2

Design and Analysis Fundamentals

In this chapter we introduce some of the fundamental notions involved in the design and analysis of algorithms. Recursion is one of the most fundamental design strategy tools, so we discuss this technique in some detail. In addition to formulating some general guidelines for algorithm design, we also classify some major design strategies that capture the underlying strategy for most of the algorithms discussed in this text. We give a brief discussion of these major design strategies, and will return to them in more detail in Part II.

The basic concepts underlying the analysis of algorithms are also introduced in this chapter. There are two basic considerations in measuring the performance (*complexity*) of an algorithm: its time and space requirements. We are mainly concerned with analyzing the time complexity of an algorithm. Hence, when using the term *complexity*, unless otherwise stated we will mean the time complexity.

We measure the complexity of an algorithm by identifying a basic operation and then counting how many times the algorithm performs that basic operation for an input of size *n*. However, the number of basic operations performed by an algorithm usually varies for inputs of a given size. Thus, we use the concepts of best-case, worst-case, and average complexity of an algorithm. Occasionally we measure the complexity of an algorithm by considering several different operations performed by the algorithm and amortizing the performance over all these operations.

Determining the complexities of an algorithm exactly is often difficult, and in practice it is usually sufficient to obtain an asymptotic approximation to these complexities. To aid our description of asymptotic behavior, in this chapter we introduce and utilize the standard notation for order and asymptotic growth rate of functions.

2.1 Guidelines for Algorithm Design

Designing an algorithm for solving a complicated task may involve finding algorithms to solve many subtasks. The standard top-down approach to algorithm design involves outlining the major steps to a solution of the given problem and then performing a process of stepwise refinement into simpler and more manageable building blocks. The key to solving the original problem then boils down to finding algorithms for building-block problems that are conceptually self-contained such as evaluating polynomials, computing powers, searching, sorting, and so forth. In this text we are concerned with finding algorithms for solving such building-block problems.

The following may be regarded as a set of general guidelines for designing and analyzing an algorithm for solving a given problem.

Guidelines for Designing an Algorithm for Solving a Given Problem

1. Identify the appropriate data structures to model the problem.
2. Once an algorithm for solving the given problem has been found, verify its correctness and analyze its performance.
3. Decide whether a search for a more efficient algorithm is worthwhile.

In the next few sections we describe some of the fundamental concepts that support the design and analysis of algorithms.

2.2 Recursion

Recursion is one of the most powerful tools in the subject of algorithms and is basis of major design strategies such as divide-and-conquer. Recursion is utilized throughout the text, both for designing algorithms and analyzing their performance. In many situations, the solution to a given problem can be expressed naturally in terms of a solution (or solutions) to a smaller or simpler input (or set of inputs) to the same problem. This expression often takes the form of a *recurrence relation,* which includes an *initial condition* stating the known solution for an initial set of inputs, and which relates the solution to a given input to an input (or set of inputs) that is closer to the known initial inputs. Exploiting such recurrence relations is the essence of *recursive* algorithmic strategy. Not only do recurrence relations yield elegant algorithms for solving many problems, recurrence relations also arise frequently in the analysis of the complexity of algorithms, as we shall see in the next chapter.

Recursive algorithms are implemented in a programming language such as C++, Java, Python, and so forth, as recursive functions or procedures. Simply stated a recursive function or procedure is one that references itself in the code. When any function or procedure is called, there is an implicit stack used to save such things as the current values stored in registers, variables, as well as a return address to be used upon resolution of the call. This overhead can be significant for recursive calls, since there may be many unresolved calls generated by the algorithm. The overhead associated with recursion can be somewhat reduced by simulating the recursion using an explicitly implement stack in the algorithm (see Appendix B for more details on this simulation).

When there is only one recursive reference in the code, it is called *single-reference* recursion. A recursive function or procedure may contain several recursive reference statements in the code and still be single-reference recursion. For example, the following code for a function *f* containing the two recursive call statements (and no others)

```
if condition A then
        call f with given parameters
else
        call f with other parameters
endif
```

is still considered single-reference recursion, since only one of the two recursive references is executed by any given *current* call of the function *f*. The recursive function *Powers* given as our next example contains this type of construction. Recursive functions or procedures, which not only contain two or more recursive references in the code, but actually execute at least two of these references in a given *current* call, are referred to as *multiple-reference* recursion. The recursive sorting algorithms *MergeSort* and *QuickSort* described later in this chapter are examples of multiple-reference recursion.

2.2.1 Exponentiation Revisited

As an illustration of the power of recursion in formulating solutions to problems, we reconsider the problem of computing x^n for a positive integer n. In Chapter 1 we presented the algorithm *Left-to-Right Binary Method* for computing x^n. The following algorithm *Powers* is directly based on the recurrence relation

$$x^n = \begin{cases} (x * x)^{n/2} & \text{if } n \text{ is even,} \\ x * (x * x)^{(n-1)/2} & \text{otherwise.} \end{cases} \qquad \textbf{init cond.}\ x^1 = x \qquad (2.1.1)$$

```
function Powers(x,n) recursive
Input: x (a real number), n (a positive integer)
Output:  x^n
    if n = 1 then
        Pow ← x                  //initial condition
    else
        if even(n) then          //n even
            Pow ← Powers(x*x,n/2)
        else                     //n odd
            Pow ← x * Powers(x*x(n – 1)/2)
        endif
    endif
    return(Pow)
end Powers
```

For input (x,n), similar to the left-to-right binary method, *Powers* performs between $\lfloor \log_2 n \rfloor$ and $2\lfloor \log_2 n \rfloor$ multiplications (see Exercise 2.4). Similar to many recursive algorithms, the correctness of *Powers* seems obvious, since it is based directly on the obviously correct recurrence relation. A formal proof of the correctness of *Powers* is given in the next chapter.

The recursive algorithm *Powers* considered in this section gave an elegant solution to the problem at hand. We will see many more examples of the power and elegance of recursion throughout the text. In fact, recursion is the basis of two important general design strategies, divide-and-conquer and dynamic programming.

2.3 Data Structures and Algorithm Design

The performance of an algorithm often depends on the choice of data organization. An abstract data type (ADT) is concerned with a theoretical description of the organization of the data and the operations to be performed on this data. The implementation of an ADT determines a *data structure.*

We assume that you are familiar with the basics of data structures, but we will review some of the ideas in order to facilitate the discussion of algorithm design and analysis.

Many algorithms discussed in this text use a list as one of the primary abstract data types. Operations associated with a list include, for example, insertion, deletion, and

accessing a particular element in the list. For convenience, when describing algorithms we often implement the list ADT as an array. However, in practice there are a number of situations where a linked list implementation would improve performance. The array implementation of a list has the important advantage of allowing quick and *direct* access to any element of the list. However, there are some disadvantages in using an array. For example, inserting an element at position i of an array $L[0{:}n-1]$ with $L[0{:}m-1]$ storing the current list requires $m-i$ array assignments. Deleting an element also requires many array assignments. Another disadvantage relates to inefficient use of space. A particular list may occupy only a small portion of an array that has been declared to handle much larger lists. A linked list avoids these disadvantages at the expense of direct access.

We assume that you are familiar with the operations of inserting and deleting elements in a linked list. We also assume that you are familiar with stacks (LIFO list) and queues (FIFO lists), and their linked list and array implementations. A review of linear data structures is given in Appendix B. While you may also be familiar with some aspects of the tree ADT, we will treat this very important ADT in some detail in Chapter 4, including the establishment of some fundamental mathematical properties of trees. Trees are the basis for a number of important data structures, such as balanced search trees for rapid key searching, heaps for maintaining priority queues, forests for maintaining disjoint sets, tries for efficient storing and retrieving of character strings, and so forth. Various other data structures will be introduced throughout the text as appropriate in the design of efficient algorithms.

Modern software design incorporates the notion of object-oriented programming, in which data and associated functions are encapsulated into a single entity called an *object*. Objects themselves are instances of a class, which implements an ADT. Many modern programming languages, such as C++, Java, C#, support classes and objects in an object-oriented programming environment. Since a high-level understanding of algorithms is emphasized in this text, for simplicity the pseudocode utilized will not explicitly mention classes and objects. For example, we represent a list of size n as $L[0{:}n-1]$. We typically will implement the list as an array or linked list, without explicitly defining a list class, with associated member functions for insertion, deletion, and so forth. This enables focusing on the algorithmic issues without the encumbrance of object-oriented details.

2.4 Major Design Strategies

The study of algorithms has led to the formulation of a classification scheme, referred to as *major design strategies*, which capture the general design strategy underlying a large percentage of the algorithms in common use today. We now briefly describe each of these strategies. In Part II we will give a fuller treatment of each of the major design strategies. The Greedy method listed first typically leads to very efficient algorithms with a small, but important domain of applicability. Whereas the remaining strategies have a progressively wider applicability, the associated algorithms usually are less efficient.

THE GREEDY METHOD

The Greedy method solves optimization problems by making locally optimal choices and hoping that these choices lead to a globally optimal solution. While the Greedy method

leads to a simple and efficient algorithm, care must be taken to verify its correctness. For example, consider the familiar problem of making change (US coins pennies, nickels, dimes, and quarters). We wish to make change using the smallest number of coins. The greedy solution is to first use as many coins of the largest denomination (that is, quarters) as possible. Then use as many coins of the next largest denomination, and so forth. This method always works for the US coins denominations. However obvious this might seem, suppose you did not have any nickels. Then to make 30 cents change, the greedy solution would use 6 coins, 1 quarter and 5 pennies. However, 3 dimes would do the job! The Greedy method is usually very efficient, but, as seen from the above example, its correctness in a given situation must always be proved.

DIVIDE-AND-CONQUER

Divide-and-Conquer is essentially a special case of recursion in which a given problem is divided into (usually) two or more subproblems of the exactly the same type, and the solution to the problem expressed in terms of the solutions to the subproblems. The sorting algorithm merge sort is a good example. Given a list to sort, merge sort divides the list in half, recursively sorts the first half and second half, respectively, and then calls a procedure that we have called *Merge* to merge the two sorted sublists into a sorting of the entire list. Not only are Divide-and-Conquer algorithms perhaps the most important design strategy for sequential algorithms, they are naturally parallelizable, since separate processors can be assigned the tasks of solving the subproblem. Of course, this simple parallelization will usually not give sufficient speedup, and further parallelization is required. For example, in mergesort, good speedup requires finding a parallel merging procedure that has good speedup over the sequential procedure *Merge*.

DYNAMIC PROGRAMMING

Dynamic Programming is a technique in which solutions to a problem are built up from solutions to subproblems (similar to Divide-and-Conquer), but where all the smallest subproblems to a given problem are solved before proceeding on to the next smallest subproblems. For example, a dynamic programming solution to sorting could be based on a "bottom-up" version of merge sort, where the two-element sublists $L[0:1]$, $L[2:3]$, ... , $L[n-2:n-1]$ are sorted first, then the four-element sublists $L[0:3]$, $L[4:7]$, ... , $L[n-4:n-1]$ are sorted (again, using *Merge*), and so forth. (Note by way of contrast that the recursive version of merge sort will sort the entire first half of the sublist before proceeding to the second half of the sublist.) Dynamic programming is usually applied to situations where the problem is to optimize an objective function, and where the optimal solution to a problem must be built up from optimal solutions to subproblems. Its efficiency results by eliminating suboptimal subproblems when moving up to larger subproblems.

BACKTRACKING AND BRANCH-AND-BOUND

Backtracking and Branch-and-Bound refer to strategies to solve problems that have associated state-space trees. Backtracking searches the state-space tree using depth-first search, whereas Branch-and-Bound uses breadth-first search. These search strategies have wide applicability, and apply to most problems whose solution depends on making a

series of decisions. The state-space tree for the problem simply models all possible decisions that can be made at each stage. For example, the children of a node in a state-space tree modeling a board game such as checkers or chess consist of all legal moves available to the player whose move is represented by the node. As another example, consider the famous Traveling Salesman Problem (TSP) in which there are n cities, and a salesman, starting from a given city, wishes to travel to all $n - 1$ remaining cities and return in such a way as to minimize the total distance traveled over all such tours. With the starting city at the root, the children of a given node consist of all cities not yet visited by the partial tour represented by the given node (cities already visited).

The state-space tree is usually quite large, so that examining each node in the tree is not usually feasible. For example, the state-space tree for TSP has $(n - 1)!$ nodes. However, by using appropriate bounding functions, the searches are often efficient due to cutting off large subtrees of the state-space tree when it is determined that no solution can lie in these subtrees. For example, in TSP one could keep track of the best tour so far generated in searching the state-space tree, and cut-off any partial tour whose distance already exceeds the distance of this currently best tour found.

2.5 Analyzing Algorithm Performance

When you can measure what you are speaking about and can express it in numbers, you know something about it. But when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind.

—Lord Kelvin

To analyze the complexity of an algorithm, we usually identify a basic operation and count how many times an algorithm performs this basic operation. Analysis based on a suitably chosen basic operation yields measurements that are proportional to actual run time behavior exhibited when running the algorithm on various computers, so that the analysis is not dependent on a particular computer. Basic operations for some sample algorithms are shown in Figure 2.1.

Problem	Input of size n	Basic operation
Searching a list	lists with n elements	comparison
Sorting a list	lists with n elements	comparison
Multiplying two matrices	two n-by-n matrices	multiplication
Prime factorization	n-digit number	division
Evaluating a polynomial	polynomial of degree n	multiplication
Traversing a tree	tree with n nodes	accessing a node
Towers of Hanoi	n disks	moving a disk

Algorithm complexity as a function of input size n

Figure 2.1

We measure the complexity of an algorithm as a function of the *input size n* to the algorithm. For example, when searching or sorting a list, the input size is the number of elements n in the list; when evaluating a polynomial, the input size is either the degree of the polynomial or the number of nonzero coefficients in the polynomial; when multiplying two square n-by-n matrices, the input size is n; when testing whether an

integer is a prime, the input size is the number of digits of the integer; when traversing a tree, the input size is the number of nodes in the tree; and so forth (see Figure 2.1).

Be careful about what you take for the measure of input size. For example, consider the problem of computing x^p for an arbitrary positive integer p. The naive algorithm takes $p - 1$ multiplications, so that if you take $n = p$ as the input size, you would have an algorithm that performs $n - 1$ multiplications, which is linear in the input size. However, since $n = p$ is a single number (not the size of an aggregate such as a list), this is a deceptive measure of the size of the input. For example in modern security schemes today, such as RSA, the power x^p can involve a power p having hundreds of (binary) digits. Then, the naïve algorithm performing $p - 1$ multiplications would not complete in real time, even for a 64 digit number. From the point of view of storing such a p, it is a relatively small number (for example, a 128-digit binary number requires only 128 bits). However, to perform $p - 1 \geq 2^{127} - 1$ multiplications would take hundreds of years on today's fastest computers.

When analyzing the complexity of an algorithm computing x^p, what is typically taken as the measure of the input size is the number n of digits of p in its base two representation, that is, n is approximately equal to $\log_2 p$. The inefficiency of the naive method then becomes apparent, since the number of multiplications performed is approximately equal to 2^n, which is exponential in the input size.

For some algorithms the number of basic operations performed is the same for any input of size n. However, for many algorithms the number of basic operations performed can differ for two inputs of the same size. For the latter algorithms, we talk about *best-case*, *worst-case* and *average* complexities as functions of the input size n. The formal definitions of these complexities require the specification of the set of inputs of size n to an algorithm. An *input* I to an algorithm is the data, such as numbers, character strings, and records, on which the operations of the algorithm are performed. Let $\mathcal{I}_n$ denote the set of all inputs of size n to an algorithm. The second column of Figure 2.1 shows what is typically considered to be $\mathcal{I}_n$ for various types of problems and their associated algorithms. For $I \in \mathcal{I}_n$ let $\tau(I)$ denote the number of basic operations that are performed when the algorithm is executed with input I.

Definition 2.5.1 Best-Case Complexity

The *best-case complexity* of an algorithm is the function $B(n)$ such that $B(n)$ equals the *minimum* value of $\tau(I)$, where I varies over all inputs of size n. That is,

$$B(n) = \min\{\tau(I) \mid I \in \mathcal{I}_n\} \tag{2.5.1}$$

Far more important than $B(n)$ is the worst-case complexity $W(n)$.

Definition 2.5.2 Worst-Case Complexity

The *worst-case complexity* of an algorithm is the function $W(n)$ such that $W(n)$ equals the *maximum* value of $\tau(I)$, where I varies over all inputs of size n. That is,

$$W(n) = \max\{\tau(I) \mid I \in \mathcal{I}_n\} \tag{2.5.2}$$

A third important complexity measure of an algorithm is its average complexity $A(n)$, which depends not only on the input size n, but also on the probability distribution on the set of all inputs of size n. More precisely, if p: $\mathscr{I}_n \rightarrow [0,1]$ is a probability function defined on the input space $\mathscr{I}_n$, then

$$A(n) = \sum_{I \in \mathscr{I}_n} \tau(I)\, p(I) = E[\tau] \qquad (2.5.3)$$

Typically we assume that each input $I \in \mathscr{I}_n$ is equally likely, so that computing $A(n)$ is the usual notion of summing up all the values of $\tau(I)$, $I \in \mathscr{I}_n$ and then dividing this sum by the cardinality of $\mathscr{I}_n$. The formal discussion of $A(n)$ is postponed until Chapter 3, since this discussion proceeds more naturally when we have established various aspects of the asymptotic behavior of functions. It is in the context of average behavior that the best-case complexity $B(n)$ is useful as a lower bound for $A(n)$. Note also that

$$B(n) \leq A(n) \leq W(n).$$

Remark

In all of the problems mentioned so far, the input size was a function of a single variable n. However, sometimes the input size is most naturally a function of more than one variable. For example, suppose we want to find an efficient sorting algorithm for lists of size n with repeated elements. The input size is then a function of n and the number m of distinct elements in the list, where $1 \leq m \leq n$ (see *BingoSort* in the exercises). In such cases best-case, worst-case and average complexities are functions of the two variables n and m. Other examples arise in the study of graphs (see Chapter 5) where in input size is often taken as a function of the number n of vertices of a graph, and the number m of edges in the graph.

2.6 Design and Analysis of Some Basic Comparison-Based List Algorithms

A *comparison-based* algorithm for searching, finding the maximum and minimum values, or sorting a list is based on making comparison involving list elements and then making decisions based on these comparisons. In particular, comparison-based algorithms make no a priori assumption about the nature of the list elements, other than knowing their relative order. For comparison-based algorithms we can often reduce or transform the sample space to a simpler sample space without affecting the average complexity. For example, the lists ($\sqrt{5}$,1.3,4,2), (108,23,123,55), (Mary,Ann,Pete,Joe), and (30.2,π,111.23,12) of size 4 can all be regarded as having the same ordering (third largest, first largest, fourth largest, second largest) as the permutation (3,1,4,2). In particular, they each require the same number of comparisons when input to a comparison-based sorting algorithm before they are put into increasing order. For this reason, we may reduce our input space for comparison-based sorting algorithms to the set of all permutations on the n integers 1, 2, . . . , n. This amounts to making two lists in $\mathscr{I}_n$ equivalent if they have the same ordering.

To illustrate the computation of $B(n)$ of $W(n)$, we consider some basic comparison-based algorithms for searching, finding a maximum element and sorting lists of size n. Searching lists is one of the most common tasks performed by computers. If no preconditioning (such as sorting) of the list is assumed, then there is no better algorithm than linear search, which is based on a sequential (linear) scan of the list. On the other hand, for sorted lists there are much more efficient searching algorithms such as binary search.

We begin with the analyses of the two algorithms linear search and binary search. Linear search and binary search are based on making comparisons between a search element and the key field of the list elements. For simplicity, in our pseudocode for linear search and binary search we do not actually identify the key field of a list element, but instead reference the entire list element.

2.6.1 Linear Search

We implement linear search as a function that returns the (first) position in a list (array of size n) $L[0{:}n-1]$ where a search element X occurs, or returns -1 if X is not in the list.

```
function LinearSearch (L[0:n – 1],X)
Input: L[0:n – 1] (a list of size n), X (a search item)
Output:    returns index of first occurrence of X in the list, or -1 if X is not in the list
    for i ← 0 to n – 1  do
        if X = L[i] then
            return(i)
        endif
    endfor
    return(-1)
end LinearSearch
```

The basic operation of *LinearSearch* is the comparison of the search element to a list element. Clearly, *LinearSearch* performs only one comparison when the input X is the first element in the list, so that the best-case complexity is $B(n) = 1$. The most comparisons are performed when X is not in the list, or when X occurs in the last position only. Thus, the worst-case complexity of *LinearSearch* is $W(n) = n$.

2.6.2 Binary Search

LinearSearch assumes nothing about the order of the elements in the list; in fact, it is an optimal algorithm when no special order is assumed. However, *LinearSearch* is not the algorithm to use when searching ordered lists, at least when direct access to each list element is possible (as with an array implementation of the list). For example, if you are looking up the word *riddle* in a dictionary, and you initially open the dictionary to the page containing the word *middle*, then you know that you only need search for the word in the pages that follow. Similarly, if you were looking up the word *fiddle* instead of *riddle*, then you would only need to search for the word in the preceding pages. This simple observation is the basis of *BinarySearch*.

The idea behind binary search is to successively cut in half the range of indices in the list where the search element X might be found. We assume that the list is sorted in

increasing order. By comparing X with the element $L[mid]$ in the middle of the list, we can determine whether X might be found in the first half of the list or the second half. We have three possibilities:

$X = L[mid]$,	X is found,
$X < L[mid]$,	search for X in $L[0:mid-1]$,
$X > L[mid]$,	search for X in $L[mid+1:n-1]$.

This process is then repeated (if necessary) for the relevant "half list." Thus, the number of elements in a sublist where X might be found is being (roughly) cut in half for each repetition. When cutting a sublist $L[low:high]$ in half, if the size of the sublist is even, then we take the midpoint index to be the smaller of the two middle indices, so that $mid = \lfloor (low + high)/2 \rfloor$. The following pseudocode implements binary search as a function with the same parameters and output as *LinearSearch*, except that the list is assumed to be sorted in increasing order.

```
function BinarySearch (L[0:n – 1], low, high, X) recursive
Input: L[0:n – 1] (an array of n list elements, sorted in increasing order)
           X (a search item)
Output:    returns an index of an occurrence of X in the sublist L[low, high], or -1 if X is
           not in the sublist
    if low > high then return(-1) endif
        mid ← ⌊(low + high)/2⌋
    case
       :X = L[mid]:  return(mid)
       :X < L[mid]:  return(BinarySearch(L[0:n – 1], low, mid-1, X))
       :otherwise: return(BinarySearch(L[0:n – 1], mid+1, high, X))
    endcase
end BinarySearch
```

Note that when searching for X in the entire list $L[0:n-1]$, we call *BinarySearch* with $low = 0$ and $high = n - 1$. It is this situation that we use when measuring the complexity of *BinarySearch*. Just as with *LinearSearch*, we use comparison of X to a list element as our basic operation when analyzing *BinarySearch*. The best-case complexity of *BinarySearch* is 1, which occurs when X is found in the midpoint position $\lfloor (n-1)/2 \rfloor$ of $L[0:n-1]$. The worst-case complexity is equal to twice the longest string of midpoints (values of *mid*) ever generated by the algorithm for an input X. In particular, if we assume that $n = 2^k - 1$ for some positive integer k, then such a string is generated by searching for $X = L[0]$. We then compare X successively to the midpoints $2^{k-1} - 1$, $2^{k-2} - 1$, . . . , 0, so that this longest string has length k. To express k in terms of n, we note that $n + 1 = 2^k$, so that we have $k = \log_2(n + 1)$. We leave it as an exercise to verify that for any n, the length of the longest string of midpoints ever generated is $\lceil \log_2(n + 1) \rceil$, so that $W(n) = 2 \lceil \log_2(n + 1) \rceil$.

2.6.3 Interpolation Search

While *BinarySearch* assumes that the list $L[0:n-1]$ is in increasing order, it always compares the search element X to the midpoint index entry in the current sublist $L[low:high]$ thereby ignoring the fact that the value of X might suggest a better place to

look. Interpolation search computes an index value in the range between *low* and *high* that on average is rather more likely to be nearer to where *X* might occur than the midpoint index. By way of motivation, consider the problem of looking up the word *algorithm* in the dictionary. It certainly would be better to open up the dictionary to a page closer to the beginning of the dictionary than the middle page.

Interpolation search computes the index *i* in the current range *low*:*high* for comparing $L[i]$ to *X* by making the assumption that the values in the original list $L[0:n-1]$ are not only increasing, but lie approximately along a straight line joining the points $(0,L[0])$ to $(n-1,L[n-1])$ (we are interpreting the list values in *L* as numbers, which can always be assumed with proper conversions). This assumption of the "linearity" of the data is essential to the efficiency of interpolation search, but not its correctness.

Unfortunately, interpolation search makes *n* comparisons in the worst case (see Exercise 2.25). However, under suitable assumptions of randomness for the elements of the list $L[0:n-1]$, it can be shown that the average performance $A(n)$ of interpolation search is approximately $\log_2(\log_2 n)$, a very slowing growing function indeed (see Exercise 2.27). The proof of this average behavior is beyond the scope of this book.

The value for the index *i* used by interpolation search (instead of *mid* used by *BinarySearch*) is computed by simply finding the point (i,X) along the line joining $(low,L[low])$ to $(high,L[high])$ corresponding to the search element *X*. More precisely, *i* is determined from the equation

$$(X - L[low])/(i - low) = (L(high) - L(low))/(high - low). \qquad (2.6.4)$$

Note that even though the entire list $L[0:n-1]$ may not be approximately linear, a given sublist might be. Thus, recalculating the value of *i* for each sublist helps make interpolation search very efficient on average. We leave the pseudocode for interpolation search and its worst-case analysis to the exercises.

2.6.4 Finding the Maximum and Minimum Elements in a List

We now consider the problem of finding the maximum (or minimum) value of an element in a list $L[0:n-1]$ of size *n*. Finding the maximum can be done with a variation of *LinearSearch* where we keep track and update the maximum (or minimum) value encountered as we scan the list.

```
function Max (L[0:n – 1])
Input: L[0:n – 1] (a list of size n)
Output:    returns the maximum value occurring in L[0:n – 1]
    MaxValue ← L[0]
    for i ← 1 to n – 1  do
        if L[i] > MaxValue then
            MaxValue ← L[i]       //update MaxValue
        endif
    endfor
    return(MaxValue)
end Max
```

When analyzing the function *Max*, we choose comparison between list elements ($L[i] > MaxValue$) as our basic operation. The only other operation performed by *Max* is the updating of *MaxValue*. However, for any input list, the number of comparisons between list elements clearly dominates the number of updates of *MaxValue*, which justifies our choice of basic operation.

Note that *Max* performs $n - 1$ comparisons for any input list of size n. Thus, the best-case, worst-case, (and average complexities) of *Max* all equal $n - 1$. It is straightforward to show that any comparison-based algorithm for finding the maximum value of an element in a list of size n must perform at least $n - 1$ comparisons for any input, so that *Max* is an optimal algorithm.

An analogous algorithm *Min* can be given for finding the minimum value in a list. Sometimes it is useful to determine *both* the maximum and the minimum values in a list $L[0:n - 1]$ (thereby determining the *range* of the data in $L[0:n - 1]$). A simple algorithm for solving this problem is to successively invoke *Max* and *Min*, having best-case and worst-case complexities equal to $2n - 2$. It turns out that any algorithm for finding the maximum and minimum in a list of size n must perform at least $\lceil 3n/2 \rceil - 2$ comparisons in the worst case. The following algorithm *MaxMin* achieves this lower bound, and therefore is an *optimal* algorithm for this problem.

To facilitate our discussion of the algorithm *MaxMin*, we introduce the following procedure *M&M*, which solves the max-min problem for a list of size 2.

```
procedure M&M(A,B,MaxValue,MinValue)
Input: A, B
Output:     MaxValue, MinValue (the maximum and minimum of A and B)
        if A ≥ B then
                MaxValue ← A
                MinValue ← B
        else
                MaxValue ← B
                MinValue ← A
        endif
end M&M
```

Procedure *MaxMin* works by pairing elements (except for one element when n is odd) in the list $L[0:n - 1]$ and computing the maximum and minimum for each pair, yielding $\lceil n/2 \rceil$ potential maxima and $\lceil n/2 \rceil$ potential minima.

```
procedure MaxMin(L[0:n – 1],MaxValue,MinValue)
Input:  L[0:n – 1] (array)
Output: MaxValue,MinValue (the maximum and minimum values in L[0:n – 1])
        if even(n) then {n is even }
                M&M(L[0],L[1],MaxValue,MinValue)
                for i ← 2 to n – 2 by 2 do
                        M&M(L[i],L[i + 1],b,a)
                        if a < MinValue then MinValue ← a endif
```

```
                    if b > MaxValue then MaxValue ← b endif
                endfor
        else //n is odd
                MaxValue ← L[0]; MinValue ← L[0];
                for i ← 1 to n – 2 by 2 do
                    M&M(L[i],L[i + 1],b,a)
                    if a < MinValue then MinValue ← a endif
                    if b > MaxValue then MaxValue ← b endif
                endfor
        endif
end MaxMin
```

For n even, $L[0]$ and $L[1]$ are compared, and then the (first) for loop of *MaxMin3* performs $3(n-2)/2$ comparisons. For n odd, there is no initial comparison, and the (second) for loop performs $3(n-1)/2$ comparisons. Thus, the best-case and worst-case complexities of *MaxMin* for input size n are both equal to $\lceil 3n/2 \rceil - 2$.

We now design and analyze some basic comparison-based sorting algorithms. We start with the simple sorting algorithm insertion sort. While insertion sort is not efficient (in the worst case) for large lists, we shall see that it does have its uses.

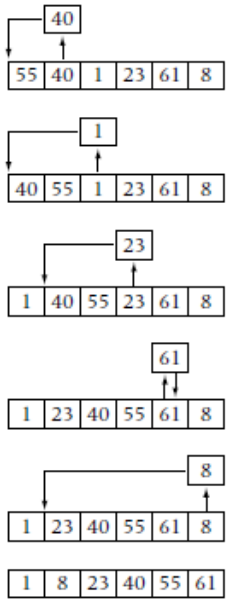
2.6.5 Insertion Sort

Array Implementation of Insertion Sort

Insertion sort sorts a given list $L[0{:}n-1]$ by successively inserting the list element $L[i]$ into its proper place in the sorted list $L[0{:}i]$, $i = 1, \ldots, n-1$. It works similarly to how a card player might insert a newly dealt card into the previously dealt hand that was already put in order. One starts a scan at one end of the hand and stops at a place where the new card can be inserted and still maintain an ordered hand. This scan can start either at the low end of the hand (forward scan), or at the high end of the hand (backward scan). A card player has no reason (other than a personal preference) to prefer one scan over the other. However, when implementing insertion sort there are several reasons for preferring one scan over the other, depending on the situation.

Given the list $L[0{:}n-1]$, clearly the sublist consisting of only the element $L[0]$ is a sorted list. Suppose (after possibly reindexing) we have a list L where the sublist $L[0{:}i-1]$ is already sorted. We then can obtain a sorted sublist $L[0{:}i]$ by inserting the element $L[i]$ in its proper position. In a backward scan, we successively compare $L[i]$ with $L[i-1]$, $L[i-2]$, and so forth, until a list element $L[position]$ is found that is not larger than $L[i]$. $L[i]$ can then be inserted at $L[position+1]$. In a forward scan, we successively compare $L[i]$ with $L[0]$, $L[1]$, and so forth, until a list element $L[position]$ is found that is not smaller than $L[i]$. $L[i]$ can then be inserted at $L[position]$.

Figure 2.2 demonstrates the action of the backward scan version of insertion sort for a list of size 6.



Action of *InsertionSort* (backward scan) for a list of size 6

Figure 2.2

We now give pseudocode for insertionsort using a backward scan.

```
procedure InsertionSort(L[0:n – 1])
Input: L[0:n – 1] (a list of size n)
Output:   L[0:n – 1] (sorted in increasing order)
    for i ← 1 to n – 1 do //insert L[i] in its proper position in L[0:i – 1]
        Current ← L[i]
        position ← i – 1
        while position ≥ 1 .and. Current < L[position] do
                  //Current must precede L[position]
            L[position +1] ← L[position]      //bump up L[position]
            position ← position – 1
        endwhile
                  //position + 1 is now the proper position for Current = L[i]
        L[position +1] ← Current
    endfor
end InsertionSort
```

When analyzing *InsertionSort*, we choose comparison between list elements (*Current* < *L*[*position*]) as our basic operation. For any input list of size *n*, the outer loop of *InsertionSort* is executed $n - 1$ times. If the input list is already sorted in increasing order, then the inner loop is iterated only once for each iteration of the outer loop. Hence, the best-case complexity of *InsertionSort* is given by

$$B(n) = n - 1 \tag{2.6.5}$$

The worst-case complexity occurs when the inner loop performs the maximum number of comparisons for each value of the outer loop variable i. For a given i, this occurs when $L[i]$ must be compared to each element $L[i-1], L[i-2], \ldots, L[0]$, so that $i-1$ comparisons are performed in the inner loop. This, in turn, occurs when the list is in strictly decreasing order. Since i varies from 1 to $n-1$, we have:

$$W(n) = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}. \tag{2.6.6}$$

Thus, $W(n)$ for *InsertionSort* is quadratic in the input size n.

We will show that the average complexity $A(n)$ for *InsertionSort* is about half of $W(n)$, and therefore is also quadratic in n. To see why this is true intuitively, it is reasonable that when inserting the $(i+1)$st element $L[i]$ into its proper position in the list $L[0], \ldots, L[i-1]$, on average $i/2$ comparisons will be made. Hence, it is reasonable that $A(n)$ should be about $1 + (1/2)[2 + 3 + \ldots + n - 1] = 1 + (1/2)[n(n-1)/2 - 1]$.

Because of its quadratic complexity, *InsertionSort* is impractical to use for sorting general large lists. However, *InsertionSort* does have five advantages:

1. *InsertionSort* works quickly on small lists. Thus, *InsertionSort* is often used as a threshold sorting conjunction with other sorting algorithms like merge sort or quick sort.
2. *InsertionSort* works quickly on large lists that are close to being sorted in the sense that no element has many larger elements occurring before it in the list. This property of *InsertionSort* makes it useful in connection with other sorts such as *ShellSort* (see the discussion in the exercises at the end of this chapter).
3. *InsertionSort* is amenable to implementation as an on-line sorting algorithm. An *on-line* sorting algorithm is one in which the entire list is not input to the algorithm in advance, but instead elements are added to the list over time. On-line sorting algorithms are required to maintain the dynamic list in sorted order.
4. *InsertionSort* is an in-place sorting algorithm. A sorting algorithm with input parameter $L[0{:}n-1]$ is called *in-place* if only a constant amount of memory is used (for temporary variables, loop control variable, sentinels, and so forth) in addition to that needed for L.
5. *InsertionSort* is a *stable* sorting algorithm in the sense that it maintains the relative order of repeated elements. More precisely, an algorithm that sorts a list $L[0{:}n-1]$ into increasing order is called stable if, given any two elements $L[i]$, $L[j]$, with $i < j$ and $L[i] = L[j]$, the final positions i', j' of $L[i]$, $L[j]$, respectively, satisfy $i' < j'$.

Stable algorithms are useful when we want to sort the elements in a list of records according to primary and secondary keys in the record, but where the sorts on these two keys take place independently. By way of illustration, suppose we have already sorted the records in some list of persons alphabetically according to name. For purposes of bulk mailing, we now wish to sort the list according to the zip code key in each record.

However, within a given zip code, we wish to maintain our alphabetical order. Clearly, a stable sorting algorithm is required.

The question arises as to why we use linear scans for finding the correct position to insert the list element $L[i]$ into the already sorted list $L[0{:}i]$, when a binary search to find this position would drastically reduce the number of comparisons made by the algorithm. For example, the worst-case complexity of *InsertionSort* would be reduced from $n(n-1)/2$ to approximately $n\log_2 n$. The catch is that this altered version would not reduce the number of array reassignments needed to insert $L[i]$, so that a quadratic number of array reassignments would still be made in the worst case. Thus, even though the altered *InsertionSort* is still comparison-based, the number of comparisons made would no longer be a true measure of the complexity of the algorithm.

Another drawback of the binary search version of *InsertionSort* is that some of the advantages of *InsertionSort* listed earlier would be lost. For example, the binary search version of *InsertionSort* would no longer be stable and would not necessarily work fast on large lists that are close to being sorted.

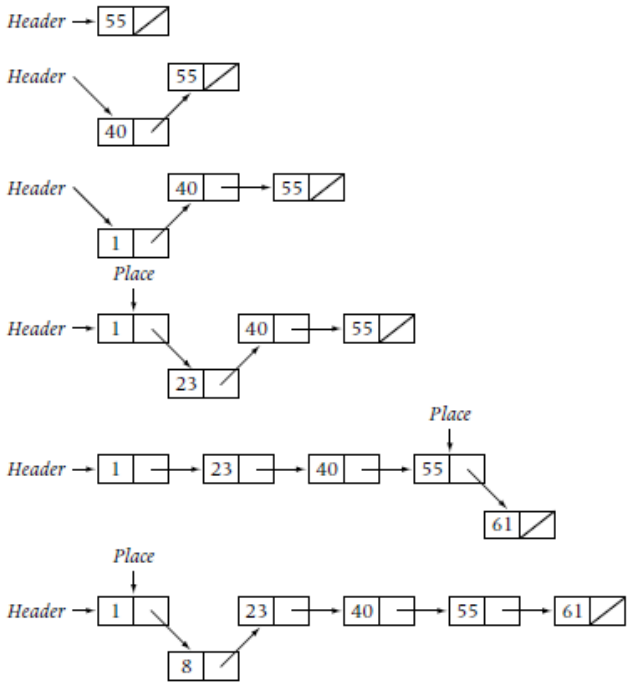
An *adjacent-key* comparison-based sorting algorithm is one in which comparisons between list elements are made only between elements that occupy *adjacent* positions in the list. These elements are interchanged if they are out of order. *InsertionSort* is essentially an adjacent-key comparison sort, since the comparison between $L[position]$ and *current* can be thought of as an assignment of *current* to $L[position]$ and then a comparison between $L[position]$ and $L[position - 1]$. For reasons of efficiency, we chose not to make the actual assignment until the correct position for insertion was determined. Another well-known adjacent-key sorting algorithm is *BubbleSort* (see Exercise 2.34).

In Chapter 3 we show that the worst-case complexity of *any* adjacent-key comparison-based sorting algorithm is at least $n(n-1)/2$. In view of this result, *InsertionSort* actually has optimal worst-case complexity for an adjacent-key comparison-based sort. If we want to design comparison-based sorting algorithms whose worst-case complexities are smaller than $n(n-1)/2$, we must look for design strategies that compare nonadjacent (far away) elements in the list. We give examples of algorithms utilizing the latter strategy later in this chapter.

Linked List Insertion Sort

We now implement insertion sort using a linked list rather than an array. To find the correct position to insert an element *X*, a forward scan of the linked list is performed using the pointer *Place*. This scan first compares *X* to the first element in the list. If *X* is not larger than this element, then the new node containing *X* is inserted at the beginning of the list. Otherwise, the scan continues down the list until the last element in the list smaller than *X* is pointed to by *Place*. The new node can then be placed immediately after the node containing this latter element.

We illustrate the linked version of insertion sort in Figure 2.3 for the same list of elements used in Figure 2.2. (The pseudocode for this linked version is left as an exercise.) In Figure 2.3, we assume that the elements are placed in nodes and inserted into the sorted linked list immediately after they are input.



Action of the linked version of *InsertionSort* for *L*[0:5]: 55 40 1 23 61 8

Figure 2.3

2.6.6 Merge Sort

Merge sort is a classical example of an algorithm based on the divide-and-conquer design strategy. Merge sort was already in use in the earliest electronic computers. In fact, it was one of the stored programs implemented by von Neumann in 1945. Given a sublist *L*[*low*:*high*] of *L*[0:*n* – 1], let *x* be any index between *low* and *high*. Let *A*, *B*, *C* denote the sublists *L*[*low*:*x*], *L*[*x*+1:*high*], *L*[*low*:*high*], respectively. The problem of sorting *C* can be solved by first sorting *A* (recursive call), then sorting *B* (another recursive call), and finally calling the procedure *Merge*, which merges the sorted lists *A* and *B* to obtain a sorted list *C*. The following recursive code for the procedure *MergeSort* implements these steps. Note in *MergeSort* that we have chosen *x* to be the midpoint of *low* and *high*. The list *L*[0:*n* – 1] is sorted by calling *MergeSort* and passing 0 to *low* and *n* – 1 to *high*.

```
procedure MergeSort(L[0:n – 1],low,high) recursive
Input: L[0:n – 1] (an array of n list elements), low,high (indices of L[0:n – 1])
Output:    L[low:high] (subarray sorted in increasing order)
    if low < high then
        mid ← ⌊(low + high)/2⌋
        MergeSort(L[0:n – 1],low,mid)
        MergeSort(L[0:n – 1],mid+1,high)
        Merge(L[0:n – 1],low,mid,high)
    endif
end MergeSort
```

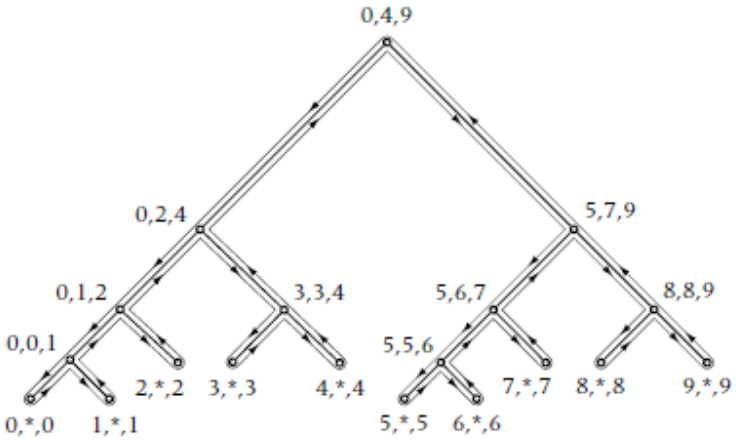
We now give the pseudocode for the procedure *Merge* called by *MergeSort*, where an auxiliary array *Temp* is used to aid in the merging process. *Merge* utilizes two variables *CurPos1* and *CurPos2* that refer to the current positions in the already sorted sublists $L[low{:}x]$ and $L[x{+}1{:}high]$, respectively. *Merge* also utilizes a third pointer *Counter*, which refers to the next available position in *Temp*. *CurPos1*, *CurPos2*, and *Counter* are initialized to *low*, $x + 1$, and *low*, respectively.

During each iteration of the while loop in *Merge*, the elements in the positions *CurPos1* and *CurPos2* are compared, and the smaller element is written to the position *Counter* in *Temp*. Then *Counter* and either *CurPos1* or *CurPos2*, depending on which one refers to the element just written to *Temp*, are incremented by one. If *CurPos1* becomes greater than *x*, then the sublist $Temp[low{:}Counter - 1]$ is a sorted list all of whose elements are not greater than any of the elements in the sublist $L[CurPos2{:}high]$. Thus, in this case we get a sorting of $L[low{:}high]$ by copying $Temp[low{:}Counter - 1]$ back onto $L[low{:}Counter - 1]$. On the other hand, if *CurPos2* becomes greater than *high*, then every element in the sublist $Temp[low{:}Counter - 1]$ is a sorted list all of whose elements are not greater than any of the elements in the sublist $L[CurPos1{:}x]$. Thus, in this case we get a sorting of $L[low{:}high]$ by first copying $L[CurPos1{:}x]$ over to $L[Counter{:}high]$, and then copying $Temp[low{:}Counter - 1]$ back onto $L[low{:}Counter - 1]$. The pseudocode for *Merge* follows.

```
procedure Merge(L[0:n – 1],low,x,high)
Input: L[0:n – 1] (an array of n list elements),
            low,x,high (indices of array L[0:n – 1]; sublists L[low:x], L[x+1:high] are
                                    assumed to be sorted in increasing order)
Output:   L[low:high] (sublist sorted in increasing order)
    CurPos1 ← low                                          //initialize pointers
    CurPos2 ← x + 1
    Counter ← low
    while CurPos1 ≤ x .and. CurPos2 ≤ high do      //while elements remain
        if   L[CurPos1] ≤ L[CurPos2] then              //to be written in both
            Temp[Counter] ← L[CurPos1]                 //sublists, merge to Temp
            CurPos1 ← CurPos1 + 1
        else
            Temp[Counter] ← L[CurPos2]
            CurPos2 ← CurPos2 + 1
        endif
        Counter ← Counter + 1
    endwhile
    if CurPos1 > x then
        for k ← low to Counter  − 1 do
                L[k] ← Temp[k]
       endfor
    else
        for k ← CurPos1 to x do
                L[k + Counter – CurPos1] ← L[k]
```

```
        endfor
        for k ← low to Counter − 1 do
                L[k] ← Temp[k]
        endfor
    endif
end Merge
```

The tree of recursive calls to *MergeSort* is illustrated in Figure 2.4. A node in the tree is labeled by the values *low*,*mid*,*high* involved in the call to *Merge*. (In leaf nodes, *mid* is not computed, as indicated by the symbol *.) Initially, *low* = 0 and *high* = 9. The path around the tree shown in Figure 2.4 indicates how the recursion resolves. Following this path amounts to a postorder traversal of the tree (see Chapter 4), where visiting a node corresponds to a call to *Merge*.



Recursive calls to merge sort for lists of size 10

Figure 2.4

We now analyze the complexity of *MergeSort*. We first discuss the worst-case complexity. Note that each call to *Merge* for merging two sublists of sizes m_1 and m_2 performs at most $m_1 + m_2 - 1$ comparisons. Consider the tree of recursive calls to *MergeSort* for a list of size n (see Figure 2.4). The leaf nodes do not generate calls to *Merge*, whereas each internal node generates a single call to *Merge*. At each level of the tree, the total number of comparisons made by all of the calls to *Merge* is at most n. The depth of the tree of recursive calls is $\lceil \log_2 n \rceil$ (see Exercise 2.36). It follows that *MergeSort* performs at most $n\lceil \log_2 n \rceil$ comparisons for any list of size n, so that $W(n) \leq n\lceil \log_2 n \rceil$.

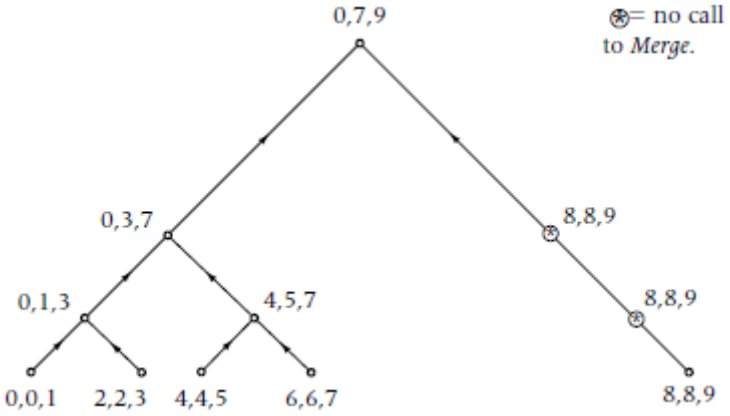
Now consider $B(n)$. Each call to *Merge* for merging two sublists of sizes m_1 and m_2 performs at least $\min\{m_1,m_2\}$ comparisons. We again consider the tree of recursive calls to *MergeSort* for a list of size n. Except for the last two levels, at each level of the tree of recursive calls, the total number of comparisons made by all of the calls to *Merge* is at least $n/2$. It follows that *MergeSort* performs at least $(n/2)(\lceil \log_2 n \rceil - 1)$ comparisons for any list of size n, so that $B(n) \geq (n/2)(\lceil \log_2 n \rceil - 1)$. Thus, we have

$$(n/2)(\lceil \log_2 n \rceil - 1) \leq B(n) \leq A(n) \leq W(n) \leq n\lceil \log_2 n \rceil .$$

Since $B(n)$, $A(n)$, and $W(n)$ are all squeezed between functions which are very close to $(n/2)\log_2 n$ and $n\log_2 n$, respectively, for asymptotic analysis purposes (where we ignore the effect of positive multiplicative constants) we simply say that these quantities have $n\log n$ complexity (note that we have omitted the base in the log, since change of base formulas show that logarithm functions to two different bases differ by a constant). It turns out that this type of complexity is optimal for $A(n)$ and $W(n)$ for *any* comparison-based sorting algorithm.

By examining Figure 2.4, we can easily come up with a bottom-up version of *MergeSort*. At the bottom level of the tree, we are merging sublists of single adjacent elements in the list. However, as the path around the tree indicates, for a given input list $L[0{:}n-1]$ *MergeSort* sorts the list $L[0{:}mid]$ before going on to any of the sublists of $L[mid+1, n-1]$. By contrast, a bottom-up version begins by dividing the list into pairs of adjacent elements, $L[0]{:}L[1]$, $L[2]{:}L[3]$, ... Next, these adjacent pairs are merged yielding the sorted lists $L[0{:}1]$, $L[2{:}3]$, ... The process is repeated by merging the adjacent pairs of sorted two-element sublists, $L[0{:}1]{:}L[2{:}3]$, $L[4{:}5]{:}L[6{:}7]$, ... Continuing this process, we arrive at the root having sorted the entire list $L[0{:}n-1]$.

Figure 2.5 shows a tree representing this bottom-up merging of adjacent sublists. Each node represents a call to *Merge* with the indicated values of *low*,*mid*,*high*. An asterisk * denotes those nodes where a call to *Merge* is not made. Note that the sublists and resulting tree are quite different from the sublists generated by the tree of recursive calls of *MergeSort* given in Figure 2.4. All the calls to *Merge* for a given level are completed before we go up to the next level. The pseudocode for the nonrecursive version of *MergeSort* based on Figure 2.5 is left to the exercises.



Bottom-up nonrecursive merge sort for a list of size n

Figure 2.5

2.6.7 Quick Sort

We now discuss the comparison-based sorting algorithm quick sort, discovered by C. A. R. Hoare in 1962. Quick sort is often the sorting algorithm of choice because of its good average behavior. Quick sort, like merge sort, is based on dividing a list into two sublists (these are actually two sublists of a rearrangement of the original list), and then sorting the sublists recursively. The difference between the two sorting strategies is that merge sort does most of the work in the combine (merge) step, whereas quick sort does most of

the work in the divide step. Quick sort is also an "in place" sort, as opposed to merge sort which used an auxiliary array for merging (in-place versions of merge sort can be written, but they are complicated).

In quick sort, the division into sublists is based on rearranging the list $L[low:high]$ with respect to a suitably chosen *pivot element* x. The list $L[low:high]$ is rearranged so that every list element in $L[low:high]$ preceding x (having smaller index than the index of x) is not larger, and every element following x in $L[low:high]$ is not smaller. For example, for the list 23,55,11,17,53,4 and pivot element $x = 23$, such a rearrangement might be 17,11,4,23,55,53. Note that after such a rearrangement, the pivot element x occupies a proper position in a sorting of the list. Thus, if the sublists on either side of x are sorted recursively, then the entire list will be sorted with no need to invoke an algorithm for combining the sorted sublists.

Procedure *QuickSort* sorts $L[low:high]$ into increasing order by first calling an algorithm *Partition* that rearranges the list with respect to pivot element $x = L[low]$ as previously described. *Partition* assumes that the element $L[high+1]$ is defined and is at least as large as $L[low]$. The output parameter *position* of *Partition* returns the index where x is placed. To sort the entire list $L[0:n-1]$, *QuickSort* would be called initially with $low = 0$ and $high = n - 1$.

```
procedure QuickSort(L[0:n – 1], low, high) recursive
Input: L[0:n – 1] (an array of n list elements)
          low, high (indices of L[0:n – 1])
        //for convenience, L[n] is assumed to have the sentinel value +∞}
Output:   L[low:high] sorted in increasing order
    if high > low then
            Partition(L[0:n – 1], low, high, position)
            QuickSort(L[0:n – 1], low, position – 1)
            QuickSort(L[0:n – 1], position+1, high)
    endif
end QuickSort
```

We now describe the algorithm *Partition*. *Partition* is based on the following clever interplay between two moving variables *moveright* and *moveleft*, which contain the indices of elements in L and are initialized to $low + 1$ and *high*, respectively.

```
while moveright < moveleft do
        moveright moves to the right (one index at a time) until it assumes the
        index of a list element not smaller than x, then it stops.
        moveleft moves to the left (one index at a time) until it assumes the index
        of a list element not larger than x, then it stops.
        if moveright < moveleft then
                interchange L[moveright] and L[moveleft]
        endif
endwhile
```

To guarantee that *moveright* actually finds an element not smaller than x, we assume that $L[high + 1]$ is defined and is not smaller than $L[low]$. As commented in the

pseudocode, this is arranged by introducing a sentinel value $L[n] = +\infty$. We leave it as an exercise to check that the condition $L[high+1] \geq L[low]$ is then automatically guaranteed for all subsequent calls to *Partition* by *QuickSort*. Of course, *Partition* could be written with explicit checking that *moveright* does not run off the list. However, this checking requires additional comparisons, and we prefer to implement the preconditioning. Figure 2.6 illustrates the movement of *moveright* (*mr*) and *moveleft* (*ml*) for a sample list $L[0:6]$. The pseudocode for *Partition* follows.

```
procedure Partition(L[0:n – 1], low, high,position)
Input: L[0:n – 1] (an array of n list elements)
       low, high (indices of L[0:n – 1])
       //L[high+1] is assumed defined and ≥ L[low]
Output:  a rearranged sublist L[low:high] such that
         L[i] ≤ L[position],     low ≤ i ≤ position,
         L[i] ≥ L[position],     position ≤ i ≤ high
         where, originally, L[low] = L[position]
         position   (the position of a proper placement of the original element L[low]
                     in the list L[low:high])
    moveright ← low
    moveleft ← high + 1
    x ← L[low]
    while moveright < moveleft do
        repeat
            moveright ← moveright + 1
        until L[moveright] ≥ x
        repeat
            moveleft ← moveleft – 1
        until L[moveleft] ≤ x
        if moveright < moveleft then
            interchange(L[moveright],L[moveleft])
        endif
    endwhile
    position ← moveleft
    L[low] ← L[position]
    L[position] ← x
end Partition
```

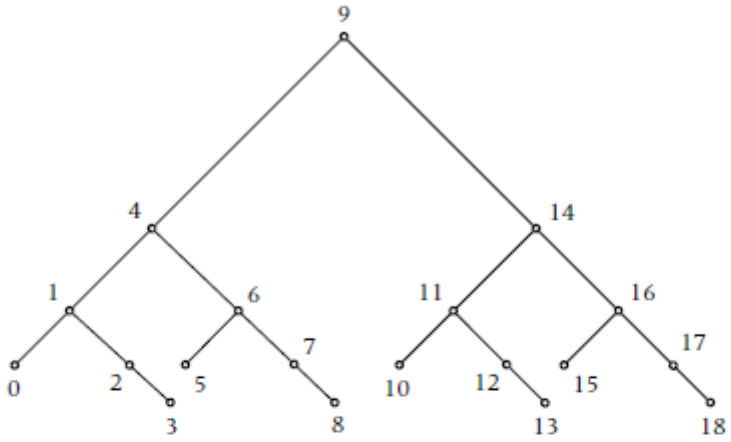
Initially:	index	0	1	2	3	4	5	6	7
	list element	23	9	23	52	15	19	47	+∞
		mr							*ml*
		Rearrange Step							
				19			23		
1st iteration:		23	9	23	52	15	19	47	+∞
				mr			*ml*		
					15	52			
2nd iteration:		23	9	19	52	15	23	47	+∞
					mr	*ml*			
3rd iteration:		23	9	19	15	52	23	47	+∞
					ml	*mr*			
		Place Step							
		15			23				
After completion of *Partition:*		23	9	19	15	52	23	47	+∞
						position = 3			

Action of *Partition* for a sample list $L[0{:}6]$

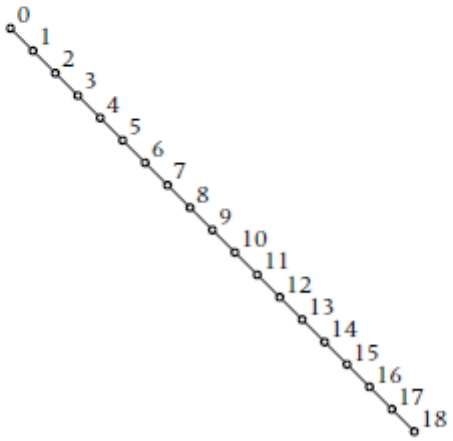
Figure 2.6

To analyze the performance of *QuickSort*, we again use list comparisons as our basic operation for analysis, where *QuickSort* is originally called with a list $L[0{:}n-1]$ of distinct elements, and $L[n] = +\infty$. A call to *Partition* with $L[low{:}high]$ performs exactly $high - low + 2$ comparisons, and this forms the basis of our analysis. We first consider the worst-case complexity.

Unfortunately, the worst-case performance of *QuickSort* occurs for a list that is already in order. In this case the recursive calls to *QuickSort* are always with the empty list $L[low{:}low-1]$ and the list $L[low+1{:}high]$ (see Figure 2.7b). Thus, the number of comparisons performed is given by $W(n) = (n + 1) + n + \ldots + 3 = (n + 2)(n + 1)/2 - 3$, so that *QuickSort* has quadratic worst-case performance. Note that a decreasing list also generates worst-case performance for *QuickSort*.



(a) Tree of recursive calls for best-case behavior of *QuickSort* for a list of size 19. Internal nodes are labeled with pivot elements in a call to *Partition*. Leaf nodes are labeled with single-element sublists that result in immediate returns. For simplicity, we do not show leaf nodes corresponding to calls to *QuickSort* with empty lists.



(b) Tree of recursive calls for worst-case behavior of *QuickSort* for a list of size 19, with nodes labeled using the same convention as in (a).

Tree of recursive calls for *QuickSort*:
(a) best-case behavior; (b) worst-case behavior

Figure 2.7

The quadratic worst-case performance is disappointing, but quick sort is popular because of its (asymptotic) *n*log *n* average behavior. We can expect this behavior since it is reasonable that on average the call to *Partition* results in dividing a given sublist into sublists of roughly equal size. In other words, for an average input, the tree of recursive calls would have a balanced nature similar to that of merge sort, so that we would have roughly $\log_2 n$ levels, with no more than $n + 1$ comparisons made by the calls to *Partition* on each level. We will prove in Chapter 3 that *QuickSort* does indeed have (asymptotic) *n*log *n* average complexity.

Remarks

For simplicity we chose the pivot element to be the first list element. A common alternative is to take the pivot element to be the median of the three elements in positions *low*,(*low* + *high*)/2, and *high*. Using this *Median of Three Rule* avoids the bad behavior exhibited by lists that are close to being sorted.
There are many different ways to design an algorithm that accomplishes the same thing as our version of *Partition*. One such alternative version is explored in Exercise 2.62.

Some improvements to *QuickSort* should be made before using it in practice. In particular, *QuickSort* as written generates $n - 1$ unresolved recursive calls (that is, $n - 1$ successive pushes of unresolved activation records) in the worst case. These $n - 1$ successive pushes will no doubt cause stack overflow when run on many computers for even modestly large values of n (that is, even for values of n for which a quadratic number of comparisons will still complete in reasonable time). Thus, it is important to rewrite *QuickSort* in order to reduce the number of unresolved recursive calls.

To reduce the number of successive recursive calls, we first note that *QuickSort* is tail recursive (see Appendix B). However, simply removing the tail recursion will result in a version that still generates $n - 1$ unresolved recursive calls for a decreasing list (however, an increasing list now generates recursive calls only with empty sublists, so that there is only one activation record on the stack generated by recursive calls at any given point in the execution for such lists). The problem is that we are always making a recursive call with the sublist to the left of the partition element, whose size is only reduced by one in the worst case. What we need to do is make the recursive call with the smaller of the two sublists on either side of the partition element, and simply redefine the relevant parameter (*low* or *high*) for the larger sublist and branch to the beginning of the code. This will reduce the number of unresolved recursive activation records on the stack to at most $\log_2 n$. Moreover, now both the worst cases of increasing or decreasing lists will only make recursive calls with empty sublists. We leave the code for this altered version of *QuickSort* and its analysis to the exercises.

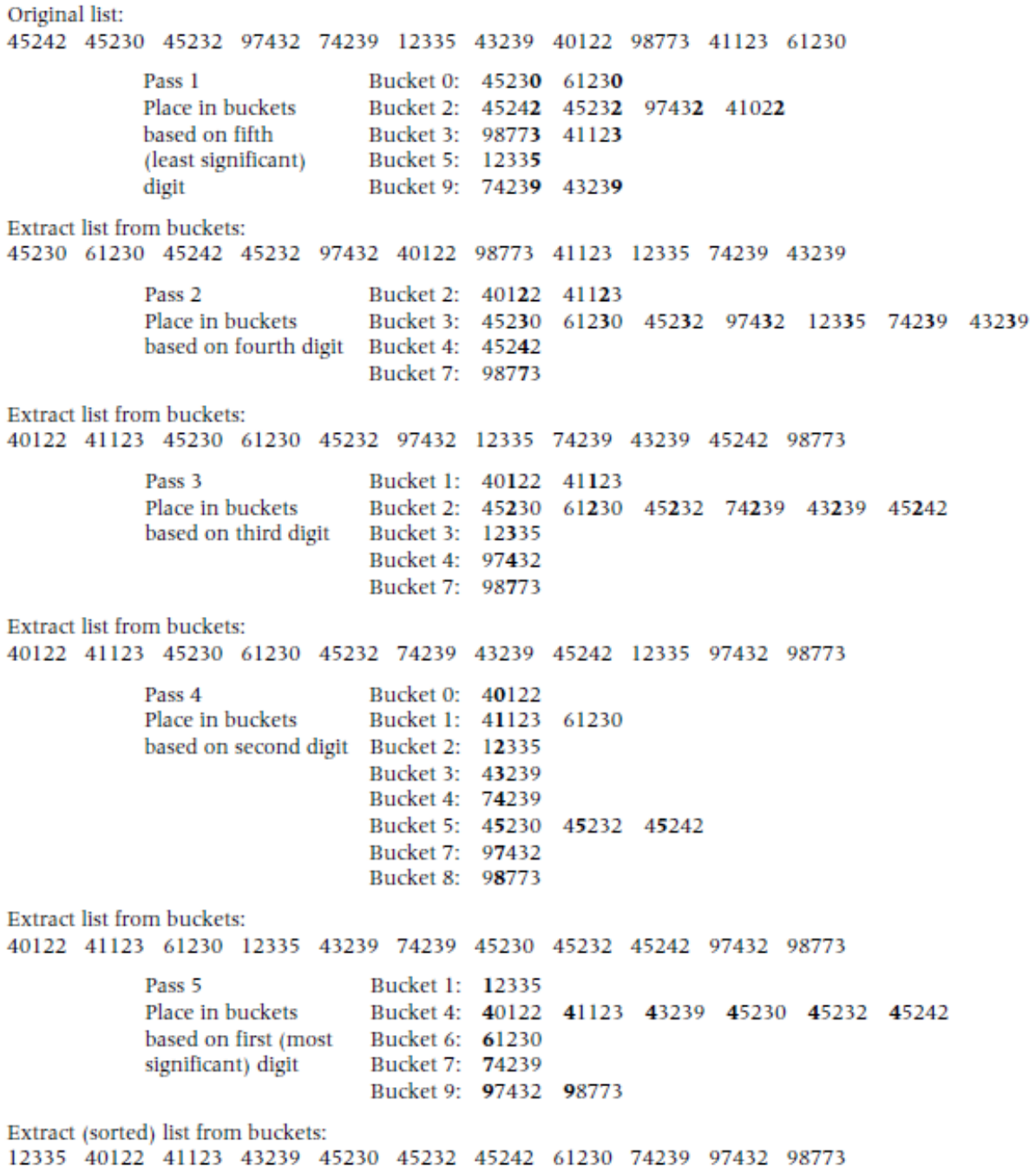
2.7 Radix Sort

The sorting algorithms that we have considered so far are comparison-based. Recall that this means that the only assumption made about the list elements is that they can be compared as to their relative order, and no additional assumptions are made about the specific nature of the elements. In this section we give an example of a sorting algorithm that is not a comparison-based algorithm, but rather depends on the special nature of the list elements. In radix sort, we assume that the list elements are strings of a given fixed length drawn from a given (ordered) alphabet and that the ordering of the strings is the usual lexicographic ordering (that is commonly used in practice). For example, we might be sorting personnel records based on social security numbers. Then the list elements are character strings of length 9 drawn from the ten character digits 0, 1, . . . , 9. Or, we might be sorting the records based on zip codes, so that the list elements are characters strings of length 5 again drawn from 0, 1, . . . , 9.

One sorting method that is sometimes used in practice when the sorting is being done by hand is to make a pass through the list, placing the records in "buckets" based on the leading character in the string. For example, suppose we are sorting zip codes. We would

use ten buckets $B_0, B_1, \ldots, B_9$. We would then make a linear scan through the list, inserting the record in B_i whenever the zip code has leading digit i, $i = 0, \ldots, 9$. The advantage of this method is that to complete the sort, we now just have to sort the records within each bucket. Sorting in each bucket could then proceed analogously, except we would place the records in sub-buckets based on the second leading digit. The disadvantage of this type of sort is the complication arising from the proliferation of buckets. However, in the case of binary strings, a fairly elegant recursive procedure similar to *Quicksort* can be developed (see Exercise 5.11).

The buckets don't proliferate if instead of looking first at the leading digit, we place the records in buckets based on the last (least significant) digit. We then repeat the process by extracting the list from the buckets in the order $B_0, B_1, \ldots, B_9$ and placing the records in these same ten buckets according to the second to the last digit. After five repetitions, the records will be sorted, as shown in Figure 2.8.



Sorting a sample list of size $n = 11$, where each element is a five-digit numerical character string, so that there are ten buckets and five passes.

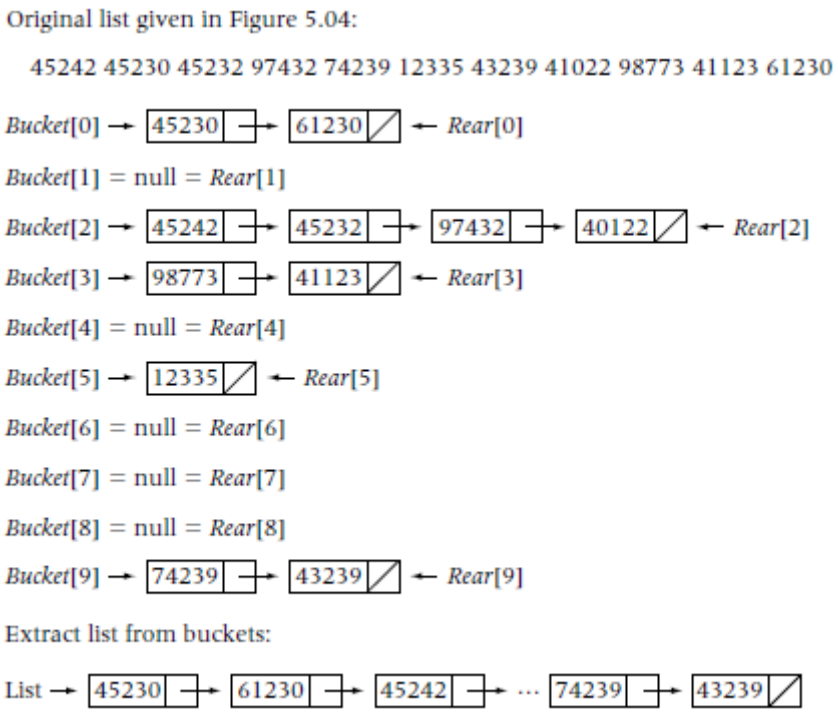
Figure 2.8

We now give a high-level description of radixsort based on the procedure illustrated in Figure 2.8. In our high-level description of the procedure *RadixSort*, we assume the built-in function *Substring*(*string*,*i*), which extracts the symbol in position *i* from *string*. We

also assume high-level procedures *Enqueue*(*Q*,*X*), which enqueues the element *X* into the queue *Q*, and *Dequeue*(*Q*,*Y*), which dequeues *Q* and assigns the dequeued element to *Y*.

```
procedure RadixSort(L[0:n – 1],k)
Input: L[0:n – 1] (a list of size n, where L[i] is a string of length k over an alphabet s_0 <
s_1 < . . . < s_m)
Output:        L[0:n – 1] (sorted in increasing order)
   for i ← k – 1  downto 0 do
   //place elements in queues B_0, B_1, . . . , B_m based on position i
      for j ← 0 to n – 1 do
         if Substring(L[j],i) = s_q then
            Enqueue(B_q,L[i])
         endif
      endfor
      //reassemble list L by successive dequeuing
      Ct ← 1
      for j ← 0 to m do
          while (.not. Empty(B_j)) do
            Dequeue(B_j,Y)
            L_Ct ← Y
           Ct ← Ct + 1
          endwhile
      endfor
  endfor
end RadixSort
```

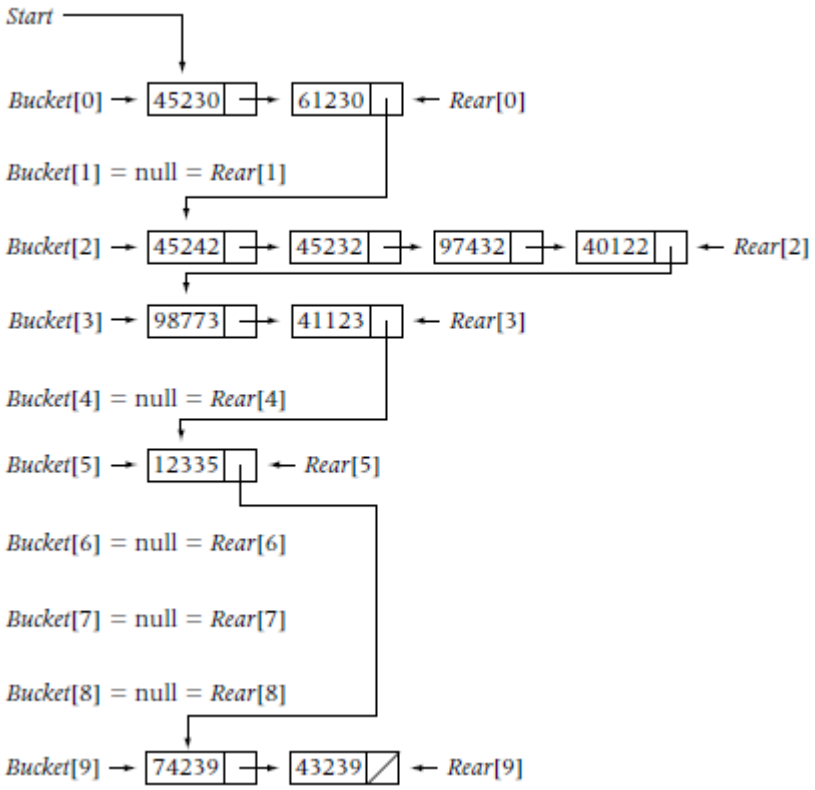
We leave the proof of the correctness of *RadixSort* as an exercise. Clearly, the (best-case, worst-case, and average) complexity of *RadixSort* is in $\Theta(kn)$. A convenient way to implement the queues used in *RadixSort* is to use linked lists. The action of the first pass of *RadixSort* is illustrated in Figure 2.9 with this implementation. We assume that the queues are pointed to by the elements in an array *Bucket*[0:9] of pointer variables. We also assume that we have an array *Rear*[0:9] of pointers, where *Rear*[*i*] points to the end of the queue whose first element is pointed to by *Bucket*[*i*], $i = 0, \ldots, 9$.



First pass of *RadixSort* for the sample list given in Figure 2.8, where the queues (buckets) are implemented using linked lists.

Figure 2.9

The extracting of the list from the buckets is a simple job of reassigning some pointers, as illustrated in Figure 2.10. Note that these pointer reassignments allow us to completely dequeue a given queue in one fell swoop! The reassembled list is a linked list pointed to by *Start*.



Extract list from buckets in the first pass of *RadixSort* for the sample list given in Figure 2.8, where the queues (buckets) and the reassembled list are implemented using linked lists.

Figure 2.10

In general, *Start* points to the first nonempty queue, that is, *Start* = *Bucket*[*i*], where *i* is the smallest integer such that *Bucket*[*i*] ≠ **null**. The reassembled linked list is then created by calling the following procedure *Reassemble*. For convenience, we write the pseudocode for *Reassemble* under the assumption that our alphabet is the ten decimal digital characters.

```
procedure Reassemble(Bucket[0:9],Rear[0:9],Start)
Input:    Bucket[0:9] (an array of pointers to queues)
          Rear[0:9] (an array of pointers to end of queues)
Output:   Start (a pointer to the reassembled linked list L)
   Ct ← 0
   while (Bucket[Ct] = null) do
      Ct ← Ct + 1
   endwhile
   Start ← Bucket[Ct]
   SaveCt ← Ct
   while(SaveCt < 9) do
      Ct ← Ct + 1
```

```
      while (Bucket[Ct] = null .and. Ct ≤ 9) do
         Ct ← Ct + 1
      endwhile
      if Ct ≤ 9 then
         Rear[SaveCt] → Next ← Bucket[Ct]
         SaveCt ← Ct
      endif
   endwhile
end Reassemble
```

2.8 Closing Remarks

For most of the algorithms in this text, our analysis of the complexity consists in identifying a basic operation and then determining the best-case, worst-case, and average complexities with respect to this basic operation. Occasionally, it is important to measure the complexity of an algorithm in terms of more than one operation. This is especially true for algorithms that involve maintaining several data structures, some of which may be altered by insertion or deletion of elements during the performance of the algorithm. *Amortized* analysis of an algorithm involves computing the maximum total number of all operations on the various data structures that are performed during the course of the algorithm for an input of size *n*. Usually this amortized total is smaller than the number obtained by simply adding together the worst-case complexities of each individual operation for inputs of size *n*. Indeed, often for an input where one of the operations achieves its worst-case complexity, another of the operations might perform much less than its worst-case complexity, so that a certain trade-off occurs for any input. When designing algorithms where amortized analysis is appropriate, it then becomes a strategy to exploit this trade-off between the various operations.

In this chapter we gave an overview of some of the major issues that arise in the design and analysis of algorithms. This overview sets the stage for the material in the rest of the text. We introduced and analyzed several important classical algorithms. Some of these algorithms were truly old, such as algorithms for exponentiation. Others, such as merge sort and quick sort, date to the period following soon after the advent of the electronic computer. Occurring throughout the rest of the text are algorithms of more recent origin, including algorithms that have become important to the efficient functioning of the Internet. Part III will be devoted to graph and network algorithms having specific application such things as search engines for the Internet.

Exercises

Section 2.2 Recursion

2.1 Give pseudocode for a recursive function that outputs the maximum value in a list of size *n*.

2.2 Give pseudocode for a recursive function that tests whether or not an input string of size *n* is a palindrome (that is, reads the same backwards and forwards).

2.3 a) Design a recursive algorithm whose input is a decimal integer and whose output is the binary representation of the input.

b) Design a recursive algorithm that computes the reverse of the result in part a), that is, converts a binary integer to its decimal equivalent.

2.4 Show that for input (x,n), *Powers* performs between $\lfloor \log_2 n \rfloor$ and $2\lfloor \log_2 n \rfloor$ multiplications.

2.5 One of the most famous sequences in computer science (and nature) is the Fibonacci sequence defined by the recurrence

$fib(n) = fib(n-1) + fib(n-2)$, **init. cond.** $fib(0) = 0$, $fib(1) = 1$.

Using the technique of generating functions, it can be shown that:

$$fib(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$$

Without using this formula, argue that

$$(1.5)^{n-2} \le fib(n) \le 2^{n-1},\ n \ge 1.$$

A formal proof requires induction, which will be discussed in Chapter 3.

2.6 Consider the following function *Fib* for computing the n^{th} Fibonacci number based directly on the recursive definition given in the previous question:

```
function Fib(n) recursive
Input: n (a nonnegative integer)
Output:  the n^th Fibonacci number
    if n ≤ 1 then
        return(n)
    else
        return(Fib(n – 1) + Fib(n – 2))
    endif
end Fib
```

a) Show that *Fib* is extremely inefficient, since it performs many redundant recalculations. How many times is $fib(k)$ computed by *Fib* when *Fib* is invoked with input n, $k = 0, 1, \ldots, n$?

b) Rewrite *Fib* so that it is still recursive but uses a table to avoid these redundant calculations.

c) Rewrite *Fib* as a purely iterative function.

2.7 Note that for the pair (89,144) given in Exercise 1.1 the algorithms *GCD* and *EuclidGCD* performed identical calculations. This phenomenon holds for infinitely many pairs of integers. One such collection can be obtained as successive pairs in the Fibonacci sequence $fib(n)$.

a) Verify that (except for the last step) *GCD* and *EuclidGCD* perform identically on any input pair $(fib(n-1), fib(n))$.

b) Obtain a formula for the number of steps required by *EuclidGCD* to compute $\gcd(fib(n-1), fib(n))$.

2.8 Show that the longest number of steps required by *EuclidGCD* for a pair of integers each having m digits or less is achieved by a suitable pair $(fib(n-1), fib(n))$.

Section 2.3 Data Structures and Algorithm Design

2.9 Given a linked list, create a linked list with the same elements but in the reverse direction.

2.10 a) Give pseudocode for the push and pop operations on a stack implemented using an array.

b) Repeat (a) for a linked list implementation.

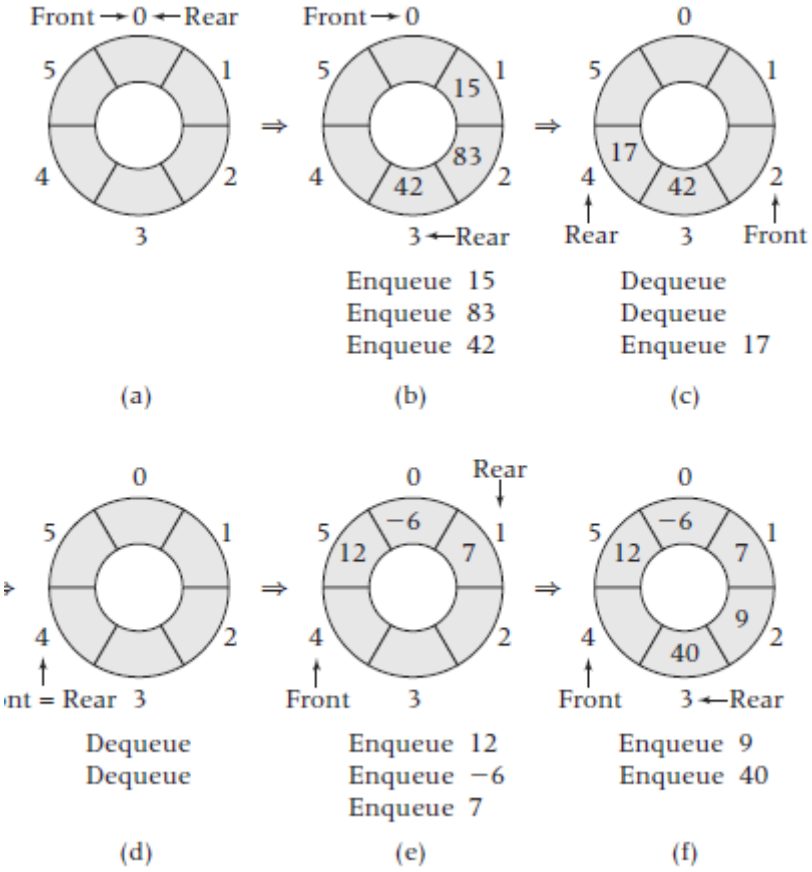
2.11 A *circular queue* is implemented using a circular array *queue*[0:*Max* – 1] and two pointers (indices) *Front* and *Rear*. We view the elements *Queue*[0], *Queue*[1], *Queue*[*Max* – 1] as positioned around a circle in a counterclockwise fashion. Initially, we have *Front* = *Rear* = 0. For nonempty queues, the variable *Rear* points to the position of the element at the rear of the queue. However, *Front* points one position counterclockwise from the element at the front of the queue. An element (in variable) *x* is enqueued by the two statements

Rear ← (*Rear* + 1) **mod** *Max*
Queue[*Rear*] ← *x*

and an element is dequeued and assigned to *x* by the two statements

Front ← (*Front* + 1) **mod** *Max*
x ← *Queue*[*Front*].

Figure 2.11 shows how the circular array *Queue* with *Max* = 6 is updated as a given sequence of enqueues and dequeues are performed.



Circular array *Queue*[0:5] after a sequence of enqueues and dequeues

Figure 2.11

If the queue is empty then *Rear* = *Front*. On the other hand, if the queue is full in the sense that the entire array *Queue* is filled, then it is also the case that *Rear* = *Front*. The question then arises as to how to distinguish between these two situations. One way is to introduce a Boolean variable, which tags the queue as being empty or full. However, maintaining such a variable may lead to a noticeable increase in the computing time, since the procedures for implementing a queue are usually called repetitively. Another solution is to consider the queue full when every position of the array *Queue* but one is filled with elements of the queue. (Of course, this wastes a single memory location in the array *Queue*, but this is hardly significant.)

a. Give pseudocode for procedures *EnqueueCirc* and *DequeueCirc* for enqueueing and dequeueing an element using the latter implementation of a circular queue.

b. Starting with an empty circular queue *Queue*[0:8], show the final state of *Queue*[0:8] and the values of *Rear* and *Front* after the following sequence of enqueues and dequeues.

enqueue 23, enqueue 108, enqueue 55, dequeue, enqueue 61, dequeue dequeue, dequeue, enqueue 44, enqueue 21, dequeue

2.12 Give pseudocode for inserting and deleting into a linked list implemented using two arrays $L[0:n-1]$ and $Next[0:n-1]$.

Section 2.5 Analyzing Algorithm Performance

2.13 Design an algorithm that tests whether or not two input lists of size n have at least one element in common. Give formulas for $B(n)$ and $W(n)$ for your algorithm.

Design and analyze an algorithm that emulates a single elimination tournament (NCAA basketball, Wimbleton tennis, and so forth) in order to find the maximum element in a list of size n. For simplicity, you may assume that n is a power of two (tournaments arrange this by giving byes when n is not a power of two).

2.14 Design a (simple) algorithm whose input is a text string T of size n and a pattern string P of size m, and determines whether P occurs as a substring of T. Give formulas for $B(n,m)$ and $W(n,m)$.

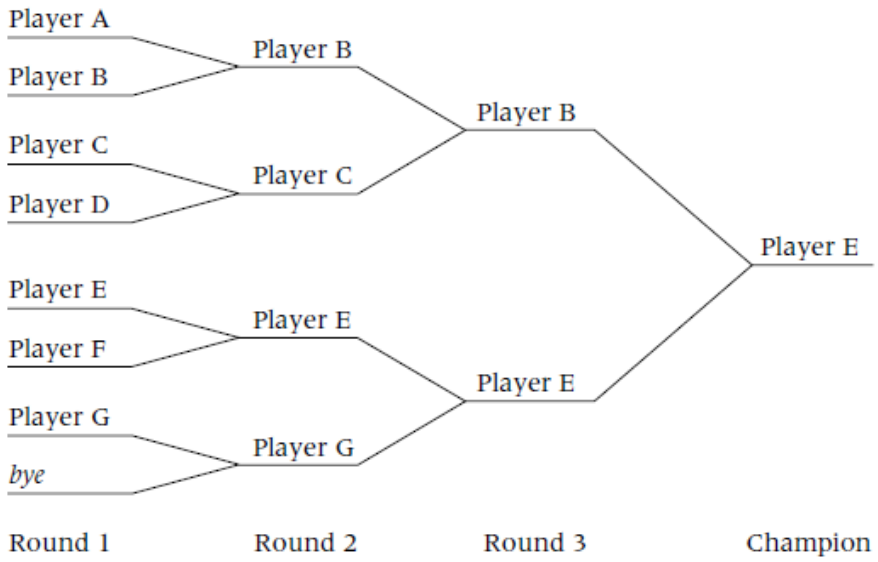
Section 2.6 Design and Analysis of Some Basic Comparison-Based List Algorithms

2.15 a) Write a procedure that finds both the largest and second-largest elements in a list $L[0:n-1]$ of size n. You should find a more efficient algorithm than simply performing two scans through the list.

b)Determine $B(n)$ and $W(n)$ for the algorithm in (a).

2.16 The idea behind a better (in fact optimal) algorithm for finding the largest and second largest element in a list, is to model the algorithm as a match-play golf or table tennis tournament. We regard the comparison of list elements as a golf or table tennis match between the elements (players). Assume for simplicity that $n = 2^k$. In the first round of the tournament, we pair the n players into $n/2$ matches. In the next round, the $n/2$ winners of the first round are paired into $n/4$ matches, and the $n/4$ winners advance into the third round. After $k = \log_2 n$ rounds, we

obtain the winner of the tournament. We draw such a single-elimination tournament in Figure 2.12. Note that to arrange for n to be a power of two, we gave Player G a *bye* in the first round.



A single-elimination tournament

Figure 2.12

The algorithm based on this tournament method makes $n/2 + n/4 + \ldots + n/2^k = n(1/2 + 1/4 + \ldots + 1/2^k) = n - 1$ comparisons in determining the largest element. The question remains as to how the algorithm can now proceed to efficiently find the runner-up (second-largest). In actual golf or table tennis tournaments, it is customary to declare the runner-up to be the player who played the winner in the last round (Player B in Figure 2.12). However, sometimes that runner-up is not really the second-best player in the tournament. (Tournament organizers attempt to minimize the latter problem by avoiding early-round matches between highest-ranked, or so-called *top-seeded*, players. Just as in Figure 2.12, they also use the method of seeding to facilitate tournament *byes*, so that the highest-ranked players don't even have to play in the first round or rounds. *Byes* allow them to assume that the first round is between $n = 2^k$ players.)

Certainly for the tournament method of determining the largest element in a list, we cannot assume that the list element that lost the last comparison made by the algorithm is the second-largest element in the list. However, the key observation here is that the second-largest must have lost a comparison to the largest element in *some* round (not necessarily the last round). Thus, for example, in our tournament shown in Figure 2.12, the candidates for second-best are Player F, Player G, and Player B. More generally, if the algorithm maintains a (dynamic) list of losers for each element, then the second-largest element must be found among the largest element's final list of losers (although a linear number of loser list updates are made, no additional comparisons are incurred). This final list contains $\log_2 n$ elements. A linear scan of these $\log_2 n$ elements (using $\log_2 n - 1$ comparisons) then determines the second-largest element in the list. Hence, the

algorithm performs a total of $n + \log_2 n - 2$ comparisons (which turns out to be an an optimal algorithm for this problem).

Write pseudocode for the algorithm just described, and prove its correctness.

2.17 Trace the action of *BinarySearch*, including listing the value of *Low*, *High*, *Mid* after each iteration, for the list

L: 2 3 5 7 11 13 17 19 23 29 31 37

for each of the following search elements X.

a. $X = 3$
b. $X = 24$
c. $X = 108$
d. $X = 13$

2.18 Show that *BinarySearch* has worst-case complexity $\lceil \log_2(n + 1) \rceil$ for any n.

2.19 Describe a modification of *BinarySearch* that returns an index in the list after which the search element can be inserted and still maintain an ordered list (or zero if the search element is smaller than any element in the list).

2.20 Design an iterative version of *BinarySearch*.

2.21 Design and analyze a variant of *BinarySearch* that performs a single check for equality between the search element and a list element.

2.22 Suppose $L[0{:}n - 1]$ is a sorted list of distinct integers. Design and prove correct an algorithm similar to *BinarySearch* that finds an index i such that $L[i] = i$ or returns 0 if no such index exists.

2.23 Give pseudocode for interpolation search, and analyze its worst-case complexity.

2.24 Give pseudocode for a recursive version of interpolation search.

2.25 It has been estimated that there are fewer than 10^{83} atoms in the known universe. Show that $\log_2(\log_2 10^{83}) < 9$.

2.26 Design and analyze a forward scan version of *InsertionSort* where the list is implemented using an array. Discuss the drawbacks of the forward scan version as compared with the backward scan version.

2.27 Show that *InsertionSort* is a *stable* sorting algorithm.

2.28 a) Design a linked list version of *InsertionSort*.
b) Give the stable (but slightly less efficient) version of the algorithm in (a).

2.29 a) Design a recursive version of *InsertionSort*.
b) Design a recursive linked list version of *InsertionSort*.

2.30 The sorting algorithm *SelectionSort* is based on the simple idea of successively selecting the largest element in a sublist $L[0{:}n - i]$ and interchanging this element with the element in position $n - i$, $i = 1, \ldots, n - 1$.
a) Give pseudocode for *SelectionSort*.
b) Analyze the complexity of *SelectionSort*.

2.31 Design a recursive version of *SelectionSort* as described in the previous exercise.

2.32 There is another well-known sorting algorithm called *BubbleSort*, which, like *SelectionSort*, results in the i^{th}-largest element being placed in its proper position at the completion of the i^{th} pass. The two algorithms differ in the manner in which this largest element is determined. During the i^{th} pass, *BubbleSort* passes sequentially through the sublist $L[0{:}n - i]$, comparing adjacent elements in the sublist and interchanging (swapping) them if they are out of order. Since this sequential pass through the sublist starts at the *beginning* of the sublist, after the

i^{th} pass we are guaranteed that the i^{th}-largest element will have "bubbled" to the end of the sublist $L[0{:}n-i]$, its proper place. In *BubbleSort*, this occurs automatically as a consequence of the interchange process, as opposed to *SelectionSort*'s method of actually determining the position in the sublist where the i^{th}-largest element occurred. Note that if on the i^{th} pass no swapping occurred, the list has been sorted. This suggests including a flag to detect this condition.

a) Give the pseudocode for *BubbleSort*.

b) Determine the best-case and worst-case complexities of *BubbleSort*.

c) Design and analyze an improvement to *BubbleSort* based on keeping track of the last positions where swaps occur in each pass and where the passes are made alternately in different directions.

2.33 Give the appropriately labeled tree of recursive calls to *MergeSort* for lists of size 18. (See Figure 2.4)

2.34 Show that the tree of recursive calls for *MergeSort* has depth $\lceil \log_2 n \rceil$. Conclude that *MergeSort* performs at most $n \lceil \log_2 n \rceil$ comparisons for any list of size n.

2.35 Design a nonrecursive version of *MergeSort* based on Figure 2.4.

2.36 Given a list $L[0{:}n-1]$, one way of maintaining a sorted order of L is to utilize an auxiliary array $Link[0{:}n-1]$. the array $Link[0{:}n-1]$ serves as a linked list determining the next highest element in L, so that the elements of L can be given in increasing order by

$$L[Start],\quad L[Link[Start]],\quad L[Link[Link[Start]]],\ \ldots .$$

Then, $Link^{n-1}[Start]$ is the index of the largest element in L, and we set $Link[Link^{n-1}[Start]] = Link^{n}[Start] = 0$ to signal the end of the linked list. Design a version of *MergeSort* that utilizes the auxiliary array *Link*.

*2.37 Develop an in-place version of *MergeSort*; that is, a version that does not use an auxiliary array and utilizes only a few additional bookkeeping variables.

2.38 a) Give pseudocode for *QuickSort* that incorporates the Median of Three Rule.

b) Show that the worst-case performance of the version of *QuickSort* in part (a) remains quadratic.

2.39 It is useful to analyze how much stacking is generated by the recursion in *QuickSort*.

a) Show that *QuickSort* may have as many as $n-1$ unresolved recursive calls active, so that the size of the stack can be as large as $n-1$.

b) Give pseudocode for a modification of *QuickSort* for which the size of the stack is at most $\log_2 n$. Verify your result.

2.40 Give a nonrecursive version of *QuickSort* (utilizing explicit stack operations).

2.41 Give a stable version of *QuickSort* that avoids making extraneous interchanges of elements having identical values.

2.42 There are a number of variants of the algorithm *Partition* used in *QuickSort*. Design and analyze one such variant based on the following strategy for partitioning an array. The elements in the array minus the pivot element p (which we take to be the first element in the array) are dynamically divided into three contiguous blocks, B_1, B_2, and B_3, where each element in B_1 is less than p, each element in B_2 is greater than or equal to p, and the status of the elements in B_3 is yet to be decided. Initially, B_1, B_2 are empty (so that B_3 is everything in the

subarray except p). The algorithm performs $n - 1$ steps, where each step consists in placing the first element in B_3 into either B_1 or B_2 as appropriate.

Section 2.7 Radix Sort

2.43 Demonstrate the action of *RadixSort* for the following sample list, where each element is a 7-digit binary string (see Figure 2.8).
1011101, 0100011, 1010110, 1111101, 0011101, 0000001, 1000000, 1010101

2.44 Demonstrate the first two passes of *RadixSort* for the sample list given in Exercise 2.43, where the buckets are implemented as linked queues (see Figures 2.9 and 2.10).

2.45 Prove the correctness of *RadixSort*.

2.46 Determine whether *RadixSort* is a stable sorting algorithm.

2.47 For binary strings stored in an array, design and analyze a recursive radix sort based on partitioning the list into two sublists according to the most significant digit.

Additional Exercises

2.48 Although for large lists *InsertionSort* is, on average, much slower than *MergeSort*, it is faster for sufficiently small lists. A useful technique to improve the efficiency of a divide-and-conquer sorting algorithm such as *MergeSort* is to employ a sorting algorithm such as *Insertionsort* when the input size is not larger than some threshold t (a value of t between 8 and 16 usually works well).
a) Write a program that computes the number of comparisons performed by *Mergesort* for a given input list and threshold t.
b) Run for thresholds t from 5 to 20 for various randomly generated lists of sizes 100, 1000, 10000. Report on which threshold t you observe to be best, and how it compares to ordinary *MergeSort*.

2.49 Repeat Exercise 5.48 for *Quicksort*.

2.50 A sloppy chef has delivered a stack of pancakes to a waiter, where the stack is all mixed up with respect to the size of the pancakes. The waiter wishes to rearrange the pancakes in order of their size, with the largest on the bottom. The waiter begins by selecting a pile of pancakes on the top of the stack and simply flips the pile over. He then keeps repeating this pancake-flipping operation until the pancakes are sorted.
a. Show that there always exists a set of flips that sorts any given stack of pancakes.
b. Describe a pancake-flipping algorithm that performs at most $2n - 2$ flips in the worst case for n pancakes.
c. Show that any pancake-flipping algorithm must perform at least $n - 1$ flips in the worst case for n pancakes.
d. Suppose now that the pancakes have a burnt side, and the waiter wishes to have the pancakes stacked in order with the unburned side up. Redo parts (a), (b), and (c) for this new problem.

Shell Sort

The Exercises 2.51-2.54 refer to the sorting algorithm Shell sort. Shell sort is often used in practice due to its simplicity of code and good (but not optimal) worst-case performance. When he designed Shell sort in 1959, Donald Shell's idea was to start out by sorting sublists of a list $L[0{:}n-1]$ of size n consisting of equally spaced, but far apart, elements. For example, if the successive elements in the sublists occupy positions k distance apart in the original list, then we have k sublists $S_0 = \{L[0], L[k], \ldots\}$, $S_1 = \{L[1], L[1+k], \ldots\}$, $\ldots$, $S_{k-1} = \{L[k-1], L[2k-1], \ldots\}$. The process of sorting these k sublists is called a *k-subsort* (note that a 1-subsort corresponds to a sorting of the entire list L). A list in which each of the sublists $S_0, S_1, \ldots, S_{k-1}$ is sorted is called *k-subsorted* (also referred to in the literature as k-ordered, or k-sorted). Each of the sublists $S_0, S_1, \ldots, S_{k-1}$ contains at most $\lceil n/k \rceil$ list elements, so for a large k, sorting each sublist is fast using a sort like *InsertionSort*. Although the list L itself is not sorted by the k-subsort, in some sense it is closer to being sorted. Shell sort works by performing a succession of k-subsorts with decreasing increments $k_0 > k_1 > \ldots > k_{m-1} = 1$.

To accomplish a k-subsort, rather than calling a sorting procedure for each sublist individually, we make a single call to the following simple variant of *InsertionSort*.

```
procedure InsertionSubsort(L[0:n – 1],k)
Input: L[0:n – 1] (a list of size n), k (a positive integer increment)
Output:      L[0:n – 1] (k-subsorted in increasing order)
      for i ← k  to n – 1 do   //insert L[i] in its proper position in the
        //already sorted sublist L[j], L[j + k] . . . , L[i – k] of S_j,   j = i mod k
            Current ← L[i]
            position ← i – k
            while position ≥ 1 .and. Current < L[position] do
                  // Current must precede L[position]
                  L[position + k] ← L[position]// bump up L[position]
                  position ← position – k
            endwhile
            // position + k is now the proper position for current = L[i]
            L[position+k] ← Current
      endfor
end InsertionSubsort
```

The following pseudocode for *ShellSort* merely consists of successively calling *InsertionSubsort* with a given set of diminishing increments.

```
procedure ShellSort(L[0:n – 1],D[0:m – 1])
Input: L[0:n – 1] (a list of size n)
                  K[0:m – 1] (an array of diminishing increments
                              K[0] > K[1] > . . . > K[m – 1] = 1)
Output:      L[0:n – 1] (sorted in increasing order)
      for i ← 0 to m – 1 do
            InsertionSubsort(L[0:n – 1],K[i])
```

```
        endfor
end ShellSort
```

The final stage of *ShellSort* consists of performing *InsertionSort* on the entire list, so that it is unquestionably correct. However, by then the list should be close to being sorted, so *InsertionSort* should work quickly on the list (see Exercise 2.53). Also, successive k-subsorting with diminishing k tends to reorder a list closer and closer to being sorted (see Exercise 2.54).

Even though Shell sort is a fairly simple algorithm and has been well-studied and used by many researchers, its analysis is not yet complete. The number of comparisons for a given input of size n performed by Shell sort depends on what set of increments is used, and the general mathematical analysis is difficult. For a general n, it is not known what set of increments exhibits optimal behavior.

A number of results are known for special choices of the increments. An order of complexity improvement over insertion sort is already achieved when only two increments k and 1 are used for a suitable k. In fact, it has been shown in the two-increment case that the optimal average complexity has order $n^{5/3}$ and is achieved when k is approximately $1.72n^{1/3}$. It has also been shown that when the increments are of the form $2^j - 1$, for all such increments less than n, then Shell sort exhibits worst-case complexity bounded above by $kn^{3/2}$ for a suitable constant k and large n. Since $2^{2j} - 1 = (2^j - 1)(2^j + 1)$, a large percentage of these increments divide one another, so we would expect to be able to do even better. Indeed, it has been shown that using increments of the form 2^i3^j for all such increments less than n yields approximately equal to $kn(\log_2 n)^2$ worst-case complexity.

2.51 a) Demonstrate the action of *ShellSort* on the list
33,2,56,23,55,78,2,98,61,108,14,60,56,77,5,3,1 with increments 5,3,1.
b) Repeat (a) with increments 5,2,1.

2.52 a) Write a program that compares the number of comparisons made for a given input list $L[0{:}n-1]$ by *InsertionSort* to that for *ShellSort* with a given set of increments $K[0{:}m-1]$.
b) Run your program with the input list and sets of increments given in Exercise 2.51
c) Repeat (b) for randomly generated lists of sizes 100, 1000, 10000.

2.53 Suppose for each list element $L[i]$ in a list $L[0{:}n-1]$ of size n, there are no more than k list elements $L[j]$ such that $j < i$ and $L[j] > L[i]$. Then *InsertionSort* makes at most $(k + 1)(n - 1)$ comparisons to sort the list.

*2.54 For any two integers k and l, suppose we perform an l-subsort on a list L that is already k-subsorted. Prove that then L remains k-subsorted—that is, L becomes *both* k-subsorted *and* l-subsorted.

Bingo Sort

In all of the sorting algorithms we have studied so far, the input size was a function of a single variable n. However, there are situations where the input size is most naturally a function of two or more variables. For example, suppose we consider the problem of finding efficient sorting algorithms for lists of size n with repeated elements. The input size is then a function of n and the number m of *distinct* elements in the list, where $1 \le m$

$\leq n$. The situation of repeated elements arises often in practice. For example, we might wish to sort the people in a large mailing list by zip codes for bulk mailing. Then the number of distinct elements m (zip codes) will be rather less than the number of people n. In this section we discuss bingo sort, which is an example of a sorting algorithm that works well for input lists with many repeated elements.

Bingo sort, which is a variation of selection sort, works as follows. Each distinct value in the list is considered to be a "bingo" value. Each pass of bingo sort corresponds to "calling out" a bingo value. The bingo values are called out in increasing order. During a given pass, all the elements having the current bingo value are placed in their correct positions in the list. The following pseudocode for bingo sort implements this process.

```
procedure BingoSort(L[0:n – 1])
Input:        L[0:n – 1] (a list of size n)
Output:       L[0:n – 1] (sorted in increasing order)
// first determine the minimum and maximum value for a list element
   MaxMin (L[0:n – 1],MaxValue,MinValue)
   Bingo ← MinValue  //initialize
   NextAvail ← 0
   NextBingo ← MaxValue
   while Bingo < MaxValue do  //move up all list elements that equal Bingo
      StartPos ← NextAvail
      for i ← StartPos to n – 1 do
         if L[i] = Bingo then  //BINGO! Interchange L[i] and L[NextAvail]
            interchange(L[i],L[NextAvail])
            NextAvail ← NextAvail + 1  // update NextAvail
         else
            if L[i] < NextBingo then NextBingo ← L[i] endif
         endif
      endfor
      Bingo ← NextBingo   //initialize for next pass
      NextBingo ← MaxValue
   endwhile
end BingoSort
```

The action of *BingoSort* is illustrated in Figure 2.13 for a sample list of size 10.

Pass 1	23	10	15	10	10	23	15	23	23	10
Start position 0	10	23	15	10	10	23	15	23	23	10
Bingo value 10	10	10	15	23	10	23	15	23	23	10
	10	10	10	23	15	23	15	23	23	10
	10	10	10	10	15	23	15	23	23	23
Pass 2										
Start position 4	10	10	10	10	15	23	15	23	23	23
Bingo value 15	10	10	10	10	15	15	23	23	23	23

Action of *BingoSort* for list 23, 10, 15, 10, 10, 23, 15, 23, 23, 10 after call to *MaxMin*

Figure 2.13

The best-case complexity $B(n,m)$ of *BingoSort* is achieved by a list where $n - m + 1$ elements have the (same) minimum value, and the worst case occurs for a list where $n - m + 1$ elements have the (same) maximum value (see Exercise 2.56).

2.55 Show that if a list has m distinct elements, then after exactly $m - 1$ passes of *BingoSort* the list is sorted into increasing order and the algorithm terminates.

2.56 a. Analyze the best-case complexity $B(n,m)$ of *BingoSort*.

b. Analyze the worst-case complexity $W(n,m)$ of *BingoSort*.