With a little cleaning up we get

```
procedure GCD2(a, b)
  while b ≠ 0 do
    t ← b; b ← a mod b; a ← t
  repeat
  return(a)
end GCD2
```

Algorithm 1.9 A refined version of Algorithm 1.8

The objective of removing recursion is to produce a more efficient but computationally equivalent iterative program. The fourteen rules stated previously need not always be followed if it is clear that one or more steps are unnecessary. Further, if your compiler translates recursive procedures into efficient code, then you may not need these rules at all. We shall return to recursive procedures and their translation as we meet the need in later chapters.

1.4 ANALYZING ALGORITHMS

Why do we bother to analyze an algorithm? For some of us analyzing algorithms is an intellectual activity that is fun. Another reason is the challenge of being able to predict the future and even though we are narrowing our predictions to algorithms, it is gratifying when we succeed. A third reason is because computer science attracts many people who enjoy being efficiency experts. Analyzing algorithms gives these people a chance to exhibit their skills by devising new ways of doing the same task even faster. This tendency has a large payoff in computing where time means money and efficiency saves dollars.

Before we can talk about how to analyze an algorithm we need to make explicit our assumptions about the kind of computer we expect the algorithm to be executed on. The assumptions we make can have important consequences with respect to how fast a problem can be solved. Though formal models of machines do exist (e.g. Turing machines or Random Access Machines), for most of this book it will be sufficient to consider our computer as a "conventional" one. By this we mean that the instructions of a program are assumed to be carried out one at a time and the major cost of an algorithm depends upon the number of operations it requires. We assume that a random access memory is available which permits one to either access or store any element in a fixed amount of time.

We admit that there are reasons to believe that these assumptions may become outmoded with future generations of machines. Already computers such as ILLIAC IV or the CDC STAR exist and offer a high degree of parallelism in the manner in which a sequence of operations can be executed. This invalidates to some extent the measurement of an algorithm's cost by the summing of its logical operations. A second though somewhat more remote factor is the dramatic decrease in the cost of logic circuits (microprocessors) to the point where configurations of these processors cause the movement of data to be more expensive than the arithmetic and logical operations. If these trends continue, a new theory of computation will be required. But until such machines becomes more pervasive the model of counting and summing logical operations on a sequential processor remains the most accurate predictor of performance and the one we will use.

Given an algorithm to be analyzed, the first task is to determine which operations are employed and what their relative costs are. These operations may include the four basic arithmetic operations on integers: addition, subtraction, multiplication and division. Other basic operations might include arithmetic on floating point numbers, comparisons, assigning values to variables and executing procedure calls. These operations typically take no more than a fixed amount of time and so we say that their time is bounded by a constant. This is not true of all operations of a computer. Some may be composed of an arbitrarily long sequence of more basic operations. For example, a comparison of two character strings may use a character compare instruction which may, in turn, use a shift and bit-compare instruction. The total time for the comparison of two strings will depend upon their lengths, while the time for each character compare is bounded by a constant.

The second task is to determine a sufficient number of data sets which cause the algorithm to exhibit all possible patterns of behavior. This is one of the important and creative tasks of algorithm analysis. It requires us to understand the workings of the algorithm well enough to concoct the data configurations which produce the best or worst or typical behavior. We will say more about this when we discuss particular algorithms.

In producing a complete analysis of the computing time of an algorithm, we distinguish between two phases: *a priori analysis* and *a posteriori testing*. In a priori analysis we obtain a function (of some relevant parameters) which bounds the algorithm's computing time. In a posteriori testing we collect actual statistics about the algorithm's consumption of time and space, while it is executing. Suppose there is the statement $x \leftarrow x + y$ somewhere in the middle of a program. We wish to determine the total time that statement will spend executing, given some initial state of input data.

This requires essentially two items of information, the statement's *frequency count* (i.e. the number of times the statement will be executed) and the time for one execution. The product of these two numbers is the total time. Since the time per execution depends on both the machine being used and the programming language together with its compiler, an a priori analysis limits itself to determining the frequency count of each statement. This number can be determined directly from the algorithm, independent of the machine it will be executed on and the programming language the algorithm is written in.

For example consider the three program segments a,b,c:

```
(a)
x ← x + y

(b)
for i ← 1 to n do
   x ← x + y
repeat

(c)
for i ← 1 to n do
   for j ← 1 to n do
      x ← x + y
   repeat
repeat
```

For each segment we assume the statement $x \leftarrow x + y$ is contained within no other loop than what is already visible. Thus for segment (a) the frequency count of this statement is 1. For segment (b) the count is n and for segment (c) it is n^2. These frequencies 1, n, n^2 are said to be different, increasing *orders of magnitude*. An order of magnitude is a common notion with which we are all familiar; for example walking, bicycling, riding in a car and flying in an airplane represent increasing orders of magnitude with respect to the distance we can travel per hour. In connection with algorithm analysis, the order of magnitude of a statement refers to its frequency of execution, while the order of magnitude of an algorithm refers to the sum of the frequencies of all of its statements. Given three algorithms for solving the same problem whose orders of magnitude are n, n^2, and n^3, naturally we will prefer the first since the second and third are progressively slower. For example, if $n = 10$ then these algorithms will require 10, 100, and 1000 units of time to execute respectively (assuming all basic operations are of equal duration). Determining the order of magnitude of an algorithm is very important and producing an algorithm which is faster by an order of magnitude is a significant accomplishment. The a priori analysis of algorithms is concerned chiefly with order of magnitude determination. Fortunately there is a convenient mathematical notation for dealing with this concept.

Asymptotic Notation

An a priori analysis of computing time ignores all of the factors which are machine or programming language dependent and concentrates on determining the order of magnitude of the frequency of execution of statements. There are several kinds of mathematical notation which are very useful for this kind of analysis. One of these is the O-notation.

Definition: $f(n) = O(g(n))$ (read as "f of n equals big oh of g of n") iff there exist two positive constants c and n_0 such that $|f(n)| \le c\,|g(n)|$ for all $n \ge n_0$.

Suppose we are determining the computing time, $f(n)$, of some algorithm. The variable n might be the number of inputs or outputs, their sum or the magnitude of one of them. Since $f(n)$ is machine dependent, an a priori analysis will not suffice to determine it. However, an a priori analysis can be used to determine a $g(n)$ such that $f(n) = O(g(n))$. When we say that *an algorithm has computing time* $O(g(n))$ we mean that if the algorithm is run on some computer on the same type of data but for increasing values of n, the resulting times will always be less than some constant times $|g(n)|$. When determining the order of magnitude of $f(n)$ we shall always try to obtain the smallest $g(n)$ such that $f(n) = O(g(n))$.

Theorem 1.1: If $A(n) = a_m n^m + \cdots + a_1 n + a_0$ is a polynomial of degree m then $A(n) = O(n^m)$.

Proof: Using the definition of $A(n)$ and a simple inequality

$$\begin{aligned}|A(n)| &\le |a_m|n^m + \cdots + |a_1|n + |a_0| \\ &\le (|a_m| + |a_{m-1}|/n + \cdots + |a_0|/n^m)n^m \\ &\le (|a_m| + \cdots + |a_0|)n^m,\ n \ge 1.\end{aligned}$$

Choosing $c = |a_m| + \cdots + |a_0|$ and $n_0 = 1$ the theorem immediately follows. □

Theorem 1.1 says that if we can describe the frequency of execution of a statement in an algorithm by a polynomial such as $A(n)$, then that statement's computing time is $O(n^m)$. However the constant in the above theorem is not the best possible. Actually we can show that any constant greater than $|a_m|$ can be used (for sufficiently large n).

If an algorithm has k statements whose orders of magnitude are $c_1 n^{m_1}$, $c_2 n^{m_2}$, ..., $c_k n^{m_k}$ then the order of magnitude of the entire algorithm is given by $c_1 n^{m_1} + \ldots + c_k n^{m_k}$ which by Theorem 1.1 is equal to $O(n^m)$ where $m = \max\{m_i\}$, $1 \le i \le k$.

If we have two algorithms which perform the same task on n inputs, and the first has a computing time which is $O(n)$ and the second $O(n^2)$, which is superior? It is easy to see that for sufficiently large values of n, the time for the second algorithm will be larger than the time for the first. For example, if the actual computing times for these algorithms are $2n$ and n^2 respectively, then algorithm one is faster (i.e. has a smaller value) than algorithm two for all $n > 2$. On the other hand if the actual computing times are $10^4 n$ and n^2 then algorithm two is faster for all $n < 10^4$. For $n > 10^4$ algorithm one is faster. So, we cannot decide which of the two algorithms is better unless we know something about the constants associated with the orders of magnitude. If the constants are comparable then the lower order algorithm is better than the higher order algorithm. But this is not the whole story. The point at which one algorithm requires fewer operations than another also depends upon the low order terms. In practice these terms and their coefficients depend on many factors, such as the language and the machine one is using. Alas, it is far more difficult to derive the entire formula for the computing time than the leading term. Thus for a priori analysis, we content ourselves with determining the order of magnitude, and the establishment of its constant will be postponed until after the program has been written and executed. We will not usually derive any terms other than the order of magnitude, unless those terms significantly influence the comparison of two algorithms.

As an example of the usefulness of improving an algorithm by an order of magnitude, suppose we have two algorithms for solving the same task which require n^2 and $n \log n$ operations on n inputs. For $n = 1024$ they require 1,048,576 versus 10,240 operations. If it takes one microsecond to perform each operation then algorithm one requires about 1.05 seconds while algorithm two requires .01 seconds on the same input. If we double n to 2048, then the operation counts become 4,194,304 versus 22,528 or roughly 4.2 seconds versus .02 seconds. When the n is doubled an $O(n^2)$ algorithm takes four times as long to complete while an $O(n \log n)$ algorithm takes only a little more than twice as long to complete. Since an n of several thousand is not especially large, we see how important an order of magnitude improvement such as this can be.

The most common computing times for algorithms we will see here are

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) \text{ and } O(2^n)$$

$O(1)$ means that the number of executions of basic operations is fixed and hence the total time is bounded by a constant. The first six orders of magnitude have an important property in common, they are bounded by a polynomial. $O(n)$, $O(n^2)$, and $O(n^3)$ are themselves polynomials referred to by their degrees: linear, quadratic, and cubic. However, there is no integer m such that n^m bounds 2^n, or

$$2^n \neq O(n^m)$$

for any integer m. The order of this formula is $O(2^n)$.

An algorithm whose computing time is bounded by $O(2^n)$ is said to require *exponential time*. As n gets large, there becomes a tremendous difference between exponential and polynomial time algorithms. If one finds an algorithm which reduces the time to solve a problem from exponential to polynomial, that is a great accomplishment. See Chapter 11 for a further discussion of polynomial versus exponential time algorithms.

Figure 1.9 and Table 1.1 show how the computing times for six of the typical functions grow with a constant equal to one. Notice how the times $O(n)$ and $O(n \log n)$ grow much more slowly than the others. For large data sets, algorithms with a complexity greater than $O(n \log n)$ are often impractical. An algorithm which is exponential will be practical only for *very* small values of n and even if we decrease the leading constant, say by a factor of 2 or 3, we will not improve the amount of data we can handle by very much. To see more precisely why a change in the constant, rather than to the order, of an algorithm produces very little improvement in running time we look at an example.

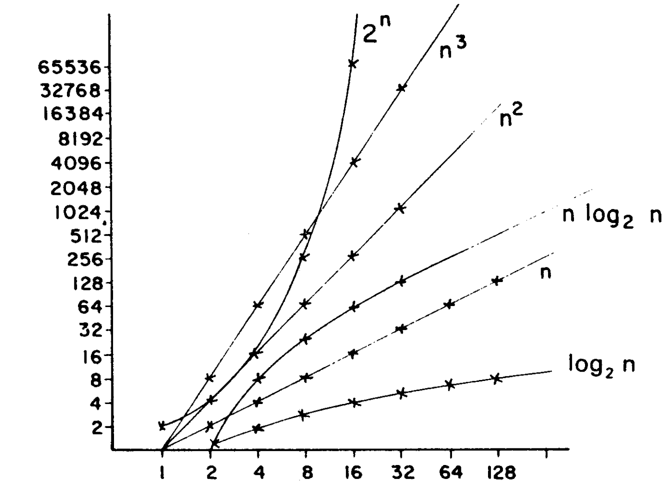

Figure 1.9 Rate of growth of common computing time functions

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Table 1.1 Values for computing functions

Example 1.5 Suppose the orders of magnitude of two algorithms are $n^2 * 2^n$ and $n * 2^n$. Both algorithms are exponential, but in one case there is an extra factor of n. The leading constants are assumed to be one. The respective frequency counts are:

n	$n * 2^n$	$n^2 * 2^n$
5	160	800
10	10240	102400
15	491520	7372800
20	20971520	419430400
30	3.2×10^{10}	9.6×10^{11}

Using the same assumption as before of one operation per microsecond, we observe that for $n = 30$ the times are roughly 8.9 hours versus 11 days. Though the extra linear factor does make a considerable difference, the exponential character of these times dominates and implies that they will both soon become intolerably long. If we were able to speed up the second algorithm by a factor of ten, so that the time is $(1/10)n^2 2^n$, then for $n > 10$ the first algorithm is still faster. Moreover, for $n = 30$ the time required by this faster version is still greater than 24 hours. The conclusion we draw from this example is this: exponential algorithms require so much time, that neither subsequent improvements in the speed of sequential computers nor improvements which effect even the leading constant of the computing time, will ever produce a much greater range of solvable problem size. One possible recourse is to devise new algorithms with much improved orders of magnitude. □

So far we have concentrated on O-notation as a means for describing an algorithm's performance. Whereas O-notation is used to express an upper bound, we might also wish to determine a function which is a lower bound. What is needed is a mathematical notation for expressing a formula which is a lower bound on the computing time of an algorithm to within a constant.

Definition: $f(n) = \Omega(g(n))$, (read as "f of n equals Ω of $g(n)$") iff there exist positive constants c and n_0 such that for all $n > n_0$, $|f(n)| \geq c|g(n)|$.

In some cases the time for an algorithm, $f(n)$, will be such that $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$. For this circumstance we will use the following notation.

Definition: $f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2, and n_0 such that for all $n > n_0$, $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$.

If $f(n) = \Theta(g(n))$ then $g(n)$ is both an upper and lower bound on $f(n)$. This means that the worst and best cases require the same amount of time to within a constant factor. As an example consider the algorithm which finds the maximum of n elements, Algorithm 1.1. The computing time for this algorithm is both $O(n)$ and $\Omega(n)$ since the **for** loop always makes $n - 1$ iterations. Thus, we say that its time is $\Theta(n)$. The procedure of algorithm 1.4 searches an array of n elements for a single value. It has a computing time which is $O(n)$ but $\Omega(1)$. In the best case it might find the value on the first comparison, but in the worst case it will look at all elements once.

An even stronger mathematical notation is given by the following.

Definition: $f(n) \sim o(g(n))$ (read as "f of n is asymptotic to $g(n)$") iff there exists a positive constant n_0 such that

$$\textit{limit} \quad f(n)/g(n) \to 1 \text{ as } n \to \infty \textit{ for } n > n_0$$

Since the ratio in the limit is one, the functions $f(n)$ and $g(n)$ must agree even closer than by a constant factor. If there is an algorithm whose exact computing time is $f(n)$ and we can determine a $g(n)$ such that f is asymptotic to g, then we will have a more precise description of the computing time than if we had used the big O-notation. In practice it implies we will know both the order of the leading term and its constant. For example if $f(n) = a_k n^k + \ldots + a_0$ then

$$f(n) = O(n^k)$$

and

$$f(n) \sim o(a_k n^k)$$

Sums of Integers

As we work to determine the frequency of execution of statements we shall often encounter expressions of the form

$$\sum_{g(n) \le i \le h(n)} f(i) \tag{1.1}$$

where $f(i)$ is a polynomial in i with rational number coefficients. The most common forms of this formula are

$$\sum_{1 \le i \le n} 1, \quad \sum_{1 \le i \le n} i, \quad \sum_{1 \le i \le n} i^2 \tag{1.2}$$

which are the first three Bernoulli polynomials. Since these sums are finite there exist formulas, polynomials in n, which are equal to these sums. The value of the first sum is easily seen to be n. But how do we determine the values of the others? One method is by using interpolation. For example we can think of the second summation as describing the set of points in two dimensional space, $(n,P(n))$, which are:

$$(1,1), (2,3), (3,6)\ (4,10), \ldots$$

$P(n)$ is the polynomial to be found. According to Lagrange's formula (see Chapter 9 for more details) we find that

$$\sum_{1 \le i \le n} i = n(n+1)/2 = \theta(n^2) \tag{1.3}$$

$$\sum_{1 \le i \le n} i^2 = n(n+1)(2n+1)/6 = \theta(n^3) \tag{1.4}$$

In general we will find that

$$\sum_{1 \le i \le n} i^k = \frac{n^{k+1}}{k+1} + \frac{n^k}{2} + \text{lower order terms} \tag{1.5}$$

Thus we can conclude that

$$\sum_{1 \le i \le n} i^k = \theta(n^{k+1}) \tag{1.6}$$

or more precisely

$$\sum_{1 \le i \le n} i^k \sim o\left(\frac{n^{k+1}}{k+1}\right) \tag{1.7}$$

PROFILING

Suppose we assume at this stage that a program solving some problem has been devised, coded, proved correct, and debugged on a computer. How do we go about producing a *performance profile*, that is determining the precise amounts of time and storage this program will consume? In order to determine exact times, our computer must be equipped with a clock whose time can be read. Using this timing capability there are many factors of the program's performance we can check. The most important test of a program is the one which confirms the earlier analysis of the order of magnitude. A program whose time has been determined to be $\Theta(n)$ or $\Theta(n \log n)$, etc. will have a performance profile which looks like the curves in Figure 1.9. Using actual timing data we should be able to determine the exact shape of this curve given the programming language and the machine we are using.

Let the program be called SOLUTION(X,Y) where X denotes the input and Y the output. When the initial analysis was first done, a consideration of possible data sets was made. This was necessary to determine at least the worst and best possbile cases of the algorithm. Let these data sets be created to be used as input to this procedure. Then a program to produce a timing profile has the following general form:

```
procedure PROFILE
  // this program outlines the form that a main program//
  //will take when testing the program SOLUTION(X,Y)//

  //initialize any variables that may be needed for SOLUTION//
  print('Test of algorithm SOLUTION. Times in milliseconds')
  loop
    read(DATA)
    if DATA = end-of-file then exit endif
    print('A new data set =', DATA)
    call STIME(t)
    //Procedure STIME initializes t to the current//
    //value of the clock. Determining the time on a//
    //computer is machine dependent and varies greatly.//
    //See a consultant at your computing center for further details.//
    call SOLUTION(DATA,OUTPUT)
    call STIME(s)
    print('Time =', s - t)
  repeat
end PROFILE
```

Algorithm 1.10 Schema for producing a program's performance profile