3

Mathematical Tools for Algorithm Analysis

Determining the complexities of an algorithm exactly is often difficult, and in practice it is usually sufficient to find asymptotic approximations to these complexities. To aid our description of asymptotic behavior, in this chapter we introduce and utilize standard notation for order and growth rate of functions. We also give you tools that can be used to help classify the growth rate of a given function as compared to that of commonly occurring growth rates such as logarithmic, linear, quadratic, and so forth.

In Chapter 2 we discussed the importance of recursion as a design strategy. In this chapter we discuss how recursive algorithms usually immediately yield recurrence relations for their complexities, and we illustrate this by giving an alternate proof of the worst-case complexity of *MergeSort*. We also give a general formula for solving some of the more commonly occurring recurrence relations that arise in algorithm analysis.

Proving correctness of an algorithm is perhaps even more fundamental to its analysis than determining its efficiency. Mathematical induction is the primary tool used to establish the correctness of algorithms. For recursive algorithms, induction on the input size is usually the relevant tool. For algorithms involving loops, establishing loop invariants (again, using induction) is usually the tool most relevant. Both techniques will be discussed in this chapter.

A function $L(n)$ is called a lower bound for, say, the worst case $W(n)$ of a problem Q, if *any* algorithm solving Q must perform at least $L(n)$ basic operations for some problem instance of Q of size n. Establishing lower bounds is an important tool in algorithm analysis, since they can be used to determine how close a given algorithm is to being optimal for the problem. In this chapter we use some simple counting arguments to establish lower bounds for such important problems as finding the maximum in a list, and sorting a list using a comparison-based sorting algorithm.

Unfortunately, there are many important problems for which the gap between a known lower bound for the problem and the performance of the best known algorithm solving the problem is wide indeed. We close this chapter with a discussion of an important class of such problems, known as NP-complete problems. Typically, a known lower bound for a given NP-complete problem is a polynomial of low degree, whereas the best known algorithm solving the problem has exponential (or at least super-polynomial) complexity. But nobody knows whether the NP-complete problems are really hard, or whether, despite over 40 years of searching by the best minds in theoretical computer science, we have simply just not found the key to solving any of them with a polynomial-complexity algorithm.

3.1 Asymptotic Behavior of Functions

Suppose $f(n) = 400n + 23$ and $g(n) = 2n^2 - 1$ are the worst-case complexities for two algorithms A and B, respectively, that solve the same problem. Which exhibits better worst-case performance? If $n \leq 200$, then $f(n)$ is greater than $g(n)$, so that algorithm B outperforms A. On the other hand, if $n > 200$, the reverse is true. When analyzing the complexity of an algorithm, it is

the behavior of the algorithm as the input size n tends to infinity (asymptotic behavior) that is usually considered to be most important. Thus, algorithm A is considered to have better worst-case behavior.

For purposes of algorithm analysis, the asymptotic behavior (*order*) of a function $f(n)$ not only ignores small values of n, but it also doesn't distinguish between $f(n)$ and $cf(n)$, where c is a positive constant. For example, *InsertionSort* performs $n^2/2 - n/2$ comparisons in the worst case for an input of size n. However, $n^2/2 - n/2$ and n^2 are considered to be asymptotically equivalent measures of the complexity of an algorithm.

There is good reason not to be too concerned about constants when we analyze algorithms. After all, the running time of an algorithm when we actually implement the algorithm in code on a computer will vary from one computer to another due to differences in processor speed, memory access, and so forth. However, these differences between the speed of two computers can generally be captured by a constant factor. It is for this reason that we analyze algorithms by identifying a basic operation and counting how many of these basic operations are performed on a given input, as opposed to measuring the running time of the algorithm on this input when implemented on some specific computer. When we talk about asymptotic growth, we will generally drop the phrase "up to a multiplicative constant," it being implicitly understood.

Of course, in practice the constants do matter. For example, if we have an algorithm with worst case $W(n) = 10^{1000}n$, then we do have a *linear* worst-case algorithm. However, in any practical situation, we would prefer to use an algorithm with worst case $W(n) = n^2$ even though the latter algorithm exhibits *quadratic* performance. It is true that eventually (i.e., for large enough n) the linear algorithm will perform fewer basic operations, but this only happens for $n > 10^{1000}$, which is an input size so large that it probably won't ever be encountered in a practical situation. On the other hand, it is theoretically important to know that there is a linear algorithm for this problem, since it gives us the hope that another linear algorithm with a smaller constant might exist for the problem.

3.1.1 The class "big oh" *O*

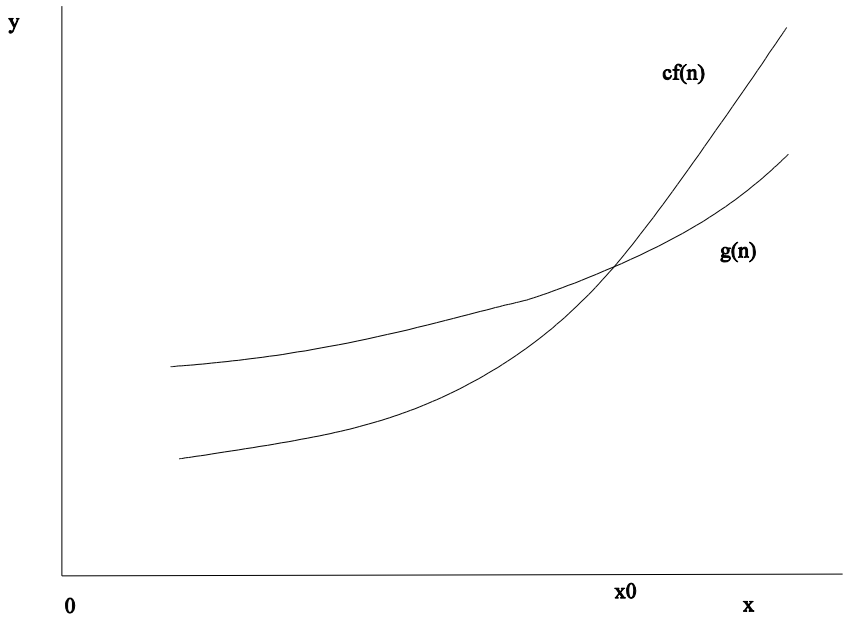
A fundamental notion in algorithm analysis, particularly in regard to the worst-case performance of an algorithm, is the "*big oh*" notation.

Definition 3.1.1

Given a function $f(n) \in \mathscr{F}$ we define $O(f(n))$ to be the set of all functions $g(n) \in \mathscr{F}$ having the property that there exist positive constants c and n_0 such that for all $n \geq n_0$,

$$g(n) \leq c\, f(n) \qquad (3.1.1)$$

If $g(n) \in O(f(n))$, we say that $g(n)$ *is big oh of* $f(n)$, and that (up to a multiplicative constant depending on g and f) $f(n)$ *grows at least as fast as* $g(n)$ (See Figure 3.7).



$g(n) \in O(f(n))$

Figure 3.7

Remarks

By abuse of language, the condition $g(n) \in O(f(n))$ is sometimes written $g(n) = O(f(n))$ (a similar remark holds for the classes $\Omega(f(n))$, and $\Theta(f(n))$ defined later). When discussing algorithm behavior in the literature, you will often see a title for a paper reads something like "An $O(f(n))$ algorithm for … (a given problem)". What is meant is that the algorithm described in the paper has worst-case $W(n) \in O(f(n))$. Also, since in asymptotic behavior we generally ignore positive constants, we would not, for example, say that *Insertionsort* is an $O(n^2/2 - n/2)$ algorithm, but simply say it is an $O(n^2)$ (or quadratic) algorithm. The reason is (exercise) that $O(n^2/2 - n/2) = O(n^2)$.

3.1.2 The Classes "big omega" Ω and "big theta" Θ

Just as big oh is useful in describing the asymptotic behavior of $W(n)$ for an algorithm A, big omega Ω is useful in describing the asymptotic behavior $B(n)$ of A. In other words, big oh is used to establish *upper bounds* on the (asymptotic) behavior of an algorithm, whereas Ω is used to establish *lower bounds* on the (asymptotic) behavior of an algorithm.

Definition 3.1.2

Given a function $f(n) \in \mathscr{F}$, we define $\Omega(f(n))$ to be the set of all functions $g(n) \in \mathscr{F}$ having the property that there exist positive constants c and n_0 (depending on g and f) such that for all $n \geq n_0$,

$$g(n) \geq cf(n). \qquad (3.1.2)$$

If $g(n) \in \Omega(f(n))$, we say that $g(n)$ *is big omega of* $f(n)$, and that (up to a multiplicative constant depending on g and f) $g(n)$ *grows at least as fast as* $gf(n)$.

Note that $g(n) \in \Omega(f(n))$ if, and only if, $f(n) \in O(g(n))$. If $f(n)$ grows at least as fast as $g(n)$ (that is, $g(n) \in O(f(n))$, and $g(n)$ grows at least as fast as $f(n)$ (that is, $g(n) \in \Omega(f(n))$, then it is natural to say that $g(n)$ grows at the same rate as $f(n)$. The following class Θ formalizes this notion.

Definition 3.1.3

Given a function $f(n) \in \mathscr{F}$, we define $\Theta(f(n))$ to be the set of all functions $g(n) \in \mathrm{F}$ having the property that there exist positive constants c_1, c_2, and n_0 (depending on g and f) such that for all $n \geq n_0$,

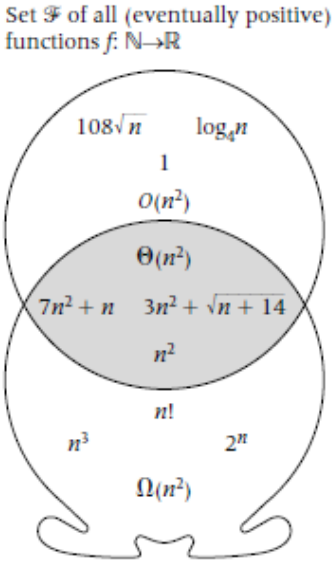
$$c_1 f(n) \leq g(n) \leq c_2 f(n). \qquad (3.1.3)$$

If $g(n) \in \Theta(f(n))$, we say that $g(n)$ *is big theta of* $f(n)$, and that (up to positive multiplicative constants) $g(n)$ *grows at the same rate as* $f(n)$.

It is easy to verify that Θ determines an equivalence relation on the set $\mathscr{F}$ (exercise), and that

$$\Theta(f(n)) = \Omega(f(n)) \cap O(f(n)). \qquad (3.1.4)$$

Sample members of the sets $O(n^2)$, $\Theta(n^2)$, and $\Omega(n^2)$ are illustrated in Figure 3.2.



Venn diagram for $O(n^2)$, $\Theta(n^2)$, and $\Omega(n^2)$ and sample set members

Figure 3.2

Remark

As we have said, big oh O is the most important of the three relations in algorithm analysis. Moreover, Ω and Θ can both be defined in terms of O, since (exercise)

$$g(n) \in \Omega(f(n)) \Leftrightarrow f(n) \in O(g(n)), \text{ and}$$
$$g(n) \in \Theta(f(n)) \Leftrightarrow f(n) \in O(g(n)) \text{ and } g(n) \in O(f(n)).$$

3.1.3 Useful Properties of the sets *O*, Θ, and Ω

Proposition 3.1.1 states some useful properties of the sets O, Θ, and Ω.

Proposition 3.1.1

The following properties hold for $f(n), g(n) \in F$.

1. Given any positive constant c, $\Omega(f(n)) = \Omega(cf(n))$, $\Theta(f(n)) = \Theta(cf(n))$, and $O(f(n)) = O(cf(n))$.
2. $g(n) \in O(f(n)) \Leftrightarrow O(g(n)) \subseteq O(f(n))$.
3. $O(f(n)) = O(g(n)) \Leftrightarrow \Omega(f(n)) = \Omega(g(n)) \Leftrightarrow \Theta(f(n)) = \Theta(g(n))$.
5. $g(n) \in O(f(n)) \Leftrightarrow f(n) \in \Omega(g(n))$. □

The proof of Proposition 3.1.1 is left to the exercises. It is common to use the notation

$$f(n) = g(n) + O(h(n)),$$

when $f(n) - g(n) \in O(h(n))$. Similar notation is used for Θ and Ω. For example, if $f(n) = 3n^3 + 4n^2 + 23n - 108$, then we could write

$$f(n) = 3n^3 + O(n^2).$$

We now give a table (Figure 3.3) that summarizes the order of the best-case, average, and worst-case complexities for most of the algorithms that we have discussed so far. The table also includes the sorting algorithm *HeapSort*, which is discussed in the next chapter.

Algorithm	$B(n)$	$A(n)$	$W(n)$
HornerEval	n	n	n
Towers	2^n	2^n	2^n
LinearSearch	1	n	n
BinarySearch	1	$\log n$	$\log n$
Max, Min, MaxMin	n	n	n
InsertionSort	n	n^2	n^2
MergeSort	$n \log n$	$n \log n$	$n \log n$
HeapSort	$n \log n$	$n \log n$	$n \log n$
QuickSort	$n \log n$	$n \log n$	n^2

Order of best-case, average, and worst-case complexities

Figure 3.3

The following proposition provides a useful test for comparing the orders of two functions $f, g \in \mathscr{F}$ when the limit of the ratio $f(n)/g(n)$ exists as n tends to infinity.

Theorem 3.1.2 The Ratio Limit Theorem

Let $f(n), g(n) \in \mathscr{F}$. If the limit of the ratio $f(n)/g(n)$ exists as n tends to infinity, then f and g are comparable. Moreover, assuming $L = \lim_{n\to\infty} f(n)/g(n)$ exists, then the following results hold.

1.	$0 < L < \infty \Rightarrow f(n) \in \Theta(g(n))$.	f and g have the same order.
2.	$L = 0 \quad \Rightarrow O(f(n)) \subset O(g(n))$	f has a *smaller* order than g.
3.	$L = \infty \quad \Rightarrow O(g(n)) \subset O(f(n))$	f has a *larger* order than g.

Proof The definition of $\lim_{n\to\infty} f(n)/g(n) = L$ states that for any given real number $\varepsilon > 0$ there exists an integer N_ε (depending on ε) such that for all $n \geq N_\varepsilon$, $|f(n)g(n) - L| < \varepsilon$ or, equivalently,

$$(L-\varepsilon)g(n) < f(n) < (L+\varepsilon)g(n), \quad \text{for all } n \geq N_\varepsilon. \tag{3.1.5}$$

***Case* 1: $0 < L < \infty$.** We are free to choose ε to be any positive real number. In particular, we may choose $\varepsilon = L/2$. Setting $c_1 = L/2$, $c_2 = 3L/2$, and $n_0 = N_{L/2}$ and substituting these values into (3.1.5) yields:

$$c_1 g(n) < f(n) < c_2 g(n), \quad \text{for all } n \geq n_0. \tag{3.1.6}$$

Hence, $f(n) \in \Theta(g(n))$.

***Case* 2: $L = 0$.** Substituting $L = 0$ into (3.1.5), for *any* positive real number ε we have

$$f(n) < \varepsilon g(n), \quad \text{for all } n \geq N_\varepsilon. \tag{3.1.7}$$

It follows immediately that $f(n) \in O(g(n))$. To prove that $O(f(n)) \subset O(g(n))$, it is necessary to show that $f(n) \notin \Omega(g(n))$. Assume to the contrary that there exist constants c and n_0 such that

$$cg(n) \leq f(n), \text{for all } n \geq n_0. \tag{3.1.8}$$

Substituting $\varepsilon = c$ in (3.1.7) yields $f(n) < cg(n)$ for all $n \geq N_c$. Thus, for any integer n larger than the maximum of n_0 and N_c, $f(n) < cg(n)$ and $cg(n) \leq f(n)$, which is a contradiction. Hence, $f(n) \notin \Omega(g(n))$, which implies that $O(f(n)) \subset O(g(n))$.

***Case* 3: $L = \infty$.** For this case, the proof that $O(g(n)) \subset O(f(n))$ is left as an exercise. ■

Remark

A partial converse to case 2 of Theorem 3.1.2 is true: If $O(f(n)) \subset O(g(n))$ and $\lim_{n\to\infty} f(n)/g(n)$ exists, then this limit must be zero. It is easy to construct examples where $O(f(n)) \subset O(g(n))$ but

$\lim_{n\to\infty} f(n)/g(n)$ does not exist (see Exercise 3.18). Similar remarks hold in the other two cases of Theorem 3.1.2.

The condition $\lim_{n\to\infty} f(n)/g(n) = 0$ is so useful in mathematics that the following special notation is used to denote the corresponding relation on the set of real-valued functions.

Definition 3.1.6

Given the function $f(n)$, we define $o(f(n))$ to be the set of all the functions $g(n)$ having the property that

$$\lim_{n\to\infty} g(n)/f(n) = 0.$$

If $f(n) \in o(g(n))$, we say that $f(n)$ *is little oh of* $g(n)$.

Remark

In the mathematical literature, $g(n) \in o(f(n))$ is usually written $g(n) = o(f(n))$.

3.1.4 Strongly Asymptotic Behavior

Definition 3.1.7

Given the function $f(n)$, we define $\sim(f(n))$ to be the set of all functions $g(n)$ having the property that

$$\lim_{n\to\infty} g(n)/f(n) = 1.$$

If $g(n) \in \sim(f(n))$, then we say that $g(n)$ is *strongly asymptotic* to $f(n)$ and denote this by writing $g(n) \sim f(n)$.

It is easy to see that $\sim$ is an equivalence relation on $\mathscr{F}$ (exercise).

The following proposition follows immediately from the Ratio Limit Theorem and the definitions of o and ~.

Proposition 3.1.5

Let $f(n), g(n) \in \mathscr{F}$. Then

1. $g(n) \sim f(n) \Rightarrow g(n) \in \Theta(f(n))$
2. $f(n) \in o(g(n)) \Rightarrow O(f(n)) \subset O(g(n))$.

We illustrate Proposition 3.1.5 by proving the following stronger version of Proposition 3.1.3 concerning the asymptotic behavior of polynomials.

Proposition 3.1.6

Let $P(n) = a_k n^k + \ldots + a_1 n + a_0$ be any polynomial of degree k, $a_k > 0$. Then $P(n) \sim a_k n^k$.

Proof Consider the ratio $(a_k n^k + \ldots + a_0)/a_k n^k$ as n tends to infinity

$$\lim_{n\to\infty} \frac{(a_k n^k + \cdots + a_0)}{a_k n^k}$$

$$= \lim_{n\to\infty}\left(1 + \frac{a_{k-1}}{a_k n} + \cdots + \frac{a_1}{a_k n^{k-1}} + \frac{a_0}{a_k n^k}\right) = 1.$$

Proposition 3.1.6 now follows immediately from property (1) of The Ratio Limit Theorem. ■

As an example, since the binomial coefficient $C(n,k)$ is a polynomial of degree k with leading coefficient $1/k!$, it follows that $C(n,k) \sim n^k/k!$.

Applying Propositions 3.1.4 and 3.1.5 requires calculating $\lim_{n\to\infty} f(n)/g(n)$, which might be difficult to do directly. When calculating $\lim_{n\to\infty} f(n)/g(n)$, a powerful tool from calculus, known as L'Hôpital's Rule, is often helpful. Application of L'Hôpital's Rule requires that $f(n)$ and $g(n)$ be extended to functions $f(x)$ and $g(x)$, respectively, both of which are differentiable for sufficiently large real numbers x.

L'Hôpital's Rule

Let $f(x)$ and $g(x)$ be functions that are differentiable for sufficiently large real numbers x. If $\lim_{x\to\infty} f(x) = \infty$ and $\lim_{x\to\infty} g(x) = \infty$, then

$$\lim_{n\to\infty} \frac{f(x)}{g(x)} = \lim_{n\to\infty} \frac{f'(x)}{g'(x)}.$$

We now apply the Ratio Limit Theorem to an example that makes use of L'Hôpital's rule. We show that $2n^{3/2} + 6n\ln n \in \Theta(n^{3/2})$ by considering the ratio $(2n^{3/2} + 6n \ln n)/n^{3/2}$ as n tends to infinity. We have:

$$\lim_{x\to\infty} \frac{(2x^{3/2} + 6x \ln x)}{x^{3/2}} = 2 + 6 \lim_{x\to\infty} \frac{\ln x}{x^{1/2}}$$

$$= 2 + 6 \lim_{x\to\infty} \frac{1/x}{(1/2)x^{-1/2}}$$

$$= 2 + 12 \lim_{x\to\infty} \frac{1}{x^{1/2}} = 2 + 0 = 2. \qquad \text{(by L'Hôpital's Rule)}$$

As our second illustration, we show that any polynomial has smaller order than any exponential function.

Proposition 3.1.7

For any nonnegative real constants k and a, with $a > 1$,

$$O(n^k) \subset O(a^n).$$

Proof Consider the ratio n^k/a^n. Repeated applications of L'Hôpital's Rule yields

$$\begin{aligned}\lim_{n\to\infty} \frac{n^k}{a^n} &= \lim_{n\to\infty} \frac{kn^{k-1}}{(\ln a)a^n} \\ &= \lim_{n\to\infty} \frac{k(k-1)n^{k-2}}{(\ln a)^2 a^n} \\ &\vdots \\ &= \lim_{n\to\infty} \frac{k!}{(\ln a)^k a^n} = 0.\end{aligned}$$

Hence, $n^k \in O(a^n)$. Proposition 3.1.7 now follows from the Ratio Limit Theorem. ■

3.2 Asymptotic Order Formulae for Three Important Series

In this section we determine the asymptotic order for three mathematical series that occur often in algorithm analysis, namely, the sum of powers $S(n,k) = 1^k + 2^k + \ldots + n^k$, for k a nonnegative integer, the sum of logarithms $L(b,n) = \log_b 1 + \log_b 2 + \ldots + \log_b n = \log_b(n!)$ and the harmonic series $H(n) = 1 + 1/2 + \ldots + 1/n$. We will see important applications of all of these formulas throughout the text.

3.2.1 Sums of Powers

$S(n,1) = 1 + 2 + \ldots + n$ is certainly one of the most frequently occurring series in algorithm analysis. For example, it will come up later this chapter in the analysis of adjacent-key comparison sorting. We will now derive the recurrence relation (3.2.1) that does not yield an explicit formula for $S(n,k)$ for a general k, but nevertheless can be used to show that $S(n,k)$ is a polynomial in n of degree $k + 1$. It is interesting that recurrence relation (3.2.1) was also known to Pascal, who in fact published it in a slightly more general form (see Exercise 3.33).

Proposition 3.2.1

Given any nonnegative k, $S(n,k) = 1^k + 2^k + \ldots + n^k$ satisfies the recurrence relation

$$\begin{aligned}S(n,k) = \frac{1}{k+1}\Bigg[(n+1)^{k+1} - 1 - \Bigg(&\binom{k+1}{0} S(n,0) + \binom{k+1}{1} S(n,1) \\ &+ \cdots + \binom{k+1}{k-1} S(n,k-1)\Bigg)\Bigg] \qquad \textbf{init.cond. } S(n,0) = n.\end{aligned} \tag{3.2.1}$$

Proof The following proof amounts to interchanging the order of summation, using the binomial theorem, and noting the resulting telescoping sum.

$$\sum_{j=0}^{k}\binom{k+1}{j}S(n,j)=\sum_{j=0}^{k}\binom{k+1}{j}\sum_{i=1}^{n}i^{j}$$

$$=\sum_{i=1}^{n}\sum_{j=0}^{k}\binom{k+1}{j}i^{j} \quad \text{(by interchanging the order of summation)}$$

$$=\sum_{i=1}^{n}\left[(1+i)^{k+1}-i^{k+1}\right] \quad \text{(by the binomial theorem)}$$

$$=(1+n)^{k+1}-1 \quad \text{(by "telescoping sum").} \qquad (3.2.2)$$

Formula (3.2.1) follows immediately from (3.2.2) by isolating the term $S(n,k)$. ■

To illustrate (3.2.1), let us apply it to obtain an explicit formula for $S_1(n)$ and $S_2(n)$. Substituting in (3.2.1) with $k = 1$, we obtain

$$S(n,1)=\frac{1}{2}\left[(n+1)^2-1-\binom{2}{0}S(n,0)\right]=\frac{1}{2}(n^2+2n-n)=\frac{1}{2}n(n+1).$$

Substituting in (3.2.1) with $k = 2$, we obtain:

$$S(n,2) = \frac{1}{3}\left[(n+1)^3-1-\left(\binom{3}{0}S(n,0)+\binom{3}{1}S(n,1)\right)\right]$$

$$= \frac{1}{3}\left[(n+1)^3-1-\left(n+3n(n+1)/2\right)\right]$$

$$= \frac{1}{3}\left[n^3+3n^2/2+n/2\right] = \frac{1}{6}n(n+1)(2n+1).$$

The exact value of $S(n,k)$ is not so important, but rather its asymptotic growth rate as determined from the following proposition.

Proposition 3.2.2

$S(n,k) = 1^k + 2^k + \ldots + n^k$ is a polynomial in n of degree $k + 1$ with leading (highest degree) coefficient $1/(k + 1)$.

Proof The proof proceeds by induction (strong form, see Appendix A) on k.

Basis step: $S(n,0) = 1^0 + 2^0 + \ldots + n^0 = n$, a polynomial of degree $0 + 1$ with leading coefficient $1/(0+1)$.

Induction step: Given a positive integer k, assume $S(n,j)$ is a polynomial in n with leading coefficient $1/(j+1)$ for all positive integers $j < k$. We must show that $S(n,k)$ is a polynomial in n of degree $k+1$ with leading coefficient $1/(k+1)$. From (3.2.1),

$$S(n,k) = \frac{(n+1)^{k+1}}{k+1} + P_k(n), \tag{3.2.3}$$

where

$$P_k(n) = \frac{-1}{k+1}\left(1 + \binom{k+1}{0}S(n,0) + \binom{k+1}{1}S(n,1) + \cdots + \binom{k+1}{k+1}S(n,k-1)\right).$$

The strong form of our induction step implies that $P_k(n)$ is a polynomial in n of degree k. By the binomial theorem, $(n+1)^{k+1}$ is a polynomial in n of degree $k+1$ with leading coefficient 1. Proposition 3.2.2 follows from this latter observation and (3.2.3). ■

Various other properties of $S(n,k)$ that follow easily from (3.2.1) and induction are developed in the exercises. For example, based on (3.2.1) it is straightforward to give a recurrence relation for the coefficients of the polynomial representing $S(n,k)$. These coefficients are known as *Bernoulli numbers* $B(j,k)$, where $B(j,k)$ is the coefficient of n^j in the polynomial representing $S(n,k)$. For example, $B(0,k) = 0$ and $B(k,k+1) = 1/(k+1)$, for $k \geq 0$, $B(k,k) = ½$, $k \geq 1$, and $B(k-1,k) = k/12$ for $k \geq 2$.

Since $S(n,k) = 1^k + 2^k + \ldots + n^k$ is a polynomial of degree $k+1$ with (highest order) leading coefficient $1/(k+1)$, it follows from Proposition 3.1.6 that $S(n,k) \sim n^k/(k+1)$.

3.2.2 Sums of Logarithms

The sum $L(b,n) = \log_b 1 + \log_b 2 + \ldots + \log_b n = \log_b(n!)$ is another frequently occurring series in algorithm analysis. For example, we will see later in this chapter that $L(2,n)$ is a lower bound for the worst-case complexity of any comparison-based sorting algorithm.

Recall that for any real constants a and $b > 1$, $\log_a n \in \Theta(\log_b n)$, so that we may use *any* base when representing this Θ-class. For convenience, we denote this class simply by $\Theta(\log n)$. Similarly, it is clear that $L(a,n) \in \Theta(L(b,n))$, so that again we may use *any* base when representing this latter Θ-class. Thus, for convenience of notation, we drop the base and simply write

$$L(n) = \log(n!) = \log 1 + \log 2 + \cdots + \log n.$$

Proposition 3.2.3 $\log(n!) \in \Theta(n \log n)$.

Proof Clearly, $L(n) \in O(n \log n)$, since

$$\log 1 + \log 2 + \cdots + \log n \leq \log n + \log n + \cdots + \log n = n \log n.$$

It remains to show that $L(n) \in \Omega(n \log n)$. Let $m = \lfloor n/2 \rfloor$. We have

$$
\begin{aligned}
L(n) &= (\log 1 + \log 2 + \cdots + \log m) + [\log(m+1) + \log(m+2) + \cdots + \log n] \\
&\geq \log(m+1) + \log(m+2) + \cdots + \log n \\
&\geq \log(m+1) + \log(m+1) + \cdots + \log(m+1) \\
&= (n-m)\log(m+1) \\
&\geq (n/2)\log(n/2) = (n/2)(\log n - \log 2).
\end{aligned}
$$

Clearly, $(n/2)(\log n - \log 2) \geq (n/2)[\log n - (1/2)\log n]$ for all $n \geq n_0$, where n_0 is sufficiently large (the constant n_0 depends on the base chosen). Thus, $L(n) \geq (1/4)\,(n \log n)$, so that $L(n) \in \Omega(n \log n)$. ■

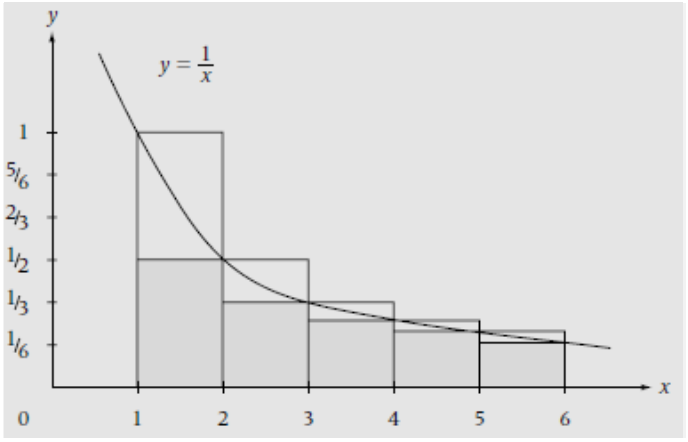
3.2.3 The Harmonic Series

Our third example, the harmonic series $H(n) = 1 + 1/2 + \ldots + 1/n$, also occurs frequently in algorithm analysis. For example, $H(n)$ occurs in the average analysis of *QuickSort* given later in this chapter.

Proposition 3.2.4 $H(n) \sim \ln n$. Thus, $H(n) \in \Theta(\log n)$.

Proof By definition, $\int_1^n 1/x\,dx = \ln n$, so that $\ln n$ is the area bounded above by the graph of $1/x$ and bounded below by the interval $1 \leq x \leq n$ on the x axis (see Figure 3.4). We see from Figure 3.4 that this area is larger than the sum of the areas of the rectangles R_i, $i = 1, \ldots, n-1$, where the base of R_i is the interval (of unit length) $i \leq x \leq i+1$, and the height of R_i equals $1/(i+1)$. Clearly, this area is smaller than the sum of the areas of the rectangles R_i*, $i = 1, \ldots, n-1$, where R_i* has the same base as R, but has height $1/i$. Thus,

$$H(n) - 1 \leq \int_1^n \frac{1}{x}dx \leq H(n) - \frac{1}{n}. \qquad (3.2.4)$$



$$H(n) - 1 \leq \int_1^n \frac{1}{x}\,dx \leq H(n) - \frac{1}{n}.$$

Figure 3.4

Proposition 3.2.4 follows after dividing each of the three terms in (3.2.4) by $H(n)$ and taking the limit as n tends to infinity. The two outside limits are 1, so the middle limit must exist and also be 1. Showing that the outside limits are 1 amounts to showing that $\lim_{n\to\infty} H(n) = \infty$ (exercise).

3.3 Recurrence Relations for Complexity

A recurrence relation typically expresses the value of a function at an input n in terms of the value of the function at a smaller value or values of the input to the function. For example, when analyzing the performance of an algorithm based on a recursive strategy (whether or not the algorithm is expressed explicitly as a recursive algorithm), we often immediately can give a recurrence relation for its complexity for inputs of size n, since it works by repeating the algorithm's operations on an input or inputs of smaller size. Recurrence relations are so natural in determining the complexity of algorithms based on a recursive strategy that we state this as a key fact.

Key Fact

When analyzing the worst-case complexity $W(n)$ of a algorithm based on a recursive strategy, a recurrence relation should immediately emerge, since worst case will usually require that the algorithm performs the repeated operations with smaller input size(s) that exhibit worst-case performance. Similar comments hold for the best case $B(n)$ and the average complexity $A(n)$, although finding (and solving) a recurrence for $A(n)$ is usually more challenging.

As our first illustration of the above key fact, we consider binary search. For an input list generating worst-case behavior, after comparing to the midpoint element, binary search always looks for the search element in the larger sublist, which has size $\lceil n/2 \rceil$. Hence, we obtain the following recurrence for $W(n)$

$$W(n) = W(\lceil n/2 \rceil) + 1, \quad \textbf{init.cond.}\ W(1) = 1 \tag{3.3.1}$$

The occurrence of the ceiling function $\lceil\ \rceil$ in (3.3.1) makes it cumbersome to solve. However, if we assume that $n = 2^k$, the relation becomes:

$$W(n) = W(n/2) + 1, \quad \textbf{init.cond.}\ W(1) = 1. \tag{3.3.2}$$

The recurrence (3.3.2) can be solved by the method of *repeated substitution* (also called *unwinding*). For example, as a first step in solving (3.3.2), note that if we simply substitute $n/2$ for n in (3.3.2), we would get $W(n/2) = W((n/2)/2) + 1$, and hence $W((n/2)/2) + 1$ can be used to replace $W(n/2)$ in (3.3.2). Repeating this substitution process we obtain

$$
\begin{aligned}
W(n) &= W\left(\frac{n}{2}\right)+1 \\
&= \left(W\left(\frac{(n/2)}{2}\right)+1\right)+1 = W\left(\frac{n}{2^2}\right)+2 \\
&= \left(W\left(\frac{(n/2^2)}{2}\right)+1\right)+2 = W\left(\frac{n}{2^3}\right)+3 \\
&\vdots \\
&= \left(W\left(\frac{(n/2^{k-1})}{2}\right)+1\right)+k-1 = W\left(\frac{n}{2^k}\right)+k = W(1)+k = 1+k
\end{aligned}
$$

Since $k = \log_2 n$, we have

$$W(n) = 1+\log_2 n, \quad n = 2^k. \tag{3.3.3}$$

We now show how formula (3.3.3) can be used to obtain an approximation for $W(n)$, for general n, which is accurate to within a single comparison. Given any positive integer n, there exists a positive integer j such that:

$$2^{j-1} \le n < 2^j. \tag{3.3.4}$$

Clearly, $W(m) \le W(n)$ for $m \le n$. Thus,

$$W(2^{j-1}) \le W(n) \le W(2^j). \tag{3.3.5}$$

By (3.3.3), $W(2^{j-1}) = j$ and $W(2^j) = j + 1$. Substituting these results into (3.3.5) yields:

$$j \le W(n) \le j+1. \tag{3.3.6}$$

By taking base-2 logarithms in (3.3.4), we obtain:

$$j-1 \le \log_2 n < j. \tag{3.3.7}$$

Combining (3.3.6) and (3.3.7) yields:

$$\log_2 n < \mathrm{W}(n) \le 2+\log_2 n, \quad n \ge 1. \tag{3.3.8}$$

Since $W(n)$ is an integer, (3.3.8) implies:

$$1+\lfloor \log_2 n \rfloor \le W(n) \le 2+\lfloor \log_2 n \rfloor, \quad n \ge 1. \tag{3.3.9}$$

It follows from (3.3.9) that $W(n) \in \Theta(\log n)$.

For many of the algorithms that we will analyze using recurrence relations, we will make the simplifying assumption that n is a power of some base b. For most functions that we will encounter, the asymptotic behavior of the function is determined by its behavior at the powers of the base b. In other words, we can "interpolate" the asymptotic behavior in between powers of b. We give a general condition for functions for which this interpolation is possible in Appendix D.

As a second example, we now consider the worst-case complexity of *MergeSort*. Again, we assume that $n = 2^k$ for some nonnegative integer k. Since the worst-case complexity of *Merge* is $n - 1$, $W(n)$ satisfies the following recurrence relation.

$$W(n) = 2W\left(\frac{n}{2}\right) + n - 1, \quad W(1) = 0 \tag{3.3.10}$$

Unwinding recurrence relation (3.3.10), we obtain

$$\begin{aligned} \mathrm{W}(n) &= 2\left(2W\left(\frac{n}{2^2}\right) + \frac{n}{2} - 1\right) + n - 1 = 2^2 W\left(\frac{n}{2^2}\right) + 2n - (1+2) \\ &= 2^2\left(2W\left(\frac{n}{2^3}\right) + \frac{n}{2^2} - 1\right) + 2n - (1+2) = 2^3 W\left(\frac{n}{2^3}\right) + 3n - (1+2+2^2) \\ &\vdots \\ &= 2^k W(1) + kn - (1 + 2 + 2^2 + \cdots + 2^{k-1}) = kn - (2^k - 1). \end{aligned}$$

Since $k = \log_2 n$, we have

$$\mathrm{W}(n) = n\log_2 n - n + 1, \text{ for } n = 2^k,\ k = 0,1,2, \ldots. \tag{3.3.11}$$

Now that we have established a formula for $W(n)$ when n is a power of two, similar to the situation for binary search we can interpolate the asymptotic behavior of $W(n)$ for all n, and conclude that $W(n) \in \Theta(n \log n)$.

Note that the recurrence relations for binary search and *MergeSort* related $W(n)$ to $W(n/2)$. More generally, whenever the recursive strategy for an algorithm repeats the algorithm's operations on an input size that is a fixed fraction n/b of the original input size n, such as in Divide-And-Conquer strategies, then the recurrence relations for $W(n)$ will be in terms of $W(n/b)$. In Appendix C we consider the general solution to a recurrence relations that relate the nth term to the (n/b)th term.

Another type of recurrence relation that often arises when analyzing algorithms relates $W(n)$ to $W(n - 1)$. For example, consider the algorithm *InsertionSort* discussed in the previous chapter. Note that for an input list $L[0{:}n - 1]$, the action of *InsertionSort* can be viewed as first executing *InsertionSort* with input list $L[0{:}n - 2]$, followed by inserting $L[n - 1]$ into the previously sorted list $L[0{:}n - 2]$. This observation immediately yields the following recurrence relation for $W(n)$.

$$W(n) = W(n - 1) + n - 1 \qquad \textbf{init. cond. } W(1) = 0. \tag{3.3.18}$$

An explicit formula for $W(n)$ is easily obtained by repeated substitution into (3.3.18). We have

$$
\begin{aligned}
W(n) &= W(n-1) + n - 1 \\
&= (W(n-2) + n - 2) + n - 1 \\
&= (W(n-3) + n - 3) + n - 2 + n - 1 \\
&\ \ \vdots \\
&= W(1) + 1 + 2 + \cdots + n - 1 \\
&= n^2/2 - n/2.
\end{aligned}
$$

Thus, $W(n) \in \Theta(n^2)$.

As another illustration a recurrence relating the nth term to the $(n-1)$st term, consider the number $t(n)$ of moves performed by *Towers* for n disks (see Appendix B). It is easily verified that $t(n)$ satisfies the recurrence relation:

$$t(n) = 2t(n-1) + 1, \quad \textbf{init cond.}\ t(1) = 1. \tag{3.3.19}$$

By repeated substitution in (3.3.19) we have:

$$
\begin{aligned}
t(n) &= 2t(n-1) + 1 \\
&= 2(2t(n-2) + 1) + 1 = 2^2(t(n-2)) + 1 + 2 \\
&= 2^2(2t(n-3) + 1) + 1 + 2 = 2^3(t(n-3)) + 1 + 2 + 2^2 \\
&\ \ \cdots \\
&= 2^{n-1}t(1) + 1 + 2 + 2 + \ldots + 2^{n-2} \\
&= 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.
\end{aligned}
$$

Thus, the Tower of Hanoi puzzle requires an exponential number of steps to solve. Using mathematical induction (see Exercise 3.51), it is easily proved that *Towers* solves the Towers of Hanoi puzzle using the fewest possible moves. In Appendix C we consider the general solution to a recurrence relations that relate the nth term to the $(n-1)$st term.

3.4 Mathematical Induction and Proving Correctness of Algorithms

In addition to the need to analyze the efficiency of an algorithm, it is even more basic to determine whether the algorithm is correct. Proving algorithms are correct is most often done with the aid of mathematical induction (see Appendix A). The induction usually utilizes one or both of the following two techniques,

1. induction on the input size of the algorithm,

2. induction to establish one or more loop invariants, which typically are stated as assertions concerning the value of a variable after each iteration of a given loop, and whose final value helps establish the correctness of the algorithm.

Induction on the input size is particularly relevant for algorithms based on a recursive strategy, since the recursive execution of the operations usually involves smaller inputs where the induction hypothesis applies. Moreover, it is usually the strong form of induction that is required since the recursion involves inputs whose size can be smaller than one less than the size

of the original input. For the purposes of induction, it is important to know what should be considered as the input size. For example, given an algorithm like *MergeSort* or *QuickSort* for sorting a list $L[0{:}n-1]$, in order to implement the recursion the additional input parameters *low* and *high* were included. In particular, the correct output for these algorithms is a sorting of $L[low{:}high]$ (so that $L[0{:}n-1]$ is sorted when called originally with $low = 0$ and $high = n-1$). Hence, the input size is $k = high - low + 1$.

Remark

A common mistake in proving correctness of recursive algorithms involving a list $L[0{:}n-1]$ is to attempt induction on n, when, in fact, n remains constant throughout the action of the algorithm (and thus is not the natural integer upon which to do the induction). The recursive algorithm (as in *BinarySearch*, *MergeSort*, *Quicksort*, etc.) usually involves input parameters with names like *low* and *high* determining a sublist $L[low, high]$ of size $high - low + 1$, whose size is diminished with the recursive calls, and so is amenable to induction.

Proof of Correctness of *BinarySearch*.

We prove correctness using strong induction the size $k = high - low + 1$ of the sublist $L[low, high]$.

Basis step: $k \leq 1$. If $k = 0$, then $low > high$ and the sublist $L[low, high]$ is an empty list. *BinarySearch* then correctly returns −1. If $k = 1$, then $low = mid = high$. If $X = L[mid]$, then *BinarySearch* correctly returns *mid*. Otherwise, *BinarySearch* is called with $low > high$, so that it correctly returns −1.

Induction step: Assume *BinarySearch* is correct for all lists of size $high - low + 1 < k$, and consider the case when $high - low + 1 = k$ If $X = L[mid]$, then *BinarySearch* correctly returns *mid*. If $X < L[mid]$, then since $L[low, high]$ is a nondecreasing list, if X is in the sublist $L[low, high]$, it must be in the sublist $L[low, mid-1]$. Since *BinarySearch* makes a recursive call with the sublist $L[low, mid-1]$ of size less than k, it follows by induction that *BinarySearch* correctly returns an index of an element where X occurs in $L[low, mid-1]$ or returns -1 if X is not in this sublist (and hence also not in the original list $L[low, high]$). A similar argument shows that *BinarySearch* is correct when $X > L[mid]$. ■

Proof of Correctness of *MergeSort* if *Merge* is Correct

We prove correctness using strong induction the size $k = high - low + 1$ of the sublist $L[low, high]$.

Basis step: $k \leq 1$. *MergeSort* is clearly correct in these cases of a one-element sublist or an empty sublist, respectively, since it simply returns without any further action.

Induction step: Given an integer k such that $1 < k \leq n$, *assume* that *MergeSort* is correct for all sublists with $high - low + 1 < k$, and consider a sublist with $high - low + 1 = k$. Then *mid* is assigned the value $\lfloor (low + high)/2 \rfloor$, and *MergeSort* makes two recursive calls, with sublists

$L[low: mid - 1]$ and $L[mid + 1: high]$. Both of these sublists have smaller size than k, so by our induction hypothesis, *MergeSort* sorts these sublists in increasing order. Hence, *MergeSort* is correct if *Merge* is correct. ■

The proof that *Merge* is correct uses loop invariants, and will be developed in the exercises. We now illustrate loop invariants by proving *HornerEval* is correct.

Proof of Correctness of *HornerEval.*

We establish the correctness of *HornerEval* by using induction to show that the following loop invariant condition holds after the kth pass of the loop in *HornerEval*, $k = 0, 1, \ldots, n$:

$$Sum \text{ has the value } a[n]*v^k + a[n-1]*v^{k-1} + \cdots + a[n-k+1]*v + a[n-k].$$

Basis step: Before the loop is entered, *Sum* is assigned the value $a[n]$, so that the claim is true for $k = 0$.

Induction step: Assuming that the claim is true for $k < n$, we show that it is also true for $k + 1$. By assumption, the value of *Sum* after the k^{th} pass is

$$a[n]*v^k + a[n-1]*v^{k-1} + \cdots + a[n-k+1]*v + a[n-k].$$

Now the $(k + 1)^{st}$ pass corresponds to the loop variable i having the value $n - k - 1$. Thus, after this pass *Sum* has the value

$$(a[n]*v^k + a[n-1]*v^{k-1} + \cdots + a[n-k+1]*v + a[n-k])*v + a[n-k-1]$$

$$= a[n]*v^{k+1} + a[n-1]*v^k + \cdots + a[n-k+1]*v^2 + a[n-k]*v + a[n-k-1].$$

Thus, our claim is true for $k + 1$. By induction, our claim is therefore proved for all $k = 0, 1, \ldots, n$. The correctness of *HornerEval* now follows from our claim, since the algorithm terminates after the n^{th} pass with the correct evaluation of the polynomial at v.

Our next illustration is proofs of the correctness of the recursive algorithm *Powers*. *Powers* is proven correct by (the strong form of) induction on the exponent n (as we have mentioned, the input size to *Powers* is really $\log_2 n$, but this will not matter in this context).

Proof of Correctness of *Powers*.

Basis step: $n = 1$. When $n = 1$, clearly *Powers* correctly returns x.

Induction step: We assume that *Powers* is correct for all exponents $1 \leq k < n$, and consider an input with $k = n$. Assume first that n is even. Then *Powers* is recursively invoked with input parameters $x*x$ and $n/2$, which, by induction assumption, results in the return of $(x*x)^{n/2} = x^n$. This result is then returned by *Powers*, so that the algorithm is correct when n is even. If n is odd, then *Powers* is recursively invoked with input parameters $x*x$ and $(n - 2)/2$, which, by induction assumption, results in the return of $(x*x)^{(n-1)/2} = x^{n-1}$. But then *Powers* returns $x*x^{n-1} = x^n$, so that the algorithm is correct in this case also. ■

Proof of Correctness of *Left-to-Right Binary Method*.

To prove correctness of *Left-to-Right Binary Method*, we use the strong form of induction on n.

Basis step: $n = 1$. When $n = 1$, clearly *Left-to-Right Binary Method* correctly returns x.

Induction step: Assume first that n is even. Then the binary expansion of n is obtained from the binary expansion of $n/2$ by adding a zero in the rightmost (least significant) position. So, by the strong form of induction, the algorithm correctly computes $x^{n/2}$. But the algorithm for input n proceeds exactly the same as with input $n/2$, except at the last stage returns the square of $x^{n/2}$, correctly returning x^n. Now assume that n is odd. Then the binary expansion of n is obtained from the binary expansion of $(n-1)/2$ by adding a one in the rightmost (least significant) position. So, by the strong form of induction, the algorithm correctly computes $x^{(n-1)/2}$. But the algorithm for input n proceeds exactly the same as with input $(n-1)/2$, except at the last stage returns x times the square of $x^{(n-1)/2}$, thereby correctly returning x^n. ■

Remark

Of course, the correctness of *Left-to-Right Binary Method* assumes that the binary expansion of n is correctly computed. We leave computation and correctness proof of an algorithm computing the binary expansion of n to the exercises. This remark also holds for the *Right-to-Left Binary Method.*

Proof of Correctness of *Right-to-Left Binary Method*

To prove the correctness of *Right-to-Left Binary Method*, we use the loop invariants determined by the values of *Pow* and *AccumPowers*. We claim that after processing the ith position in the binary presentation of n (the ith position corresponds to 2^{i-1}), $Pow = n^{2^1}$ and the value of *AccumPowers* is the product of all terms x^{2^j} as j runs from 0 to $i-1$ and $b_j \neq 0$. Our proof uses induction on the index position of the binary representation of n.

Basis step: $i = 1$. The value of $Pow = x^2$ and the value of *AccumPowers* is 1 if $b_0 = 0$, and x if $b_0 = 1$. Thus the basis step is verified.

Induction step: Assume that after we complete the processing at the $(i-1)$st position $(i < m)$, $Pow = n^{2^{i-1}}$ and the value of *AccumPowers* is the product of all terms x^{2^j} as j runs from 0 to $i-2$ and $b_j \neq 0$. At the ith position, if $b_{i-1} = 0$ then *AccumPowers* is not changed, whereas if $b_i = 1$ then we multiply *AccumPowers* by $n^{2^{i-1}}$, so that *AccumPowers* has the appropriate value in either case. After *AccumPowers* is updated, then *Pow* is squared, yielding the value n^{2^1}. Thus, *Pow* and *AssumPowers* have the correct values at the ith position. Letting $i = m$ completes the verification of correctness of *Right-to-Left Binary Method*.

Remarks

The *Right-to-Left Binary Method* models the way the recursive algorithm *Powers* computes x^n. It is an interesting exercise to verify this by tracing the stack of recursive calls that result in invoking *Powers* on some sample examples. In Exercise 3.x we give an iterative algorithm *Powers2* that is directly based on modeling this stack, and eliminating the stack altogether as can be done in general whenever we are dealing with *tail recursion* (see Appendix C) as is the case with *Powers*.

3.5 Establishing Lower Bounds for Problems

Once we have found one or more algorithms solving a given problem, it is important to know whether these algorithms are the best possible. Lower bound theory aids us in answering this question. The purpose of lower bound theory is to obtain (as sharp as possible) lower bounds for the complexities of *any* algorithm that solves the problem. An algorithm whose complexity equals a lower bound that has been established is an optimal algorithm for the problem and the lower bound is sharp. Such a sharp lower bound is the *complexity of the problem.* For example, we don't need to look for a better algorithm for polynomial evaluation than *HornerEval*, since any correct algorithm for evaluating an *n*th-degree polynomial must perform at least *n* multiplications.

In this section we introduce lower bound theory by determining sharp lower bounds for a comparison-based searching algorithm to find the maximum value in a list of size *n* and for sorting a list of size *n* using an adjacent-key comparison-based sorting algorithm. A lower bound of $n - 1$ for the best-case, worst-case, and average complexities of any comparison-based algorithm for finding the maximum is established using a simple counting technique. Lower bounds of $n(n - 1)/2$ for the worst-case complexity and $n(n - 1)/4$ for the average complexity of adjacent-key comparison-based sorting are obtained using simple properties of permutations. In the next section, we also establish a $\Omega(n \log n)$ lower bound for the worst-case complexity of *any* comparison-based sorting algorithm.

3.5.1 Lower Bound for Finding the Maximum

Consider the problem of using a comparison-based algorithm to find the maximum value of an element in a list $L[0{:}n - 1]$ of size *n*. Clearly, each list element must participate in at least one comparison, so that $\lceil n/2 \rceil$ is trivially a lower bound for the problem. We now use a simple counting argument 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. The algorithm *Max* given in Section 3.3 has best-case, worst-case, and average complexities all equal $n - 1$, so that *Max* is an optimal algorithm.

Suppose for any convenience that we restrict attention to lists that contain *distinct* elements. When a comparison-based algorithm compares two distinct elements *X*, *Y*, we say that *X wins* the comparison if $X > Y$, otherwise *X loses*. Thus, each comparison generates exactly one loss, and this is the unit of work that we associate with the comparison. Now we simply count the number of total losses that must occur if the algorithm is to correctly determine the maximum element. We claim that any comparison-based algorithm must generate at least $n - 1$ units of work. More precisely, we claim that each of the $n - 1$ elements that it *not* the maximum must lose at least one

comparison. Indeed, suppose there is an input list $L[0{:}n-1]$ that causes the algorithm to terminate with two list elements $L[i]$ and $L[j]$ *both* never having lost a comparison. Assume for definiteness that the algorithm declares $L[i]$ to be the maximum. Now consider the input list $L'[0{:}n-1]$ that is identical to L except that $L'[j]$ is increased so that it is larger than $L[i]$ and remains different from any other list element. By the definition of comparison-based algorithms, our algorithm performs *identically* on L and L'. To see this, note that the algorithm could only perform differently when making comparisons involving $L'[j]$. But $L[j]$ won all its comparisons, and since $L'[j] \geq L[j]$, so does $L'[j]$. Thus, the algorithm must again declare $L'[i] = L[i]$ to be the maximum element in L', which is a contradiction.

Thus, we have established that any comparison-based algorithm for finding the maximum element in a list of size n (of distinct elements) must perform at least $n - 1$ comparisons. Since *Max* performs $n - 1$ comparisons for *any* input list of size n, it is an optimal algorithm for the problem, as summarized in the following proposition.

Proposition 3.5.1

The problem of finding the maximum element in a list of size n using a comparison-based algorithm has worst-case, best-case, and average complexities all equal to $n - 1$. Moreover, *Max* is an optimal algorithm for the problem.

3.5.2 Lower Bounds for Adjacent-Key Comparison Sorting

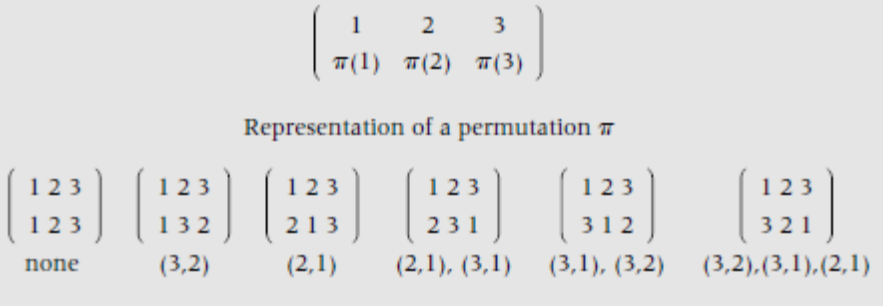
We now consider the problem of sorting a list using an adjacent-key comparison-based algorithm. As discussed earlier, the sorting algorithm *InsertionSort* can be viewed as an adjacent-key comparison sort. Recall that *InsertionSort* has worst-case complexity $W(n) = n(n-1)/2$. In this section we show that *any* adjacent-key comparison sort performs at least $n(n-1)/2$ comparisons to sort a list $L[0{:}n-1]$ of size n in the worst case. Hence, *InsertionSort* achieves optimal worst-case complexity for an adjacent-key comparison-based sort.

We may assume without loss of generality that the input lists of size n to the algorithm are chosen from the set of all permutations of the set of integers $\{1, 2, \ldots, n\}$. We associate with each permutation a combinatorial entity called an *inversion* of the permutation and show that the maximum number of inversions in a permutation on n symbols is given by $n(n-1)/2$. Since interchanging adjacent list elements (keys) removes at most one inversion, the maximum number of inversions then serves as a lower bound for the problem of adjacent-key comparison-based sorting.

Proposition 3.5.2

A lower bound for the worst-case complexity $W(n)$ of any adjacent-key comparison-based sorting algorithm is $n(n-1)/2$.

Proof Given a permutation π of $\{1,2, \ldots, n\}$, an *inversion* of π is a pair $(\pi(a), \pi(b))$, $a, b \in \{1,2, \ldots, n\}$, where $a < b$ but $\pi(a) > \pi(b)$. In Figure 3.5 we illustrate the inversions corresponding to the six permutations of $\{1,2,3\}$.



All permutations of {1,2,3} and their associated inversions

Figure 3.5

Note that any sorting algorithm must ultimately remove all inversions. Note also that a comparison of list elements utilized by any adjacent-key comparison-based sorting algorithm by definition considers only inversions of the form$(\pi(i), \pi(i + 1))$. Removing any such inversion (by interchanging $\pi(i)$ and $\pi(i + 1)$) results in a decrease of exactly one inversion (see Exercise 3.52). Thus, the worst-case complexity of any adjacent-key comparison-based sorting algorithm is bounded below by the maximum number of inversions in a permutation of $\{1,2, \ldots, n\}$.

Consider the permutation corresponding to a list sorted in decreasing order—that is, the permutation $\pi(i) = n - i + 1$, $i = 1, \ldots, n$. Clearly, every pair $(\pi(i), \pi(j))$, $1 \leq i < j \leq n$ is an inversion of π. The number of such pairs is given by the binomial coefficient,

$$\binom{n}{2} = n(n-1)/2$$

thereby establishing the lower bound given in Proposition 3.5.2. ■

Since *InsertionSort* performs $n(n - 1)/2$ comparisons in the worst case, it follows from Proposition 3.5.2 that *InsertionSort* is an (exactly) optimal worst-case adjacent-key comparison-based sorting algorithm.

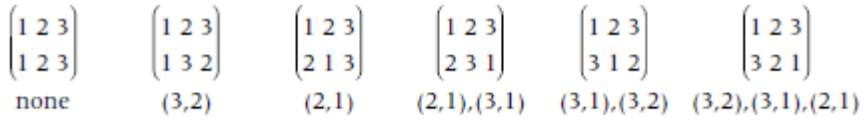
We now consider a lower bound for the average behavior of adjacent-key comparison-based sorting algorithms. The following proposition gives us a lower bound that is only half of that found for the worst case.

Proposition 3.5.3

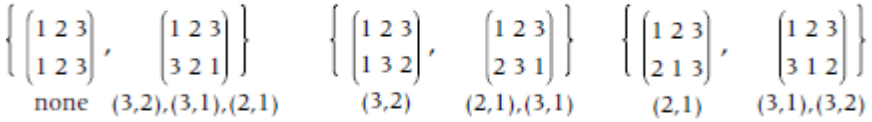
A lower bound for the average complexity $A(n)$ of any adjacent-key comparison-based sorting algorithm is $n(n - 1)/4$.

Proof For any input permutation π, any sorting algorithm must ultimately remove all the inversions in π, so that the average complexity of any adjacent-key comparison sorting algorithm is bounded below by the average number of inversions $\iota(n)$ in a permutation of $\{1,2, \ldots, n\}$. We now give a quick proof of the formula $\iota(n) = n(n - 1)/4$, from which Proposition 3.5.3 follows. Given any permutation π of $\{1,2, \ldots, n\}$, its *reverse* permutation π_{rev} is defined by

$\pi_{rev}(i) = \pi(n - i + 1)$, $i \in \{1,2, \ldots, n\}$. Now note that given any $x, y \in \{1,2, \ldots, n\}$., where $x > y$, the pair (x,y) occurs as an inversion in *exactly one* of the permutations π,π_{rev} (see Figure 3.6b). Thus, the permutations π,π_{rev} have a total of exactly $n(n-1)/2$ inversions between them. We can sum the number of inversions over all permutations of $\{1,2, \ldots, n\}$ by summing over each (unordered) pair $\{\pi,\pi_{rev}\}$, thereby obtaining $(n!/2)(n(n-1)/2)$ inversions altogether. The average is then obtained by dividing this latter number by $n!$, yielding $\iota(n) = n(n-1)/4$. ■



(a) All permutations of {1,2,3} and their inversions



(b) Partition of the permutations of {1,2,3} into pairs $\{\pi,\pi_{rev}\}$, with each pair $\{\pi,\pi_{rev}\}$ yielding a full set of inversions

Permutations of {1,2,3}

Figure 3.6

In the next section we will show that the average complexity $A(n)$ of *InsertionSort* is given by $A(n) = n^2/4 + 3n/4 - H(n)$, where $H(n)$ is the harmonic series $1 + 2 + \ldots + 1/n$. Hence, $A(n)$ is asymptotically very close to $n^2/4$. Thus, by Proposition 3.5.3, *InsertionSort* has basically optimal average complexity.

3.5.3 Lower Bound for Comparison-Based Sorting in General

In the previous section we proved that adjacent key comparison-based sorting algorithms must perform at least $n(n-1)/2$ comparisons in the worst case for inputs of size n, and $n(n-1)/4$ comparisons on average. Similar arguments show that for any *fixed* (independent of n) positive integer k, if a comparison-based algorithm always makes comparisons between list elements that occupy positions no more than k indices apart, then the worst case and average performance remains quadratic in the input size n (see Exercise 3.53). Thus, to improve on quadratic performance, we need to compare list elements that are far apart (that is, the difference between index positions of some of the compared elements must *grow* as a function of the input size). Of course, it is easy to design a comparison-based algorithm that does compare far apart elements, but whose performance is still quadratic. What we need is a clever way to use comparison based sorting such as *MergeSort*. But the question arises as to how good we can do in general with comparison-based sorting. The answer for the worst-case complexity is found in the following proposition.

Proposition 3.5.4

A lower bound for the worst-case complexity $W(n)$ of any comparison-based sorting algorithm is $\log_2 n! \in \Omega(n \log n)$.

Proof Given any comparison-based algorithm having worst-case complexity $m = W(n)$, we define a sequence of m partitions $C_1, \ldots, C_m$ of the set of all permutations of $\{1, \ldots, n\}$ as follows. C_1 is a bipartition of the set of permutations of $\{1, \ldots, n\}$ into two sets S_0 and S_1, where a permutation π is placed in S_0 if, when input to algorithm, the elements that are compared in the first comparison are in order and placed in S_1 if they are out of order(determine an inversion). Inductively, assuming $C_1, \ldots, C_k$ have been defined, we define C_{k+1} to be a refinement of C_k as follows. Consider any set S in C_k. We bipartition S into two sets X and Y by placing a permutation $\pi \in S$ in Y if the algorithm performs at least $k + 1$ comparisons with π as input and the elements that are compared in the $(k + 1)^{st}$ comparison are out of order, and letting X be the set of remaining permutations in S. If any of the resulting sets of C_{k+1} are empty, we simply remove them.

After m comparisons the algorithm has terminated for each input permutation. We now show that each set in C_m contains exactly one permutation. Suppose, to the contrary that some set S of C_m contains two permutations π_1 and π_2. Then, since the algorithm is comparison-based, it can easily be shown by induction that, for each comparison performed by the algorithm, precisely the same list positions were compared with π_1 as input versus π_2 and the relative order of these elements was the same. It follows that π_1 and π_2 must be the same permutation (a contradiction). Thus, since there are $n!$ permutations, by the pigeon hole principle, $|C_m| \geq n!$. Clearly, $|C_m| \leq 2^m$ so that we have

$$2^m \geq |C_m| \geq n! \Rightarrow m \geq \log_2 n! \in \Omega(n \log n).$$ ■

MergeSort is an example of a sorting algorithm whose worst-case performance is $O(n \log n)$, so that it exhibits order-optimal behavior in the worst case. *MergeSort* is also order-optimal on average, since it turns out that $\Omega(n \log n)$ is also a lower bound for the average behavior of comparison-based sorting. Since *QuickSort* has quadratic worst-case complexity, it is not order-optimal in the worst case. However, it is often used since its average complexity belongs to $O(n \log n)$, so it has order-optimal average complexity, and performs well in practice. We now discuss average behavior of algorithms more formally. We assume that the reader is familiar with the discussion of probability that is contained in Appendix E.

3.6 Probabilistic Analysis of Algorithms

In practice, a given algorithm is usually run repeatedly with varying inputs of size n. Thus, another important measure of the performance of an algorithm is its *average complexity* $A(n)$. Let τ: $\mathcal{I}_n \rightarrow N$ denote the mapping (random variable) that sends an input I in $\mathcal{I}_n$ to the number of basic operations performed by the algorithm on input I. The average complexity $A(n)$ of the algorithm is defined as the *expected* number $E[\tau]$ of basic operations performed, *and depends on*

the probability distribution imposed on the sample space I_n (see Appendix E for definitions of the terms used in this section). For now, we assume that I_n is a finite set and that each input $I \in \mathcal{I}_n$ has probability $p(I)$ of occurring as the input to the algorithm. An important special case is when each input is equally likely (the *uniform distribution*), in which case $p(I) = 1/|\mathcal{I}_n|$, and average complexity is then the familiar

$$A(n) = \left(\sum_{I \in \mathcal{I}_n} \tau(I) \right) / |I_n| = E[\tau]$$

Since our interest is in algorithm analysis, throughout this chapter we specialize the formulas given in Appendix E for the expectation of a random variable X defined on a sample space S to the case where $X = \tau$ and $S = \mathcal{I}_n$. The general definition of $A(n)$ follows.

Definition 3.6.1 Average Complexity

Given a probability function p: $\mathcal{I}_n \rightarrow [0,1]$ defined on the finite input set $\mathcal{I}_n$, the *average complexity* of an algorithm is defined as:

$$A(n) = \sum_{I \in \mathcal{I}_n} \tau(I)\, p(I) = E[\tau] \qquad (3.6.1)$$

Formula (3.6.1) for $A(n)$ is rarely used directly, since it is simply too cumbersome to examine each term in I_n directly. Also, the growth of the summation as a function of the input size n is usually hard to estimate, much less calculate exactly. Thus, when analyzing the average complexity of a given algorithm, we usually seek closed-form expressions or estimates for $A(n)$, or formulas that allow some gathering of terms in (3.6.1). For example, let p_i denote probability that the algorithm performs exactly i basic operations; that is $p_i = p(\tau = i)$. Then:

$$A(n) = E[\tau] = \sum_{i=1}^{W(n)} i p_i, \qquad (3.6.2)$$

Formula (3.6.2) is often useful in practice, and follows from (3.6.1) by simply gathering up, for each i between 1 and $W(n)$, all the inputs I such that $p(I) = i$. We illustrate the use of (3.6.2) by computing the average complexity of *LinearSearch*.

3.6.1 Average Complexity of *LinearSearch*

To simplify the discussion of the average behavior of *LinearSearch*, we assume that the search element X is in the list $L[0{:}n-1]$ and is equally likely to be found in any of the n positions. Note that i comparisons are performed when X is found at position i in the list. Thus, the probability that *LinearSearch* performs i comparisons is given by $p_i = 1/n$. Substituting these probabilities into (3.6.2) yields:

$$A(n) = \sum_{i=1}^{W(n)} ip_i = \sum_{i=1}^{n} i\frac{1}{n} = \left(\frac{n(n+1)}{2}\right)\frac{1}{n} = \frac{n+1}{2}. \qquad (3.6.3)$$

Formula (3.6.3) is intuitively correct, since under our assumptions X is equally likely to be found in either half of the list. To see that (3.6.3) truly reflects average behavior, suppose that we run *LinearSearch* m times (m large) with a fixed list $L[0{:}n-1]$ and with search element X being randomly chosen as one of the list elements. Let m_i denote the number of runs in which X was found at position i. Then the total number of comparisons performed over the m runs is given by $1m_1 + 2m_2 + \ldots + nm_n$. Dividing this expression by m gives the average number $A_m(n)$ of comparisons over the entire m runs, as follows:

$$A_m(n) = 1\left(\frac{m_1}{m}\right) + 2\left(\frac{m_2}{m}\right) + \cdots + n\left(\frac{m_n}{m}\right). \qquad (3.3.6)$$

Note that for large m, the ratios m_i/m occurring in (3.3.6) approach the probability p_i that X occurs in the ith position. Since we have assumed that X is equally likely to be found in any of the n positions, each m_i is approximately equal to m/n. Hence, substituting $m_i = m/n$ into (3.3.6) yields:

$$A_m(n) \approx \frac{1+2+\cdots+n}{n} = \frac{n+1}{2} = A(n) \qquad (3.6.5)$$

To summarize, *LinearSearch* has best-case, worst-case, and average complexities 1, n, and $(n + 1)/2$, respectively. The best-case complexity is a constant independent of the input size n, whereas the worst-case and average complexities are both linear functions of n. For simplicity, we say that *LinearSearch* has *constant* best-case complexity and *linear* worst-case and average complexities.

We calculated $A(n)$ for *LinearSearch* under the assumption that the search element X is in the list. In the exercises we examine a more general situation where we assume that X is in the list with probability p, $0 \le p \le 1$.

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

3.6.2 Two Major Techniques for Computing Average Behavior

There are two techniques that are particularly useful in computing $A(n)$, namely,

1. partitioning the algorithm into disjoint stages,

or,

2. partitioning the input space into disjoint subsets.

Technique (1) leads to the application of the fundamental additive property of expectation.

Additivity of Expectation. Suppose the random variable $\tau = \tau_1 + \tau_2 + \ldots + \tau_m$. Then:

$$A(n) = E[\tau] = E[\tau_1] + E[\tau_2] + \ldots + E[\tau_m]. \qquad (3.6.6)$$

Technique (2) leads to the application of the additivity of expectation together with the notion of conditional expectation.

Partitioning $\mathscr{I}_n$ and Conditional Expectation. Suppose the $\mathscr{I}_n$ is partitioned into disjoint sets $S_1, \ldots, S_m$. This partition is also equivalently determined (see Appendix E) by defining a second random variable Y: $\mathscr{I}_n \rightarrow [1,m]$ where $Y(i) = i$ if I belongs to S_i, so that $S_i = Y^{-1}(i)$, $i = 1, \ldots, m$. Then if $E[\tau|Y = i]$ denotes the conditional expectation of τ given that $Y = i$, we have

$$A(n) = E[\tau] = E[\tau|Y = 1]P(Y = 1) + \ldots + E[\tau|Y = m]P(Y = m). \qquad (3.6.7)$$

We illustrate the use of (3.6.6) by computing $A(n)$ for *InsertionSort*, and the use of (3.6.7) by computing $A(n)$ for *QuickSort*. The analysis of both *InsertionSort* and *Quicksort* involve the harmonic series $H(n) = 1 + 1/2 + \ldots + 1/n$.

3.6.3 Average Complexity of *InsertionSort*

Since *InsertionSort* is a comparison-based algorithm, we can assume without loss of generality that inputs to *InsertionSort* are permutations of $\{1,2, \ldots, n\}$. We also assume that each permutation is equally likely to be the input to *InsertionSort*. Unlike our analysis of *LinearSearch*, we cannot compute $A(n) = E[\tau]$ directly by applying formula (3.6.2). Instead, we partition the algorithm *InsertionSort* into $n - 1$ stages. The i^{th} stage consists of inserting the $(i + 1)$st element $L[i]$ into its proper position in the sublist $L[0:i - 1]$, where the latter sublist has already been sorted by the algorithm. Let τ_i denote the number of comparisons performed in stage i, so that $\tau = \tau_1 + \ldots + \tau_{n-1}$ and, by (6.2.5),

$$A(n) = E[\tau] = E[\tau_1] + E[\tau_2] + \cdots + E[\tau_{n-1}]. \qquad (3.6.8)$$

We now calculate $E[\tau_i]$, $i = 1, \ldots, n - 1$, using formula (3.6.7). We have

$$E[\tau_i] = \sum_{j=1}^{i} jP(\tau_i = j). \qquad (3.6.9)$$

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

$$P(\tau_i = j) = \frac{1}{i+1}, \quad j = 1, \ldots, i-1,$$
$$P(\tau_i = i) = \frac{2}{i+1}, \quad i = 1, \ldots, n-1. \tag{3.6.10}$$

Substituting (3.6.10) into (3.6.9) and simplifying yields

$$E[\tau_i] = \left(\sum_{j=1}^{i} \frac{j}{i+1} \right) + \frac{i}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}, \quad i = 1, \ldots, n-1. \tag{3.6.11}$$

Substituting (3.6.11) into (3.6.8), we have

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

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

Often it is possible to find a recurrence relation expressing $A(n)$ in terms of one or more of the values $A(m)$ with $m < n$. Of course, if the algorithm itself is written recursively, then the recurrence relation is usually easy to find. In general, finding a recurrence relation for $A(n)$ often involves utilizing the techniques of partitioning the algorithm or the input space. The analysis of the average complexity of *QuickSort* uses both of these techniques.

3.6.4 Average Complexity of *QuickSort*

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

$$A(n) = E[\tau] = E[\tau_1] + E[\tau_2] = n + 1 + E[\tau_2]. \tag{3.6.12}$$

We compute $E[\tau_2]$ using (3.6.7) by introducing the random variable Y that maps an input list $L[0{:}n-1]$ into the proper place for $L[0]$ as determined by a call to *Partition*. The uniform distribution assumption on the input space implies that:

$$P(Y = i) = \frac{1}{n}, \quad i = 0, \ldots, n-1. \tag{3.6.13}$$

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

$$E[\tau_2 \mid Y = i] = \mathrm{A}(i) + \mathrm{A}(n-i-1), \quad i = 0, \ldots, n-1. \tag{3.6.14}$$

Combining (3.6.7), (3.6.12), (3.6.13), and (3.6.14), we have

$$\begin{aligned} \mathrm{A}(n) &= (n+1) + \sum_{i=0}^{n-1} E[\tau_2 \mid Y = i] P(Y = i) \\ &= (n+1) + \sum_{i=0}^{n-1} (\mathrm{A}(i) + \mathrm{A}(n-i-1)) \left(\frac{1}{n}\right) \\ &= (n+1) + \frac{2}{n}(\mathrm{A}(0) + \mathrm{A}(1) + \cdots + \mathrm{A}(n-1)), \\ &\textbf{init.cond.}\ \mathrm{A}(0) = \mathrm{A}(1) = 0. \end{aligned} \tag{3.6.15}$$

Recurrence relation (3.6.15) is an example of what is sometimes referred to as a *full history* recurrence relation, since it relates $A(n)$ to *all* of the previous values $A(i)$, $0 \le i \le n-1$. Fortunately, with some algebraic manipulation, we can transform (3.6.15) into a simpler recurrence relation relating $A(n)$ to just $A(n\text{-}1)$. The trick is to first observe that

$$nA(n) = n(n+1) + 2(A(0) + A(1) + \cdots + A(n-2) + A(n-1)). \tag{3.6.16}$$

Substituting $n-1$ for n in (3.6.16), yields

$$(n-1)A(n-1) = n(n-1) + 2(A(0) + A(1) + \cdots + A(n-2)). \tag{3.6.17}$$

Hence, subtracting (3.6.17) from (3.6.16) we obtain

$$nA(n) - (n-1)A(n-1) = 2n + 2A(n-1). \tag{3.6.18}$$

Rewriting (3.6.18) by moving the term involving $A(n-1)$ to the right-hand side and dividing both sides by $n(n+1)$ yields

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2}{n+1}. \tag{3.6.19}$$

Letting $t(n) = A(n)/(n+1)$, (3.6.19) becomes

$$t(n) = t(n-1) + \frac{2}{n+1}. \tag{3.6.20}$$

Recurrence relation (3.6.20) directly unwinds to yield

$$\begin{aligned} t(n) &= 2\left(\frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{(n+1)}\right) \\ &= 2H(n+1) - 3, \end{aligned} \tag{3.6.21}$$

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

$$A(n) \sim 2n \ln n. \tag{3.6.22}$$

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

3.7 Hard Problems

It is a curious fact in the theory of algorithms that most problems of practical importance are either solvable by algorithms having small degree polynomial (worst-case) complexity, or the best known solutions have complexity that grows faster than *any* polynomial (*super-polynomial*). An example of the latter problem is the famous *Traveling Salesman Problem* (TSP), which asks for a minimum length tour of a given set of n cities from a given starting city. Since there are $(n-1)!$ possible tours, even for a relatively small value of n it is computationally infeasible to find an optimal (minimum distance) tour by a brute force examination of each tour. While better algorithms exist for TSP than brute force examination, all known algorithms are super-polynomial. TSP has many important applications, and whole books have been written about the problem. However, finding an algorithm having polynomial complexity solving TSP, or proving that no such algorithm exists, remains a mystery (it is generally believed that there is no polynomial-complexity algorithm for TSP).

Another problem for which no polynomial algorithm has been found is the problem of factoring an integer n. The fact that factoring is generally believed to be a truly hard problem (that is, requiring a super-polynomial algorithm) is the basis for the most widely used Internet security encryption schemes. Yet another problem that appears to be hard is the so-called *discrete logarithm* problem, which for a given positive base b asks for the value of n if the value of b^n is known.

3.6.1 One-way Functions and Sharing Secrets

The fact that the discrete logarithm problem is hard, yet the inverse problem of computing b^n is easy, that is, it can be computed efficiently, has an important application to cryptography. For example, consider the following problem: Alice and Bob wish to send confidential information to one another over some communication medium (e-mail, telephone conversation, fax, and so forth). However, they know that Eve Dropper has the ability to monitor all communication sent over this medium. Moreover, Alice and Bob have not shared in advance any secret information

that could be used for encrypting information. We now describe a communication protocol based on exponentiation modulo p that can be used by Alice and Bob to share secrets.

Alice and Bob first send messages, monitored by Eve, in which they agree to use a conventional cryptographic system for communication. The cryptographic system, also assumed to be known by Eve, requires that they exchange a secret key value, which we assume is an arbitrary positive integer. While the key value is arbitrary, it must be known by both Alice and Bob. The following question arises: Can Alice and Bob publicly (that is, known to Eve) exchange information that allows them to efficiently determine a key, but at the same time makes it infeasible for Eve to discern the value of this key?

The answer lies in the notion of *one-way* functions. One-way functions are efficiently computable, but their inverses are infeasible to compute. In 1976, Diffie and Hellman first pointed out the utility of one-way functions in cryptology and identified exponentiation modulo p, where p is an integer with several hundred decimal digits, as an example. We now describe how efficient exponentiation yields an effective method of exchanging a secret key.

Alice and Bob send messages agreeing on the value of p, as well as on an integer b between 2 and $p - 1$. Of course, then Eve also knows the values of p and b. Next, Alice and Bob randomly choose integers m and n, respectively, between 1 and $p - 1$. Alice does not know n, and Bob does not know m (Eve knows neither n nor m). Using a modification *PowersMod* of *Powers* in which all calculations are carried out modulo p (see Exercise 3.69b), Alice then calculates and sends the integer b^m **mod** p to Bob. Similarly, Bob calculates and sends the integer b^n **mod** p to Alice. Here comes the secret: Using *PowersMod* again, Alice calculates $(b^n \textbf{ mod } p)^m$ **mod** p, and Bob calculates $(b^m \textbf{ mod } p)^n$ **mod** p. Lo and behold, they have just calculated the same number b^{mn} **mod** p, which is the secret key they now share.

Because Eve has monitored all exchanges between Alice and Bob, she now knows p, b, b^m **mod** p, and b^n **mod** p. Thus, theoretically Eve can determine the secret key b^{mn} **mod** p by calculating m from her knowledge of b^m **mod** p and then calculating $(b^n \textbf{ mod } p)^m$ **mod** p. The problem of determining m from the known values of b, p, and b^m **mod** p is (a special case of) the *discrete logarithm* problem. The naive algorithm solving the discrete logarithm problem is to compute b^i **mod** p to b^m **mod** p as i takes on successive values 1, 2, 3, . . . , until we reach an i such that b^i **mod** $p = b^m$ **mod** p. Since this would take $p/2$ iterations on average, the naive algorithm is obviously infeasible. (Recall that p is an integer having more than a hundred digits!) While there are better algorithms for the discrete logarithm problem, unfortunately for Eve no efficient algorithm is known.

The algorithm *PowersMod* which Alice and Bob used to calculate powers of b requires special techniques for multiplying large integers. A classical divide-and-conquer method for multiplying large integers is discussed in Chapter 4. Since all calculations are reduced modulo p, *PowersMod* always multiplies or squares numbers less than p, so that each such operation takes time bounded above by a constant $C(p)$ depending only on p. Thus, if Alice and Bob choose p, m, and n having 200 decimal digits each, then they can exchange their secret key after performing 3000 multiplications of 200-digit numbers and 3000 reductions of a 400-digit number modulo p. All these computations can be done within a reasonable time.

There are literally thousands of important problems such as the three we have just mentioned that are generally believed to be hard (since nobody has found polynomial solutions for any of them), but which have not been proven to be hard (that is, super-polynomial lower bounds have not been found for any of these problems). Settling the question of whether these problems are truly hard is the most important open question in theoretical computer science today.

Interestingly, the problem of factoring and finding the discrete logarithm have both been shown to admit polynomial quantum algorithms, but, as we have already mentioned, the open question here is whether or not quantum computers of sufficiently large size will ever be practical.

3.7.2 NP-Complete Problems

A decision problem is a problem having a yes or no answer to its inputs. For example, the prime testing problem is the decision problem that asks whether or not a given integer n is prime. Often optimization problems have associated decision problem versions obtained by adding a (goodness measure) parameter and asking whether there is a solution as good as the parameter. For example, in TSP, one could add the parameter k to the problem, and ask whether or not there exists a tour whose total length is not greater than k. Of course, if the optimization problem is solved, the associated decision version is also solved immediately (just compare the length of the optimal tour to the parameter k). Sometimes, at least in special cases, there is a polynomial procedure to solve the optimization problem given a solution to the general decision problem. For example, such a polynomial procedure exists for TSP if all the distances between cities are integers of bounded size.

There is a class of decision problems called NP (for Non-deterministic Polynomial) that can be thought of as problems where candidate solutions to "yes" instances can be constructed ("guessed") in polynomial time, and a given candidate can be checked to see if it is a solution in polynomial time. For example, given TSP with the parameter k, a candidate solution is simply a permutation of the $n - 1$ cities to be visited, and such a permutation can be constructed (guessed) in linear time. Moreover, given a candidate solution, checking whether or not the length of the tour corresponding to this permutation is not larger than k can clearly be done in linear time, since it amounts to adding n numbers.

The class P is the class of decision problems having a polynomial deterministic solution to all instances. Note that it is clear that $P \subseteq NP$, since the candidate solution to "yes" instances can be taken to be the solution constructed by the algorithm in polynomial time. Since allowing nondeterminism (perfect guessing) should help, one would expect that $P \neq NP$, but this important question remains unsolved despite the efforts of the best theoretical computer scientists over more than the last 40 years.

The class of NP-complete problems are the problems in NP that, roughly speaking, are as hard to solve (up to polynomial factors) as any other problem in NP. Somewhat more precisely, problem A is as hard to solve as problem B if there is a mapping t from the inputs of size n to problem B to the inputs of size at most $p(n)$ to problem A for a suitable polynomial p, such that t is constructible in polynomial time (and space), and such that an input I to problem B is a "yes" instance if, and only if, $t(I)$ is a "yes" instance to problem A. We say that problem B is (polynomially) *reducible* to problem A. A problem A in NP is NP-*complete* if *every* problem in NP is reducible to A. It is certainly not obvious that NP-complete problems exist. But, if NP-complete problems exist, and if any one of them belongs to P, then every NP problem belongs to P (which would imply that P = NP, an equality generally believed to be false).

The first NP-complete example was found by Cook, who showed in 1970 that the problem of determining whether or not the variables in a given Boolean expression can be assigned values (true or false) in such a way as to make the Boolean expression true (called a satisfying

assignment). Levin found an example about the same time. Since then thousands of problems have been shown to be NP-complete. Many NP-complete problems have important applications, so it is frustrating not to know whether they are all super-polynomial in complexity. However, it is true that sometimes important special cases of a given NP-complete problem can be solved, or in some useful sense approximately solved, in polynomial time. We will discuss NP-complete problems and approximation algorithms in more detail in Chapter 10.

Interestingly, prime testing was not known to be in P until 2002, when it was shown to be solvable by a 6^{th} degree polynomial in the size of the input. Good probabilistic algorithms for prime testing have been known for some time, and these algorithms still are more efficient in practice than the recently found deterministic polynomial-time algorithm. However, showing that prime testing is in P was nevertheless a major achievement.

3.8 Closing Remarks

An algorithm's complexity is often described in the literature in terms of belonging to an *O*-class, since an *O*-class bound for $W(n)$ also gives a bound on the computing time for any input of size n. Also, the *O*-class value for complexity of an algorithm automatically applies to the problem itself. Another reason for using the *O*-class notation is that it might yield a much simpler measure of the complexity of an algorithm than the more exact order formula using Θ. For example, consider the naïve algorithm for testing whether an integer m is prime by simply successively testing the potential divisors $2,3, \ldots, m^{1/2}$. This algorithm performs only one division when m is even, but will perform $m^{1/2}$ divisions when m is a prime. Thus, the algorithm performs $O(m^{1/2})$ divisions, that is, has $O((\sqrt{2})^n)$ complexity where the input size is the number of binary digits n of m. In this case, the Θ–class for the exact worst-case complexity contains no simply expressible function.

The asymptotic classes O, Θ, Ω do not take into account the size of the constants involved. In practice, as we mentioned before, the constants sometimes do matter. For example, there are algorithms known for solving certain problems that have linear order, but for which the explicit constants involved are so large that the algorithms are totally impractical to implement. Of course, for sufficiently large input size n, a linear algorithm will outperform, say, a quadratic algorithm, but the latter algorithm might have sufficiently small associated explicit and implicit constants that it outperforms the linear algorithm for any input size n small enough to be of practical interest. The linear algorithm in such cases is then mostly of theoretical interest but nevertheless important: It gives hope that a linear algorithm with reasonably small constants can be found.

Exercises

Section 3.1 Asymptotic Behavior of Functions

3.1 Prove each of the following directly from the definition of Θ.

a) $100{,}000{,}000 \in \Theta(1)$

b) $n^2/2 + 2n - 5 \in \Theta(n^2)$

c) $\log_{10}(n^2) \in \Theta(\log_2 n)$

3.2 Prove each of the following directly from the definitions of O and Ω.

a) $17n^{1/6} \in O(n^{1/5})$

b) $1000n^2 \in O(n^2)$

c) $n/1000 - 500 \in \Omega(n)$

d) $30n\log_2 n - 23 \in \Omega(n)$

e) $O(n!) \subset O((n+1)!)$

3.3 Repeat Exercise 3.1 using the Ratio Limit Theorem.

3.4 Repeat Exercise 3.2 using the Ratio Limit Theorem.

3.5 Using the Ratio Limit Theorem and (possibly) L'Hôpital's rule, prove each of the following.

a) $(n^3 + n)/(2n + \ln n) \in \Theta(n^2)$

b) $O(n^4) \subset O(3^n)$

c) $n2^n + n^9 \in O(e^n)$

d) $n^{1/2} \in \Omega((\log n)^4)$

e) $n^{\log_2 n} \in O(2^n)$

3.6 Using the Ratio Limit Theorem, prove the following.

$O(108) \subset O(\ln n) \subset O(n) \subset O(n \ln n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(3^n)$

3.7 For any constants k and $b > 1$, show that

$O(n^k) \subset O(n^{\ln n}) \subset O(b^n)$.

3.8 Prove property (1) of Proposition 3.1.2: For any positive constant c,

$\Omega(f(n)) = \Omega(cf(n))$, $\Theta(f(n)) = \Theta(cf(n))$, and $O(f(n)) = O(cf(n))$.

3.9 Prove property (3.1.4):

$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in \Omega(g(n)) \cap O(g(n))$.

3.10 Show that $f(n) \in \Theta(g(n))$ if, and only if, $\Theta(f(n)) = \Theta(g(n))$.

3.11 Prove properties (3) and (5) of Proposition 3.1.2:

a) if $f(n) \in O(g(n))$, then $O(f(n)) \subseteq O(g(n))$.

b) $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$.

3.12 Prove property (4) of Proposition 3.1.2:

$O(f(n)) = O(g(n)) \Leftrightarrow \Omega(f(n)) = \Omega(g(n)) \Leftrightarrow \Theta(f(n)) = \Theta(g(n))$.

3.13 Prove each of the following.

a) $4n^3 + \sqrt{n} \sim 4n^3 - n^2 + 500$

b) $2^n + 50n^5 + 13n^2 + 42 \sim 2^n$

c) $(6n^2 + 50\sqrt{n})/(3\sqrt{n} + 5\ln n) \sim 2n^{3/2}$

d) $30n - 50 = O(n^2/23)$

e) $1000 \log_2 n = O(\sqrt{n})$

3.14 Suppose a and c are positive constants.

a) If f is a polynomial, show that $f(n + c) \sim f(n)$ and $f(cn) \in \Theta(f(n))$.

b) If $f(n) = a^n$, show that $f(n + c) \in \Theta(f(n))$ and $O(f(n)) \subset O(f(cn))$, $c > 1$.

c) If $f(n) = n!$, show that $O(f(n)) \subset O(f(n + c))$, ($c$ a positive integer).

3.15 Let $P(n)$ be any polynomial of degree k whose leading coefficient is positive, and let a be any real number $a > 1$. Show that $P(n) = O(a^n)$.

3.16 Show that if $1 < a < b$, then $a^n = O(b^n)$.

3.17 Complete the proof of the Ratio Limit Theorem by proving case 3.

3.18 The Ratio Limit Theorem states that $\lim_{n\to\infty} f(n)/g(n) = 0$ implies that $O(f(n)) \subset O(g(n))$.
a) Show that a partial converse is true: If $O(f(n)) \subset O(g(n))$ and $\lim_{n\to\infty} f(n)/g(n)$ exists, then this limit must be zero.
b) Construct examples where $O(f(n)) \subset O(g(n))$ but $\lim_{n\to\infty} f(n)/g(n)$ does not exist.
c) Give examples similar to (b) for the other two cases of the Ratio Limit Theorem.

3.19 Prove the following result from the definition of o and ~. If $f(n)$ and $g(n)$ are functions such that $g(n) \in O(f(n))$, then

$$f(n) \pm g(n) \sim f(n).$$

3.20 The notions of Ω, Θ, O can be viewed as binary relations on the set F of (eventually positive) functions f: N $\to$ R. A (*binary*) *relation* R on a set S is any subset of the Cartesian product $S \times S$. For $x, y \in S$, we say that x is *related* to y, written xRy, if the ordered pair $(x, y) \in$ R. Note that Θ determines a relation on the set F by defining $f \Theta g$ to mean $g(n) \in \Theta(f(n))$. In a similar way, Ω and O determine relations on F.

A very important class of relations on a set S are the so-called equivalence relations. Equivalence relations on S correspond precisely to partitions of S into pair-wise disjoint subsets (see Exercise 3.22).

Definition
A relation R on S is said to be an *equivalence* relation if the following three properties are satisfied.

1. *Reflexive Property:* xRx, $\forall\ x \in$ S.
2. *Symmetric Property:* xR$y \Rightarrow y$Rx, $\forall\ x, y \in$ S.
3. *Transitive Property:* xRy and yR$z \Rightarrow x$Rz, $\forall\ x, y, z \in$ S.

a) Show that the relation Θ is an equivalence relation on $\mathscr{F}$ (that is, verify properties (1), (2), and (3) of Proposition 3.1.1).
b) Show that relations O and Ω are reflexive and transitive but are not symmetric.

3.21 Given an equivalence relation R on a set S, each element $x \in$ S determines an *equivalence class*, denoted by $[x]$, consisting of all elements y such that xRy. It is easily proved that $[x] = [y]$ iff xRy. A finite or infinite collection of subsets of a set S is said to be a *partition* of S if the sets are pairwise disjoint and their union is S.
a) Given an equivalence relation R on a set S, show that the set of equivalence classes is a partition of the set S.
b) Conversely, given any partition of the set S, show there is a unique equivalence relation on S whose equivalence classes are precisely the subsets of the given partition (define xRy iff x and y lie in the same subset).

3.22 a) Show that the relation ~ is actually an equivalence relation on $\mathscr{F}$.
b) Show that ~ is a stronger relation than Θ in the sense that ~ refines Θ. Given two relations R_1, R_2 on a set S, we say that R_2 *refines* R_1 if $x R_2 y \Rightarrow x R_1 y$. If both R_1 and R_2 are equivalence relations, and R_2 refines R_1, then each R_2-equivalence class is contained in an R_1-equivalence class. We say that the R_2-equivalence classes form a *refinement* of the R_1-equivalence classes.

3.23 Consider the relation K on $\mathscr{F}$ defined as follows:

$$f \text{ K} g \text{ iff } \lim_{n\to\infty} \frac{f(n)}{g(n)} = L, \quad \text{where } 0 < L < \infty.$$

The value of L varies with f and g.

a) Show that K is an equivalence relation on $\mathscr{F}$.

b) Show that K refines Θ.

c) Show that K is refined by $\sim$.

3.24 Show that the functions $f(n) = n^3$ and $g(n) = n^4(n \textbf{ mod } 2) + n^2$ are not comparable.

3.25 Find two strictly increasing functions $f, g \in \mathscr{F}$ whose orders are not comparable.

Section 3.2 Asymptotic Order Formulae for Three Important Series

3.26 Obtain a formula for the order of $S(n) = \sum_{i-1}^{n} (\log i)^2$.

3.27 Obtain an approximation for $\log n! = \sum_{i=1}^{n} (\log i)$ by using a technique similar to that used in Section 3.7 for approximating the harmonic series. (*Hint:* Use $\int_1^n \log x dx$.)

3.28 Show that $\lim_{n\to\infty} H(n) = \infty$.

3.29 Show directly from the definition of $S(n,k)$ that $S(n,k) \in \Theta(n^{k+1})$ (that is, without using recurrence relation (2.2.5)).

3.30 Show that $S(n, -k) = 1 + \left(\frac{1}{2}\right)^k + \left(\frac{1}{3}\right)^k + \cdots + \left(\frac{1}{n}\right)^k \in \Theta(1)$, for all integers $k \geq 2$.

3.31 Proposition 7.7.2 states that $S(n,k) = 1^k + 2^k + \ldots + n^k$ is a polynomial in n of degree $k + 1$ whose leading coefficient is $1/(k + 1)$. Using induction and Proposition 3.2.1, show that the coefficients of n^k and n^{k-1} in $S(n,k)$ are $1/2$ and $k/12$, respectively, for $k > 1$.

3.32 a) $S(n,k) = 1^k + 2^k + \ldots + n^k$ is a polynomial in n of degree $k + 1$ whose constant term is zero. Let $B(j,k)$ denote the coefficient of n^j in polynomial $S(n,k)$, $j = 1, \ldots, k + 1$. Using the recurrence relation (3.2.1) for $S(n,k)$, obtain a recurrence relation for $B(j,k)$.

b) Using the recurrence relation for $B(j,k)$, obtained in (a), give pseudocode for the procedure *SumOfPowers*, which outputs the (coefficients of) the polynomial $S(n,k)$.

c) Modify your algorithm to avoid truncation errors when dividing by large integers by storing the coefficients $B(j,k)$ as fractions in lowest form. These fractions can be stored as pairs of integers (*Numer*,*Denom*), where *Numer* is the numerator and *Denom* is the denominator. Obtain the lowest form of the fraction represented by (*Numer*,*Denom*) by employing Euclid's gcd algorithm.

3.33 Generalize the recurrence relation (3.2.1) for $S(n,k)$ to a recurrence relation for the sum of the kth powers of the first n terms in the arbitrary arithmetic progression

$$S(a,d,n,k) = a^k + (a+d)^k + \cdots + (a+(n-1)d)^k.$$

Section 3.3 Recurrence Relations for Complexities

3.34 Derive and solve a recurrence relation for the best-case complexity $B(n)$ of *MergeSort*.

3.35 Solve the following recurrence relations:

a) $t(n) = 3t(n-1) + n, \quad n \geq 1, \quad \textbf{init.cond.} \quad t(0) = 1.$

b) $t(n) = 4t(n-1) + 5, \quad n \geq 1, \quad \textbf{init.cond.} \quad t(0) = 2.$

c) $t(n) = 2t(n/3) + n, \quad n \geq 1, \textbf{init.cond.} \quad t(0) = 0.$

3.36 Determine the Θ-class of $t(n)$ where

$t(n) = 2t(n-1) + n^4 + 1$, **init. cond.** $t(0) = 0$.

3.37 A very natural question to ask related to *MergeSort* is whether we can do better by dividing the list into more than two parts. It turns out we can't do any better, at least in the worst case. In this exercise you will verify this fact mathematically. *TriMergeSort*, given below, is a variant of *MergeSort*, where the list is split into three equal parts instead of two. For convenience, assume that $n = 3^k$, for some nonnegative integer k.

```
procedure TriMergeSort(Low,High) recursive
Input: Low,High (indices of a global array L[0:n-1], initially Low = 0 and
                                        High = n – 1 = 3^k – 1)
Output:      sublist L[Low:High] is sorted in nondecreasing order
     if High ≤ Low then return endif
     Third ← Low + ⌊(High – Low + 1 )/3⌋
     TwoThirds ← Low + 2*⌊(High – Low + 1 )/3⌋
     TriMergeSort(Low,Third)
     TriMergeSort(Third+1,TwoThirds)
     TriMergeSort(TwoThirds+1,High)
       //merge sublists L[Low:Third] and L[Third:TwoThirds] of sizes n/3 and n/3
     Merge(Low,Third,TwoThirds)
      //merge sublists L[Low:TwoThirds] and L[TwoThirds:High] of sizes 2n/3 and n/3
     Merge(Low,TwoThirds,High)
end TriMergeSort
```

a) Give a recurrence relation for the worst-case complexity $W(n)$ of *TriMergeSort* for an input list of size n.

b) Solve the recurrence formula you have given in (a) to obtain an explicit formula for the worst-case complexity $W(n)$ of *TriMergeSort*.

c) Which is more efficient in the worst case, *MergeSort* or *TriMergeSort*? Discuss.

d) Repeat (a), (b), and (c) for the best-case complexity $B(n)$.

3.38 Design and analyze a variant of *TriMergeSort* which uses a single call to a merge procedure for merging three sorted sublists, where three pointers are used, one for each sublist.

3.39 In this section we established a formula for the worst-case complexity $W(n)$ of *MergeSort* when n is a power of two. Find an approximation to $W(n)$ for a general n, showing that $W(n) \in \Theta(n \log n)$.

Section 3.4 Mathematical Induction and Proving Correctness of Algorithms

3.40 Prove by induction that

$$C(n,0) + C(n,1) + C(n,2) + \cdots + C(n,n) = 2^n.$$

3.41 Prove by induction that

$$C(n,0) - C(n,1) + \cdots + (-1)^i C(n,i) + \cdots + (-1)^n C(n,n) = 0.$$

3.42 Prove by induction that

$$fib(1) + fib(2) + \ldots + fib(n) = fib(n+2) - 1$$

where $fib(n)$ denotes the n^{th} Fibonacci number.

3.43 a) Prove by induction that

$$\begin{pmatrix} fib(n) \\ fib(n+1) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

b) Briefly describe how the preceding formula can be employed to design an algorithm for computing $fib(n)$ using only at most $8\log_2 n$ multiplications

3.44 Consider the famous Towers of Hanoi puzzle (see Appendix B). Prove by induction that the minimum number of moves needed to solve the Towers of Hanoi puzzle is $2^n - 1$ (so that the solution given in Appendix B is optimal).

3.45 Suppose you have a collection of n lines in the plane such that no two are parallel and no three meet in a point. Find a formula for the number of regions in the plane determined by these lines, and prove your formula using induction.

3.43 Prove *Merge* is correct.

3.44 Prove *QuickSort* is correct.

3.45 Prove *BubbleSort* (see Exercise 2.34) is correct

3.46 Prove *InsertionSort* is correct

3.47 Prove *BinarySearch* is correct

3.48 Write an iterative version of binary search, and prove the correctness of your algorithm.

3.49 Prove *TriMergeSort* (see Exercise 3.37) is correct.

3.50 We remarked that *Right-to-Left Binary Powers* is directly related to how the recursive algorithm *Powers* computes x^n, except that *Powers* does not require the explicit binary representation of n. The following is a direct translation of *Powers* into an iterative version that is based on a canonical way of removing tail recursion. Of course, *Powers*2 is very similar to *Right-to-Left Binary Powers* , but (as with *Powers*) without requiring the explicit binary representation of n.

```
function Powers2(x,n)
  Input: x (a real number), n (a nonnegative number)
  Output: x^n
    if n = 0 then return(1)  endif
    AccumPowers ← 1
    Pow ← x
    while n > 1 do
       if even(n) then
          n ← n/2
       else
          AccumPowers ← AccumPowers*Pow
          n ← (n - 1)/2
       endif
       Pow ← Pow*Pow
    endwhile
    return(Pow*AccumPowers)
end Powers 2
```

Prove the correctness of *Powers2* using (strong) induction on n (and without explicit use of the binary representation of n).

3.51 a) Using mathematical induction prove that the Fibonacci numbers satisfy:

$$(1.5)^{n-2} \leq fib(n) \leq 2^{n-1},\ n \geq 1.$$

b) Applying the upper bound for $fib(n)$ in part (a) and Exercise 2.7 from Chapter 2, show that the worst-case complexity of *EuclidGCD* belongs to $O(n)$.

Section 3.5 Establishing Lower Bounds for Problems

3.52 a) Show that removing an inversion $(\pi(i),\pi(i+1))$ by interchanging $\pi(i)$ and $\pi(i+1)$ Results in a decrease of (exactly) one in the total number of inversions in the permutation π.

b) Generalize the result in part a. by showing that interchanging $\pi(i)$ and $\pi(i+k)$ results in a decrease of at most $2k-1$ in the total number of inversions in the permutation π.

3.53 Consider any comparison-based sorting algorithm, where each comparison involves elements that are at most k positions apart (that is, elements indexed in positions i and $i+j$ for some $j \leq k$). Use the result from Exercise 3.52 to show that the worst-case complexity $W(n)$ of such an algorithm is at least $(n^2/2 - n/2)/(2k-1)$. Note that this generalizes Proposition 3.5.2.

3.53 Show that any comparison-based algorithm for merging two sorted lists of size $n/2$ to form a sorted list of size n cannot have complexity belonging to $O(n^{1-\varepsilon})$ for any $\varepsilon > 0$. Hint: we have already seen that $\Omega(n\log n)$ is a lower bound for comparison-based sorting.

3.55 a) For the algorithm *InsertionSort*, exhibit the permutations in each set of the partition C_p, $p = 1,2,3$ as described in the proof of Proposition 3.5.4.

b) Repeat part a) for *MergeSort*.

Section 3.6 Probabilistic Analysis of Algorithms

For Exercises 3.56-3.69, we often refer to Appendix E for propositions and formulas.

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

a) Give a table showing the probability distribution for rolling these dice (the sample space is shown in Figure E.1).

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

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

3.57 Consider the random variable $X = r_1 + r_2$ defined on the sample space S given in Exercise 3.56.

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

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

3.58 Repeat Exercise 3.56 for the random variable $X = \max\{r_1,r_2\}$.

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

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

3.61 Give an alternate derivation of the expectation of a binomial distribution using Proposition E.1.4. (*Hint:* Let X_i be the random variable that has the value 1 if there was a success in the ith trial, and 0 otherwise.)

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

3.63 Prove Proposition E.1.1.

3.64 Prove Propositions E.1.3 and E.1.4.

3.65 Verify that the probabilities given by (E.1.12) satisfy the three axioms for a probability distribution on $\{0,1, \ldots, n\}$.

3.66 Verify that P_F defined by (E.1.24) satisfies the three axioms for a probability distribution on F.

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

3.68 Show that formula (E.1.30) of Proposition E.1.7 reduces to formula (E.1.15) when $X = Y$.

3.69 Prove the following generalization of (E.1.5) to n events $E_1,E_2, \ldots, E_n$:
$$P(E_1 \cap E_2 \cap \ldots \cap E_n) = P(E_1)P(E_2 \mid E_1)P(E_3 \mid E_1 \cap E_2)\ldots P(E_n \mid E_1 \cap E_2 \cap \ldots \cap E_{n-1}).$$

3.70 In Chapter 2 we described a (worst-case optimal) algorithm *MaxMin* for determining the maximum and minimum elements in a list $L[0:n-1]$. The following simple algorithm *MaxMin2* for solving this problem has better best-case complexity, but it is not as efficient in the worst-case. It becomes an interesting question as to what is its average behavior.

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

a) Show that $B(n) = n - 1$ and $W(n) = 2n - 2$ for *MaxMin2*

b) Show that $A(n) \sim 2n - \ln n$, so that $A(n)$ is strongly asymptotic to $W(n)$.
Hint: The stated result for $A(n)$ follows from the curious fact that the average number of times $\lambda(n)$ that the maximum is updated by *MaxMin* (and hence also by *MaxMin2*) over all permutations of size n is strongly asymptotic to $\ln n$. To prove this, let $Y: I_n \rightarrow [1,n]$ be defined by $Y(\pi)$ = the position where the maximum occurs for input permutation π. Using (3.6.6) and the fact that $P(Y=i) = 1/n$, $i = 1, \ldots, n$, show that $\lambda(n) = \lambda(n\text{-}1) + 1/n$. The result follows by unwinding this recurrence relation for $\lambda(n)$.

Section 3.7 Hard Problems

3.71 a) Show that
$(x + y)$ **mod** $p = ((x$ **mod** $p) + (y$ **mod** $p))$ **mod** p
(xy) **mod** $p = ((x$ **mod** $p)(y$ **mod** $p))$ **mod** p.

b) Give pseudocode for a modification *PowersMod* of *Powers* in which all calculations are carried out modulo p. The correctness of *PowersMod* follows from (a).