PART I

INTRODUCTION TO ALGORITHMS

1

Introduction and Preliminaries

The use of algorithms did not begin with the introduction of computers. In fact, people have been using algorithms as long as they have been solving problems systematically. While a completely rigorous definition of an algorithm uses the notion of a Turing Machine, the following definition will suffice for our purposes. Informally, an *algorithm* is a complete, step-by-step procedure for solving a specific problem. Each step must be unambiguously expressed in terms of a finite number of rules and guaranteed to terminate in a finite number of applications of these rules. A rule typically calls for the execution of one or more operations. A *sequential* algorithm performs these operations one at a time in sequence, whereas a *parallel* or *distributed* algorithm can perform many operations simultaneously.

In this text, the operations allowed in a sequential algorithm are restricted to instructions found in a typical high-level procedural computer language. These instructions include, for example, arithmetical operations, logical comparisons, and transfer of control. A parallel or distributed algorithm the same operations as a sequential algorithm (in addition to communication operations between processors), but a given operation can be performed on multiple data instances simultaneously. For instance, a parallel algorithm might add one to each element in an array of numbers in a single parallel step.

When designing algorithms, it is important to consider the various models and architectures that will implement the algorithms. We discuss these models in a bit more detail later in this chapter. In this chapter we also provide some historical background for the study of sequential, parallel and distributed algorithms, and give a brief trace of how algorithms have developed from ancient times to the present.

1.1 Algorithms from Ancient to Modern Times

The algorithms discussed in this section solve some classical problems in arithmetic. Some algorithms that are commonly executed on today's computers were originally developed more than three thousand years ago. The examples we present illustrate the fact that the most straightforward algorithm for solving a given problem often is not the most efficient.

1.1.1 Evaluating Powers

An ancient problem in arithmetic is the efficient evaluation of integer powers of a number x. The naive approach to evaluating x^n is to repeatedly multiply x by itself $n – 1$ times, yielding the following algorithm.

```
function NaivePowers($x$,$n$)
Input: $x$ (a real number), $n$ (a positive integer)
Output: $x^n$
    $Product \leftarrow x$
    for $i \leftarrow 1$ to $n - 1$ do
        $Product \leftarrow Product * x$
    endfor
    return($Product$)
end NaivePowers
```

Clearly, *NaivePowers* performs $n - 1$ multiplications. In particular to compute x^{32} *NaivePowers* uses 31 multiplications. However, after a little thought we could have computed x^{32} using only 5 multiplications, by successive squaring operations, yielding: $x^2, x^4, x^8, x^{16}, x^{32}$. Notice that this simple successive squaring works only for exponents that are a power of 2. In fact we can do much better than the naive algorithm for general n by using an algorithm which has been known for over two millennia, and already was referenced in Pingala's Hindu classic *Chandah-sutra* circa 200 BC. To see how this algorithm might arise, consider the problem of computing x^{108}. By repeatedly applying the simple (recurrence) formula

$$x^n = \begin{cases} (x^{n/2})^2 & n \ \mathit{even}, \\ x * (x^{(n-1)/2})^2 & n \ \mathit{odd}, \end{cases}$$

we obtain the collection of reductions:

$$x^{108} = (x^{54})^2,\ x^{54} = (x^{27})^2,\ x^{27} = x(x^{13})^2,\ x^{13} = x(x^6)^2,\ x^6 = (x^3)^2,\ x^3 = x(x^1)^2,\ x^1 = x(x^0)^2.$$

Now x^{108} can be computed by working our way through these reductions from right to left, involving the sequence of powers (on the left-hand-side of the equalities) 1,3,6,13,27,54,108. Note that when we compute the next power of x, we simply square the previous power of x if the exponent is even, and we multiply the square of the previous power of x by x if the exponent is odd. Determining whether the exponent is even or odd can be conveniently obtained from its binary (base 2) expansion, since it amounts to simply checking whether the least significant binary digit is 0 or 1. Writing the exponents in binary, we obtain the sequence 1, 11, 110, 1101, 11011, 110110, 1101100. Note that taking the least significant digit in each number in the sequence is equivalent to simply scanning the last binary number in the sequence (that is, the original number 108) from left to right, which is why this method is called **left-to-right binary exponentiation**. Note also that, in practice, we can always omit the first step (which always yields x), so that we can start our scan from the second most significant digit. High-level pseudocode for the algorithm follows:

```
function Left-to-Right Binary Method for computing $x^n$
Input: $x$ (a real number), $n$ (a positive integer)
Output: $x^n$
    Compute the binary representation of $n$
    Pow ← x
    Scan binary representation from left to right starting with second position
    if 0 is encountered
        Pow ← Pow * Pow
    else
        Pow ← x * Pow* Pow
    endif
    return(Product)
end Left-to-Right Binary Method
```

Compute binary representation of 108 obtaining 110110.

$$x \rightarrow x^2*x = x^3 \rightarrow (x^3)^2 = x^6 \rightarrow x*(x^6)^2 = x^{13} \rightarrow x*(x^{13})^2 = x^{27} \rightarrow (x^{27})^2*x$$
$$= x^{54} \rightarrow (x^{54})^2 = x^{108}$$

Action of left-to-right binary powers in computing x^{108}

Figure 1.1

The above method of computing x^{108} required only 9 multiplications, as opposed to the 107 multiplications required by the naïve method. For a general positive integer n, the left-to-right binary method for computing requires between $\log_2 n$ and $2\log_2 n$ multiplications (see Exercise 1.1). For large n, the difference between the $n - 1$ multiplications required by *NaivePowers* and the at most $2 \log_2 n$ multiplications required the left-to-right binary method is dramatic indeed. We illustrate the dramatic difference between n and $\log_2 n$ in Figure 1.2. For example, 10^{83} is estimated to be more than the number of atoms in the known universe. However, $\log_2 10^{83}$ is smaller than 276.

$n = 2^m$	$m = \log_2 n$	$n - 1$
1	0	0
2	1	1
16	4	15
128	7	127
1,024	10	1,023
1,048,576	20	1,048,575
562,949,953,421,312	49	562,949,953,421,311

Table of values of $\log_2 n$ versus $n - 1$

Figure 1.2

The left-to-right binary exponentiation method illustrates nicely the fact that the simplest algorithm for solving a problem is often much more inefficient than a more clever but perhaps more complicated algorithm. We state this as an important key fact for algorithm design.

Key Fact

The simplest algorithm for solving a problem is many times not the most efficient. Therefore, when designing an algorithm, don't just settle for any algorithm that works.

Remarks

1. The binary method for exponentiation does not always yield the minimum number of multiplications. For example, computing x^{15} by the binary method requires 6 multiplications. However, it can be done using only 5 multiplications (See exercise 1.3).

2. The binary method allows computers to perform the multiplications required to compute x^n where n is an integer with hundreds of binary digits. However, for such values of n, the successive powers of x being computed are growing exponentially, and will quickly exceed the storage capacity of any conceivable computer that will ever be built. In practice, such as in internet security communication protocols, such as RSA, which involve computing x^n for n having hundreds of digits, the exponential growth in the size of the successive powers is avoided by always reducing the powers modulo some fixed integer p at each stage (called *modular exponentiation*).

There is another method for computing x^n that is called **right-to-left binary exponentiation** since it is based on scanning of the binary expansion of n from right to left. The method was mentioned by al-Kashi in 1427, and is similar to a method used by the Egyptians for multiplication as early as 2000BC. The algorithm is directly based on the law of exponents $x^{y+z} = x^y x^z$. Consider the binary expansion of n,

$$n = 2^m + b_{m-1}2^{m-1} + b_{m-2}2^{m-2} + \ldots + b_0.$$

Letting $z = b_{m-1}2^{m-1} + b_{m-2}2^{m-2} + \ldots + b_0$, we have $n = 2^m + z$ and $x^n = x^{2^m+z} = x^{2^m}x^z$.

Note (by the extended law of exponents) that x^z is the product of all terms x^{2^i} as i runs from 0 to $m – 1$ and $b_i \neq 0$. Hence, to compute x^n, we compute x^{2^m} by initializing a variable *Pow* to x, and scanning the binary expansion of n right-to-left from the leftmost digit, squaring *Pow* at each digit position. Clearly, this will result in *Pow* having the value x^{2^m} at the end of the scan. We also initialize a variable *AccumPowers* to one, and during the scan if we encounter a one at digit position i (corresponding to $2^i – 1$), we multiply *AccumPowers* by the current value $x^{2^{i-1}}$ of *Pow* (i.e., the value before *Pow* is squared). Hence, the value of *AccumPowers* is then the product of all terms x^{2j} as j runs from 0 to $i – 1$ and $b_j \neq 0$. In particular, at the conclusion of the scan (ending at the digit position m corresponding to 2^{m-1}), *AccumPowers* has the value x^z. Finally, the value x^n is computed by taking the product of *Pow* and *AccumPowers*.

```
function Right-to-Left Binary Method for computing $x^n$
Input:    $x$ (a real number), $n$ (a positive integer)
Output:  $x^n$
        Compute the binary representation of $n$
        Pow ← $x$
        AccumPowers ← 1
        Scan binary representation from right to left omitting the last (most significant) binary digit
          if 1 is encountered
              AccumPowers ← Pow * AccumPowers
          endif
        Pow ← Pow * Pow
        return(Pow*AccumPowers)
end Right-to-Left Binary Method
```

1.1.2 The Euclidean Algorithm

One of the oldest problems in number theory is to determine the *greatest common divisor* gcd(a,b) of two positive integers a and b. The greatest common divisor of a and b is the largest positive integer k that divides both a and b with no remainder. The problem of calculating gcd(a,b) was already known to the ancient Greek mathematicians. A naive algorithm computes the prime factorization of a and b and collects common prime powers whose product is then equal to gcd(a,b). However, for large a and b, computing the prime factorizations is very time consuming, even on today's fastest computers. A more efficient algorithm was published in *Euclid's Elements* (c. 300 B.C.). Euclid's algorithm is a refinement of an algorithm that was known 200 years earlier. This earlier algorithm is based on the observation that for $a \geq b$, an integer divides both a and b if, and only if, it divides $a - b$ and b. Thus,

$$\gcd(a,b) = \gcd(a-b,b), \quad a \geq b, \quad \gcd(a,a) = a. \tag{1.1.1}$$

Formula (1.1.1) yields the following algorithm for computing gcd(a,b).

```
function NaiveGCD($a,b$)
Input: $a,b$ (two positive integers)
Output: gcd($a,b$) (the greatest common divisor of $a$ and $b$)
    while $a \neq b$ do
        if $a > b$ then
            $a \leftarrow a - b$
        else
            $b \leftarrow b - a$
        endif
    endwhile
    return($a$)
end GCDNaive
```

After each iteration of the while loop in *NaiveGCD*, the larger the previous values of a and b is replaced by a strictly smaller positive number. Hence, *NaiveGCD* eventually terminates, having calculated the gcd of the original a and b.

Euclid's gcd algorithm refines the algorithm *NaiveGCD* by utilizing the fact that if $a > b$ and $a - b$ is still greater than b, then $a - b$ in turn is replaced by $a - 2b$, and so forth. Hence if a is not a multiple of b, then a is eventually replaced by $r = a - qb$, where r is the remainder when a is divided by b. Thus, all these successive subtractions can be replaced by the single invocation a **mod** b, where **mod** is the built-in function defined by

$$a \textbf{ mod } b = a - b\left\lfloor \frac{a}{b} \right\rfloor, \quad a \text{ and } b \text{ integers}, \quad b \neq 0,$$

where $\lfloor x \rfloor$ denotes the largest integer less than or equal to x.

For example, when calculating gcd(108,8), the thirteen subtractions 108 – 8, 100 – 8, 92 –8, . . . , 12 – 8 executed by the algorithm *NaiveGCD* can be replaced by the single calculation 108 **mod** 8 = 4.

The preceding discussion leads to an algorithm based on the following formula:

$$\gcd(a,b) = \gcd(b, a \textbf{ mod } b). \qquad (1.1.2)$$

Note that when a is a multiple of b, $\gcd(a,b) = b$, and a **mod** $b = 0$. So the usual convention $\gcd(b,0) = b$ shows that (1.1.2) remains valid when a is a multiple of b.

Euclid's description of the gcd algorithm based on (1.1.2) was complicated by the fact that the algebraic concept of zero was not yet formalized. The following is a modern version of Euclid's algorithm.

```
function EuclidGCD($a$,$b$)
Input: $a$,$b$ (two nonnegative integers)
Output:     gcd($a$,$b$) (the greatest common divisor of $a$ and $b$)
    while $b \neq 0$ do
        Remainder ← a mod b
        $a \leftarrow b$
        b ← Remainder
    endwhile
    return($a$)
end EuclidGCD
```

We illustrate *EuclidGCD* for input $a = 10724$, $b = 864$. We have

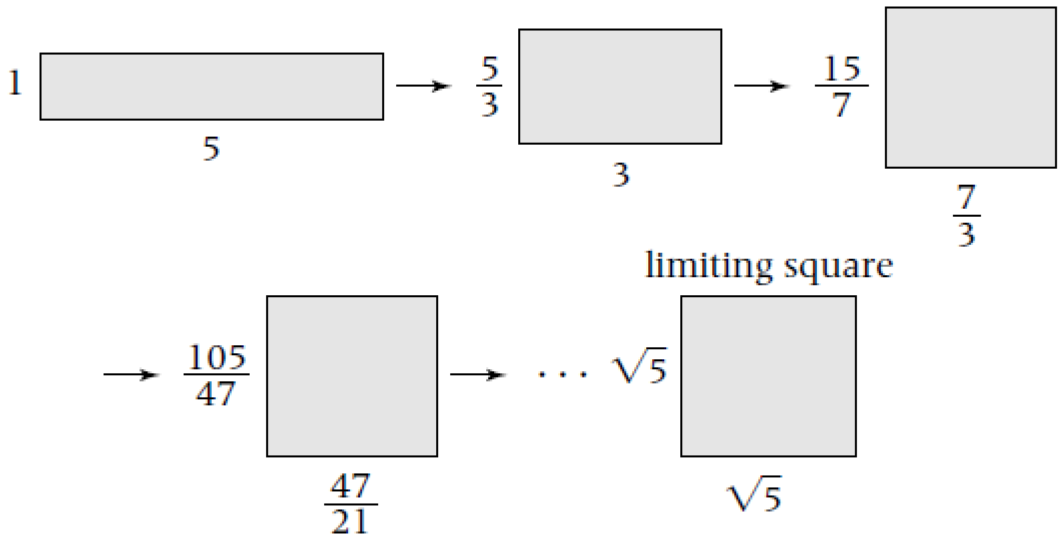
$$\gcd(10724,864) = \gcd(864,356) = \gcd(356,152) = \gcd(152,52) = \gcd(52,48) = \gcd(48,4) = \gcd(4,0) = 4.$$

The problem of computing $\gcd(a,b)$ has very important applications to modern computing, particularly in as it occurs in cryptography and commonly used data security systems (see Chapter 11). It turns out that the **while** loop of *EuclidGCD* never executes more than (roughly) $\log_2(\max(a,b))$ times, so that the algorithm can be executed rapidly even for integers a and b having hundreds of digits.

1.1.3 Babylonian Square Roots

Another mathematical problem gained special significance in the sixth century B.C. when the Pythagorean school of geometers made the startling discovery that the length of the hypotenuse of a right triangle with legs both equal to 1 cannot be expressed as the ratio of two integers. This conclusion is equivalent to saying that $\sqrt{2}$ is not a rational number and therefore its decimal expansion can never be completely calculated. Long before this discovery of irrational numbers people were interested in calculating the square root of a given positive number a to any desired degree of accuracy. A square root algorithm was already known to the Babylonians by 1500 B.C. and is perhaps the first nontrivial mathematical algorithm.

The Babylonian method for calculating $\sqrt{a}$ is based on averaging two points on either side of $\sqrt{a}$. The Babylonians may have discovered this algorithm by considering the problem of laying out a square plot of a given area. For example, consider an area of 5. As a first approximation, they may have considered a rectangular plot of dimensions 1 by 5. If they replaced one dimension by the average of the previous two dimensions, they would obtain a "more square" plot of dimension 3 by 5/3. If they next replaced one of the new dimensions by the average of 3 and 5/3, the dimensions of the plot would be 7/3 by 5/(7/3) (roughly 2.33 by 2.14). More repetitions of this technique lead to plots having sides that are better and better approximations to $\sqrt{5}$ (see Figure 1.3).



Increasingly better approximations of $\sqrt{5}$

Figure 1.3

We can use the Babylonian square root algorithm to calculate the square root of any positive number *a*. Start with an initial guess $x = x_1$ for $\sqrt{a}$; any guess will do, but a good initial guess leads to more rapid convergence. We calculate successive approximations $x_2, x_3, \ldots$ to $\sqrt{a}$ using the formula

$$x_i = \frac{x_{i-1} + (a / x_{i-1})}{2}, \quad i = 2,3,\ldots$$

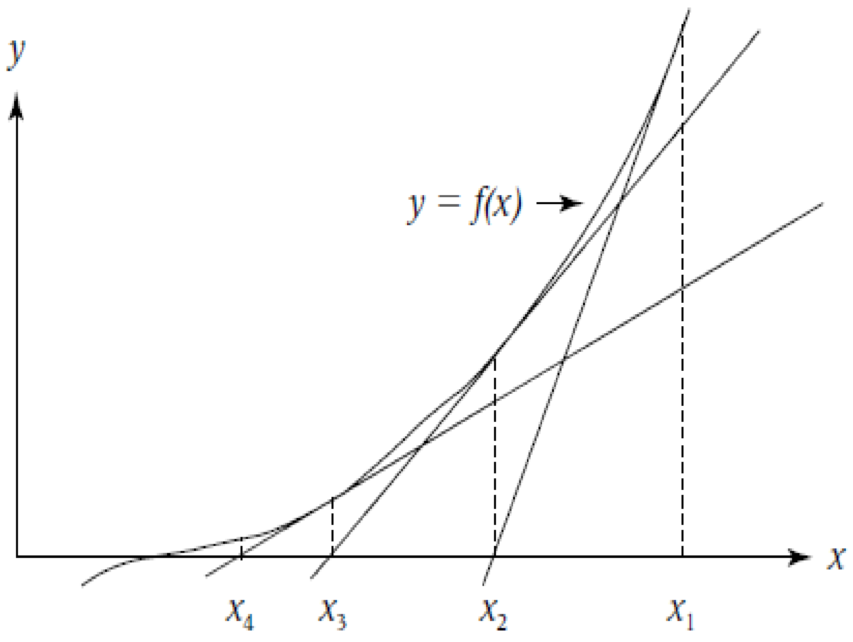
We write the Babylonian square root algorithm as a function whose input parameters are the number *a* and a positive real number *error* measuring the desired accuracy for computing $\sqrt{a}$. For simplicity, /we use *a* as our initial approximation to $\sqrt{a}$.

```
function BabylonianSQRT(a,error)
Input: a (a positive number), error (a positive real number)
Output:     $\sqrt{a}$ accurate to within error
    $x \leftarrow a$
    while $|x - a/x| > error$ do
        $x \leftarrow (x + a/x)/2$
    endwhile
    return($x$)
end BabylonianSQRT
```

Finding square roots is a special instance of the problem of determining the (approximate) roots of a polynomial (note that $\sqrt{a}$ is the positive root of the polynomial $x^2 - a$). More generally, suppose $f(x)$ is a real-valued function, and we wish to find a value of x (called a *zero of f*) such that $f(x) = 0$. If f is a continuous function, then the intermediate value theorem of calculus guarantees that a zero occurs in any closed interval $[a, b]$ where $f(a)$ and $f(b)$ have opposite signs. An algorithm (called the *bisection method*) for determining a zero of f proceeds by bisecting such an interval $[a, b]$ in half, then narrowing the search for a zero to one of the two subintervals where a sign change of f occurs. By repeating this process n times, an approximation to a zero is computed that is no more than $(b - a)2^n$ from an actual zero. We leave the pseudocode for the bisection method to the exercises.

For the case when $f(x)$ is a differentiable function, a more efficient method for finding zeros of f was developed by Sir Isaac Newton in the seventeenth century. Newton's method is based on constructing the tangent line to the graph of f at an initial guess of a zero x_1. The point x_2 where this tangent line crosses the x axis is taken as the second approximation to a zero of f. This process is then repeated at x_2, yielding a third approximation x_3 (see Figure 1.4). Successive iterations of yield points $x_{2,}$, x_3, . . . given by the formula (see Exercise 1.16)

$$x_i = x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})}, \quad i = 2,3,\ldots \tag{1.1.3}$$



Newton's method for finding a zero of a differentiable function f

Figure 1.4

Curiously, when applied to the polynomial $x^2 - a$ Newton's method yields exactly the Babylonian square root algorithm. In general, certain conditions need to be imposed on the function f and starting point x_1 to guarantee that the points x_i given by (1.1.3) actually converge to a zero.

1.1.4 Evaluating Polynomials

A basic problem in mathematics is the problem of evaluating a polynomial $p(x) = a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$ at a particular value of v of x. The most straightforward solution to the problem would be to compute each term a_ix^i independently, $i = 1, \ldots, n$, and sum the individual terms. However, when computing each power v^i, $i = 1, \ldots, n$, it is more efficient to obtain v^i by multiplying the already calculated v^{i-1} by v. This simple observation leads to the following algorithm.

```
function PolyEval(a[0:n],v)
Input: a[0:n] (an array of real numbers), v (a real number)
Output:    the value of the polynomial $a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$ at $x = v$
    Sum ← a[0]
    Product ← 1
    for i ← 1 to n do
        Product ← Product * v
        Sum ← Sum + a[i] * Product
    endfor
    return(Sum)
end PolyEval
```

PolyEval clearly does $2n$ multiplications and n additions, and this might seem the best that we can do. However, there is a simple algorithm for polynomial evaluation that cuts the number of multiplications in half. This algorithm goes under the name of Horner's rule, since W. G. Horner popularized the method in 1819, but the algorithm was actually devised by Sir Isaac Newton in 1699. Horner's rule for polynomial evaluation is based on a clever parenthesizing of the polynomial. For example, a fourth-degree polynomial $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ is rewritten as

$$(((a_4 * x + a_3) * x + a_2) * x + a_1) * x + a_0.$$

This rewriting can be done for a polynomial of *any* degree n, yielding the following algorithm.

```
function HornerEval(a[0:n],v)
Input: a[0:n] (an array of real numbers), v (a real number)
Output:    the value of the polynomial $a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$ at $x = v$
    Sum ← a[n]
    for i ← n – 1 downto 0 do
        Sum ← Sum * v + a[i]
    endfor
    return(Sum)
end HornerEval
```

Clearly, *HornerEval* performs n multiplications and n additions when evaluating a polynomial of degree n. It can be proved that any algorithm for evaluating an nth-degree polynomial that only uses multiplications or divisions and additions or subtractions must perform at least n multiplications or divisions and at least n additions or subtractions. Hence, *HornerEval* is an optimal algorithm for evaluating polynomials.

Algorithms for various other mathematical problems have been developed over the years. These old algorithms were mostly concerned with algebraic rules for calculating numbers and solving arithmetic equations. Indeed, the word *algorithm* itself is taken from the name of the ninth-century Arabic mathematician and astronomer Al-Khowarizmi, who wrote a famous book (*Al Jabr*, c. 825 A.D.) on the manipulation of numbers and equations. One of Al-Khowarizmi's accomplishments involved converting Jewish-calendar dates to Islamic dates. In fact, many early algorithms involved calendars and fixing the days for feasts.

1.4 Closing Remarks

We have discussed several problems having a long history and presented some algorithms for solving these problems. These examples illustrated the fact that the most straightforward and simple algorithm for solving a given problem is often not the most efficient. Designing more efficient algorithms frequently requires utilizing more sophisticated data structures than were used in this chapter. The next chapter will give a review of some standard data structures, and other data structures will be introduced as needed throughout the text.

Measuring the performance of an algorithm was discussed informally in this chapter. In the next chapter we will formalize performance measures by introducing the notions of best-case, worst-case, and average complexities of an algorithm. In order to describe the asymptotic nature of these complexities, in Chapter 3 we will formalize the notion of asymptotic growth rates of functions.

Exercises

Section 1.1 Algorithms Predating Computers

1.1 Trace the action of the left-to-right binary method to compute:
- a) x^{123}
- b) x^{64}
- c) x^{65}
- d) x^{711}

1.2 Repeat exercise 1.1 for the right-to-left binary method.

1.3 Show that computing x^{15} by either of the right-to-left of left-to-right binary methods requires 6 multiplications, and demonstrate that it can be done using only 5 multiplications.

1.4 For a general positive integer n, show that the left-to-right binary method for computing requires between $\log_2 n$ and $2\log_2 n$ multiplications.

1.5 Give pseudocode for implementing the left-to-right binary method when
- a. the number is input as a binary number
- b. the number is input as a decimal number.

1.6 Show that the right-to-left binary method requires the same number of multiplications as the left-to-right binary method.

1.7 Trace the action of the algorithm *GCD* for the following input pairs.
a) (24,108)
b) (23,108)
c) (89,144)
d) (1953,1937)

1.8 Repeat Exercise 1.7 for the algorithm *EuclidGCD.*

1.9 The l*east common multiple* lcm(a,b) of two positive integers a and b is the smallest integer divisible by both a and b. For example, lcm(12,20) = 60. Give a formula for lcm(a,b) in terms of gcd(a,b).

1.10 Let a and b be positive integers and let g = gcd(a,b). Then, g can be expressed as an integer linear combination of a and b, that is there exists integers s and t (not necessarily positive) such $sa + tb = g$. (We will see uses for this in Chapter 18 in connection with the RSA public key cryptosystem).
a) Design and give pseudocode for an *extended Euclid's algorithm*, which inputs integers a and b and outputs integers g, s, t such that $g = sa + tb$.
b) Prove that g is the smallest integer that can be expressed as an integer linear combination of a and b.

1.11 Trace the action of the algorithm *BabylonianSQRT* for the following input values of a and *error* = 0.001.
a) $a = 6$
b) $a = 23$
c) $a = 16$

1.12 A continuous function $f(x)$ such that $f(a)$ and $f(b)$ have opposite signs must have a zero (a point x such that $f(x) = 0$) in the interval (a, b). By checking whether $f(a)$ and $f((a + b)/2)$ have opposite signs, we can determine whether the zero occurs in the subinterval $[a, (a + b)/2]$ or the subinterval $[(a + b)/2, b]$. The *bisection method* for approximating a zero of $f(x)$ is based on repeating this process until a zero is obtained to within a predescribed error. Give pseudocode for the bisection method algorithm *Bisec*($f(x)$,a,b,*error*) for finding an approximation to a zero of a continuous function $f(x)$ in the interval $[a, b]$ accurate to within *error*.

1.13 a) Show how to use the bisection method of Exercise 1.11 to compute $\sqrt{c}$.
b) For $c = 6$ and *error* = 0.00001, compare the efficiency (number of iterations) of *BabylonianSQRT* for computing $\sqrt{c}$ with the bisection method algorithm on the interval [1, 6]. (You might want to write a program for this.)

1.14 For each of the following, do three iterations of the bisection method.
a) $f(x) = x^3 – 6$, initial interval [1, 3]
b) $f(x) = x^3 – 26$, initial interval [1, 3]
c) $f(x) = 2^x – 10$, initial interval [3, 5]

1.15 Show that Newton's method reduces to *BabylonianSQRT* in the special case when $f(x) = x^2 – a$.

1.16 Use calculus and the description given in Figure 1.4 to derive Newton's formula (1.1.3).

1.17 Trace the action of Horner's rule for the polynomial

$$7x^5 – 3x^3 + 2x^2 + x – 5.$$

1.18 In practice, when writing a program requiring interactive input of numeric data, it is useful to check whether the user has entered any "bad characters" (that is, characters not corresponding to a digit between 0 and 9, inclusive). To prevent the program from crashing, we can input the data as a character string, test for bad characters, and convert it to an integer if none are found. Devise an algorithm based on Horner's Rule for converting a string of alphanumeric digits to its numeric value. Assume a function *ConvertDigit* exists for converting an alphanumeric digit to its integer equivalent (for example, *ConvertDigit* ('5') =5). Also assume that the alpha-numeric digits are input one at a time in an *online* fashion, so that the total number of digits is not known in advance.

1.19 To evaluate a polynomial degree n at v and $-v$, one could simply call *HornerEval* twice, involving $2n$ multiplications and $2n$ additions. Describe a modification of *HornerEval* that solves this particular evaluation problem using only $n + 1$ multiplications and $n + 1$ additions. A generalization of this process is the basis of the Fast Fourier Transform (see Chapter 7.)