} print $key[x]$
  \item \hspace{1em} \texttt{INORDER-TREE-WALK(right[$x$])}
\end{enumerate}
Chapter 13  Binary Search Trees

As an example, the inorder tree walk prints the keys in each of the two binary search trees from Figure 13.1 in the order 2, 3, 5, 5, 7, 8. The correctness of the algorithm follows by induction directly from the binary-search-tree property. It takes \( \Theta(n) \) time to walk an \( n \)-node binary search tree, since after the initial call, the procedure is called recursively exactly twice for each node in the tree—once for its left child and once for its right child.

Exercises

13.1-1
Draw binary search trees of height 2, 3, 4, 5, and 6 on the set of keys \{1, 4, 5, 10, 16, 17, 21\}.

13.1-2
What is the difference between the binary-search-tree property and the heap property (7.1)? Can the heap property be used to print out the keys of an \( n \)-node tree in sorted order in \( O(n) \) time? Explain how or why not.

13.1-3
Give a nonrecursive algorithm that performs an inorder tree walk. (Hint: There is an easy solution that uses a stack as an auxiliary data structure and a more complicated but elegant solution that uses no stack but assumes that two pointers can be tested for equality.)

13.1-4
Give recursive algorithms that perform preorder and postorder tree walks in \( \Theta(n) \) time on a tree of \( n \) nodes.

13.1-5
Argue that since sorting \( n \) elements takes \( \Omega(n \log n) \) time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of \( n \) elements takes \( \Omega(n \log n) \) time in the worst case.

13.2  Querying a binary search tree

The most common operation performed on a binary search tree is searching for a key stored in the tree. Besides the \texttt{SEARCH} operation, binary search trees can support such queries as \texttt{MINIMUM}, \texttt{MAXIMUM}, \texttt{SUCCESSOR}, and \texttt{PREDECESSOR}. In this section, we shall examine these operations and show that each can be supported in time \( O(h) \) on a binary search tree of height \( h \).
13.2 Querying a binary search tree

Figure 13.2 Queries on a binary search tree. To search for the key 13 in the tree, the path 15 \(\rightarrow\) 6 \(\rightarrow\) 7 \(\rightarrow\) 13 is followed from the root. The minimum key in the tree is 2, which can be found by following left pointers from the root. The maximum key 20 is found by following right pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

Searching

We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key \(k\), TREE-SEARCH returns a pointer to a node with key \(k\) if one exists; otherwise, it returns NIL.

\[
\text{TREE-SEARCH}(x, k)
\]

\begin{verbatim}
1  if x = NIL or k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE-SEARCH(left[x], k)
5  else return TREE-SEARCH(right[x], k)
\end{verbatim}

The procedure begins its search at the root and traces a path downward in the tree, as shown in Figure 13.2. For each node \(x\) it encounters, it compares the key \(k\) with \(key[x]\). If the two keys are equal, the search terminates. If \(k\) is smaller than \(key[x]\), the search continues in the left subtree of \(x\), since the binary-search-tree property implies that \(k\) could not be stored in the right subtree. Symmetrically, if \(k\) is larger than \(key[k]\), the search continues in the right subtree. The nodes encountered during the recursion form a path downward from the root of the tree, and thus the running time of TREE-SEARCH is \(O(h)\), where \(h\) is the height of the tree.

The same procedure can be written iteratively by "unrolling" the recursion into a while loop. On most computers, this version is more efficient.
Iterative-Tree-Search($x, k$)
1 \textbf{while} $x \neq \text{NIL}$ and $k \neq \text{key}[x]$
2 \hspace{1em} \textbf{do} \text{if} $k < \text{key}[x]$
3 \hspace{2em} \text{then} $x \leftarrow \text{left}[x]$
4 \hspace{2em} \text{else} $x \leftarrow \text{right}[x]$
5 \textbf{return} $x$

Minimum and maximum

An element in a binary search tree whose key is a minimum can always be found by following left child pointers from the root until a NIL is encountered, as shown in Figure 13.2. The following procedure returns a pointer to the minimum element in the subtree rooted at a given node $x$.

Tree-Minimum($x$)
1 \textbf{while} left[$x$] \neq \text{NIL}
2 \hspace{1em} \textbf{do} $x \leftarrow \text{left}[x]$
3 \textbf{return} $x$

The binary-search-tree property guarantees that Tree-Minimum is correct. If a node $x$ has no left subtree, then since every key in the right subtree of $x$ is at least as large as key[$x$], the minimum key in the subtree rooted at $x$ is key[$x$]. If node $x$ has a left subtree, then since no key in the right subtree is smaller than key[$x$] and every key in the left subtree is not larger than key[$x$], the minimum key in the subtree rooted at $x$ can be found in the subtree rooted at left[$x$].

The pseudocode for Tree-Maximum is symmetric.

Tree-Maximum($x$)
1 \textbf{while} right[$x$] \neq \text{NIL}
2 \hspace{1em} \textbf{do} $x \leftarrow \text{right}[x]$
3 \textbf{return} $x$

Both of these procedures run in $O(h)$ time on a tree of height $h$, since they trace paths downward in the tree.

Successor and predecessor

Given a node in a binary search tree, it is sometimes important to be able to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the successor of a node $x$ is the node with the smallest key greater than key[$x$]. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys. The following procedure returns the successor of a node $x$ in a binary search tree if it exists, and NIL if $x$ has the largest key in the tree.