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.

where 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:

```
x ← x + y                  (a)

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

for i ← 1 to n do          (c)
    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 $|f(n)| \leq c\,|g(n)|$ for all $n \geq n_0$.

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

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

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 $m = \max\{m_i\}$, $1 \leq i \leq k$.

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

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

$O(1)$ 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 below by $\Omega(2^n)$ 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.

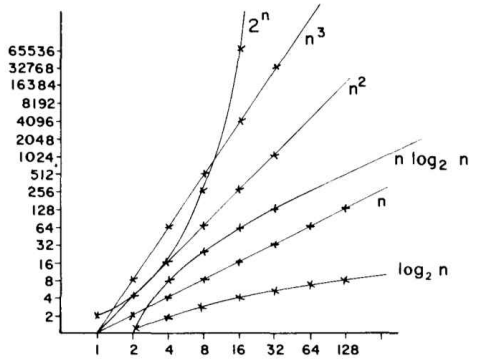


Figure 1.9 Rate of growth of common computing time functions

$\log n$	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 omega 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

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

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

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

Chapter 3

DIVIDE-AND-CONQUER

3.1 THE GENERAL METHOD

Given a function to compute on n inputs the *divide-and-conquer* strategy suggests splitting the inputs into k distinct subsets, $1 < k \leq n$ yielding k subproblems. These subproblems must be solved and then a method must be found to combine subsolutions into a solution of the whole. If the subproblems are still relatively large, then the divide-and-conquer strategy may possibly be reapplied. Often the subproblems resulting from a divide-and-conquer design are of the *same* type as the original problem. For those cases the reapplication of the divide-and-conquer principle is naturally expressed by a recursive procedure. Now smaller and smaller subproblems of the same kind are generated, eventually producing subproblems that are small enough to be solved without splitting.

To be more precise suppose we consider the divide-and-conquer strategy when it splits the input into two subproblems of the same kind as the original problem. This splitting is typical of many of the problems we will see here. We can write a control abstraction which mirrors the way an actual program based upon divide-and-conquer will look. By a *control abstraction* we informally mean a procedure whose flow of control is clear, but whose primary operations are specified by other procedures whose precise meaning is left undefined. Let the n inputs be stored (or pointed at) by the array $A(1{:}n)$ and we will assume this array is global to Algorithm 3.1. Procedure DANDC is a function which is initially invoked as DANDC(1, n). DANDC (p, q) solves a problem instance defined by the inputs $A(p{:}q)$.

```
procedure DANDC(p, q)
  global n, A(1:n); integer m,p,q;        //1 ≤ p ≤ q ≤ n//
  if SMALL(p,q)
    then return (G(p, q))
    else m ← DIVIDE(p,q)      //p ≤ m < q//
      return(COMBINE(DANDC(p,m), DANDC(m + 1,q)))
  endif
end DANDC
```

Algorithm 3.1 Control abstraction for divide-and-conquer

SMALL(p,q) is a Boolean valued function which determines if the input size $q - p + 1$ is small enough so that the answer can be computed without splitting. If this is so the function G is invoked. Otherwise the function DIVIDE(p,q) is called. This function returns an integer which specifies where the input is to be split. Let m = DIVIDE(p,q). The input is split so that $A(p{:}m)$ and $A(m + 1, q)$ define instances of two subproblems. The solutions x and y respectively of these two subproblems are obtained by recursive application of DANDC. COMBINE(x, y) is a function which determines the solution to $A(p{:}q)$ using the solutions x and y to the two subproblems $A(p{:}m)$ and $A(m + 1, q)$. If the sizes of the two subproblems are approximately equal then the computing time of DANDC is naturally described by the recurrence relation

$$T(n) = \begin{cases} g(n), & \text{n small} \\ 2T(n/2) + f(n), & \text{otherwise} \end{cases} \tag{3.1}$$

where $T(n)$ is the time for DANDC on n inputs, $g(n)$ is the time to compute the answer directly for small inputs and $f(n)$ is the time for DIVIDE and COMBINE. Recurrence relations will often arise for divide-and-conquer based algorithms and we will see how to work with them as they arise.

For divide-and-conquer based algorithms which produce subproblems of the same type as the original problem it is very natural to first describe such an algorithm using recursion. But to gain efficiency it may be desirable to translate the resulting program into iterative form. Algorithm 3.2 shows the result of applying the translation rules of section 1.3 to Algorithm 3.1.

```
procedure DANDC1 (p,q)
  //iterative version of DANDC//
  //declare a stack of appropriate size//
  local s, t
  top ← 0      //set the stack to empty//
  L1: while not SMALL(p, q) do
      m ← DIVIDE(p,q)  //determine how to split the input//
      STACK gets p, q, m, 0, 2      //process the first recursive call;//
                                    //increment top//
      q ← m
  repeat
  t ← G(p, q)
  while top ≠ 0 do
    p, q, m, s, ret removed from STACK    //decrement top appropri-//
                                          //ately//
    if ret = 2
      then STACK gets p, q, m, t, 3   //process the second recursive call//
        p ← m + 1
        go to L1
      else t ← COMBINE(s, t)   //combine two solutions into one//
    endif
  repeat
  return(t)
end DANDC1
```

Algorithm 3.2 Iterative form of divide-and-conquer control abstraction

3.2 BINARY SEARCH

Let a_i, $1 \leq i \leq n$ be a list of elements which are sorted in nondecreasing order. Consider the problem of determining whether a given element x is present in the list. In case x is present, we are to determine a value j such that $a_j = x$. If x is not in the list then j is to be set to zero. Divide-and-conquer suggests breaking up any instance $I = (n, a_1, \ldots, a_n, x)$ of this search problem into subinstances. One possibility is to pick an index k and obtain three instances: $I1 = (k - 1, a_1, \ldots, a_{k-1}, x)$, $I2 = (1, a_k, x)$, and $I3 = (n - k, a_{k+1}, \ldots, a_n, x)$. The search problem for two of these three instances is easily solved by comparing x with a_k. If $x = a_k$ then $j = k$ and $I1$ and $I3$ need not be solved. If $x < a_k$ then for $I2$ and $I3$, $j = 0$ and only $I1$ remains to be solved. If $x > a_k$ then for $I1$ and $I2$, $j = 0$ and

only $I3$ remains to be solved. After a comparison with a_k, the instance remaining to be solved (if any) can be solved by using this divide-and-conquer scheme again. If k is always chosen such that a_k is the middle element (i.e. $k = \lfloor (n + 1)/2 \rfloor$ then the resulting search algorithm is known as binary search.

Algorithm 3.3 describes this binary search method [illegible] SPARKS. Procedure BINSRCH has three inputs, A, n [illegible] put, j. The **while** loop continues processing as long as there are more elements left to check. The **case** statement permits the selection of the three alternatives. The first two conditions are checked for, and if they do not occur, the "**else** clause" is automatically executed. At the conclusion of the procedure either $j = 0$ if x is not present, or $A(j) = x$.

```
procedure BINSRCH(A,n,x,j)
  //given an array A(1:n) of elements in nondecreasing order,//
  //n ≥ 0, determine if x is present, and if so, set j such that x = A(j)//
  //else j = 0.//
  integer low, high, mid, j, n;
  low ← 1; high ← n
  while low ≤ high do
      mid ← ⌊(low + high)/2⌋
    case
      : x < A(mid) : high ← mid − 1
      : x > A(mid) : low ← mid + 1
      : else : j ← mid; return
    endcase
  repeat
  j ← 0
end BINSRCH
```

Algorithm 3.3 Binary Search

Is BINSRCH an algorithm? We must be sure that all of the operations such as comparisons between x and $A(mid)$ are well defined. If the elements of A are integers, reals, or character strings then the relational operators will correctly carry out the comparisons. This will be true for those languages which offer these data types. Does BINSRCH terminate? We observe that *low* and *high* are integer variables such that each time through the loop either x is found or *low* is increased by at least one or *high* is decreased by

at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if x is not present.

Let us select the nine entries,

$$-15, -6, 0, 7, 9, 23, 54, 82, 101$$

place them in $A(1:9)$, and simulate the steps that BIN ou as it searches for different values of x. Only the variabl d n, need to be traced as we simulate the algorithm. We shall try the following values for x: 101, −14, and 82 for two successful searches and one unsuccessful search.

$x = 101$	*low*	*high*	*mid*
	1	9	5
	6	9	7
	8	9	8
	9	9	9
			found

$x = -14$	*low*	*high*	*mid*
	1	9	5
	1	4	2
	1	1	1
	2	1	not found

$x = 82$	*low*	*high*	*mid*
	1	9	5
	6	9	7
	8	9	8
			found

Table 3.1 Three examples of binary search on nine elements

These examples may give us a little more confidence about Algorithm 3.3, but they by no means prove that it is correct. Proofs of programs are very useful because they establish the correctness of the program for *all* possible inputs, while testing gives much less in the way of guarantees. Unfortunately, program proving is a very difficult process and the complete proof of a program can be many times longer than the program itself. We shall content ourselves with an "informal proof" of BINSRCH.

Theorem 3.1 Procedure BINSRCH(A, n, x, j) works correctly.

Proof: We assume that all statements work as expected and that comparisons such as $x > A(mid)$ are appropriately carried out. Initially $low = 1$, $high = n$, $n \geq 0$ and $A(1) \leq \ldots \leq A(n)$. If $n = 0$ the **while** loop is not entered and j is set to zero. Otherwise we observe that each time through the loop the possible elements to be checked for equality with x are $A(low)$,

Can we expect another searching algorithm to be significantly better than binary search in the worst case? This question will be pursued rigorously in chapter 10. But we can anticipate the answer here which is no. The method for proving such an assertion is to view the binary decision tree as a general model for any searching algorithm which depends upon comparisons of entire elements. Viewed in this way, we observe that the *longest* path to discover any element is minimized by binary search, ar so any alternative algorithm will be no better from thi

Before we end this section there is an interestin bi search which is useful for programming languages which require two co parisons to implement the **case** statement of procedure BINSRCH. This variation appears as Algorithm 3.4. The correctness proof of this algorithm is left as an exercise.

```
procedure BINSRCH1(A,n,x,j)
  //Same specifications as BINSRCH except n > 0.//
  integer low, high, mid, j, n;
  low ← 1; high ← n + 1   //high is always one more than is possible//
  while low < high − 1 do
    mid ← ⌊(low + high)/2⌋
    if x < A(mid)      //only one comparison in the loop//
       then high ← mid
       else low ← mid   //x ≥ A(mid)//
    endif
  repeat
  if x = A(low) then j ← low      //x is present//
                else j ← 0        //x is not present//
  endif
end BINSRCH1
```

Algorithm 3.4 Binary search using one comparison per cycle

The virtue of this procedure is that it uses only one comparison between x and $A(mid)$ within the **while** loop. The **case** statement of BINSRCH can be implemented using the arithmetic-if statement in FORTRAN. In a language such as PL/*I* or Pascal, it may be implemented by the code equivalent to:

```
if x < A(mid) then high ← mid − 1
              else if x > A(mid) then low ← mid + 1
                                 else j ← mid; return
              endif
endif
```

3.4 MERGESORT

As another example of divide-and-conquer, we inves[t] 128/643 [al]gorithm which has the nice property that in the worst ca[se] [co]mplexity is $O(n \log_2 n)$. This algorithm is called mergesort. We shall assume throughout that the elements are to be sorted in nondecreasing order. Given a sequence of n elements (also called keys) $A(1), \ldots, A(n)$ the general idea is to imagine them split into two sets $A(1), \ldots, A(\lfloor n/2 \rfloor)$ and $A(\lfloor n/2 \rfloor + 1), \ldots, A(n)$. Each set is individually sorted and the resulting sequences are merged to produce a single sorted sequence of n elements. Thus we have another ideal example of the divide-and-conquer strategy where the splitting is into two equal size sets and the combining operation is the merging of two sorted sets into one.

Procedure MERGESORT describes this process very succinctly using recursion and a subprocedure MERGE which merges together two sorted sets.

```
procedure MERGESORT(low, high)
  //A(low : high) is a global array containing high - low + 1 ≥ 0//
  //values which represent the elements to be sorted.//
  integer low, high;
  if low < high
    then mid ← ⌊(low + high)/2⌋   //find where to split the set//
         call MERGESORT(low, mid)   //sort one subset//
         call MERGESORT(mid + 1, high)   //sort the other subset//
         call MERGE(low, mid, high)   //combine the results//
  endif
end MERGESORT
```

Algorithm 3.7 Mergesort

```
procedure MERGE(low, mid, high)
  //A(low:high) is a global array containing two sorted subsets  //
  //in A(low:mid) and in A(mid + 1:high).  //
  //The objective is to merge these sorted sets into  //
  //a single sorted set residing in A(low:high). An auxiliary array B is used//
  integer h, i, j, k, low, mid, high;  //low ≤ mid < high//
  global A(low:high); local B(low:high)
  h ← low; i ← low; j ← mid + 1;
  while h ≤ mid and j ≤ high do  //while both sets are not exhausted//
    if A(h) ≤ A(j) then B(i) ← A(h); h ← h + 1
                   else B(i) ← A(j); j ← j + 1
    endif
    i ← i + 1
  repeat
  if h > mid then for k ← j to high do  //handle any remaining elements//
                    B(i) ← A(k);  i ← i + 1
                  repeat
             else for k ← h to mid do
                    B(i) ← A(k);  i ← i + 1
                  repeat
  endif
  for k ← low to high do  //copy the merged sets back into A//
    A(k) ← B(k)
  repeat
end MERGE
```

Algorithm 3.8 Merging two sorted sets using auxiliary storage

Before executing procedure MERGESORT, the n elements should be placed in $A(1{:}n)$ and the auxiliary array $B(1{:}n)$ should also be declared. Then **call** MERGESORT(1, n) will cause the keys to be rearranged into nondecreasing order in A.

Consider the array of ten elements A = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520). Procedure MERGESORT begins by splitting A into two subfiles of size five. The elements in $A(1{:}5)$ are then split into two subfiles of size three and two. Then the items in $A(1{:}3)$ are split into subfiles of size two and one. The two values in $A(1{:}2)$ are split a final time into one element subfiles and now the merging begins. Note that no actual movement of data has yet taken place. A record of the subfiles is implicitly

maintained by the recursive mechanism. Pictorially the file can now be viewed as

(310|285|179|652, 351|423, 861, 254, 450, 520)

with the vertical bars indicating the boundaries of subfiles. $A(1)$ and $A(2)$ are merged to yield

(285, 310|179|652, 351|423, 861, 254, 450, 520)

Then $A(3)$ is merged with $A(1{:}2)$ producing

(179, 285, 310|652, 351|423, 861, 254, 450, 520)

Next, elements $A(4)$ and $A(5)$ are merged

(179, 285, 310|351, 652|423, 861, 254, 450, 520)

followed by the merging of $A(1{:}3)$ and $A(4{:}5)$ to give

(179, 285, 310, 351, 652|423, 861, 254, 450, 520)

At this point the algorithm has returned to the first invocation of MERGESORT and it is about to process the second recursive call. Repeated recursive calls are invoked producing the following subfiles:

(179, 285, 310, 351, 652|423|861|254|450, 520)

$A(6)$ and $A(7)$ are merged and then $A(8)$ is merged with $A(6{:}7)$ giving

(179, 285, 310, 351, 652|254, 423, 861|450, 520)

Next $A(9)$ and $A(10)$ are merged followed by $A(6{:}8)$ and $A(9{:}10)$

(179, 285, 310, 351, 652|254, 423, 450, 520, 861)

At this point there are two sorted subfiles and the final merge produces the fully sorted result

(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	
A:	–	50,	10,	25,	30,	15,	70,	35,	55,	
LINK	0,	0,	0,	0,	0,	0,	0,	0,	0,	
q r p										
1 2 2	2	0	1	0	0	0	0	0	0	(10, 50)
3 4 3	3	0	1	4	0	0	0	0	0	(10, 50), (25, 30)
2 3 2	2	0	3	4	1	0	0	0	0	(10, 25, 30, 50)
5 6 5	5	0	3	4	1	6	0	0	0	(10, 25, 30, 50), (15, 70)
7 8 7	7	0	3	4	1	6	0	8	0	(10, 25, 30, 50), (15, 70), (35, 55)
5 7 5	5	0	3	4	1	7	0	8	6	(10, 25, 30, 50, 55, 70), (35, 55)
2 5 2	2	8	5	4	7	3	0	1	6	(10, 15, 25, 30, 35, 50, 55, 70)

Table 3.3 Example of how the LINK array changes when MERGESORT1 is applied to $A(1{:}8) = (50, 10, 25, 30, 15, 70, 35, 55)$.

3.5 QUICKSORT

The divide-and-conquer approach may be used to arrive at an efficient sorting method different from mergesort. In mergesort, the file $A(1{:}n)$ was divided at its midpoint into subfiles which were independently sorted and later merged. In quicksort, the division into two subfiles is made such that the sorted subfiles do not need to be later merged. This is accomplished by rearranging the elements in $A(1{:}n)$ such that $A(i) \leq A(j)$ for all i between 1 and m and all j between $m + 1$ and n for some m, $1 \leq m \leq n$. Thus, the elements in $A(1{:}m)$ and $A(m + 1{:}n)$ may be independently sorted. No merge is needed. The rearrangement of the elements is accomplished by picking some element of A, say $t = A(s)$, and then reordering the other elements so that all elements appearing before t in $A(1{:}n)$ are less than or equal to t and all elements appearing after t are greater than or equal to t. This rearranging is referred to as partitioning.

Procedure PARTITION of Algorithm 3.12 (due to C. A. R. Hoare) accomplishes an in-place partitioning of the elements of $A(m{:}p - 1)$. It is assumed that $A(p) \geq A(m)$ and that $A(m)$ is the partitioning element. If $m = 1$ and $p - 1 = n$ then $A(n + 1)$ must be defined and must be greater than or equal to those elements in $A(1{:}n)$. The assumption that $A(m)$ is the partition element is merely for convenience and we shall see that other choices for the partitioning element than the first item in the set will be better in practice. The procedure INTERCHANGE(x, y) performs the assignments: temp $\leftarrow x$; $x \leftarrow y$; $y \leftarrow$ temp.

```
procedure PARTITION(m, p)
  //Within A(m), A(m + 1), ..., A(p - 1) the elements are//
  //rearranged in such a way that if initially t = A(m),//
  //then after completion A(q) = t, for some q between m and p - 1,//
  //A(k) ≤ t for m ≤ k < q and A(k) ≥ t for q < k < p.//
  //The final value of p is q//
  integer m, p, i; global A(m: p)
  v ← A(m); i ← m    //A(m) is the partition element//
  loop
    loop i ← i + 1 until A(i) ≥ v repeat    //i moves left to right//
    loop p ← p - 1 until A(p) ≤ v repeat    //p moves right to left//
    if i < p
      then call INTERCHANGE(A(i), A(p))    //exchange A(i) and A(p)//
      else exit
    endif
  repeat
  A(m) ← A(p); A(p) ← v        //the partition element belongs at position p//
end PARTITION
```

Algorithm 3.12 Partition the set $A(m:p-1)$ about $A(m)$

As an example of how PARTITION works consider the following array of 9 elements. The procedure is initially invoked as **call** PARTITION(1, 10). The vertical bars connected by a horizontal line indicate those elements which were interchanged to produce the next row. $A(1) = 65$ is the partitioning element and it is eventually (in the sixth row) determined to be the 5th smallest element of the set. Notice that the remaining elements are unsorted but they are partitioned about $A(5) = 65$.

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

Using Hoare's clever method of partitioning a set of elements about a chosen element we can directly devise a divide-and-conquer method for completely sorting n elements. Following a call to procedure PARTITION two sets S_1 and S_2 are produced. All elements in S_1 are less than or equal to the elements in S_2. Hence S_1 and S_2 may be sorted independently. Each set will be sorted by reusing procedure PARTITION. Algorithm 3.13 describes the complete process as a program.


```
procedure QUICKSORT(p, q)
  //sorts the elements A(p), ..., A(q) which reside//
  //in the global array A(1:n) into ascending order;//
  //A(n + 1) is considered to be defined//
  //and must be ≥ all elements in A(p:q); A(n + 1) = +∞//
  integer p, q; global n, A(1:n)
  if p < q
    then j ← q + 1
      call PARTITION(p, j)
      call QUICKSORT(p, j - 1) //j is the position of the partitioning//
                               //element//
      call QUICKSORT(j + 1, q)
  endif
end QUICKSORT
```

Algorithm 3.13 Sorting by partitioning

Analysis of Quicksort

In analyzing QUICKSORT, we shall count only the number of element comparisons $C(n)$. It is easy to see that the frequency count of other operations is of the same order as $C(n)$. We make the following assumptions:

(i) the n elements to be sorted are distinct;
(ii) the partitioning element v in PARTITION is chosen using a random selection process.

If RANDOM(i, j) is a function that generates a random integer in the interval $[i, j]$, then the selection element is chosen by replacing the statements $v \leftarrow A(m)$; $i \leftarrow m$ in PARTITION by $i \leftarrow$ RANDOM$(m, p - 1)$; $v \leftarrow A(i)$; $A(i) \leftarrow A(m)$; $i \leftarrow m$.

First, let us obtain the worst case value $C_w(n)$ of $C(n)$. The number of element comparisons in each call of PARTITION is at most $p - m + 1$.

Chapter 4

THE GREEDY METHOD

4.1 THE GENERAL METHOD

The greedy method is perhaps the most straightforward design technique we shall be considering in this text, and what's more it can be applied to a wide variety of problems. Most, though not all, of these problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a *feasible* solution. We are required to find a feasible solution that either maximizes or minimizes a given *objective function*. A feasible solution that does this is called an *optimal solution*. There is usually an obvious way to determine a feasible solution, but not necessarily an optimal solution.

The greedy method suggests that one can devise an algorithm which works in stages, considering one input at a time. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure. This measure may or may not be the objective function. In fact, several different optimization measures may be plausible for a given problem. Most of these, however, will result in algorithms that generate suboptimal solutions.

We can describe the greedy method abstractly, but more precisely than above, by considering the following control abstraction.

```
procedure GREEDY(A,n)
  //A(1:n) contains the n inputs//
  solution ← φ  //initialize the solution to empty//
  for i ← 1 to n do
    x ← SELECT(A)
    if FEASIBLE(solution,x)
      then solution ← UNION(solution,x)
    endif
  repeat
  return (solution)
end GREEDY
```


Algorithm 4.1 Greedy method control abstraction

The function SELECT selects an input from A, removes it and assigns its value to x. FEASIBLE is a Boolean-valued function which determines if x can be included into the solution vector. UNION actually combines x with solution and updates the objective function. Procedure GREEDY describes the essential way that a greedy based algorithm will look, once a particular problem is chosen and the procedures SELECT, FEASIBLE and UNION are properly implemented.

4.2 OPTIMAL STORAGE ON TAPES

There are n programs that are to be stored on a computer tape of length L. Associated with each program i is a length l_i, $1 \leq i \leq n$. Clearly, all programs can be stored on the tape if and only if the sum of the lengths of the programs is at most L. We shall assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. Hence, if the programs are stored in the order $I = i_1, i_2, \ldots, i_n$, the time t_j needed to retrieve program i_j is proportional to $\sum_{1 \leq k \leq j} l_{i_k}$. If all programs are retrieved equally often then the expected or *mean retrieval time* (MRT) is $(1/n)\sum_{1 \leq j \leq n} t_j$. In the optimal storage on tape problem, we are required to find a permutation for the n programs so that when they are stored on the tape in this order the MRT is minimized. Minimizing the MRT is equivalent to minimizing $D(I) = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} l_{i_k}$.

4.3 KNAPSACK PROBLEM

Now, let us try to apply the greedy method to solve a more complex problem. This problem is the knapsack problem. We are given n objects and a knapsack. Object i has a weight w_i and the knapsack has a capacity M. If a fraction x_i, $0 \leq x_i \leq 1$, of object i is placed into the knapsack then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is M, we require the total weight of all chosen objects to be at most M. Formally, the problem may be stated as:

$$\text{maximize} \sum_{1 \leq i \leq n} p_i x_i \tag{4.1}$$

$$\text{subject to} \sum_{1 \leq i \leq n} w_i x_i \leq M \tag{4.2}$$

$$\text{and} \quad 0 \leq x_i \leq 1, \quad 1 \leq i \leq n \tag{4.3}$$

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying (4.2) and (4.3) above. An optimal solution is a feasible solution for which (4.1) is maximum.

Example 4.2 Consider the following instance of the knapsack problem: $n = 3, M = 20, (p_1, p_2, p_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\Sigma w_i x_i$	$\Sigma p_i x_i$
i)	(1/2, 1/3, 1/4)	16.5	24.25
ii)	(1, 2/15, 0)	20	28.2
iii)	(0, 2/3, 1)	20	31
iv)	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution (iv) yields the maximum profit. As we shall soon see, this solution is optimal for the given problem instance. □

In case the sum of all the weights is $\leq M$, then clearly $x_i = 1, 1 \leq i \leq n$ is an optimal solution. So, let us assume the sum of weights exceeds M. Now all the x_i's cannot be 1. Another observation to make is that all optimal solutions will fill the knapsack exactly. This is true because we can always increase by a fractional amount the contribution of some object i until the total weight is exactly M.

(Algorithm 4.3) obtains solutions corresponding to this strategy. Note that solutions corresponding to the first two strategies can be obtained using this algorithm if the objects are initially in the appropriate order. Disregarding the time to initially sort the objects, each of the three strategies outlined above requires only $O(n)$ time.

```
procedure GREEDY_KNAPSACK(P, W, M, X, n)
  //P(1:n) and W(1:n) contain the profits and weights respectively of the n//
  //objects ordered so that P(i)/W(i) ≥ P(i + 1)/W(i + 1). M is the//
  //knapsack size and X(1:n) is the solution vector//
  real P(1:n), W(1:n), X(1:n), M, cu;
  integer i, n;
  X ← 0  //initialize solution to zero//
  cu ← M  //cu = remaining knapsack capacity//
  for i ← 1 to n do
    if W(i) > cu then exit endif
    X(i) ← 1
    cu ← cu - W(i)
  repeat
  if i ≤ n then X(i) ← cu/W(i) endif
end GREEDY_KNAPSACK
```

Algorithm 4.3 Algorithm for greedy strategies for the knapsack problem

We have seen that when one applies the greedy method to the solution of the knapsack problem there are at least three different measures one can attempt to optimize when determining which object to include next. These measures are total profit, capacity used and the ratio of accumulated profit divided by capacity used. Once an optimization measure has been chosen, the greedy method suggests choosing objects for inclusion into the solution in such a way that each choice optimizes the measure at that time. Thus a greedy method using profit as its measure will at each step choose an object that increases the profit the most. If the capacity measure is used, the next object included will increase this the least. While greedy based algorithms using the first two measures do not guarantee optimal solutions for the knapsack problem, Theorem 4.3 shows that a greedy algorithm using the third strategy always obtains an optimal solution. This theorem is proved by comparing the greedy solution to any optimal solution. If the two solutions differ, then we find the first x_i at which they differ. Next, it

Chapter 5

DYNAMIC PROGRAMMING

5.1 THE GENERAL METHOD

Dynamic Programming is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions. In earlier chapters we have seen many problems that can be viewed this way. Some examples are:

Example 5.1 [Knapsack] The solution to the knapsack problem (Section 4.3) may be viewed as the result of a sequence of decisions. We have to decide the values of x_i, $1 \leq i \leq n$. First we may make a decision on x_1, then on x_2, then on x_3 etc. An optimal sequence of decisions will maximize the objective function $\sum p_i x_i$. (It will also satisfy the constraints $\sum w_i x_i \leq M$ and $0 \leq x_i \leq 1$.) □

Example 5.2 [Optimal Merge Patterns] This problem was discussed in Section 4.4. An optimal merge pattern tells us which pair of files should be merged at each step. As a decision sequence, the problem calls for us to decide which pair of files should be merged first; which pair second; which pair third, etc. An optimal sequence of decisions is a least cost sequence. □

Example 5.3 [Shortest Path] One way to find a shortest path from vertex i to vertex j in a directed graph G is to decide which vertex should be the second vertex, which the third, which the fourth; etc. until vertex j is reached. An optimal sequence of decisions is one which results in a path of least length. □

For some of the problems that may be viewed in this way, an optimal sequence of decisions may be found by making the decisions one at a time and never making an erroneous decision. This is true for all problems solv-

$A^{(0)}$	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

$A^{(1)}$	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

$A^{(2)}$	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

$A^{(3)}$	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

Figure 5.5 Matrices A^k produced by ALL_PATHS for the digraph of Figure 5.4

The time needed by procedure ALL_PATHS is especially easy to determine because the looping is independent of the data in the matrix A. Line 9 is iterated n^3 times and so the time for procedure ALL_PATHS is $\theta(n^3)$. An exercise examines the extensions needed to actually obtain the i to j paths with these lengths. Some speed-up can be obtained by noticing that the innermost **for** loop need be executed only when $A(i, k)$ and $A(k, j)$ are not equal to ∞.

5.4 OPTIMAL BINARY SEARCH TREES

Definition A *binary search tree* T is a binary tree; either it is empty or each node in the tree contains an identifier and

(i) all identifiers in the left subtree of T are less (numerically or alphabetically) than the identifier in the root node T;

(ii) all identifiers in the right subtree are greater than the identifier in the root node T;

(iii) the left and right subtrees of T are also binary search trees.

Note that the definition of a binary search tree requires that all identifiers in the tree be distinct. For a given set of identifiers, several different binary search trees are possible. Figure 5.6 shows two possible binary search trees for a subset of the reserved words of SPARKS.

To determine whether an identifier X is present in a binary search tree, X is compared with the root. If X is less than the identifier in the root, then the search continues in the left subtree; if X equals the identifier in the root, the search terminates successfully; otherwise the search continues in the right subtree. This is formalized in procedure SEARCH.

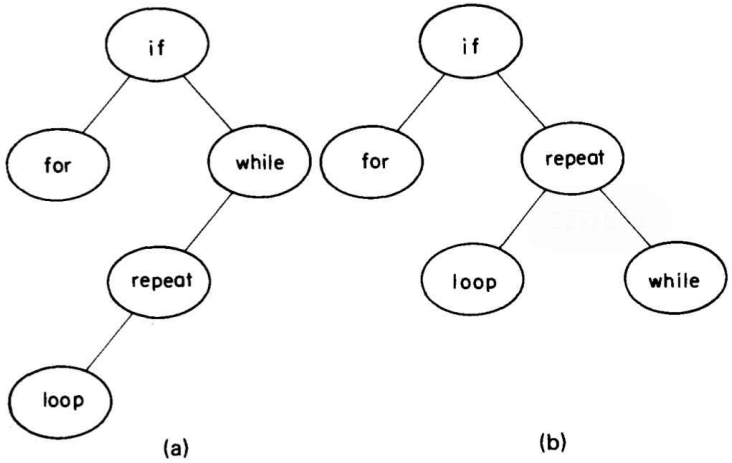


Figure 5.6 Two possible binary search trees

```
     procedure SEARCH(T, X, i)
       //Search the binary search tree T for X. Each node of the tree has//
       //fields LCHILD, IDENT, RCHILD. If X is not in T then set i = //
       //0. Otherwise, set i such that IDENT(i) = X.//
1      i ← T
2      while i ≠ 0 do
3        case
4             :X < IDENT(i): i ← LCHILD(i)   //search left subtree//
5             :X = IDENT(i): return
6             :X > IDENT(i): i ← RCHILD(i)   //search right subtree//
7        endcase
8      repeat
9    end SEARCH
```

Algorithm 5.4 Searching a binary search tree

Given a fixed set of identifiers, we wish to create a binary search tree organization. We may expect different binary search trees for the same identifier set to have different performance characteristics. The tree of Figure 5.6(a), in the worst case, requires four comparisons to find an iden-

the search to the range $R(i, j-1) \le k \le R(i+1, j)$. In this case the computing time becomes $O(n^2)$ (see exercises). Procedure OBST (Algorithm 5.5) uses this result to obtain in $O(n^2)$ time the values of $W(i, j)$, $R(i, j)$ and $C(i, j)$, $0 \le i \le j \le n$. The actual tree T_{0n} may be constructed from the values of $R(i, j)$ in $O(n)$ time. The algorithm for this is left as an exercise.

```
procedure OBST(P, Q, n)
  //Given n distinct identifiers a_1 < a_2 < ... < a_n and probabilities//
  //P(i), 1 ≤ i ≤ n and Q(i), 0 ≤ i ≤ n this algorithm computes the cost//
  //C(i, j) of optimal binary search trees T_ij for identifiers a_{i+1}, ..., a_j.//
  //It also computes R(i, j), the root of T_ij. W(i, j) is the weight of T_ij//
  real P(n), Q(0:n), C(0:n, 0:n), W(0:n, 0:n)
  integer R(0:n, 0:n)
  for i ← 0 to n - 1 do
    (W(i, i), R(i, i), C(i, i)) ← (Q(i), 0, 0)   //initialize//
    (W(i, i + 1), R(i, i + 1), C(i, i + 1)) ← (Q(i) + Q(i + 1) + P(i + 1),
      i + 1, Q(i) + Q(i + 1) + P(i + 1))   //optimal trees with one node//
  repeat
  (W(n, n), R(n, n), C(n, n)) ← (Q(n), 0, 0)
  for m ← 2 to n do   //find optimal trees with m nodes//
    for i ← 0 to n - m do
      j ← i + m
      W(i, j) ← W(i, j - 1) + P(j) + Q(j)
      k ← a value of l in the range R(i, j - 1) ≤ l ≤ R(i + 1, j) that
        minimizes {C(i, l - 1) + C(l, j)}   //solve (5.12) using Knuth's//
                                            //result//
      C(i, j) ← W(i, j) + C(i, k - 1) + C(k, j)
      R(i, j) ← k
    repeat
  repeat
end OBST
```

Algorithm 5.5 Finding a minimum cost binary search tree

5.5 0/1—KNAPSACK

The terminology and notation used in this section is the same as in section 5.1. A solution to the knapsack problem may be obtained by making a sequence of decisions on the variables $x_1, x_2, \ldots, x_n$. A decision on variable x_i involves deciding which of the values 0 or 1 is to be assigned to it. Let us

a. Give an example of three matrices for which the number of multiplications in $(A_1 \cdot A_2) \cdot A_3$ and $A_1 \cdot (A_2 \cdot A_3)$ differ at least by a factor of 1000.

b. How many different ways are there to compute the product of n matrices?

c. Design a dynamic programming algorithm for finding an optimal order of multiplying n matrices.

4 Warshall's and Floyd's Algorithms

In this section, we look at two well-known algorithms: Warshall's algorithm for computing the transitive closure of a directed graph and Floyd's algorithm for the all-pairs shortest-paths problem. These algorithms are based on essentially the same idea: exploit a relationship between a problem and its simpler rather than smaller version. Warshall and Floyd published their algorithms without mentioning dynamic programming. Nevertheless, the algorithms certainly have a dynamic programming flavor and have come to be considered applications of this technique.

Warshall's Algorithm

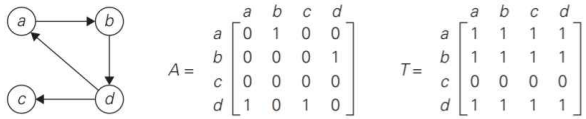
Recall that the adjacency matrix $A = \{a_{ij}\}$ of a directed graph is the boolean matrix that has 1 in its ith row and jth column if and only if there is a directed edge from the ith vertex to the jth vertex. We may also be interested in a matrix containing the information about the existence of directed paths of arbitrary lengths between vertices of a given graph. Such a matrix, called the transitive closure of the digraph, would allow us to determine in constant time whether the jth vertex is reachable from the ith vertex.

Here are a few application examples. When a value in a spreadsheet cell is changed, the spreadsheet software must know all the other cells affected by the change. If the spreadsheet is modeled by a digraph whose vertices represent the spreadsheet cells and edges indicate cell dependencies, the transitive closure will provide such information. In software engineering, transitive closure can be used for investigating data flow and control flow dependencies as well as for inheritance testing of object-oriented software. In electronic engineering, it is used for redundancy identification and test generation for digital circuits.

DEFINITION The ***transitive closure*** of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{t_{ij}\}$, in which the element in the ith row and the jth column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, t_{ij} is 0.

An example of a digraph, its adjacency matrix, and its transitive closure is given in Figure 8.11.

We can generate the transitive closure of a digraph with the help of depth-first search or breadth-first search. Performing either traversal starting at the ith



(a) (b) (c)

FIGURE 8.11 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

vertex gives the information about the vertices reachable from it and hence the columns that contain 1's in the ith row of the transitive closure. Thus, doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.

Since this method traverses the same digraph several times, we should hope that a better algorithm can be found. Indeed, such an algorithm exists. It is called ***Warshall's algorithm*** after Stephen Warshall, who discovered it [War62]. It is convenient to assume that the digraph's vertices and hence the rows and columns of the adjacency matrix are numbered from 1 to n. Warshall's algorithm constructs the transitive closure through a series of $n \times n$ boolean matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}. \quad \textbf{(8.9)}$$

Each of these matrices provides certain information about directed paths in the digraph. Specifically, the element $r_{ij}^{(k)}$ in the ith row and jth column of matrix $R^{(k)}$ $(i, j = 1, 2, \ldots, n, k = 0, 1, \ldots, n)$ is equal to 1 if and only if there exists a directed path of a positive length from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k. Thus, the series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph. (Recall that the adjacency matrix contains the information about one-edge paths, i.e., paths with no intermediate vertices.) $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate; thus, with more freedom, so to speak, it may contain more 1's than $R^{(0)}$. In general, each subsequent matrix in series (8.9) has one more vertex to use as intermediate for its paths than its predecessor and hence may, but does not have to, contain more 1's. The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.

The central point of the algorithm is that we can compute all the elements of each matrix $R^{(k)}$ from its immediate predecessor $R^{(k-1)}$ in series (8.9). Let $r_{ij}^{(k)}$, the element in the ith row and jth column of matrix $R^{(k)}$, be equal to 1. This means that there exists a path from the ith vertex v_i to the jth vertex v_j with each intermediate vertex numbered not higher than k:

v_i, a list of intermediate vertices each numbered not higher than k, v_j. **(8.10)**

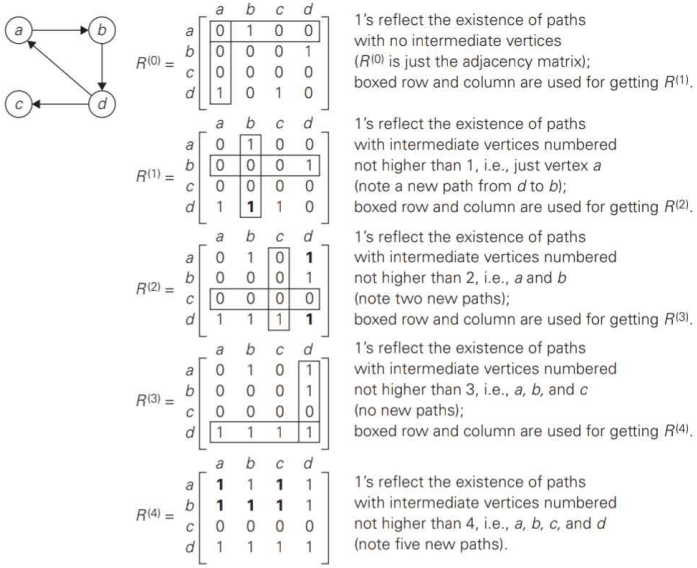


FIGURE 8.13 Application of Warshall's algorithm to the digraph shown. New 1's are in bold.

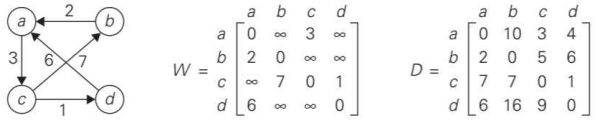
335/593

Here is pseudocode of Warshall's algorithm.

ALGORITHM *Warshall*$(A[1..n, 1..n])$

```
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
R^(0) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R^(k)[i, j] ← R^(k−1)[i, j] or (R^(k−1)[i, k] and R^(k−1)[k, j])
return R^(n)
```

Several observations need to be made about Warshall's algorithm. First, it is remarkably succinct, is it not? Still, its time efficiency is only $\Theta(n^3)$. In fact, for sparse graphs represented by their adjacency lists, the traversal-based algorithm



(a) (b) (c)

FIGURE 8.14 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

mentioned at the beginning of this section has a better asymptotic efficiency than Warshall's algorithm (why?). We can speed up the above implementation of Warshall's algorithm for some inputs by restructuring its innermost loop (see Problem 4 in this section's exercises). Another way to make the algorithm run faster is to treat matrix rows as bit strings and employ the bitwise *or* operation available in most modern computer languages.

As to the space efficiency of Warshall's algorithm, the situation is similar to that of computing a Fibonacci number and some other dynamic programming algorithms. Although we used separate matrices for recording intermediate results of the algorithm, this is, in fact, unnecessary. Problem 3 in this section's exercises asks you to find a way of avoiding this wasteful use of the computer memory. Finally, we shall see below how the underlying idea of Warshall's algorithm can be applied to the more general problem of finding lengths of shortest paths in weighted graphs.

Floyd's Algorithm for the All-Pairs Shortest-Paths Problem

Given a weighted connected graph (undirected or directed) ***hort***
paths problem asks to find the distances—i.e., the length: atl
from each vertex to all other vertices. This is one of se s of
problem involving shortest paths in graphs. Because of its important applications to communications, transportation networks, and operations research, it has been thoroughly studied over the years. Among recent applications of the all-pairs shortest-path problem is precomputing distances for motion planning in computer games.

It is convenient to record the lengths of shortest paths in an $n \times n$ matrix D called the ***distance matrix***: the element d_{ij} in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex. For an example, see Figure 8.14.

We can generate the distance matrix with an algorithm that is very similar to Warshall's algorithm. It is called ***Floyd's algorithm*** after its co-inventor Robert W. Floyd.[1] It is applicable to both undirected and directed weighted graphs provided

1. Floyd explicitly referenced Warshall's paper in presenting his algorithm [Flo62]. Three years earlier, Bernard Roy published essentially the same algorithm in the proceedings of the French Academy of Sciences [Roy59].

that they do not contain a cycle of a negative length. (The distance between any two