

DESIGN AND ANALYSIS OF ALGORITHMS

UNIT - 1

INTRODUCTION ,

ELEMENTARY DATA STRUCTURES

Chapter 1

INTRODUCTION

1.1 WHAT IS AN ALGORITHM?

The word algorithm comes from the name of a Persian author, Abu Ja'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.), who wrote a textbook on mathematics. This word has taken on a special significance in computer science, where “algorithm” has come to refer to a method that can be used by a computer for the solution of a problem. This is what makes algorithm different from words such as process, technique, or method.

Definition 1.1 [Algorithm]: An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. **Input.** Zero or more quantities are externally supplied.
2. **Output.** At least one quantity is produced.
3. **Definiteness.** Each instruction is clear and unambiguous.
4. **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness.** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible. □

An algorithm is composed of a finite set of steps, each of which may require one or more operations. The possibility of a computer carrying out these operations necessitates that certain constraints be placed on the type of operations an algorithm can include.

Criteria 1 and 2 require that an algorithm produce one or more *outputs* and have zero or more *inputs* that are externally supplied. According to criterion 3, each operation must be *definite*, meaning that it must be perfectly clear what should be done. Directions such as “add 6 or 7 to x ” or “compute $5/0$ ” are not permitted because it is not clear which of the two possibilities should be done or what the result is.

The fourth criterion for algorithms we assume in this book is that they *terminate* after a finite number of operations. A related consideration is that the time for termination should be reasonably short. For example, an algorithm could be devised that decides whether any given position in the game of chess is a winning position. The algorithm works by examining all possible moves and countermoves that could be made from the starting position. The difficulty with this algorithm is that even using the most modern computers, it may take billions of years to make the decision. We must be very concerned with analyzing the efficiency of each of our algorithms.

Criterion 5 requires that each operation be *effective*; each step must be such that it can, at least in principle, be done by a person using pencil and paper in a finite amount of time. Performing arithmetic on integers is an example of an effective operation, but arithmetic with real numbers is not, since some values may be expressible only by infinitely long decimal expansion. Adding two such numbers would violate the effectiveness property.

Algorithms that are definite and effective are also called *computational procedures*. One important example of computational procedures is the operating system of a digital computer. This procedure is designed to control the execution of jobs, in such a way that when no jobs are available, it does not terminate but continues in a waiting state until a new job is entered. Though computational procedures include important examples such as this one, we restrict our study to computational procedures that always terminate.

To help us achieve the criterion of definiteness, algorithms are written in a programming language. Such languages are designed so that each legitimate sentence has a unique meaning. A *program* is the expression of an algorithm in a programming language. Sometimes words such as procedure, function, and subroutine are used synonymously for program. Most readers of this book have probably already programmed and run some algorithms on a computer. This is desirable because before you study a concept in general, it helps if you had some practical experience with it. Perhaps you had some difficulty getting started in formulating an initial solution to a problem, or perhaps you were unable to decide which of two algorithms was better. The goal of this book is to teach you how to make these decisions.

The study of algorithms includes many important and active areas of research. There are four distinct areas of study one can identify:

1. *How to devise algorithms* — Creating an algorithm is an art which may never be fully automated. A major goal of this book is to study vari-

ous design techniques that have proven to be useful in that they have often yielded good algorithms. By mastering these design strategies, it will become easier for you to devise new and useful algorithms. Many of the chapters of this book are organized around what we believe are the major methods of algorithm design. The reader may now wish to glance back at the table of contents to see what these methods are called. Some of these techniques may already be familiar, and some have been found to be so useful that books have been written about them. Dynamic programming is one such technique. Some of the techniques are especially useful in fields other than computer science such as operations research and electrical engineering. In this book we can only hope to give an introduction to these many approaches to algorithm formulation. All of the approaches we consider have applications in a variety of areas including computer science. But some important design techniques such as linear, nonlinear, and integer programming are not covered here as they are traditionally covered in other courses.

2. *How to validate algorithms* — Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. We refer to this process as *algorithm validation*. The algorithm need not as yet be expressed as a program. It is sufficient to state it in any precise way. The purpose of the validation is to assure us that this algorithm will work correctly independently of the issues concerning the programming language it will eventually be written in. Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is referred to as *program proving* or sometimes as *program verification*. A proof of correctness requires that the solution be stated in two forms. One form is usually as a program which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in the predicate calculus. The second form is called a *specification*, and this may also be expressed in the predicate calculus. A proof consists of showing that these two forms are equivalent in that for every given legal input, they describe the same output. A complete proof of program correctness requires that each statement of the programming language be precisely defined and all basic operations be proved correct. All these details may cause a proof to be very much longer than the program.

3. *How to analyze algorithms* — This field of study is called analysis of algorithms. As an algorithm is executed, it uses the computer's central processing unit (CPU) to perform operations and its memory (both immediate and auxiliary) to hold the program and data. *Analysis of algorithms* or *performance analysis* refers to the task of determining how much computing time and storage an algorithm requires. This is a challenging area which sometimes requires great mathematical skill. An important result of this study is that it allows you to make quantitative judgments about the value of one algorithm over another. Another result is that it allows you to predict whether the software will meet any efficiency constraints that exist.

Questions such as how well does an algorithm perform in the best case, in the worst case, or on the average are typical. For each algorithm in the text, an analysis is also given. Analysis is more fully described in Section 1.3.2.

4. *How to test a program* — Testing a program consists of two phases: debugging and profiling (or performance measurement). *Debugging* is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them. However, as E. Dijkstra has pointed out, “debugging can only point to the presence of errors, but not to their absence.” In cases in which we cannot verify the correctness of output on sample data, the following strategy can be employed: let more than one programmer develop programs for the same problem, and compare the outputs produced by these programs. If the outputs match, then there is a good chance that they are correct. A proof of correctness is much more valuable than a thousand tests (if that proof is correct), since it guarantees that the program will work correctly for all possible inputs. *Profiling* or *performance measurement* is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results. These timing figures are useful in that they may confirm a previously done analysis and point out logical places to perform useful optimization. A description of the measurement of timing complexity can be found in Section 1.3.5. For some of the algorithms presented here, we show how to devise a range of data sets that will be useful for debugging and profiling.

These four categories serve to outline the questions we ask about algorithms throughout this book. As we can’t hope to cover all these subjects completely, we content ourselves with concentrating on design and analysis, spending less time on program construction and correctness.

EXERCISES

1. Look up the words algorism and algorithm in your dictionary and write down their meanings.
2. The name al-Khowarizmi (algorithm) literally means “from the town of Khowarazm.” This city is now known as Khiva, and is located in Uzbekistan. See if you can find this country in an atlas.
3. Use the WEB to find out more about al-Khowarizmi, e.g., his dates, a picture, or a stamp.

1.2 ALGORITHM SPECIFICATION

1.2.1 Pseudocode Conventions

In computational theory, we distinguish between an algorithm and a program. The latter does not have to satisfy the finiteness condition. For example, we can think of an operating system that continues in a “wait” loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs always terminate, we use “algorithm” and “program” interchangeably in this text.

We can describe an algorithm in many ways. We can use a natural language like English, although if we select this option, we must make sure that the resulting instructions are definite. Graphic representations called *flowcharts* are another possibility, but they work well only if the algorithm is small and simple. In this text we present most of our algorithms using a pseudocode that resembles C and Pascal.

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces: { and }. A compound statement (i.e., a collection of simple statements) can be represented as a block. The body of a procedure also forms a block. Statements are delimited by ;.
3. An identifier begins with a letter. The data types of variables are not explicitly declared. The types will be clear from the context. Whether a variable is global or local to a procedure will also be evident from the context. We assume simple data types such as integer, float, char, boolean, and so on. Compound data types can be formed with **records**. Here is an example:

```
node = record
    {   datatype_1  data_1;
        :
        datatype_n  data_n;
        node          *link;
    }
```

In this example, *link* is a pointer to the record type *node*. Individual data items of a record can be accessed with → and period. For instance if *p* points to a record of type *node*, *p* → *data_1* stands for the value of the first field in the record. On the other hand, if *q* is a record of type *node*, *q.data_1* will denote its first field.

4. Assignment of values to variables is done using the assignment statement

$$\langle \text{variable} \rangle := \langle \text{expression} \rangle;$$

5. There are two boolean values **true** and **false**. In order to produce these values, the logical operators **and**, **or**, and **not** and the relational operators $<$, \leq , $=$, \neq , \geq , and $>$ are provided.
6. Elements of multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the (i, j) th element of the array is denoted as $A[i, j]$. Array indices start at zero.
7. The following looping statements are employed: **for**, **while**, and **repeat-until**. The **while** loop takes the following form:

```
while <condition> do
{
    <statement 1>
    :
    <statement n>
}
```

As long as $\langle \text{condition} \rangle$ is **true**, the statements get executed. When $\langle \text{condition} \rangle$ becomes **false**, the loop is exited. The value of $\langle \text{condition} \rangle$ is evaluated at the top of the loop.

The general form of a **for** loop is

```
for variable := value1 to value2 step step do
{
    <statement 1>
    :
    <statement n>
}
```

Here value1 , value2 , and step are arithmetic expressions. A variable of type integer or real or a numerical constant is a simple form of an arithmetic expression. The clause “**step step**” is optional and taken as $+1$ if it does not occur. step could either be positive or negative. variable is tested for termination at the start of each iteration. The **for** loop can be implemented as a **while** loop as follows:

```

variable := value1;
fin := value2;
incr := step;
while ((variable − fin) * step ≤ 0) do
{
    ⟨statement 1⟩
    :
    ⟨statement n⟩
    variable := variable + incr;
}

```

A **repeat-until** statement is constructed as follows:

```

repeat
    ⟨statement 1⟩
    :
    ⟨statement n⟩
until ⟨condition⟩

```

The statements are executed as long as ⟨*condition*⟩ is **false**. The value of ⟨*condition*⟩ is computed after executing the statements.

The instruction **break;** can be used within any of the above looping instructions to force exit. In case of nested loops, **break;** results in the exit of the innermost loop that it is a part of. A **return** statement within any of the above also will result in exiting the loops. A **return** statement results in the exit of the function itself.

8. A conditional statement has the following forms:

```

if ⟨condition⟩ then ⟨statement⟩
if ⟨condition⟩ then ⟨statement 1⟩ else ⟨statement 2⟩

```

Here ⟨*condition*⟩ is a boolean expression and ⟨*statement*⟩, ⟨*statement 1*⟩, and ⟨*statement 2*⟩ are arbitrary statements (simple or compound).

We also employ the following **case** statement:

```

case
{
    :⟨condition 1⟩: ⟨statement 1⟩
    :
    :⟨condition n⟩: ⟨statement n⟩
    :else: ⟨statement n + 1⟩
}

```

Here $\langle \text{statement } 1 \rangle$, $\langle \text{statement } 2 \rangle$, etc. could be either simple statements or compound statements. A **case** statement is interpreted as follows. If $\langle \text{condition } 1 \rangle$ is **true**, $\langle \text{statement } 1 \rangle$ gets executed and the **case** statement is exited. If $\langle \text{statement } 1 \rangle$ is **false**, $\langle \text{condition } 2 \rangle$ is evaluated. If $\langle \text{condition } 2 \rangle$ is **true**, $\langle \text{statement } 2 \rangle$ gets executed and the **case** statement exited, and so on. If none of the conditions $\langle \text{condition } 1 \rangle, \dots, \langle \text{condition } n \rangle$ are true, $\langle \text{statement } n+1 \rangle$ is executed and the **case** statement is exited. The **else** clause is optional.

9. Input and output are done using the instructions **read** and **write**. No format is used to specify the size of input or output quantities.
10. There is only one type of procedure: **Algorithm**. An algorithm consists of a heading and a body. The heading takes the form

Algorithm *Name* ($\langle \text{parameter list} \rangle$)

where *Name* is the name of the procedure and ($\langle \text{parameter list} \rangle$) is a listing of the procedure parameters. The body has one or more (simple or compound) statements enclosed within braces { and }. An algorithm may or may not return any values. Simple variables to procedures are passed by value. Arrays and records are passed by reference. An array name or a record name is treated as a pointer to the respective data type.

As an example, the following algorithm finds and returns the maximum of n given numbers:

```

1  Algorithm Max(A, n)
2  // A is an array of size n.
3  {
4      Result := A[1];
5      for i := 2 to n do
6          if A[i] > Result then Result := A[i];
7      return Result;
8  }
```

In this algorithm (named **Max**), *A* and *n* are procedure parameters. *Result* and *i* are local variables.

Next we present two examples to illustrate the process of translating a problem into an algorithm.

Example 1.1 [Selection sort] Suppose we must devise an algorithm that sorts a collection of $n \geq 1$ elements of arbitrary type. A simple solution is given by the following

From those elements that are currently unsorted, find the smallest and place it next in the sorted list.

Although this statement adequately describes the sorting problem, it is not an algorithm because it leaves several questions unanswered. For example, it does not tell us where and how the elements are initially stored or where we should place the result. We assume that the elements are stored in an array a , such that the i th integer is stored in the i th position $a[i]$, $1 \leq i \leq n$. Algorithm 1.1 is our first attempt at deriving a solution.

```

1  for  $i := 1$  to  $n$  do
2  {
3      Examine  $a[i]$  to  $a[n]$  and suppose
4      the smallest element is at  $a[j]$ ;
5      Interchange  $a[i]$  and  $a[j]$ ;
6  }

```

Algorithm 1.1 Selection sort algorithm

To turn Algorithm 1.1 into a pseudocode program, two clearly defined subtasks remain: finding the smallest element (say $a[j]$) and interchanging it with $a[i]$. We can solve the latter problem using the code

$$t := a[i]; a[i] := a[j]; a[j] := t;$$

The first subtask can be solved by assuming the minimum is $a[i]$, checking $a[i]$ with $a[i + 1], a[i + 2], \dots$, and, whenever a smaller element is found, regarding it as the new minimum. Eventually $a[n]$ is compared with the current minimum, and we are done. Putting all these observations together, we get the algorithm `SelectionSort` (Algorithm 1.2).

The obvious question to ask at this point is, Does `SelectionSort` work correctly? Throughout this text we use the notation $a[i : j]$ to denote the array elements $a[i]$ through $a[j]$.

Theorem 1.1 Algorithm `SelectionSort(a, n)` correctly sorts a set of $n \geq 1$ elements; the result remains in $a[1 : n]$ such that $a[1] \leq a[2] \leq \dots \leq a[n]$.

Proof: We first note that for any i , say $i = q$, following the execution of lines 6 to 9, it is the case that $a[q] \leq a[r]$, $q < r \leq n$. Also observe that when i becomes greater than q , $a[1 : q]$ is unchanged. Hence, following the last execution of these lines (that is, $i = n$), we have $a[1] \leq a[2] \leq \dots \leq a[n]$.

We observe at this point that the upper limit of the **for** loop in line 4 can be changed to $n - 1$ without damaging the correctness of the algorithm. \square

```

1  Algorithm SelectionSort( $a, n$ )
2  // Sort the array  $a[1 : n]$  into nondecreasing order.
3  {
4      for  $i := 1$  to  $n$  do
5      {
6           $j := i;$ 
7          for  $k := i + 1$  to  $n$  do
8              if ( $a[k] < a[j]$ ) then  $j := k;$ 
9               $t := a[i]; a[i] := a[j]; a[j] := t;$ 
10     }
11 }
```

Algorithm 1.2 Selection sort**1.2.2 Recursive Algorithms**

A recursive function is a function that is defined in terms of itself. Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body. An algorithm that calls itself is *direct recursive*. Algorithm \mathcal{A} is said to be *indirect recursive* if it calls another algorithm which in turn calls \mathcal{A} . These recursive mechanisms are extremely powerful, but even more importantly, many times they can express an otherwise complex process very clearly. For these reasons we introduce recursion here.

Typically, beginning programmers view recursion as a somewhat mystical technique that is useful only for some very special class of problems (such as computing factorials or Ackermann's function). This is unfortunate because any algorithm that can be written using assignment, the **if-then-else** statement, and the **while** statement can also be written using assignment, the **if-then-else** statement, and recursion. Of course, this does not say that the resulting algorithm will necessarily be easier to understand. However, there are many instances when this will be the case. When is recursion an appropriate mechanism for algorithm exposition? One instance is when the problem itself is recursively defined. Factorial fits this category, as well as binomial coefficients, where

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} = \frac{n!}{m!(n-m)!}$$

The following two examples show how to develop a recursive algorithm. In the first example, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.

Example 1.2 [Towers of Hanoi] The Towers of Hanoi puzzle is fashioned after the ancient Tower of Brahma ritual (see Figure 1.1). According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks. The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top. Besides this tower there were two other diamond towers (labeled B and C). Since the time of creation, Brahman priests have been attempting to move the disks from tower A to tower B using tower C for intermediate storage. As the disks are very heavy, they can be moved only one at a time. In addition, at no time can a disk be on top of a smaller disk. According to legend, the world will come to an end when the priests have completed their task.

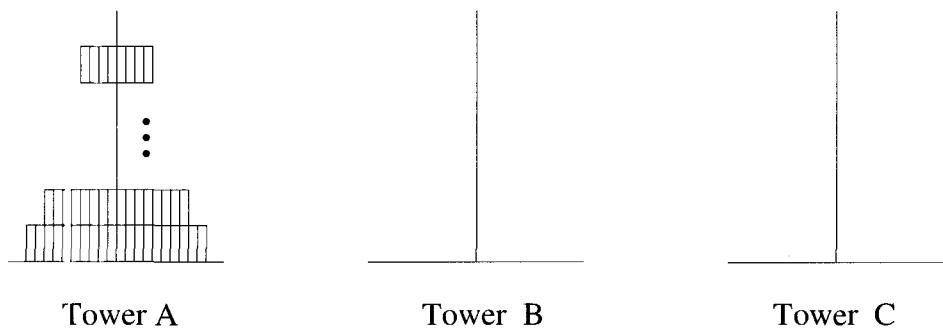


Figure 1.1 Towers of Hanoi

A very elegant solution results from the use of recursion. Assume that the number of disks is n . To get the largest disk to the bottom of tower B, we move the remaining $n - 1$ disks to tower C and then move the largest to tower B. Now we are left with the task of moving the disks from tower C to tower B. To do this, we have towers A and B available. The fact that tower B has a disk on it can be ignored as the disk is larger than the disks being moved from tower C and so any disk can be placed on top of it. The recursive nature of the solution is apparent from Algorithm 1.3. This algorithm is invoked by `TowersOfHanoi(n , A, B, C)`. Observe that our solution for an n -disk problem is formulated in terms of solutions to two $(n - 1)$ -disk problems. \square

Example 1.3 [Permutation generator] Given a set of $n \geq 1$ elements, the problem is to print all possible permutations of this set. For example, if the set is $\{a, b, c\}$, then the set of permutations is $\{(a, b, c), (a, c, b), (b, a, c),$

```

1  Algorithm TowersOfHanoi( $n, x, y, z$ )
2  // Move the top  $n$  disks from tower  $x$  to tower  $y$ .
3  {
4      if ( $n \geq 1$ ) then
5      {
6          TowersOfHanoi( $n - 1, x, z, y$ );
7          write ("move top disk from tower",  $x$ ,
8          "to top of tower",  $y$ );
9          TowersOfHanoi( $n - 1, z, y, x$ );
10     }
11 }
```

Algorithm 1.3 Towers of Hanoi

$(b, c, a), (c, a, b), (c, b, a)\}$. It is easy to see that given n elements, there are $n!$ different permutations. A simple algorithm can be obtained by looking at the case of four elements (a, b, c, d) . The answer can be constructed by writing

1. a followed by all the permutations of (b, c, d)
2. b followed by all the permutations of (a, c, d)
3. c followed by all the permutations of (a, b, d)
4. d followed by all the permutations of (a, b, c)

The expression “followed by all the permutations” is the clue to recursion. It implies that we can solve the problem for a set with n elements if we have an algorithm that works on $n - 1$ elements. These considerations lead to Algorithm 1.4, which is invoked by $\text{Perm}(a, 1, n)$. Try this algorithm out on sets of length one, two, and three to ensure that you understand how it works. \square

EXERCISES

1. Horner’s rule is a means for evaluating a polynomial at a point x_0 using a minimum number of multiplications. If the polynomial is $A(x) = a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0$, Horner’s rule is

```

1  Algorithm Perm( $a, k, n$ )
2  {
3      if ( $k = n$ ) then write ( $a[1 : n]$ ); // Output permutation.
4      else //  $a[k : n]$  has more than one permutation.
5          // Generate these recursively.
6          for  $i := k$  to  $n$  do
7              {
8                   $t := a[k]; a[k] := a[i]; a[i] := t;$ 
9                  Perm( $a, k + 1, n$ );
10                 // All permutations of  $a[k + 1 : n]$ 
11                  $t := a[k]; a[k] := a[i]; a[i] := t;$ 
12             }
13 }
```

Algorithm 1.4 Recursive permutation generator

$$A(x_0) = (\cdots (a_n x_0 + a_{n-1}) x_0 + \cdots + a_1) x_0 + a_0$$

Write an algorithm to evaluate a polynomial using Horner's rule.

2. Given n boolean variables x_1, x_2, \dots , and x_n , we wish to print all possible combinations of truth values they can assume. For instance, if $n = 2$, there are four possibilities: true, true; true, false; false, true; and false, false. Write an algorithm to accomplish this.
3. Devise an algorithm that inputs three integers and outputs them in nondecreasing order.
4. Present an algorithm that searches an unsorted array $a[1 : n]$ for the element x . If x occurs, then return a position in the array; else return zero.
5. The factorial function $n!$ has value 1 when $n \leq 1$ and value $n * (n - 1)!$ when $n > 1$. Write both a recursive and an iterative algorithm to compute $n!$.
6. The Fibonacci numbers are defined as $f_0 = 0$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$ for $i > 1$. Write both a recursive and an iterative algorithm to compute f_i .
7. Give both a recursive and an iterative algorithm to compute the binomial coefficient $\binom{n}{m}$ as defined in Section 1.2.2, where $\binom{n}{0} = \binom{n}{n} = 1$.

8. Ackermann's function $A(m, n)$ is defined as follows:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{otherwise} \end{cases}$$

This function is studied because it grows very fast for small values of m and n . Write a recursive algorithm for computing this function. Then write a nonrecursive algorithm for computing it.

9. The *pigeonhole principle* states that if a function f has n distinct inputs but less than n distinct outputs, then there exist two inputs a and b such that $a \neq b$ and $f(a) = f(b)$. Present an algorithm to find a and b such that $f(a) = f(b)$. Assume that the function inputs are $1, 2, \dots$, and n .
10. Give an algorithm to solve the following problem: Given n , a positive integer, determine whether n is the sum of all of its divisors, that is, whether n is the sum of all t such that $1 \leq t < n$, and t divides n .
11. Consider the function $F(x)$ that is defined by “if x is even, then $F(x) = x/2$; else $F(x) = F(F(3x + 1))$.” Prove that $F(x)$ terminates for all integers x . (*Hint:* Consider integers of the form $(2i + 1)2^k - 1$ and use induction.)
12. If S is a set of n elements, the *powerset* of S is the set of all possible subsets of S . For example, if $S = \{a, b, c\}$, then $\text{powerset}(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Write a recursive algorithm to compute $\text{powerset}(S)$.

1.3 PERFORMANCE ANALYSIS

One goal of this book is to develop skills for making evaluative judgments about algorithms. There are many criteria upon which we can judge an algorithm. For instance:

1. Does it do what we want it to do?
2. Does it work correctly according to the original specifications of the task?
3. Is there documentation that describes how to use it and how it works?

4. Are procedures created in such a way that they perform logical sub-functions?
5. Is the code readable?

These criteria are all vitally important when it comes to writing software, most especially for large systems. Though we do not discuss how to reach these goals, we try to achieve them throughout this book with the pseudocode algorithms we write. Hopefully this more subtle approach will gradually infect your own program-writing habits so that you will automatically strive to achieve these goals.

There are other criteria for judging algorithms that have a more direct relationship to performance. These have to do with their computing time and storage requirements.

Definition 1.2 [Space/Time complexity] The *space complexity* of an algorithm is the amount of memory it needs to run to completion. The *time complexity* of an algorithm is the amount of computer time it needs to run to completion. \square

Performance evaluation can be loosely divided into two major phases: (1) *a priori* estimates and (2) *a posteriori* testing. We refer to these as *performance analysis* and *performance measurement* respectively.

1.3.1 Space Complexity

Algorithm **abc** (Algorithm 1.5) computes $a + b + b * c + (a + b - c)/(a + b) + 4.0$; Algorithm **Sum** (Algorithm 1.6) computes $\sum_{i=1}^n a[i]$ iteratively, where the $a[i]$'s are real numbers; and **RSum** (Algorithm 1.7) is a recursive algorithm that computes $\sum_{i=1}^n a[i]$.

```

1  Algorithm abc( $a, b, c$ )
2  {
3      return  $a + b + b * c + (a + b - c)/(a + b) + 4.0$ ;
4  }
```

Algorithm 1.5 Computes $a + b + b * c + (a + b - c)/(a + b) + 4.0$

The space needed by each of these algorithms is seen to be the sum of the following components:

```
1  Algorithm Sum( $a, n$ )
2  {
3       $s := 0.0;$ 
4      for  $i := 1$  to  $n$  do
5           $s := s + a[i];$ 
6      return  $s;$ 
7  }
```

Algorithm 1.6 Iterative function for sum

```
1  Algorithm RSum( $a, n$ )
2  {
3      if ( $n \leq 0$ ) then return 0.0;
4      else return RSum( $a, n - 1$ ) +  $a[n];$ 
5  }
```

Algorithm 1.7 Recursive function for sum

1. A fixed part that is independent of the characteristics (e.g., number, size) of the inputs and outputs. This part typically includes the instruction space (i.e., space for the code), space for simple variables and fixed-size component variables (also called *aggregate*), space for constants, and so on.
2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics), and the recursion stack space (insofar as this space depends on the instance characteristics).

The space requirement $S(P)$ of any algorithm P may therefore be written as $S(P) = c + S_P(\text{instance characteristics})$, where c is a constant.

When analyzing the space complexity of an algorithm, we concentrate solely on estimating $S_P(\text{instance characteristics})$. For any given problem, we need first to determine which instance characteristics to use to measure the space requirements. This is very problem specific, and we resort to examples to illustrate the various possibilities. Generally speaking, our choices are limited to quantities related to the number and magnitude of the inputs to and outputs from the algorithm. At times, more complex measures of the interrelationships among the data items are used.

Example 1.4 For Algorithm 1.5, the problem instance is characterized by the specific values of a , b , and c . Making the assumption that one word is adequate to store the values of each of a , b , c , and the result, we see that the space needed by `abc` is independent of the instance characteristics. Consequently, $S_P(\text{instance characteristics}) = 0$. \square

Example 1.5 The problem instances for Algorithm 1.6 are characterized by n , the number of elements to be summed. The space needed by n is one word, since it is of type *integer*. The space needed by a is the space needed by variables of type array of floating point numbers. This is at least n words, since a must be large enough to hold the n elements to be summed. So, we obtain $S_{\text{Sum}}(n) \geq (n + 3)$ (n for $a[]$, one each for n , i , and s). \square

Example 1.6 Let us consider the algorithm `RSum` (Algorithm 1.7). As in the case of `Sum`, the instances are characterized by n . The recursion stack space includes space for the formal parameters, the local variables, and the return address. Assume that the return address requires only one word of memory. Each call to `RSum` requires at least three words (including space for the values of n , the return address, and a pointer to $a[]$). Since the depth of recursion is $n + 1$, the recursion stack space needed is $\geq 3(n + 1)$. \square

1.3.2 Time Complexity

The time $T(P)$ taken by a program P is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. Also, we may assume that a compiled program will be run several times without recompilation. Consequently, we concern ourselves with just the run time of a program. This run time is denoted by t_P (instance characteristics).

Because many of the factors t_P depends on are not known at the time a program is conceived, it is reasonable to attempt only to estimate t_P . If we knew the characteristics of the compiler to be used, we could proceed to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on, that would be made by the code for P . So, we could obtain an expression for $t_P(n)$ of the form

$$t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \dots$$

where n denotes the instance characteristics, and c_a, c_s, c_m, c_d , and so on, respectively, denote the time needed for an addition, subtraction, multiplication, division, and so on, and ADD, SUB, MUL, DIV , and so on, are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and so on, that are performed when the code for P is used on an instance with characteristic n .

Obtaining such an exact formula is in itself an impossible task, since the time needed for an addition, subtraction, multiplication, and so on, often depends on the numbers being added, subtracted, multiplied, and so on. The value of $t_P(n)$ for any given n can be obtained only experimentally. The program is typed, compiled, and run on a particular machine. The execution time is physically clocked, and $t_P(n)$ obtained. Even with this experimental approach, one could face difficulties. In a multiuser system, the execution time depends on such factors as system load, the number of other programs running on the computer at the time program P is run, the characteristics of these other programs, and so on.

Given the minimal utility of determining the exact number of additions, subtractions, and so on, that are needed to solve a problem instance with characteristics given by n , we might as well lump all the operations together (provided that the time required by each is relatively independent of the instance characteristics) and obtain a count for the total number of operations. We can go one step further and count only the number of program steps.

A *program step* is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics. For example, the entire statement

```
return  $a + b + b * c + (a + b - c)/(a + b) + 4.0;$ 
```

of Algorithm 1.5 could be regarded as a step since its execution time is independent of the instance characteristics (this statement is not strictly true, since the time for a multiply and divide generally depends on the numbers involved in the operation).

The number of steps any program statement is assigned depends on the kind of statement. For example, comments count as zero steps; an assignment statement which does not involve any calls to other algorithms is counted as one step; in an iterative statement such as the **for**, **while**, and **repeat-until** statements, we consider the step counts only for the control part of the statement. The control parts for **for** and **while** statements have the following forms:

for $i := \langle expr \rangle$ **to** $\langle expr1 \rangle$ **do**

while $(\langle expr \rangle)$ **do**

Each execution of the control part of a **while** statement is given a step count equal to the number of step counts assignable to $\langle expr \rangle$. The step count for each execution of the control part of a **for** statement is one, unless the counts attributable to $\langle expr \rangle$ and $\langle expr1 \rangle$ are functions of the instance characteristics. In this latter case, the first execution of the control part of the **for** has a step count equal to the sum of the counts for $\langle expr \rangle$ and $\langle expr1 \rangle$ (note that these expressions are computed only when the loop is started). Remaining executions of the **for** statement have a step count of one; and so on.

We can determine the number of steps needed by a program to solve a particular problem instance in one of two ways. In the first method, we introduce a new variable, *count*, into the program. This is a global variable with initial value 0. Statements to increment *count* by the appropriate amount are introduced into the program. This is done so that each time a statement in the original program is executed, *count* is incremented by the step count of that statement.

Example 1.7 When the statements to increment *count* are introduced into Algorithm 1.6, the result is Algorithm 1.8. The change in the value of *count* by the time this program terminates is the number of steps executed by Algorithm 1.6.

Since we are interested in determining only the change in the value of *count*, Algorithm 1.8 may be simplified to Algorithm 1.9. For every initial value of *count*, Algorithms 1.8 and 1.9 compute the same final value for *count*. It is easy to see that in the **for** loop, the value of *count* will increase by a total of $2n$. If *count* is zero to start with, then it will be $2n + 3$ on termination. So each invocation of **Sum** (Algorithm 1.6) executes a total of $2n + 3$ steps. \square

```
1  Algorithm Sum( $a, n$ )
2  {
3       $s := 0.0;$ 
4       $count := count + 1;$  // count is global; it is initially zero.
5      for  $i := 1$  to  $n$  do
6      {
7           $count := count + 1;$  // For for
8           $s := s + a[i]; count := count + 1;$  // For assignment
9      }
10      $count := count + 1;$  // For last time of for
11      $count := count + 1;$  // For the return
12     return  $s;$ 
13 }
```

Algorithm 1.8 Algorithm 1.6 with count statements added

```
1  Algorithm Sum( $a, n$ )
2  {
3      for  $i := 1$  to  $n$  do  $count := count + 2;$ 
4       $count := count + 3;$ 
5  }
```

Algorithm 1.9 Simplified version of Algorithm 1.8

Example 1.8 When the statements to increment *count* are introduced into Algorithm 1.7, Algorithm 1.10 is obtained. Let $t_{\text{RSum}}(n)$ be the increase in the value of *count* when Algorithm 1.10 terminates. We see that $t_{\text{RSum}}(0) = 2$. When $n > 0$, *count* increases by 2 plus whatever increase results from the invocation of RSum from within the **else** clause. From the definition of t_{RSum} , it follows that this additional increase is $t_{\text{RSum}}(n - 1)$. So, if the value of *count* is zero initially, its value at the time of termination is $2 + t_{\text{RSum}}(n - 1)$, $n > 0$.

```

1  Algorithm RSum(a, n)
2  {
3      count := count + 1; // For the if conditional
4      if (n ≤ 0) then
5      {
6          count := count + 1; // For the return
7          return 0.0;
8      }
9      else
10     {
11         count := count + 1; // For the addition, function
12                     // invocation and return
13         return RSum(a, n - 1) + a[n];
14     }
15 }
```

Algorithm 1.10 Algorithm 1.7 with count statements added

When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count, for example,

$$t_{\text{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\text{RSum}}(n - 1) & \text{if } n > 0 \end{cases}$$

These recursive formulas are referred to as *recurrence relations*. One way of solving any such recurrence relation is to make repeated substitutions for each occurrence of the function t_{RSum} on the right-hand side until all such occurrences disappear:

$$\begin{aligned}
 t_{\text{RSum}}(n) &= 2 + t_{\text{RSum}}(n - 1) \\
 &= 2 + 2 + t_{\text{RSum}}(n - 2) \\
 &= 2(2) + t_{\text{RSum}}(n - 2) \\
 &\vdots \\
 &= n(2) + t_{\text{RSum}}(0) \\
 &= 2n + 2, \quad n \geq 0
 \end{aligned}$$

So the step count for RSum (Algorithm 1.7) is $2n + 2$. \square

The step count is useful in that it tells us how the run time for a program changes with changes in the instance characteristics. From the step count for Sum, we see that if n is doubled, the run time also doubles (approximately); if n increases by a factor of 10, the run time increases by a factor of 10; and so on. So, the run time grows *linearly* in n . We say that Sum is a linear time algorithm (the time complexity is linear in the instance characteristic n).

Definition 1.3 [Input size] One of the instance characteristics that is frequently used in the literature is the *input size*. The input size of any instance of a problem is defined to be the number of words (or the number of elements) needed to describe that instance. The input size for the problem of summing an array with n elements is $n + 1$, n for listing the n elements and 1 for the value of n (Algorithms 1.6 and 1.7). The problem tackled in Algorithm 1.5 has an input size of 3. If the input to any problem instance is a single element, the input size is normally taken to be the number of bits needed to specify that element. Run times for many of the algorithms presented in this text are expressed as functions of the corresponding input sizes. \square

Example 1.9 [Matrix addition] Algorithm 1.11 is to add two $m \times n$ matrices a and b together. Introducing the *count*-incrementing statements leads to Algorithm 1.12. Algorithm 1.13 is a simplified version of Algorithm 1.12 that computes the same value for *count*. Examining Algorithm 1.13, we see that line 7 is executed n times for each value of i , or a total of mn times; line 5 is executed m times; and line 9 is executed once. If *count* is 0 to begin with, it will be $2mn + 2m + 1$ when Algorithm 1.13 terminates.

From this analysis we see that if $m > n$, then it is better to interchange the two **for** statements in Algorithm 1.11. If this is done, the step count becomes $2mn + 2n + 1$. Note that in this example the instance characteristics are given by m and n and the input size is $2mn + 2$. \square

The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement. This figure is often arrived at by first determining the number of

```

1  Algorithm Add( $a, b, c, m, n$ )
2  {
3      for  $i := 1$  to  $m$  do
4          for  $j := 1$  to  $n$  do
5               $c[i, j] := a[i, j] + b[i, j];$ 
6  }
```

Algorithm 1.11 Matrix addition

```

1  Algorithm Add( $a, b, c, m, n$ )
2  {
3      for  $i := 1$  to  $m$  do
4          {
5               $count := count + 1;$  // For ‘for  $i$ ’
6              for  $j := 1$  to  $n$  do
7                  {
8                       $count := count + 1;$  // For ‘for  $j$ ’
9                       $c[i, j] := a[i, j] + b[i, j];$ 
10                      $count := count + 1;$  // For the assignment
11                 }
12                  $count := count + 1;$  // For loop initialization and
13                             // last time of ‘for  $j$ ’
14             }
15              $count := count + 1;$  // For loop initialization and
16                             // last time of ‘for  $i$ ’
17 }
```

Algorithm 1.12 Matrix addition with counting statements

```

1  Algorithm Add( $a, b, c, m, n$ )
2  {
3      for  $i := 1$  to  $m$  do
4      {
5           $count := count + 2;$ 
6          for  $j := 1$  to  $n$  do
7               $count := count + 2;$ 
8      }
9       $count := count + 1;$ 
10 }

```

Algorithm 1.13 Simplified algorithm with counting only

steps per execution (s/e) of the statement and the total number of times (i.e., frequency) each statement is executed. *The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.* By combining these two quantities, the total contribution of each statement is obtained. By adding the contributions of all statements, the step count for the entire algorithm is obtained.

In Table 1.1, the number of steps per execution and the frequency of each of the statements in Sum (Algorithm 1.6) have been listed. The total number of steps required by the algorithm is determined to be $2n + 3$. It is important to note that the frequency of the **for** statement is $n + 1$ and not n . This is so because i has to be incremented to $n + 1$ before the **for** loop can terminate.

Table 1.2 gives the step count for RSum (Algorithm 1.7). Notice that under the s/e (steps per execution) column, the **else** clause has been given a count of $1 + t_{RSum}(n - 1)$. This is the total cost of this line each time it is executed. It includes all the steps that get executed as a result of the invocation of RSum from the **else** clause. The frequency and total steps columns have been split into two parts: one for the case $n = 0$ and the other for the case $n > 0$. This is necessary because the frequency (and hence total steps) for some statements is different for each of these cases.

Table 1.3 corresponds to algorithm Add (Algorithm 1.11). Once again, note that the frequency of the first **for** loop is $m + 1$ and not m . This is so as i needs to be incremented up to $m + 1$ before the loop can terminate. Similarly, the frequency for the second **for** loop is $m(n + 1)$.

When you have obtained sufficient experience in computing step counts, you can avoid constructing the frequency table and obtain the step count as in the following example.

Statement	s/e	frequency	total steps
1 Algorithm Sum(a, n)	0	—	0
2 {	0	—	0
3 $s := 0.0;$	1	1	1
4 for $i := 1$ to n do	1	$n + 1$	$n + 1$
5 $s := s + a[i];$	1	n	n
6 return $s;$	1	1	1
7 }	0	—	0
Total			$2n + 3$

Table 1.1 Step table for Algorithm 1.6

Statement	s/e	frequency		total steps	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
1 Algorithm RSum(a, n)	0	—	—	0	0
2 {					
3 if ($n \leq 0$) then	1	1	1	1	1
4 return 0.0;	1	1	0	1	0
5 else return					
6 RSum($a, n - 1$) + $a[n];$	$1 + x$	0	1	0	$1 + x$
7 }	0	—	—	0	0
Total				2	$2 + x$

$$x = t_{\text{RSum}}(n - 1)$$

Table 1.2 Step table for Algorithm 1.7

Statement	s/e	frequency	total steps
1 Algorithm Add(a, b, c, m, n)	0	—	0
2 {	0	—	0
3 for $i := 1$ to m do	1	$m + 1$	$m + 1$
4 for $j := 1$ to n do	1	$m(n + 1)$	$mn + m$
5 $c[i, j] := a[i, j] + b[i, j];$	1	mn	mn
6 }	0	—	0
Total			$2mn + 2m + 1$

Table 1.3 Step table for Algorithm 1.11

Example 1.10 [Fibonacci numbers] The Fibonacci sequence of numbers starts as

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

Each new term is obtained by taking the sum of the two previous terms. If we call the first term of the sequence f_0 , then $f_0 = 0$, $f_1 = 1$, and in general

$$f_n = f_{n-1} + f_{n-2}, \quad n \geq 2$$

Fibonacci (Algorithm 1.14) takes as input any nonnegative integer n and prints the value f_n .

To analyze the time complexity of this algorithm, we need to consider the two cases (1) $n = 0$ or 1 and (2) $n > 1$. When $n = 0$ or 1, lines 4 and 5 get executed once each. Since each line has an s/e of 1, the total step count for this case is 2. When $n > 1$, lines 4, 8, and 14 are each executed once. Line 9 gets executed n times, and lines 11 and 12 get executed $n - 1$ times each (note that the last time line 9 is executed, i is incremented to $n + 1$, and the loop exited). Line 8 has an s/e of 2, line 12 has an s/e of 2, and line 13 has an s/e of 0. The remaining lines that get executed have s/e's of 1. The total steps for the case $n > 1$ is therefore $4n + 1$. \square

Summary of Time Complexity

The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function it was written for. The number of steps is itself a function of the instance characteristics. Although any specific instance may have several characteristics (e.g., the number of inputs, the number of outputs, the magnitudes of the inputs and outputs), the number

```

1  Algorithm Fibonacci( $n$ )
2  // Compute the  $n$ th Fibonacci number.
3  {
4      if ( $n \leq 1$ ) then
5          write ( $n$ );
6      else
7      {
8           $fnm2 := 0; fnm1 := 1;$ 
9          for  $i := 2$  to  $n$  do
10         {
11              $fn := fnm1 + fnm2;$ 
12              $fnm2 := fnm1; fnm1 := fn;$ 
13         }
14         write ( $fn$ );
15     }
16 }
```

Algorithm 1.14 Fibonacci numbers

of steps is computed as a function of some subset of these. Usually, we choose those characteristics that are of importance to us. For example, we might wish to know how the computing (or run) time (i.e., time complexity) increases as the number of inputs increase. In this case the number of steps will be computed as a function of the number of inputs alone. For a different algorithm, we might be interested in determining how the computing time increases as the magnitude of one of the inputs increases. In this case the number of steps will be computed as a function of the magnitude of this input alone. Thus, before the step count of an algorithm can be determined, we need to know exactly which characteristics of the problem instance are to be used. These define the variables in the expression for the step count. In the case of Sum, we chose to measure the time complexity as a function of the number n of elements being added. For algorithm Add, the choice of characteristics was the number m of rows and the number n of columns in the matrices being added.

Once the relevant characteristics (n, m, p, q, r, \dots) have been selected, we can define what a step is. A *step* is any computation unit that is independent of the characteristics (n, m, p, q, r, \dots) . Thus, 10 additions can be one step; 100 multiplications can also be one step; but n additions cannot. Nor can $m/2$ additions, $p + q$ subtractions, and so on, be counted as one step.

A systematic way to assign step counts was also discussed. Once this has been done, the time complexity (i.e., the total step count) of an algorithm can be obtained using either of the two methods discussed.

The examples we have looked at so far were sufficiently simple that the time complexities were nice functions of fairly simple characteristics like the number of inputs and the number of rows and columns. For many algorithms, the time complexity is not dependent solely on the number of inputs or outputs or some other easily specified characteristic. For example, the searching algorithm you wrote for Exercise 4 in Section 1.2, may terminate in one step if x is the first element examined by your algorithm, or it may take two steps (this happens if x is the second element examined), and so on. In other words, knowing n alone is not enough to estimate the run time of your algorithm.

We can extricate ourselves from the difficulties resulting from situations when the chosen parameters are not adequate to determine the step count uniquely by defining three kinds of step counts: best case, worst case, and average. The *best-case step count* is the minimum number of steps that can be executed for the given parameters. The *worst-case step count* is the maximum number of steps that can be executed for the given parameters. The *average step count* is the average number of steps executed on instances with the given parameters.

Our motivation to determine step counts is to be able to compare the time complexities of two algorithms that compute the same function and also to predict the growth in run time as the instance characteristics change.

Determining the exact step count (best case, worst case, or average) of an algorithm can prove to be an exceedingly difficult task. Expending immense effort to determine the step count exactly is not a very worthwhile endeavor, since the notion of a step is itself inexact. (Both the instructions $x := y$; and $x := y + z + (x/y) + (x * y * z - x/z)$; count as one step.) Because of the inexactness of what a step stands for, the exact step count is not very useful for comparative purposes. An exception to this is when the difference between the step counts of two algorithms is very large, as in $3n + 3$ versus $100n + 10$. We might feel quite safe in predicting that the algorithm with step count $3n+3$ will run in less time than the one with step count $100n+10$. But even in this case, it is not necessary to know that the exact step count is $100n + 10$. Something like, “it’s about $80n$ or $85n$ or $75n$,” is adequate to arrive at the same conclusion.

For most situations, it is adequate to be able to make a statement like $c_1n^2 \leq t_P(n) \leq c_2n^2$ or $t_Q(n, m) = c_1n + c_2m$, where c_1 and c_2 are non-negative constants. This is so because if we have two algorithms with a complexity of $c_1n^2 + c_2n$ and c_3n respectively, then we know that the one with complexity c_3n will be faster than the one with complexity $c_1n^2 + c_2n$ for sufficiently large values of n . For small values of n , either algorithm could be faster (depending on c_1 , c_2 , and c_3). If $c_1 = 1$, $c_2 = 2$, and $c_3 = 100$, then

$c_1n^2 + c_2n \leq c_3n$ for $n \leq 98$ and $c_1n^2 + c_2n > c_3n$ for $n > 98$. If $c_1 = 1$, $c_2 = 2$, and $c_3 = 1000$, then $c_1n^2 + c_2n \leq c_3n$ for $n \leq 998$.

No matter what the values of c_1 , c_2 , and c_3 , there will be an n beyond which the algorithm with complexity c_3n will be faster than the one with complexity $c_1n^2 + c_2n$. This value of n will be called the *break-even point*. If the break-even point is zero, then the algorithm with complexity c_3n is always faster (or at least as fast). The exact break-even point cannot be determined analytically. The algorithms have to be run on a computer in order to determine the break-even point. To know that there is a break-even point, it is sufficient to know that one algorithm has complexity $c_1n^2 + c_2n$ and the other c_3n for some constants c_1 , c_2 , and c_3 . There is little advantage in determining the exact values of c_1 , c_2 , and c_3 .

1.3.3 Asymptotic Notation (O , Ω , Θ)

With the previous discussion as motivation, we introduce some terminology that enables us to make meaningful (but inexact) statements about the time and space complexities of an algorithm. In the remainder of this chapter, the functions f and g are nonnegative functions.

Definition 1.4 [Big “oh”] The function $f(n) = O(g(n))$ (read as “ f of n is big oh of g of n ”) iff (if and only if) there exist positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all n , $n \geq n_0$. \square

Example 1.11 The function $3n + 2 = O(n)$ as $3n + 2 \leq 4n$ for all $n \geq 2$. $3n + 3 = O(n)$ as $3n + 3 \leq 4n$ for all $n \geq 3$. $100n + 6 = O(n)$ as $100n + 6 \leq 101n$ for all $n \geq 6$. $10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$. $1000n^2 + 100n - 6 = O(n^2)$ as $1000n^2 + 100n - 6 \leq 1001n^2$ for $n \geq 100$. $6 * 2^n + n^2 = O(2^n)$ as $6 * 2^n + n^2 \leq 7 * 2^n$ for $n \geq 4$. $3n + 3 = O(n^2)$ as $3n + 3 \leq 3n^2$ for $n \geq 2$. $10n^2 + 4n + 2 = O(n^4)$ as $10n^2 + 4n + 2 \leq 10n^4$ for $n \geq 2$. $3n + 2 \neq O(1)$ as $3n + 2$ is not less than or equal to c for any constant c and all $n \geq n_0$. $10n^2 + 4n + 2 \neq O(n)$. \square

We write $O(1)$ to mean a computing time that is a constant. $O(n)$ is called *linear*, $O(n^2)$ is called *quadratic*, $O(n^3)$ is called *cubic*, and $O(2^n)$ is called *exponential*. If an algorithm takes time $O(\log n)$, it is faster, for sufficiently large n , than if it had taken $O(n)$. Similarly, $O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$. These seven computing times— $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, and $O(2^n)$ —are the ones we see most often in this book.

As illustrated by the previous example, the statement $f(n) = O(g(n))$ states only that $g(n)$ is an upper bound on the value of $f(n)$ for all n , $n \geq n_0$. It does not say anything about how good this bound is. Notice

that $n = O(2^n)$, $n = O(n^{2.5})$, $n = O(n^3)$, $n = O(2^n)$, and so on. For the statement $f(n) = O(g(n))$ to be informative, $g(n)$ should be as small a function of n as one can come up with for which $f(n) = O(g(n))$. So, while we often say that $3n + 3 = O(n)$, we almost never say that $3n + 3 = O(n^2)$, even though this latter statement is correct.

From the definition of O , it should be clear that $f(n) = O(g(n))$ is not the same as $O(g(n)) = f(n)$. In fact, it is meaningless to say that $O(g(n)) = f(n)$. The use of the symbol $=$ is unfortunate because this symbol commonly denotes the equals relation. Some of the confusion that results from the use of this symbol (which is standard terminology) can be avoided by reading the symbol $=$ as “is” and not as “equals.”

Theorem 1.2 obtains a very useful result concerning the order of $f(n)$ (that is, the $g(n)$ in $f(n) = O(g(n))$) when $f(n)$ is a polynomial in n .

Theorem 1.2 If $f(n) = a_m n^m + \dots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof:

$$\begin{aligned} f(n) &\leq \sum_{i=0}^m |a_i| n^i \\ &\leq n^m \sum_{i=0}^m |a_i| n^{i-m} \\ &\leq n^m \sum_{i=0}^m |a_i| \quad \text{for } n \geq 1 \end{aligned}$$

So, $f(n) = O(n^m)$ (assuming that m is fixed). \square

Definition 1.5 [Omega] The function $f(n) = \Omega(g(n))$ (read as “ f of n is omega of g of n ”) iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n , $n \geq n_0$. \square

Example 1.12 The function $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$ (the inequality holds for $n \geq 0$, but the definition of Ω requires an $n_0 > 0$). $3n + 3 = \Omega(n)$ as $3n + 3 \geq 3n$ for $n \geq 1$. $100n + 6 = \Omega(n)$ as $100n + 6 \geq 100n$ for $n \geq 1$. $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for $n \geq 1$. $6 * 2^n + n^2 = \Omega(2^n)$ as $6 * 2^n + n^2 \geq 2^n$ for $n \geq 1$. Observe also that $3n + 3 = \Omega(1)$, $10n^2 + 4n + 2 = \Omega(n)$, $10n^2 + 4n + 2 = \Omega(1)$, $6 * 2^n + n^2 = \Omega(n^{100})$, $6 * 2^n + n^2 = \Omega(n^{50.2})$, $6 * 2^n + n^2 = \Omega(n^2)$, $6 * 2^n + n^2 = \Omega(n)$, and $6 * 2^n + n^2 = \Omega(1)$. \square

As in the case of the big oh notation, there are several functions $g(n)$ for which $f(n) = \Omega(g(n))$. The function $g(n)$ is only a lower bound on $f(n)$. For the statement $f(n) = \Omega(g(n))$ to be informative, $g(n)$ should be as large a function of n as possible for which the statement $f(n) = \Omega(g(n))$ is true. So, while we say that $3n + 3 = \Omega(n)$ and $6 * 2^n + n^2 = \Omega(2^n)$, we almost never say that $3n + 3 = \Omega(1)$ or $6 * 2^n + n^2 = \Omega(1)$, even though both of these statements are correct.

Theorem 1.3 is the analogue of Theorem 1.2 for the omega notation.

Theorem 1.3 If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Proof: Left as an exercise. □

Definition 1.6 [Theta] The function $f(n) = \Theta(g(n))$ (read as “ f of n is theta of g of n ”) iff there exist positive constants c_1, c_2 , and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$. □

Example 1.13 The function $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$, and $n_0 = 2$. $3n + 3 = \Theta(n)$, $10n^2 + 4n + 2 = \Theta(n^2)$, $6 * 2^n + n^2 = \Theta(2^n)$, and $10 * \log n + 4 = \Theta(\log n)$. $3n + 2 \neq \Theta(1)$, $3n + 3 \neq \Theta(n^2)$, $10n^2 + 4n + 2 \neq \Theta(n)$, $10n^2 + 4n + 2 \neq \Theta(1)$, $6 * 2^n + n^2 \neq \Theta(n^2)$, $6 * 2^n + n^2 \neq \Theta(n^{100})$, and $6 * 2^n + n^2 \neq \Theta(1)$. □

The theta notation is more precise than both the the big oh and omega notations. The function $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.

Notice that the coefficients in all of the $g(n)$ ’s used in the preceding three examples have been 1. This is in accordance with practice. We almost never find ourselves saying that $3n + 3 = O(3n)$, that $10 = O(100)$, that $10n^2 + 4n + 2 = \Omega(4n^2)$, that $6 * 2^n + n^2 = O(6 * 2^n)$, or that $6 * 2^n + n^2 = \Theta(4 * 2^n)$, even though each of these statements is true.

Theorem 1.4 If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

Proof: Left as an exercise. □

Definition 1.7 [Little “oh”] The function $f(n) = o(g(n))$ (read as “ f of n is little oh of g of n ”) iff

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

□

Example 1.14 The function $3n + 2 = o(n^2)$ since $\lim_{n \rightarrow \infty} \frac{3n+2}{n^2} = 0$. $3n + 2 = o(n \log n)$. $3n + 2 = o(n \log \log n)$. $6 * 2^n + n^2 = o(3^n)$. $6 * 2^n + n^2 = o(2^n \log n)$. $3n + 2 \neq o(n)$. $6 * 2^n + n^2 \neq o(2^n)$. □

Analogous to o is the notation ω defined as follows.

Definition 1.8 [Little omega] The function $f(n) = \omega(g(n))$ (read as “ f of n is little omega of g of n ”) iff

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$$

□

Example 1.15 Let us reexamine the time complexity analyses of the previous section. For the algorithm Sum (Algorithm 1.6) we determined that $t_{\text{Sum}}(n) = 2n + 3$. So, $t_{\text{Sum}}(n) = \Theta(n)$. For Algorithm 1.7, $t_{\text{RSum}}(n) = 2n + 2 = \Theta(n)$. □

Although we might all see that the O , Ω , and Θ notations have been used correctly in the preceding paragraphs, we are still left with the question, Of what use are these notations if we have to first determine the step count exactly? The answer to this question is that the asymptotic complexity (i.e., the complexity in terms of O , Ω , and Θ) can be determined quite easily without determining the exact step count. This is usually done by first determining the asymptotic complexity of each statement (or group of statements) in the algorithm and then adding these complexities. Tables 1.4 through 1.6 do just this for Sum, RSum, and Add (Algorithms 1.6, 1.7, and 1.11).

Statement	s/e	frequency	total steps
1 Algorithm Sum(a, n)	0	—	$\Theta(0)$
2 {	0	—	$\Theta(0)$
3 $s := 0.0;$	1	1	$\Theta(1)$
4 for $i := 1$ to n do	1	$n + 1$	$\Theta(n)$
5 $s := s + a[i];$	1	n	$\Theta(n)$
6 return $s;$	1	1	$\Theta(1)$
7 }	0	—	$\Theta(0)$
Total			$\Theta(n)$

Table 1.4 Asymptotic complexity of Sum (Algorithm 1.6)

Although the analyses of Tables 1.4 through 1.6 are carried out in terms of step counts, it is correct to interpret $t_P(n) = \Theta(g(n))$, $t_P(n) = \Omega(g(n))$, or $t_P(n) = O(g(n))$ as a statement about the computing time of algorithm P . This is so because each step takes only $\Theta(1)$ time to execute.

Statement	s/e	frequency		total steps	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
1 Algorithm RSum(a, n)	0	—	—	0	$\Theta(0)$
2 {	0	—	—	0	$\Theta(0)$
3 if ($n \leq 0$) then	1	1	1	1	$\Theta(1)$
4 return 0.0;	1	1	0	1	$\Theta(0)$
5 else return					
6 RSum($a, n - 1$) + $a[n]$;	$1 + x$	0	1	0	$\Theta(1 + x)$
7 }	0	—	—	0	$\Theta(0)$
Total				2	$\Theta(1 + x)$

$$x = t_{\text{RSum}}(n - 1)$$

Table 1.5 Asymptotic complexity of RSum (Algorithm 1.7).

Statement	s/e	frequency	total steps
1 Algorithm Add(a, b, c, m, n)	0	—	$\Theta(0)$
2 {	0	—	$\Theta(0)$
3 for $i := 1$ to m do	1	$\Theta(m)$	$\Theta(m)$
4 for $i := 1$ to n do	1	$\Theta(mn)$	$\Theta(mn)$
5 $c[i, j] := a[i, j] + b[i, j]$;	1	$\Theta(mn)$	$\Theta(mn)$
6 }	0	—	$\Theta(0)$
Total			$\Theta(mn)$

Table 1.6 Asymptotic complexity of Add (Algorithm 1.11)

After you have had some experience using the table method, you will be in a position to arrive at the asymptotic complexity of an algorithm by taking a more global approach. We elaborate on this method in the following examples.

Example 1.16 [Permutation generator] Consider `Perm` (Algorithm 1.4). When $k = n$, we see that the time taken is $\Theta(n)$. When $k < n$, the `else` clause is entered. At this time, the second `for` loop is entered $n - k + 1$ times. Each iteration of this loop takes $\Theta(n + t_{\text{Perm}}(k + 1, n))$ time. So, $t_{\text{Perm}}(k, n) = \Theta((n - k + 1)(n + t_{\text{Perm}}(k + 1, n)))$ when $k < n$. Since $t_{\text{Perm}}(k + 1, n)$ is at least n when $k + 1 \leq n$, we get $t_{\text{Perm}}(k, n) = \Theta((n - k + 1)t_{\text{Perm}}(k + 1, n))$ for $k < n$. Using the substitution method, we obtain $t_{\text{Perm}}(1, n) = \Theta(n(n!))$, $n \geq 1$. \square

Example 1.17 [Magic square] The next example we consider is a problem from recreational mathematics. A magic square is an $n \times n$ matrix of the integers 1 to n^2 such that the sum of every row, column, and diagonal is the same. Figure 1.2 gives an example magic square for the case $n = 5$. In this example, the common sum is 65.

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Figure 1.2 Example magic square

H. Coxeter has given the following simple rule for generating a magic square when n is odd:

Start with 1 in the middle of the top row; then go up and left, assigning numbers in increasing order to empty squares; if you fall off the square imagine the same square as tiling the plane and continue; if a square is occupied, move down instead and continue.

The magic square of Figure 1.2 was formed using this rule. Algorithm 1.15 is for creating an $n \times n$ magic square for the case in which n is odd. This results from Coxeter's rule.

The magic square is represented using a two-dimensional array having n rows and n columns. For this application it is convenient to number the rows (and columns) from 0 to $n - 1$ rather than from 1 to n . Thus, when the algorithm “falls off the square,” the **mod** operator sets i and/or j back to 0 or $n - 1$.

The time to initialize and output the square is $\Theta(n^2)$. The third **for** loop (in which key ranges over 2 through n^2) is iterated $n^2 - 1$ times and each iteration takes $\Theta(1)$ time. So, this **for** loop takes $\Theta(n^2)$ time. Hence the overall time complexity of **Magic** is $\Theta(n^2)$. Since there are n^2 positions in which the algorithm must place a number, we see that $\Theta(n^2)$ is the best bound an algorithm for the magic square problem can have. \square

Example 1.18 [Computing x^n] Our final example is to compute x^n for any real number x and integer $n \geq 0$. A naive algorithm for solving this problem is to perform $n - 1$ multiplications as follows:

```
power := x;
for i := 1 to n - 1 do power := power * x;
```

This algorithm takes $\Theta(n)$ time. A better approach is to employ the “repeated squaring” trick. Consider the special case in which n is an integral power of 2 (that is, in which n equals 2^k for some integer k). The following algorithm computes x^n .

```
power := x;
for i := 1 to k do power := power2;
```

The value of $power$ after q iterations of the **for** loop is x^{2^q} . Therefore, this algorithm takes only $\Theta(k) = \Theta(\log n)$ time, which is a significant improvement over the run time of the first algorithm.

Can the same algorithm be used when n is not an integral power of 2? Fortunately, the answer is yes. Let $b_k b_{k-1} \cdots b_1 b_0$ be the binary representation of the integer n . This means that $n = \sum_{q=0}^k b_q 2^q$. Now,

$$x^n = x^{\sum_{q=0}^k b_q 2^q} = (x)^{b_0} * (x^2)^{b_1} * (x^4)^{b_2} * \cdots * (x^{2^k})^{b_k}$$

Also observe that b_0 is nothing but $n \bmod 2$ and that $\lfloor n/2 \rfloor$ is $b_k b_{k-1} \cdots b_1$ in binary form. These observations lead us to **Exponentiate** (Algorithm 1.16) for computing x^n .

```

1  Algorithm Magic( $n$ )
2  // Create a magic square of size  $n$ ,  $n$  being odd.
3  {
4      if (( $n \bmod 2$ ) = 0) then
5      {
6          write (" $n$  is even"); return;
7      }
8      else
9      {
10         for  $i := 0$  to  $n - 1$  do // Initialize square to zero.
11         {
12             for  $j := 0$  to  $n - 1$  do  $\text{square}[i, j] := 0$ ;
13              $\text{square}[0, (n - 1)/2] := 1$ ; // Middle of first row
14             // ( $i, j$ ) is the current position.
15              $j := (n - 1)/2$ ;
16             for key := 2 to  $n^2$  do
17             {
18                 // Move up and left. The next two if statements
19                 // may be replaced by the mod operator if
20                 //  $-1 \bmod n$  has the value  $n - 1$ .
21                 if ( $i \geq 1$ ) then  $k := i - 1$ ; else  $k := n - 1$ ;
22                 if ( $j \geq 1$ ) then  $l := j - 1$ ; else  $l := n - 1$ ;
23                 if ( $\text{square}[k, l] \geq 1$ ) then  $i := (i + 1) \bmod n$ ;
24                 else // square[ $k, l$ ] is empty.
25                 {
26                      $i := k$ ;  $j := l$ ;
27                 }
28                  $\text{square}[i, j] := \text{key}$ ;
29             }
30             // Output the magic square.
31             for  $i := 0$  to  $n - 1$  do
32                 for  $j := 0$  to  $n - 1$  do write ( $\text{square}[i, j]$ );
33         }
}

```

Algorithm 1.15 Magic square

```

1  Algorithm Exponentiate( $x, n$ )
2  // Return  $x^n$  for an integer  $n \geq 0$ .
3  {
4       $m := n$ ;  $power := 1$ ;  $z := x$ ;
5      while ( $m > 0$ ) do
6      {
7          while (( $m \bmod 2$ ) = 0) do
8          {
9               $m := \lfloor m/2 \rfloor$ ;  $z := z^2$ ;
10         }
11          $m := m - 1$ ;  $power := power * z$ ;
12     }
13     return  $power$ ;
14 }
```

Algorithm 1.16 Computation of x^n

Proving the correctness of this algorithm is left as an exercise. The variable m starts with the value of n , and after every iteration of the innermost **while** loop (line 7), its value decreases by a factor of at least 2. Thus there will be only $\Theta(\log n)$ iterations of the **while** loop of line 7. Each such iteration takes $\Theta(1)$ time. Whenever control exits from the innermost **while** loop, the value of m is odd and the instructions $m := m - 1$; $power := power * z$; are executed once. After this execution, since m becomes even, either the innermost **while** loop is entered again or the outermost **while** loop (line 5) is exited (in case $m = 0$). Therefore the instructions $m := m - 1$; $power := power * z$; can only be executed $O(\log n)$ times. In summary, the overall run time of **Exponentiate** is $\Theta(\log n)$. \square

1.3.4 Practical Complexities

We have seen that the time complexity of an algorithm is generally some function of the instance characteristics. This function is very useful in determining how the time requirements vary as the instance characteristics change. The complexity function can also be used to compare two algorithms P and Q that perform the same task. Assume that algorithm P has complexity $\Theta(n)$ and algorithm Q has complexity $\Theta(n^2)$. We can assert that algorithm P is faster than algorithm Q for sufficiently large n . To see the validity of this assertion, observe that the computing time of P is bounded

from above by cn for some constant c and for all n , $n \geq n_1$, whereas that of Q is bounded from below by dn^2 for some constant d and all n , $n \geq n_2$. Since $cn \leq dn^2$ for $n \geq c/d$, algorithm P is faster than algorithm Q whenever $n \geq \max\{n_1, n_2, c/d\}$.

You should always be cautiously aware of the presence of the phrase “sufficiently large” in an assertion like that of the preceding discussion. When deciding which of the two algorithms to use, you must know whether the n you are dealing with is, in fact, sufficiently large. If algorithm P runs in $10^6 n$ milliseconds, whereas algorithm Q runs in n^2 milliseconds, and if you always have $n \leq 10^6$, then, other factors being equal, algorithm Q is the one to use.

To get a feel for how the various functions grow with n , you are advised to study Table 1.7 and Figure 1.3 very closely. It is evident from Table 1.7 and Figure 1.3 that the function 2^n grows very rapidly with n . In fact, if an algorithm needs 2^n steps for execution, then when $n = 40$, the number of steps needed is approximately $1.1 * 10^{12}$. On a computer performing one billion steps per second, this would require about 18.3 minutes. If $n = 50$, the same algorithm would run for about 13 days on this computer. When $n = 60$, about 310.56 years are required to execute the algorithm and when $n = 100$, about $4 * 10^{13}$ years are needed. So, we may conclude that the utility of algorithms with exponential complexity is limited to small n (typically $n \leq 40$).

$\log n$	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	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

Table 1.7 Function values

Algorithms that have a complexity that is a polynomial of high degree are also of limited utility. For example, if an algorithm needs n^{10} steps, then using our 1-billion-steps-per-second computer, we need 10 seconds when $n = 10$, 3171 years when $n = 100$, and $3.17 * 10^{13}$ years when $n = 1000$. If the algorithm’s complexity had been n^3 steps instead, then we would need one second when $n = 1000$, 110.67 minutes when $n = 10,000$, and 11.57 days when $n = 100,000$.

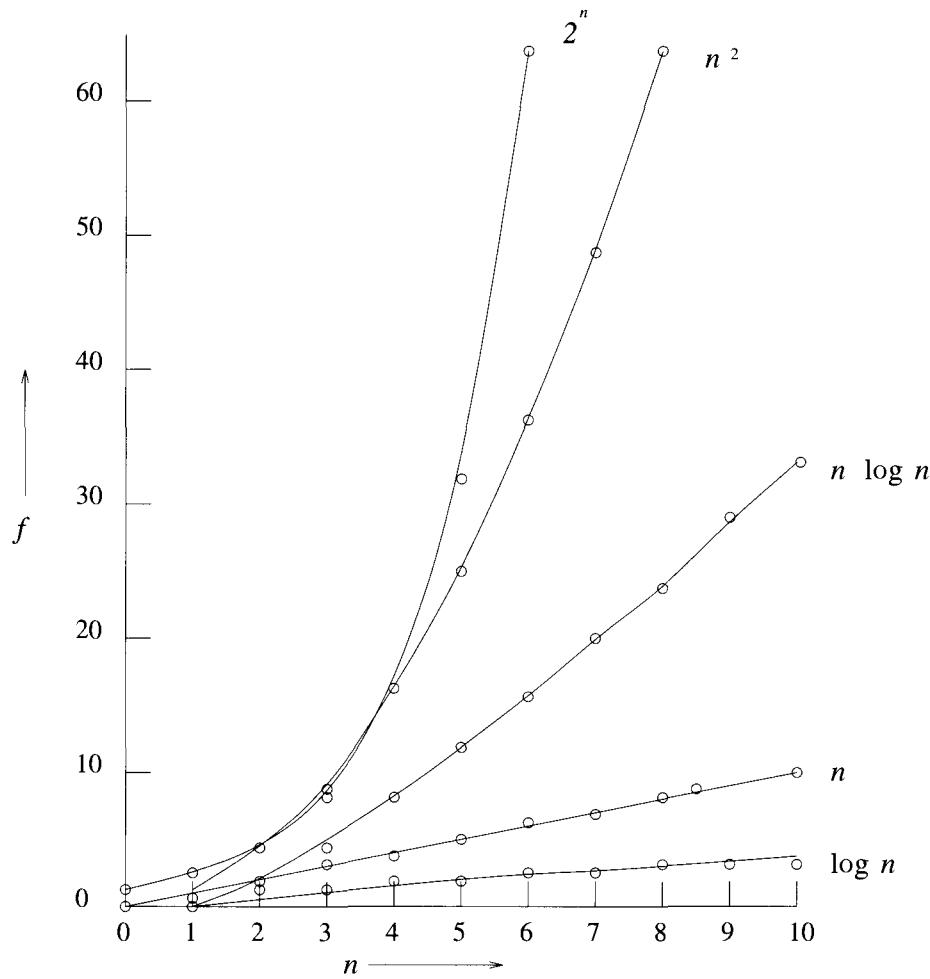


Figure 1.3 Plot of function values

Table 1.8 gives the time needed by a one-billion-steps-per-second computer to execute an algorithm of complexity $f(n)$ instructions. You should note that currently only the fastest computers can execute about 1 billion instructions per second. From a practical standpoint, it is evident that for reasonably large n (say $n > 100$), only algorithms of small complexity (such as n , $n \log n$, n^2 , and n^3) are feasible. Further, this is the case even if you could build a computer capable of executing 10^{12} instructions per second. In this case, the computing times of Table 1.8 would decrease by a factor of 1000. Now, when $n = 100$, it would take 3.17 years to execute n^{10} instructions and $4 * 10^{10}$ years to execute 2^n instructions.

n	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
	$f(n) = n$	$f(n) = n \log_2 n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = n^4$	$f(n) = n^{10}$	$f(n) = 2^n$
10	.01 μ s	.03 μ s	.1 μ s	1 μ s	10 μ s	10 s	1 μ s
20	.02 μ s	.09 μ s	.4 μ s	8 μ s	160 μ s	2.84 hr	1 ms
30	.03 μ s	.15 μ s	.9 μ s	27 μ s	810 μ s	6.83 d	1 s
40	.04 μ s	.21 μ s	1.6 μ s	64 μ s	2.56 ms	121.36 d	18.3 min
50	.05 μ s	.28 μ s	2.5 μ s	125 μ s	6.25 ms	3.1 yr	13 d
100	.1 μ s	.66 μ s	10 μ s	1 ms	100 ms	3171 yr	$4 * 10^{13}$ yr
1,000	1 μ s	9.96 μ s	1 ms	1 s	16.67 min	$3.17 * 10^{13}$ yr	$32 * 10^{283}$ yr
10,000	10 μ s	130 μ s	100 ms	16.67 min	115.7 d	$3.17 * 10^{23}$ yr	
100,000	100 μ s	1.66 ms	10 s	11.57 d	3171 yr	$3.17 * 10^{33}$ yr	
1,000,000	1 ms	19.92 ms	16.67 min	31.71 yr	$3.17 * 10^7$ yr	$3.17 * 10^{43}$ yr	

Table 1.8 Times on a 1-billion-steps-per-second computer

1.3.5 Performance Measurement

Performance measurement is concerned with obtaining the space and time requirements of a particular algorithm. These quantities depend on the compiler and options used as well as on the computer on which the algorithm is run. Unless otherwise stated, all performance values provided in this book are obtained using the Gnu C++ compiler, the default compiler options, and the Sparc 10/30 computer workstation.

In keeping with the discussion of the preceding section, we do not concern ourselves with the space and time needed for compilation. We justify this by the assumption that each program (after it has been fully debugged) is compiled once and then executed several times. Certainly, the space and time needed for compilation are important during program testing, when more time is spent on this task than in running the compiled code.

We do not consider measuring the run-time space requirements of a program. Rather, we focus on measuring the computing time of a program. To obtain the computing (or run) time of a program, we need a clocking procedure. We assume the existence of a program `GetTime()` that returns the current time in milliseconds.

Suppose we wish to measure the worst-case performance of the sequential search algorithm (Algorithm 1.17). Before we can do this, we need to (1) decide on the values of n for which the times are to be obtained and (2) determine, for each of the above values of n , the data that exhibit the worst-case behavior.

```

1  Algorithm SeqSearch( $a, x, n$ )
2  // Search for  $x$  in  $a[1 : n]$ .  $a[0]$  is used as additional space.
3  {
4       $i := n; a[0] := x;$ 
5      while ( $a[i] \neq x$ ) do  $i := i - 1;$ 
6      return  $i;$ 
7  }
```

Algorithm 1.17 Sequential search

The decision on which values of n to use is based on the amount of timing we wish to perform and also on what we expect to do with the times once they are obtained. Assume that for Algorithm 1.17, our intent is simply to predict how long it will take, in the worst case, to search for x , given the size n of a . An asymptotic analysis reveals that this time is $\Theta(n)$. So, we expect a plot of the times to be a straight line. Theoretically, if we know the times for any two values of n , the straight line is determined, and we can obtain the time for all other values of n from this line. In practice, we need the times for more than two values of n . This is so for the following reasons:

1. Asymptotic analysis tells us the behavior only for sufficiently large values of n . For smaller values of n , the run time may not follow the asymptotic curve. To determine the point beyond which the asymptotic curve is followed, we need to examine the times for several values of n .
2. Even in the region where the asymptotic behavior is exhibited, the times may not lie exactly on the predicted curve (straight line in the case of Algorithm 1.17) because of the effects of low-order terms that are discarded in the asymptotic analysis. For instance, an algorithm with asymptotic complexity $\Theta(n)$ can have time complexity $c_1n + c_2 \log n + c_3$ or, for that matter, any other function of n in which the highest-order term is c_1n for some constant c_1 , $c_1 > 0$.

It is reasonable to expect that the asymptotic behavior of Algorithm 1.17 begins for some n that is smaller than 100. So, for $n > 100$, we obtain the

run time for just a few values. A reasonable choice is $n = 200, 300, 400, \dots, 1000$. There is nothing magical about this choice of values. We can just as well use $n = 500, 1,000, 1,500, \dots, 10,000$ or $n = 512, 1,024, 2,048, \dots, 2^{15}$. It costs us more in terms of computer time to use the latter choices, and we probably do not get any better information about the run time of Algorithm 1.17 using these choices.

For n in the range $[0, 100]$ we carry out a more-refined measurement, since we are not quite sure where the asymptotic behavior begins. Of course, if our measurements show that the straight-line behavior does not begin in this range, we have to perform a more-detailed measurement in the range $[100, 200]$, and so on, until the onset of this behavior is detected. Times in the range $[0, 100]$ are obtained in steps of 10 beginning at $n = 0$.

Algorithm 1.17 exhibits its worst-case behavior when x is chosen such that it is not one of the $a[i]$'s. For definiteness, we set $a[i] = i$, $1 \leq i \leq n$, and $x = 0$. At this time, we envision using an algorithm such as Algorithm 1.18 to obtain the worst-case times.

```

1   Algorithm TimeSearch()
2   {
3       for  $j := 1$  to 1000 do  $a[j] := j$ ;
4       for  $j := 1$  to 10 do
5           {
6                $n[j] := 10 * (j - 1)$ ;  $n[j + 10] := 100 * j$ ;
7           }
8       for  $j := 1$  to 20 do
9           {
10           $h := \text{GetTime}()$ ;
11           $k := \text{SeqSearch}(a, 0, n[j])$ ;
12           $h1 := \text{GetTime}()$ ;
13           $t := h1 - h$ ;
14          write ( $n[j], t$ );
15      }
16  }
```

Algorithm 1.18 Algorithm to time Algorithm 1.17

The timing results of this algorithm is summarized in Table 1.9. The times obtained are too small to be of any use to us. Most of the times are zero; this indicates that the precision of our clock is inadequate. The nonzero times are just noise and are not representative of the time taken.

n	time	n	time
0	0	100	0
10	0	200	0
20	0	300	1
30	0	400	0
40	0	500	1
50	0	600	0
60	0	700	0
70	0	800	1
80	0	900	0
90	0	1000	0

Table 1.9 Timing results of Algorithm 1.18. Times are in milliseconds.

To time a short event, it is necessary to repeat it several times and divide the total time for the event by the number of repetitions.

Since our clock has an accuracy of about one-tenth of a second, we should not attempt to time any single event that takes less than about one second. With an event time of at least ten seconds, we can expect our observed times to be accurate to one percent.

The body of Algorithm 1.18 needs to be changed to that of Algorithm 1.19. In this algorithm, $r[i]$ is the number of times the search is to be repeated when the number of elements in the array is $n[i]$. Notice that rearranging the timing statements as in Algorithm 1.20 or 1.21 does not produce the desired results. For instance, from the data of Table 1.9, we expect that with the structure of Algorithm 1.20, the value output for $n = 0$ will still be 0. This is because there is a chance that in every iteration of the **for** loop, the clock does not change between the two times `GetTime()` is called. With the structure of Algorithm 1.21, we expect the algorithm never to exit the **while** loop when $n = 0$ (in reality, the loop will be exited because occasionally the measured time will turn out to be a few milliseconds).

Yet another alternative is shown in Algorithm 1.22. This approach can be expected to yield satisfactory times. It cannot be used when the timing procedure available gives us only the time since the last invocation of `GetTime`. Another difficulty is that the measured time includes the time needed to read the clock. For small n , this time may be larger than the time to run `SeqSearch`. This difficulty can be overcome by determining the time taken by the timing procedure and subtracting this time later.

```

1  Algorithm TimeSearch()
2  {
3      // Repetition factors
4       $r[21] := \{0, 200000, 200000, 150000, 100000, 100000, 100000,$ 
5           $50000, 50000, 50000, 50000, 50000, 50000, 50000,$ 
6           $50000, 50000, 25000, 25000, 25000, 25000\};$ 
7      for  $j := 1$  to 1000 do  $a[j] := j;$ 
8      for  $j := 1$  to 10 do
9      {
10          $n[j] := 10 * (j - 1); n[j + 10] := 100 * j;$ 
11     }
12     for  $j := 1$  to 20 do
13     {
14          $h := \text{GetTime}();$ 
15         for  $i := 1$  to  $r[j]$  do  $k := \text{SeqSearch}(a, 0, n[j]);$ 
16          $h1 := \text{GetTime}();$ 
17          $t1 := h1 - h;$ 
18          $t := t1; t := t/r[j];$ 
19         write ( $n[j], t1, t$ );
20     }
21 }
```

Algorithm 1.19 Timing algorithm

```

1   $t := 0;$ 
2  for  $i := 1$  to  $r[j]$  do
3  {
4       $h := \text{GetTime}();$ 
5       $k := \text{SeqSearch}(a, 0, n[j]);$ 
6       $h1 := \text{GetTime}();$ 
7       $t := t + h1 - h;$ 
8  }
9   $t := t/r[j];$ 
```

Algorithm 1.20 Improper timing construct

```
1  t := 0;
2  while (t < DESIRED_TIME) do
3  {
4      h := GetTime();
5      k := SeqSearch(a, 0, n[j]);
6      h1 := GetTime();
7      t := t + h1 - h;
8  }
```

Algorithm 1.21 Another improper timing construct

```
1  h := GetTime(); t := 0;
2  while (t < DESIRED_TIME) do
3  {
4      k := SeqSearch(a, 0, n[j]);
5      h1 := GetTime();
6      t := h1 - h;
7  }
```

Algorithm 1.22 An alternate timing construct

Timing results of Algorithm 1.19, is given in Table 1.10. The times for n in the range [0, 1000] are plotted in Figure 1.4. Values in the range [10, 100] have not been plotted. The linear dependence of the worst-case time on n is apparent from this graph.

n	t_1	t	n	t_1	t
0	308	0.002	100	1683	0.034
10	923	0.005	200	3359	0.067
20	1181	0.008	300	4693	0.094
30	1087	0.011	400	6323	0.126
40	1384	0.014	500	7799	0.156
50	1691	0.017	600	9310	0.186
60	999	0.020	700	5419	0.217
70	1156	0.023	800	6201	0.248
80	1306	0.026	900	6994	0.280
90	1460	0.029	1000	7725	0.309

Times are in milliseconds

Table 1.10 Worst-case run times for Algorithm 1.17

The graph of Figure 1.4 can be used to predict the run time for other values of n . We can go one step further and get the equation of the straight line. The equation of this line is $t = c + mn$, where m is the slope and c the value for $n = 0$. From the graph, we see that $c = 0.002$. Using the point $n = 600$ and $t = 0.186$, we obtain $m = (t - c)/n = 0.184/600 = 0.0003067$. So the line of Figure 1.4 has the equation $t = 0.002 + 0.0003067n$, where t is the time in milliseconds. From this, we expect that when $n = 1000$, the worst-case search time will be 0.3087 millisecond, and when $n = 500$, it will be 0.155 millisecond. Compared to the observed times of Table 1.10, we see that these figures are very accurate!

Summary of Running Time Calculation

To obtain the run time of a program, we need to plan the experiment. The following issues need to be addressed during the planning stage:

1. What is the accuracy of the clock? How accurate do our results have to be? Once the desired accuracy is known, we can determine the length of the shortest event that should be timed.

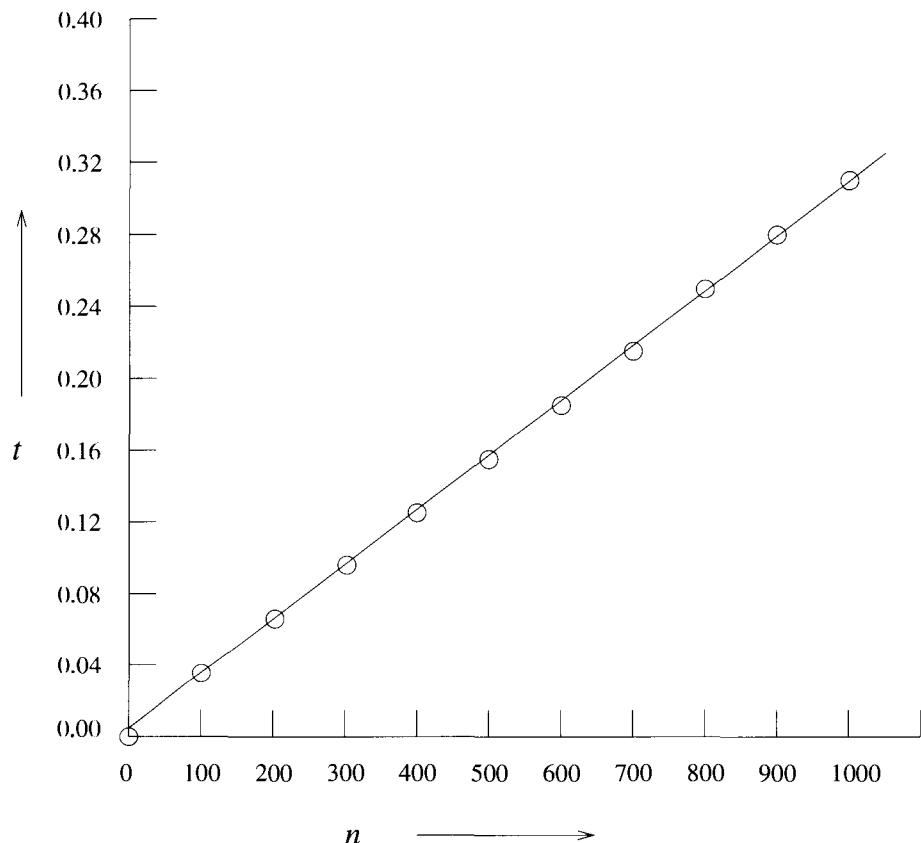


Figure 1.4 Plot of the data in Table 1.10

2. For each instance size, a repetition factor needs to be determined. This is to be chosen such that the event time is at least the minimum time that can be clocked with the desired accuracy.
3. Are we measuring worst-case or average performance? Suitable test data need to be generated.
4. What is the purpose of the experiment? Are the times being obtained for comparative purposes, or are they to be used to predict run times? If the latter is the case, then contributions to the run time from such sources as the repetition loop and data generation need to be subtracted (in case they are included in the measured time). If the former is the case, then these times need not be subtracted (provided they are the same for all programs being compared).
5. In case the times are to be used to predict run times, then we need to fit a curve through the points. For this, the asymptotic complexity should be known. If the asymptotic complexity is linear, then a least-squares straight line can be fit; if it is quadratic, then a parabola can be used (that is, $t = a_0 + a_1n + a_2n^2$). If the complexity is $\Theta(n \log n)$, then a least-squares curve of the form $t = a_0 + a_1n + a_2n \log_2 n$ can be fit. When obtaining the least-squares approximation, one should discard data corresponding to small values of n , since the program does not exhibit its asymptotic behavior for these n .

Generating Test Data

Generating a data set that results in the worst-case performance of an algorithm is not always easy. In some cases, it is necessary to use a computer program to generate the worst-case data. In other cases, even this is very difficult. In these cases, another approach to estimating worst-case performance is taken. For each set of values of the instance characteristics of interest, we generate a suitably large number of random test data. The run times for each of these test data are obtained. The maximum of these times is used as an estimate of the worst-case time for this set of values of the instance characteristics.

To measure average-case times, it is usually not possible to average over all possible instances of a given characteristic. Although it is possible to do this for sequential search, it is not possible for a sort algorithm. If we assume that all keys are distinct, then for any given n , $n!$ different permutations need to be used to obtain the average time. Obtaining average-case data is usually much harder than obtaining worst-case data. So, we often adopt the strategy outlined above and simply obtain an estimate of the average time on a suitable set of test data.

Whether we are estimating worst-case or average time using random data, the number of instances that we can try is generally much smaller than the total number of such instances. Hence, it is desirable to analyze the algorithm being tested to determine classes of data that should be generated for the experiment. This is a very algorithm-specific task, and we do not go into it here.

EXERCISES

1. Compare the two functions n^2 and $2^n/4$ for various values of n . Determine when the second becomes larger than the first.
2. Prove by induction:
 - (a) $\sum_{i=1}^n i = n(n+1)/2$, $n \geq 1$
 - (b) $\sum_{i=1}^n i^2 = n(n+1)(2n+1)/6$, $n \geq 1$
 - (c) $\sum_{i=0}^n x^i = (x^{n+1} - 1)/(x - 1)$, $x \neq 1$, $n \geq 0$
3. Determine the frequency counts for all statements in the following two algorithm segments:

<pre> 1 for i := 1 to n do 2 for j := 1 to i do 3 for k := 1 to j do 4 x := x + 1; </pre>	<pre> 1 i := 1; 2 while (i ≤ n) do 3 { 4 x := x + 1; 5 i := i + 1; 6 } </pre>
---	--

(a)

(b)

4. (a) Introduce statements to increment *count* at all appropriate points in Algorithm 1.23.
- (b) Simplify the resulting algorithm by eliminating statements. The simplified algorithm should compute the same value for *count* as computed by the algorithm of part (a).
- (c) What is the exact value of *count* when the algorithm terminates? You may assume that the initial value of *count* is 0.
- (d) Obtain the step count for Algorithm 1.23 using the frequency method. Clearly show the step count table.
5. Do Exercise 4 for Transpose (Algorithm 1.24).
6. Do Exercise 4 for Algorithm 1.25. This algorithm multiplies two $n \times n$ matrices *a* and *b*.

```

1  Algorithm D( $x, n$ )
2  {
3       $i := 1$ ;
4      repeat
5      {
6           $x[i] := x[i] + 2$ ;  $i := i + 2$ ;
7      } until ( $i > n$ );
8       $i := 1$ ;
9      while ( $i \leq \lfloor n/2 \rfloor$ ) do
10     {
11          $x[i] := x[i] + x[i + 1]$ ;  $i := i + 1$ ;
12     }
13 }
```

Algorithm 1.23 Example algorithm

```

1  Algorithm Transpose( $a, n$ )
2  {
3      for  $i := 1$  to  $n - 1$  do
4          for  $j := i + 1$  to  $n$  do
5          {
6               $t := a[i, j]$ ;  $a[i, j] := a[j, i]$ ;  $a[j, i] := t$ ;
7          }
8      }
```

Algorithm 1.24 Matrix transpose

```

1  Algorithm Mult( $a, b, c, n$ )
2  {
3      for  $i := 1$  to  $n$  do
4          for  $j := 1$  to  $n$  do
5              {
6                   $c[i, j] := 0$ ;
7                  for  $k := 1$  to  $n$  do
8                       $c[i, j] := c[i, j] + a[i, k] * b[k, j]$ ;
9              }
10 }
```

Algorithm 1.25 Matrix multiplication

7. (a) Do Exercise 4 for Algorithm 1.26. This algorithm multiplies two matrices a and b , where a is an $m \times n$ matrix and b is an $n \times p$ matrix.

```

1  Algorithm Mult( $a, b, c, m, n, p$ )
2  {
3      for  $i := 1$  to  $m$  do
4          for  $j := 1$  to  $p$  do
5              {
6                   $c[i, j] := 0$ ;
7                  for  $k := 1$  to  $n$  do
8                       $c[i, j] := c[i, j] + a[i, k] * b[k, j]$ ;
9              }
10 }
```

Algorithm 1.26 Matrix multiplication

- (b) Under what conditions is it profitable to interchange the two outermost **for** loops?
8. Show that the following equalities are correct:
- $5n^2 - 6n = \Theta(n^2)$
 - $n! = O(n^n)$
 - $2n^22^n + n \log n = \Theta(n^22^n)$
 - $\sum_{i=0}^n i^2 = \Theta(n^3)$

- (e) $\sum_{i=0}^n i^3 = \Theta(n^4)$.
- (f) $n^{2^n} + 6 * 2^n = \Theta(n^{2^n})$
- (g) $n^3 + 10^6 n^2 = \Theta(n^3)$
- (h) $6n^3 / (\log n + 1) = O(n^3)$
- (i) $n^{1.001} + n \log n = \Theta(n^{1.001})$
- (j) $n^{k+\epsilon} + n^k \log n = \Theta(n^{k+\epsilon})$ for all fixed k and ϵ , $k \geq 0$ and $\epsilon > 0$
- (k) $10n^3 + 15n^4 + 100n^2 2^n = O(100n^2 2^n)$
- (l) $33n^3 + 4n^2 = \Omega(n^2)$
- (m) $33n^3 + 4n^2 = \Omega(n^3)$

9. Show that the following equalities are incorrect:

- (a) $10n^2 + 9 = O(n)$
- (b) $n^2 \log n = \Theta(n^2)$
- (c) $n^2 / \log n = \Theta(n^2)$
- (d) $n^3 2^n + 6n^2 3^n = O(n^3 2^n)$

10. Prove Theorems 1.3 and 1.4.

11. Analyze the computing time of `SelectionSort` (Algorithm 1.2).

12. Obtain worst-case run times for `SelectionSort` (Algorithm 1.2). Do this for suitable values of n in the range $[0, 100]$. Your report must include a plan for the experiment as well as the measured times. These times are to be provided both in a table and as a graph.

13. Consider the algorithm `Add` (Algorithm 1.11).

- (a) Obtain run times for $n = 1, 10, 20, \dots, 100$.
- (b) Plot the times obtained in part (a).

14. Do the previous exercise for matrix multiplication (Algorithm 1.26).

15. A complex-valued matrix X is represented by a pair of matrices (A, B) , where A and B contain real values. Write an algorithm that computes the product of two complex-valued matrices (A, B) and (C, D) , where $(A, B) * (C, D) = (A + iB) * (C + iD) = (AC - BD) + i(AD + BC)$. Determine the number of additions and multiplications if the matrices are all $n \times n$.

Chapter 2

ELEMENTARY DATA STRUCTURES

Now that we have examined the fundamental methods we need to express and analyze algorithms, we might feel all set to begin. But, alas, we need to make one last diversion, and that is a discussion of data structures. One of the basic techniques for improving algorithms is to structure the data in such a way that the resulting operations can be efficiently carried out. In this chapter, we review only the most basic and commonly used data structures. Many of these are used in subsequent chapters. We should be familiar with stacks and queues (Section 2.1), binary trees (Section 2.2), and graphs (Section 2.6) and be able to refer to the other structures as needed.

2.1 STACKS AND QUEUES

One of the most common forms of data organization in computer programs is the ordered or linear list, which is often written as $a = (a_1, a_2, \dots, a_n)$. The a_i 's are referred to as *atoms* and they are chosen from some set. The null or empty list has $n = 0$ elements. A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top*. A *queue* is an ordered list in which all insertions take place at one end, the *rear*, whereas all deletions take place at the other end, the *front*.

The operations of a stack imply that if the elements A, B, C, D, and E are inserted into a stack, in that order, then the first element to be removed (deleted) must be E. Equivalently we say that the last element to be inserted into the stack is the first to be removed. For this reason stacks are sometimes referred to as **Last In First Out (LIFO)** lists. The operations of a queue require that the first element that is inserted into the queue is the first one to be removed. Thus queues are known as **First In First Out (FIFO)** lists. See Figure 2.1 for examples of a stack and a queue each containing the same

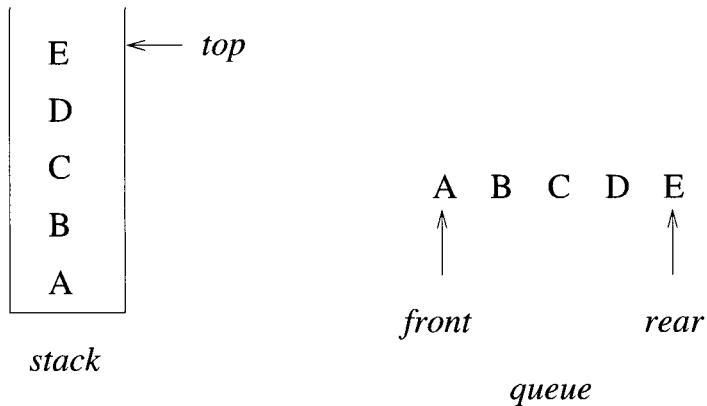


Figure 2.1 Example of a stack and a queue

five elements inserted in the same order. Note that the data object queue as defined here need not correspond to the concept of queue that is studied in queuing theory.

The simplest way to represent a stack is by using a one-dimensional array, say $stack[0 : n - 1]$, where n is the maximum number of allowable entries. The first or bottom element in the stack is stored at $stack[0]$, the second at $stack[1]$, and the i th at $stack[i - 1]$. Associated with the array is a variable, typically called *top*, which points to the top element in the stack. To test whether the stack is empty, we ask “if ($top < 0$)”. If not, the topmost element is at $stack[top]$. Checking whether the stack is full can be done by asking “if ($top \geq n - 1$)”. Two more substantial operations are inserting and deleting elements. The corresponding algorithms are Add and Delete (Algorithm 2.1).

Each execution of Add or Delete takes a constant amount of time and is independent of the number of elements in the stack.

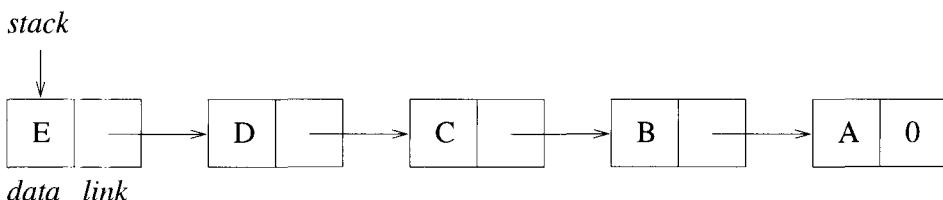
Another way to represent a stack is by using links (or pointers). A *node* is a collection of data and link information. A stack can be represented by using nodes with two fields, possibly called *data* and *link*. The data field of each node contains an item in the stack and the corresponding link field points to the node containing the next item in the stack. The link field of the last node is zero, for we assume that all nodes have an address greater than zero. For example, a stack with the items A, B, C, D, and E inserted in that order, looks as in Figure 2.2.

```

1  Algorithm Add(item)
2  // Push an element onto the stack. Return true if successful;
3  // else return false. item is used as an input.
4  {
5      if (top  $\geq$  n - 1) then
6      {
7          write ("Stack is full!"); return false;
8      }
9      else
10     {
11         top := top + 1; stack[top] := item; return true;
12     }
13 }

1  Algorithm Delete(item)
2  // Pop the top element from the stack. Return true if successful
3  // else return false. item is used as an output.
4  {
5      if (top < 0) then
6      {
7          write ("Stack is empty!"); return false;
8      }
9      else
10     {
11         item := stack[top]; top := top - 1; return true;
12     }
13 }

```

Algorithm 2.1 Operations on a stack**Figure 2.2** Example of a five-element, linked stack

```

// Type is the type of data.
node =record
{
    Type data; node *link;
}

1  Algorithm Add(item)
2  {
3      // Get a new node.
4      temp := new node;
5      if (temp ≠ 0) then
6      {
7          (temp → data) := item; (temp → link) := top;
8          top := temp; return true;
9      }
10     else
11     {
12         write ("Out of space!");
13         return false;
14     }
15 }

1  Algorithm Delete(item)
2  {
3      if (top = 0) then
4      {
5          write ("Stack is empty!");
6          return false;
7      }
8      else
9      {
10         item := (top → data); temp := top;
11         top := (top → link);
12         delete temp; return true;
13     }
14 }

```

Algorithm 2.2 Link representation of a stack

The variable *top* points to the topmost node (the last item inserted) in the list. The empty stack is represented by setting *top* := 0. Because of the way the links are pointing, insertion and deletion are easy to accomplish. See Algorithm 2.2.

In the case of Add, the statement *temp* := **new node**; assigns to the variable *temp* the address of an available node. If no more nodes exist, it returns 0. If a node exists, we store appropriate values into the two fields of the node. Then the variable *top* is updated to point to the new top element of the list. Finally, **true** is returned. If no more space exists, it prints an error message and returns **false**. Referring to Delete, if the stack is empty, then trying to delete an item produces the error message "Stack is empty!" and **false** is returned. Otherwise the top element is stored as the value of the variable *item*, a pointer to the first node is saved, and *top* is updated to point to the next node. The deleted node is returned for future use and finally **true** is returned.

The use of links to represent a stack requires more storage than the sequential array *stack*[0 : *n* − 1] does. However, there is greater flexibility when using links, for many structures can simultaneously use the same pool of available space. Most importantly the times for insertion and deletion using either representation are independent of the size of the stack.

An efficient queue representation can be obtained by taking an array *q*[0 : *n* − 1] and treating it as if it were circular. Elements are inserted by increasing the variable *rear* to the next free position. When *rear* = *n* − 1, the next element is entered at *q*[0] in case that spot is free. The variable *front* always points one position counterclockwise from the first element in the queue. The variable *front* = *rear* if and only if the queue is empty and we initially set *front* := *rear* := 0. Figure 2.3 illustrates two of the possible configurations for a circular queue containing the four elements J1 to J4 with *n* > 4.

To insert an element, it is necessary to move *rear* one position clockwise. This can be done using the code

```
if (rear = n − 1) then rear := 0;  
else rear := rear + 1;
```

A more elegant way to do this is to use the built-in modulo operator which computes remainders. Before doing an insert, we increase the rear pointer by saying *rear* := (*rear* + 1) **mod** *n*. Similarly, it is necessary to move *front* one position clockwise each time a deletion is made. An examination of Algorithm 2.3(a) and (b) shows that by treating the array circularly, addition and deletion for queues can be carried out in a fixed amount of time or $O(1)$.

One surprising feature in these two algorithms is that the test for queue full in AddQ and the test for queue empty in DeleteQ are the same. In the

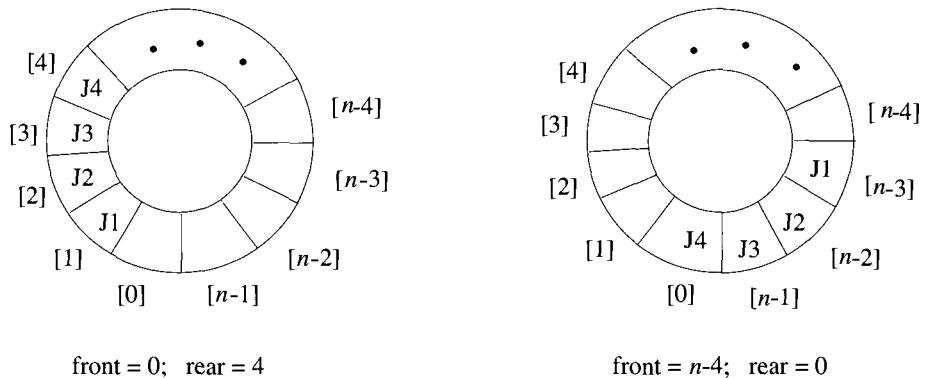


Figure 2.3 Circular queue of capacity $n - 1$ containing four elements J1, J2, J3, and J4

case of AddQ, however, when $front = rear$, there is actually one space free, $q[rear]$, since the first element in the queue is not at $q[front]$ but is one position clockwise from this point. However, if we insert an item there, then we cannot distinguish between the cases full and empty, since this insertion leaves $front = rear$. To avoid this, we signal queue full and permit a maximum of $n - 1$ rather than n elements to be in the queue at any time. One way to use all n positions is to use another variable, tag , to distinguish between the two situations; that is, $tag = 0$ if and only if the queue is empty. This however slows down the two algorithms. Since the AddQ and DeleteQ algorithms are used many times in any problem involving queues, the loss of one queue position is more than made up by the reduction in computing time.

Another way to represent a queue is by using links. Figure 2.4 shows a queue with the four elements A, B, C, and D entered in that order. As with the linked stack example, each node of the queue is composed of the two fields *data* and *link*. A queue is pointed at by two variables, *front* and *rear*. Deletions are made from the front, and insertions at the rear. Variable *front* = 0 signals an empty queue. The procedures for insertion and deletion in linked queues are left as exercises.

EXERCISES

1. Write algorithms for AddQ and DeleteQ, assuming the queue is represented as a linked list.

```

1  Algorithm AddQ(item)
2  // Insert item in the circular queue stored in q[0 : n − 1].
3  // rear points to the last item, and front is one
4  // position counterclockwise from the first item in q.
5  {
6      rear := (rear + 1) mod n; // Advance rear clockwise.
7      if (front = rear) then
8      {
9          write ("Queue is full!");
10         if (front = 0) then rear := n − 1;
11         else rear := rear − 1;
12         // Move rear one position counterclockwise.
13         return false;
14     }
15     else
16     {
17         q[rear] := item; // Insert new item.
18         return true;
19     }
20 }
```

(a) Addition of an element

```

1  Algorithm DeleteQ(item)
2  // Removes and returns the front element of the queue q[0 : n − 1].
3  {
4      if (front = rear) then
5      {
6          write ("Queue is empty!");
7          return false;
8      }
9      else
10     {
11         front := (front + 1) mod n; // Advance front clockwise.
12         item := q[front]; // Set item to front of queue.
13         return true;
14     }
15 }
```

(b) Deletion of an element

Algorithm 2.3 Basic queue operations

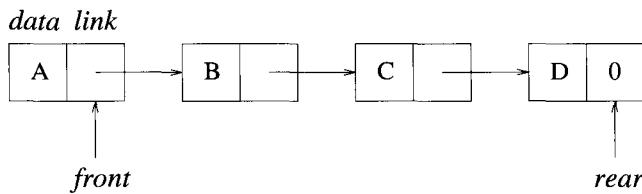


Figure 2.4 A linked queue with four elements

2. A linear list is being maintained circularly in an array $c[0 : n - 1]$ with f and r set up as for circular queues.

 - Obtain a formula in terms of f , r , and n for the number of elements in the list.
 - Write an algorithm to delete the k th element in the list.
 - Write an algorithm to insert an element y immediately after the k th element.

What is the time complexity of your algorithms for parts (b) and (c)?

3. Let $X = (x_1, \dots, x_n)$ and $Y = (y_1, \dots, y_m)$ be two linked lists. Write an algorithm to merge the two lists to obtain the linked list $Z = (x_1, y_1, x_2, y_2, \dots, x_m, y_m, x_{m+1}, \dots, x_n)$ if $m \leq n$ or $Z = (x_1, y_1, x_2, y_2, \dots, x_n, y_n, y_{n+1}, \dots, y_m)$ if $m > n$.
 4. A double-ended queue (deque) is a linear list for which insertions and deletions can occur at either end. Show how to represent a deque in a one-dimensional array and write algorithms that insert and delete at either end.
 5. Consider the hypothetical data object $X2$. The object $X2$ is a linear list with the restriction that although additions to the list can be made at either end, deletions can be made from one end only. Design a linked list representation for $X2$. Specify initial and boundary conditions for your representation.

2.2 TREES

Definition 2.1 [Tree] A *tree* is a finite set of one or more nodes such that there is a specially designated node called the *root* and the remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, \dots, T_n , where each of these sets is a tree. The sets T_1, \dots, T_n are called the *subtrees* of the root. \square

2.2.1 Terminology

There are many terms that are often used when referring to trees. Consider the tree in Figure 2.5. This tree has 13 nodes, each data item of a node being a single letter for convenience. The root contains A (we usually say node A), and we normally draw trees with their roots at the top. The number of subtrees of a node is called its *degree*. The degree of A is 3, of C is 1, and of F is 0. Nodes that have degree zero are called *leaf* or *terminal* nodes. The set $\{K, L, F, G, M, I, J\}$ is the set of leaf nodes of Figure 2.5. The other nodes are referred to as *nonterminals*. The roots of the subtrees of a node X are the *children* of X . The node X is the *parent* of its children. Thus the children of D are H, I, and J, and the parent of D is A.

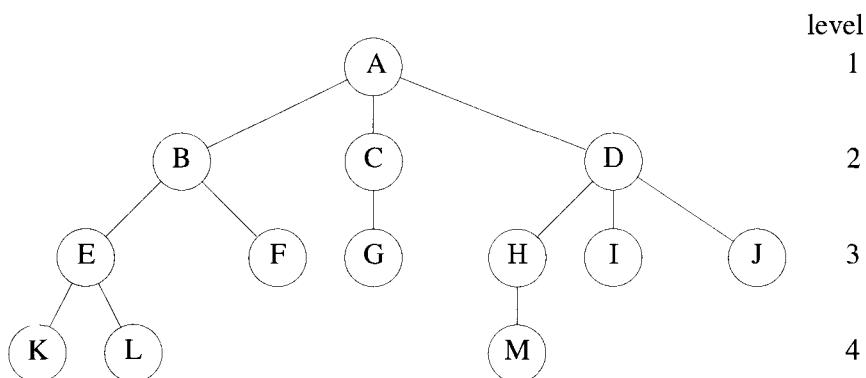


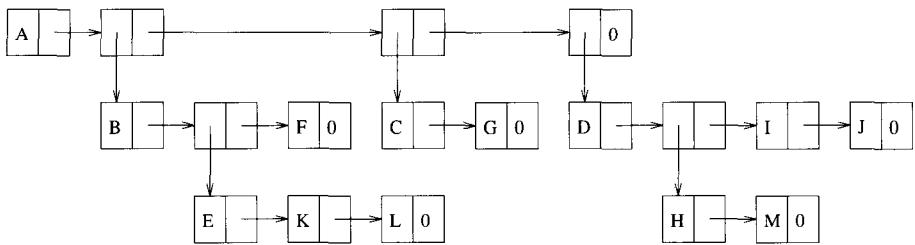
Figure 2.5 A sample tree

Children of the same parent are said to be *siblings*. For example H, I, and J are siblings. We can extend this terminology if we need to so that we can ask for the grandparent of M, which is D, and so on. The *degree* of a tree is the maximum degree of the nodes in the tree. The tree in Figure 2.5 has degree 3. The *ancestors* of a node are all the nodes along the path from the root to that node. The ancestors of M are A, D, and H.

The *level* of a node is defined by initially letting the root be at level one. If a node is at level p , then its children are at level $p + 1$. Figure 2.5 shows the levels of all nodes in that tree. The *height* or *depth* of a tree is defined to be the maximum level of any node in the tree.

A *forest* is a set of $n \geq 0$ disjoint trees. The notion of a forest is very close to that of a tree because if we remove the root of a tree, we get a forest. For example, in Figure 2.5 if we remove A, we get a forest with three trees.

Now how do we represent a tree in a computer's memory? If we wish to use a linked list in which one node corresponds to one node in the tree, then a node must have a varying number of fields depending on the number of children. However, it is often simpler to write algorithms for a data representation in which the node size is fixed. We can represent a tree using a fixed node size list structure. Such a list representation for the tree of Figure 2.5 is given in Figure 2.6. In this figure nodes have three fields: *tag*, *data*, and *link*. The fields *data* and *link* are used as before with the exception that when *tag* = 1, *data* contains a pointer to a list rather than a data item. A tree is represented by storing the root in the first node followed by nodes that point to sublists each of which contains one subtree of the root.



The *tag* field of a node is one if it has a down-pointing arrow; otherwise it is zero.

Figure 2.6 List representation for the tree of Figure 2.5

2.2.2 Binary Trees

A binary tree is an important type of tree structure that occurs very often. It is characterized by the fact that any node can have at most two children; that is, there is no node with degree greater than two. For binary trees we distinguish between the subtree on the left and that on the right, whereas for other trees the order of the subtrees is irrelevant. Furthermore a binary tree is allowed to have zero nodes whereas any other tree must have at least one node. Thus a binary tree is really a different kind of object than any other tree.

Definition 2.2 A *binary tree* is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the *left* and *right* subtrees. \square

Figure 2.7 shows two sample binary trees. These two trees are special kinds of binary trees. Figure 2.7(a) is a *skewed tree*, skewed to the left.

There is a corresponding tree skewed to the right, which is not shown. The tree in Figure 2.7(b) is called a *complete* binary tree. This kind of tree is defined formally later on. Notice that for this tree all terminal nodes are on two adjacent levels. The terms that we introduced for trees, such as degree, level, height, leaf, parent, and child, all apply to binary trees in the same way.

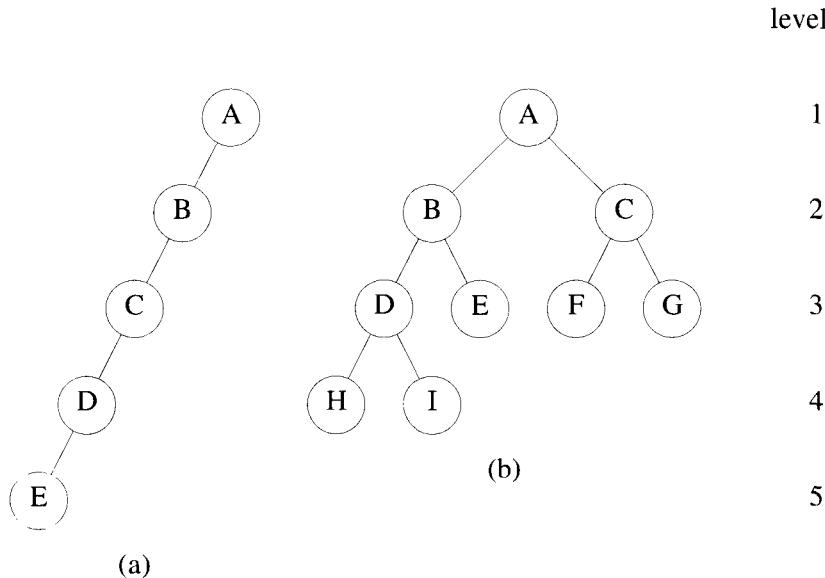


Figure 2.7 Two sample binary trees

Lemma 2.1 The maximum number of nodes on level i of a binary tree is 2^{i-1} . Also, the maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k > 0$. \square

The binary tree of depth k that has exactly $2^k - 1$ nodes is called a *full* binary tree of depth k . Figure 2.8 shows a full binary tree of depth 4. A very elegant sequential representation for full binary trees results from sequentially numbering the nodes, starting with the node on level one, then going to those on level two, and so on. Nodes on any level are numbered from left to right (see Figure 2.8). A binary tree with n nodes and depth k is *complete* iff its nodes correspond to the nodes that are numbered one to n in the full binary tree of depth k . A consequence of this definition is that in a complete tree, leaf nodes occur on at most two adjacent levels. The nodes

of an n -node complete tree may be compactly stored in a one-dimensional array, $\text{tree}[1 : n]$, with the node numbered i being stored in $\text{tree}[i]$. The next lemma shows how to easily determine the locations of the parent, left child, and right child of any node i in the binary tree without explicitly storing any link information.

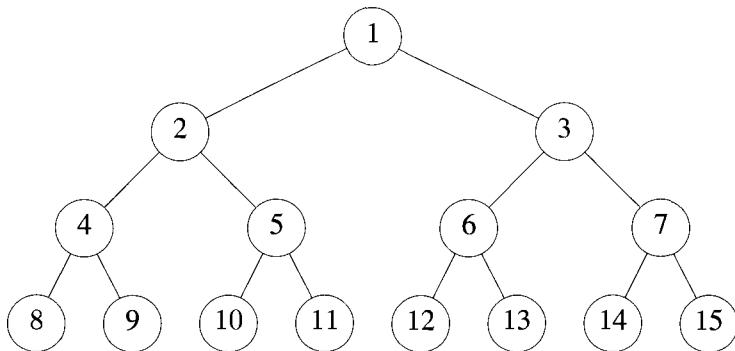


Figure 2.8 Full binary tree of depth 4

Lemma 2.2 If a complete binary tree with n nodes is represented sequentially as described before, then for any node with index i , $1 \leq i \leq n$, we have:

1. $\text{parent}(i)$ is at $\lfloor i/2 \rfloor$ if $i \neq 1$. When $i = 1$, i is the root and has no parent.
2. $\text{lchild}(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, i has no left child.
3. $\text{rchild}(i)$ is at $2i + 1$ if $2i + 1 \leq n$. If $2i + 1 > n$, i has no right child. \square

This representation can clearly be used for all binary trees though in most cases there is a lot of unutilized space. For complete binary trees the representation is ideal as no space is wasted. For the skewed tree of Figure 2.7, however, less than a third of the array is utilized. In the worst case a right-skewed tree of depth k requires $2^k - 1$ locations. Of these only k are occupied.

Although the sequential representation, as in Figure 2.9, appears to be good for complete binary trees, it is wasteful for many other binary trees. In addition, the representation suffers from the general inadequacies of sequential representations. Insertion or deletion of nodes requires the movement

tree	A	B	-	C	-	-	-	D	-	...	E
tree	A	B	C	D	E	F	G	H	I		
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	...	(16)

Figure 2.9 Sequential representation of the binary trees of Figure 2.7

of potentially many nodes to reflect the change in level number of the remaining nodes. These problems can be easily overcome through the use of a linked representation. Each node has three fields: *lchild*, *data*, and *rchild*. Although this node structure makes it difficult to determine the parent of a node, for most applications it is adequate. In case it is often necessary to be able to determine the parent of a node, then a fourth field, *parent*, can be included with the obvious interpretation. The representation of the binary trees of Figure 2.7 using a three-field structure is given in Figure 2.10.

2.3 DICTIONARIES

An abstract data type that supports the operations insert, delete, and search is called a *dictionary*. Dictionaries have found application in the design of numerous algorithms.

Example 2.1 Consider the database of books maintained in a library system. When a user wants to check whether a particular book is available, a *search* operation is called for. If the book is available and is issued to the user, a *delete* operation can be performed to remove this book from the set of available books. When the user returns the book, it can be *inserted* back into the set. \square

It is essential that we are able to support the above-mentioned operations as efficiently as possible since these operations are performed quite frequently. A number of data structures have been devised to realize a dictionary. At a very high level these can be categorized as comparison methods and direct access methods. Hashing is an example of the latter. We elaborate only on binary search trees which are an example of the former.

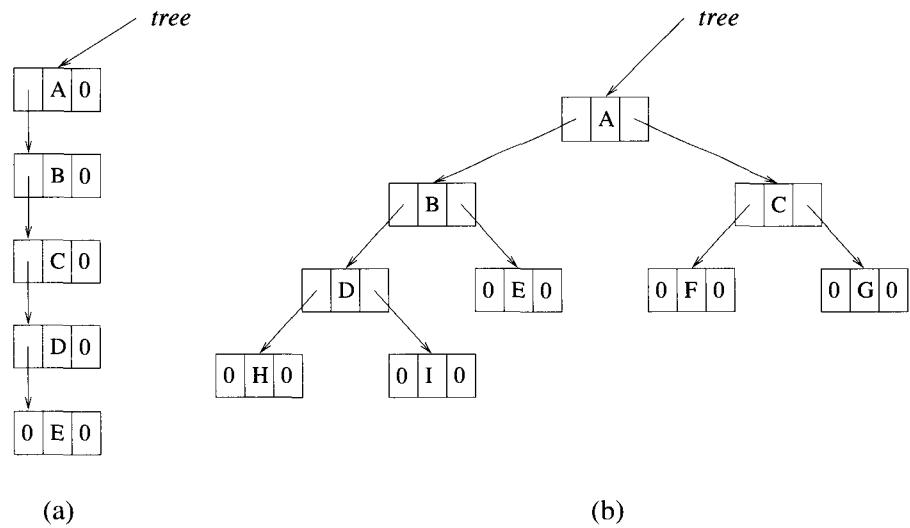


Figure 2.10 Linked representations for the binary trees of Figure 2.7

2.3.1 Binary Search Trees

Definition 2.3 [Binary search tree] A *binary search tree* is a binary tree. It may be empty. If it is not empty, then it satisfies the following properties:

1. Every element has a key and no two elements have the same key (i.e., the keys are distinct).
2. The keys (if any) in the left subtree are smaller than the key in the root.
3. The keys (if any) in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees. □

A *binary search tree* can support the operations search, insert, and delete among others. In fact, with a binary search tree, we can search for a data element both by key value and by rank (i.e., find an element with key x , find the fifth-smallest element, delete the element with key x , delete the fifth-smallest element, insert an element and determine its rank, and so on).

There is some redundancy in the definition of a binary search tree. Properties 2, 3, and 4 together imply that the keys must be distinct. So, property 1 can be replaced by the property: The root has a key.

Some examples of binary trees in which the elements have distinct keys are shown in Figure 2.11. The tree of Figure 2.11(a) is not a binary search tree, despite the fact that it satisfies properties 1, 2, and 3. The right subtree fails to satisfy property 4. This subtree is not a binary search tree, as its right subtree has a key value (22) that is smaller than that in the subtree's root (25). The binary trees of Figure 2.11(b) and (c) are binary search trees.

Searching a Binary Search Tree

Since the definition of a binary search tree is recursive, it is easiest to describe a recursive search method. Suppose we wish to search for an element with key x . An element could in general be an arbitrary structure that has as one of its fields a *key*. We assume for simplicity that the element just consists of a *key* and use the terms element and key interchangeably. We begin at the root. If the root is 0, then the search tree contains no elements and the search is unsuccessful. Otherwise, we compare x with the key in the root. If x equals this key, then the search terminates successfully. If x is less than the key in the root, then no element in the right subtree can have key value x , and only the left subtree is to be searched. If x is larger than the key in the root, only the right subtree needs to be searched. The subtrees can be searched recursively as in Algorithm 2.4. This function assumes a linked

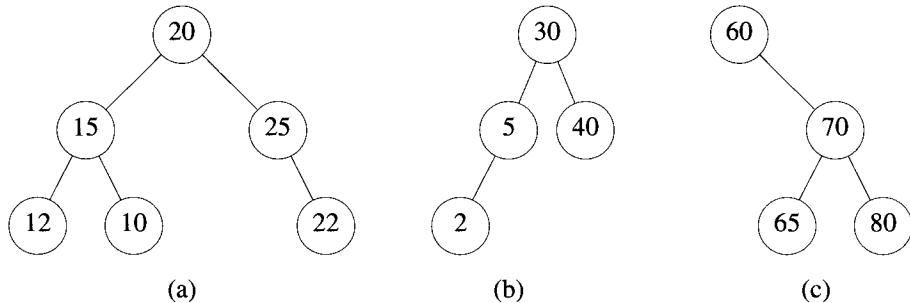


Figure 2.11 Binary trees

representation for the search tree. Each node has the three fields $lchild$, $rchild$, and $data$. The recursion of Algorithm 2.4 is easily replaced by a **while** loop, as in Algorithm 2.5.

```

1 Algorithm Search( $t, x$ )
2 {
3     if ( $t = 0$ ) then return 0;
4     else if ( $x = t \rightarrow data$ ) then return  $t$ ;
5         else if ( $x < t \rightarrow data$ ) then
6             return Search( $t \rightarrow lchild, x$ );
7         else return Search( $t \rightarrow rchild, x$ );
8 }
```

Algorithm 2.4 Recursive search of a binary search tree

If we wish to search by rank, each node should have an additional field *leftsize*, which is one plus the number of elements in the left subtree of the node. For the search tree of Figure 2.11(b), the nodes with keys 2, 5, 30, and 40, respectively, have *leftsize* equal to 1, 2, 3, and 1. Algorithm 2.6 searches for the *k*th-smallest element.

As can be seen, a binary search tree of height h can be searched by key as well as by rank in $O(h)$ time.

```

1  Algorithm |Search( $x$ )
2  {
3       $found := \text{false}$ ;
4       $t := \text{tree}$ ;
5      while ( $(t \neq 0)$  and not  $found$ ) do
6      {
7          if ( $x = (t \rightarrow \text{data})$ ) then  $found := \text{true}$ ;
8          else if ( $x < (t \rightarrow \text{data})$ ) then  $t := (t \rightarrow lchild)$ ;
9          else  $t := (t \rightarrow rchild)$ ;
10     }
11     if (not  $found$ ) then return 0;
12     else return  $t$ ;
13 }
```

Algorithm 2.5 Iterative search of a binary search tree

```

1  Algorithm Searchk( $k$ )
2  {
3       $found := \text{false}$ ;  $t := \text{tree}$ ;
4      while ( $(t \neq 0)$  and not  $found$ ) do
5      {
6          if ( $k = (t \rightarrow \text{leftsize})$ ) then  $found := \text{true}$ ;
7          else if ( $k < (t \rightarrow \text{leftsize})$ ) then  $t := (t \rightarrow lchild)$ ;
8          else
9          {
10               $k := k - (t \rightarrow \text{leftsize})$ ;
11               $t := (t \rightarrow rchild)$ ;
12          }
13      }
14      if (not  $found$ ) then return 0;
15      else return  $t$ ;
16 }
```

Algorithm 2.6 Searching a binary search tree by rank

Insertion into a Binary Search Tree

To insert a new element x , we must first verify that its key is different from those of existing elements. To do this, a search is carried out. If the search is unsuccessful, then the element is inserted at the point the search terminated. For instance, to insert an element with key 80 into the tree of Figure 2.12(a), we first search for 80. This search terminates unsuccessfully, and the last node examined is the one with key 40. The new element is inserted as the right child of this node. The resulting search tree is shown in Figure 2.12(b). Figure 2.12(c) shows the result of inserting the key 35 into the search tree of Figure 2.12(b).

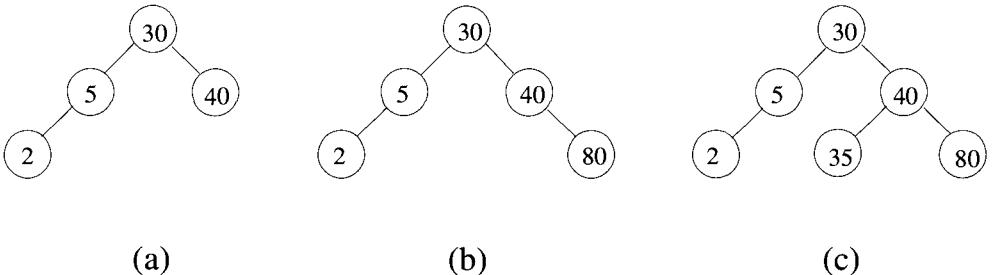


Figure 2.12 Inserting into a binary search tree

Algorithm 2.7 implements the insert strategy just described. If a node has a *leftsize* field, then this is updated too. Regardless, the insertion can be performed in $O(h)$ time, where h is the height of the search tree.

Deletion from a Binary Search Tree

Deletion of a leaf element is quite easy. For example, to delete 35 from the tree of Figure 2.12(c), the left-child field of its parent is set to 0 and the node disposed. This gives us the tree of Figure 2.12(b). To delete the 80 from this tree, the right-child field of 40 is set to 0; this gives the tree of Figure 2.12(a). Then the node containing 80 is disposed.

The deletion of a nonleaf element that has only one child is also easy. The node containing the element to be deleted is disposed, and the single child takes the place of the disposed node. So, to delete the element 5 from the tree of Figure 2.12(b), we simply change the pointer from the parent node (i.e., the node containing 30) to the single-child node (i.e., the node containing 2).

```

1  Algorithm Insert( $x$ )
2  // Insert  $x$  into the binary search tree.
3  {
4       $found := \text{false}$ ;
5       $p := tree$ ;
6      // Search for  $x$ .  $q$  is the parent of  $p$ .
7      while ( $(p \neq 0)$  and not  $found$ ) do
8      {
9           $q := p$ ; // Save  $p$ .
10         if ( $x = (p \rightarrow data)$ ) then  $found := \text{true}$ ;
11         else if ( $x < (p \rightarrow data)$ ) then  $p := (p \rightarrow lchild)$ ;
12         else  $p := (p \rightarrow rchild)$ ;
13     }
14
15     // Perform insertion.
16     if (not  $found$ ) then
17     {
18          $p := \text{new } TreeNode$ ;
19          $(p \rightarrow lchild) := 0$ ;  $(p \rightarrow rchild) := 0$ ;  $(p \rightarrow data) := x$ ;
20         if ( $tree \neq 0$ ) then
21         {
22             if ( $x < (q \rightarrow data)$ ) then  $(q \rightarrow lchild) := p$ ;
23             else  $(q \rightarrow rchild) := p$ ;
24         }
25     }
26 }
```

Algorithm 2.7 Insertion into a binary search tree

When the element to be deleted is in a nonleaf node that has two children, the element is replaced by either the largest element in its left subtree or the smallest one in its right subtree. Then we proceed to delete this replacing element from the subtree from which it was taken. For instance, if we wish to delete the element with key 30 from the tree of Figure 2.13(a), then we replace it by either the largest element, 5, in its left subtree or the smallest element, 40, in its right subtree. Suppose we opt for the largest element in the left subtree. The 5 is moved into the root, and the tree of Figure 2.13(b) is obtained. Now we must delete the second 5. Since this node has only one child, the pointer from its parent is changed to point to this child. The tree of Figure 2.13(c) is obtained. We can verify that regardless of whether the replacing element is the largest in the left subtree or the smallest in the right subtree, it is originally in a node with a degree of at most one. So, deleting it from this node is quite easy. We leave the writing of the deletion procedure as an exercise. It should be evident that a deletion can be performed in $O(h)$ time if the search tree has a height of h .

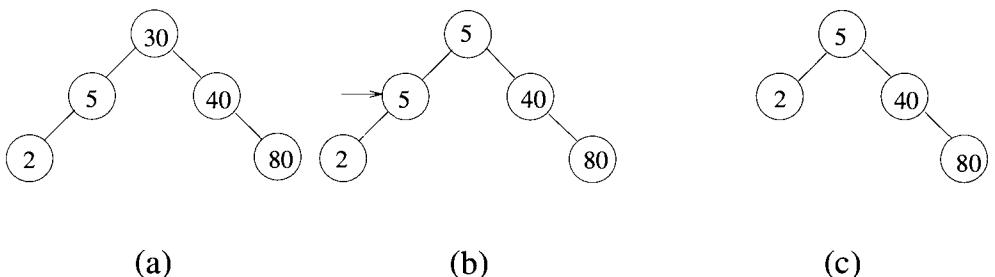


Figure 2.13 Deletion from a binary search tree

Height of a Binary Search Tree

Unless care is taken, the height of a binary search tree with n elements can become as large as n . This is the case, for instance, when Algorithm 2.7 is used to insert the keys $[1, 2, 3, \dots, n]$, in this order, into an initially empty binary search tree. It can, however, be shown that when insertions and deletions are made at random using the procedures given here, the height of the binary search tree is $O(\log n)$ on the average.

Search trees with a worst-case height of $O(\log n)$ are called *balanced search trees*. Balanced search trees that permit searches, inserts, and deletes to be performed in $O(\log n)$ time are listed in Table 2.1. Examples include AVL trees, 2-3 trees, Red-Black trees, and B-trees. On the other hand splay trees

take $O(\log n)$ time for each of these operations in the *amortized* sense. A description of these balanced trees can be found in the book by E. Horowitz, S. Sahni, and D. Mehta cited at the end of this chapter.

Data structure	search	insert	delete
Binary search tree	$O(n)$ (wc)	$O(n)$ (wc)	$O(n)$ (wc)
	$O(\log n)$ (av)	$O(\log n)$ (av)	$O(\log n)$ (av)
AVL tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)	$O(\log n)$ (wc)
2-3 tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Red-Black tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)	$O(\log n)$ (wc)
B-tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Splay tree	$O(\log n)$ (am)	$O(\log n)$ (am)	$O(\log n)$ (am)

Table 2.1 Summary of dictionary implementations. Here (wc) stands for worst case, (av) for average case, and (am) for amortized cost.

2.3.2 Cost Amortization

Suppose that a sequence I1, I2, D1, I3, I4, I5, I6, D2, I7 of insert and delete operations is performed on a set. Assume that the *actual cost* of each of the seven inserts is one. (We use the terms *cost* and *complexity* interchangeably.) By this, we mean that each insert takes one unit of time. Further, suppose that the delete operations D1 and D2 have an actual cost of eight and ten, respectively. So, the total cost of the sequence of operations is 25.

In an amortization scheme we charge some of the actual cost of an operation to other operations. This reduces the charged cost of some operations and increases that of others. The *amortized cost* of an operation is the total cost charged to it. The cost transferring (amortization) scheme is required to be such that the sum of the amortized costs of the operations is greater than or equal to the sum of their actual costs. If we charge one unit of the cost of a delete operation to each of the inserts since the last delete operation (if any), then two units of the cost of D1 get transferred to I1 and I2 (the charged cost of each increases by one), and four units of the cost of D2 get transferred to I3 to I6. The amortized cost of each of I1 to I6 becomes two, that of I7 becomes equal to its actual cost (that is, one), and that of each of D1 and D2 becomes 6. The sum of the amortized costs is 25, which is the same as the sum of the actual costs.

Now suppose we can prove that no matter what sequence of insert and delete operations is performed, we can charge costs in such a way that the amortized cost of each insertion is no more than two and that of each deletion

is no more than six. This enables us to claim that the actual cost of any insert/delete sequence is no more than $2 * i + 6 * d$, where i and d are, respectively, the number of insert and delete operations in the sequence. Suppose that the actual cost of a deletion is no more than ten and that of an insertion is one. Using actual costs, we can conclude that the sequence cost is no more than $i + 10 * d$. Combining these two bounds, we obtain $\min\{2 * i + 6 * d, i + 10 * d\}$ as a bound on the sequence cost. Hence, using the notion of cost amortization, we can obtain tighter bounds on the complexity of a sequence of operations.

The amortized time complexity to perform insert, delete, and search operations in splay trees is $O(\log n)$. This amortization is over n operations. In other words, the total time taken for processing an arbitrary sequence of n operations is $O(n \log n)$. Some operations may take much longer than $O(\log n)$ time, but when amortized over n operations, each operation costs $O(\log n)$ time.

EXERCISES

1. Write an algorithm to delete an element x from a binary search tree t . What is the time complexity of your algorithm?
2. Present an algorithm to start with an initially empty binary search tree and make n random insertions. Use a uniform random number generator to obtain the values to be inserted. Measure the height of the resulting binary search tree and divide this height by $\log_2 n$. Do this for $n = 100, 500, 1,000, 2,000, 3,000, \dots, 10,000$. Plot the ratio height/ $\log_2 n$ as a function of n . The ratio should be approximately constant (around 2). Verify that this is so.
3. Suppose that each node in a binary search tree also has the field *leftsize* as described in the text. Design an algorithm to insert an element x into such a binary search tree. The complexity of your algorithm should be $O(h)$, where h is the height of the search tree. Show that this is the case.
4. Do Exercise 3, but this time present an algorithm to delete the element with the k th-smallest key in the binary search tree.
5. Find an efficient data structure for representing a subset S of the integers from 1 to n . Operations we wish to perform on the set are
 - **INSERT(i)** to insert the integer i to the set S . If i is already in the set, this instruction must be ignored.
 - **DELETE** to delete an arbitrary member from the set.
 - **MEMBER(i)** to check whether i is a member of the set.

Your data structure should enable each one of the above operations in constant time (irrespective of the cardinality of S).

6. Any algorithm that merges two sorted lists of size n and m , respectively, must make at least $n + m - 1$ comparisons in the worst case. What implications does this have on the run time of any comparison-based algorithm that combines two binary search trees that have n and m elements, respectively?
7. It is known that every comparison-based algorithm to sort n elements must make $O(n \log n)$ comparisons in the worst case. What implications does this have on the complexity of initializing a binary search tree with n elements?

2.4 PRIORITY QUEUES

Any data structure that supports the operations of search min (or max), insert, and delete min (or max, respectively) is called a *priority queue*.

Example 2.2 Suppose that we are selling the services of a machine. Each user pays a fixed amount per use. However, the time needed by each user is different. We wish to maximize the returns from this machine under the assumption that the machine is not to be kept idle unless no user is available. This can be done by maintaining a priority queue of all persons waiting to use the machine. Whenever the machine becomes available, the user with the smallest time requirement is selected. Hence, a priority queue that supports delete min is required. When a new user requests the machine, his or her request is put into the priority queue.

If each user needs the same amount of time on the machine but people are willing to pay different amounts for the service, then a priority queue based on the amount of payment can be maintained. Whenever the machine becomes available, the user willing to pay the most is selected. This requires a delete max operation. \square

Example 2.3 Suppose that we are simulating a large factory. This factory has many machines and many jobs that require processing on some of the machines. An *event* is said to occur whenever a machine completes the processing of a job. When an event occurs, the job has to be moved to the queue for the next machine (if any) that it needs. If this queue is empty, the job can be assigned to the machine immediately. Also, a new job can be scheduled on the machine that has become idle (provided that its queue is not empty).

To determine the occurrence of events, a priority queue is used. This queue contains the finish time of all jobs that are presently being worked on.

The next event occurs at the least time in the priority queue. So, a priority queue that supports delete min can be used in this application. \square

The simplest way to represent a priority queue is as an unordered linear list. Suppose that we have n elements in this queue and the delete max operation is to be supported. If the list is represented sequentially, additions are most easily performed at the end of this list. Hence, the insert time is $\Theta(1)$. A deletion requires a search for the element with the largest key, followed by its deletion. Since it takes $\Theta(n)$ time to find the largest element in an n -element unordered list, the delete time is $\Theta(n)$. Each deletion takes $\Theta(n)$ time. An alternative is to use an ordered linear list. The elements are in nondecreasing order if a sequential representation is used. The delete time for each representation is $\Theta(1)$ and the insert time $O(n)$. When a *max heap* is used, both additions and deletions can be performed in $O(\log n)$ time.

2.4.1 Heaps

Definition 2.4 [Heap] A *max (min) heap* is a complete binary tree with the property that the value at each node is at least as large as (as small as) the values at its children (if they exist). Call this property the *heap property*. \square

In this section we study in detail an efficient way of realizing a priority queue. We might first consider using a queue since inserting new elements would be very efficient. But finding the largest element would necessitate a scan of the entire queue. A second suggestion might be to use a sorted list that is stored sequentially. But an insertion could require moving all of the items in the list. What we want is a data structure that allows *both* operations to be done efficiently. One such data structure is the max heap.

The definition of a max heap implies that one of the largest elements is at the root of the heap. If the elements are distinct, then the root contains the largest item. A max heap can be implemented using an array $a[]$.

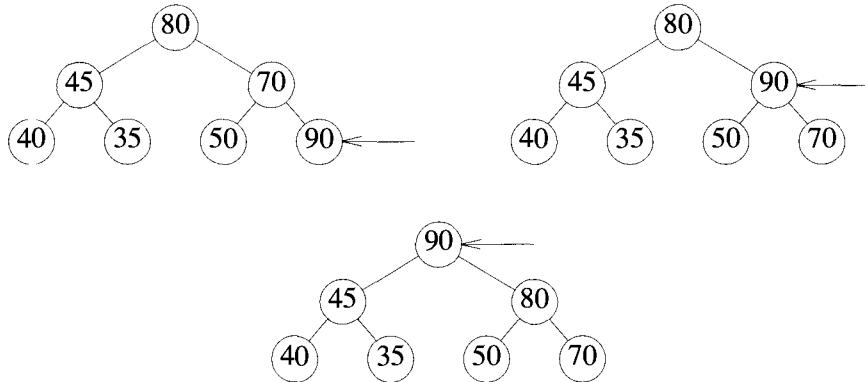
To insert an element into the heap, one adds it “at the bottom” of the heap and then compares it with its parent, grandparent, greatgrandparent, and so on, until it is less than or equal to one of these values. Algorithm `Insert` (Algorithm 2.8) describes this process in detail.

Figure 2.14 shows one example of how `Insert` would insert a new value into an existing heap of six elements. It is clear from Algorithm 2.8 and Figure 2.14 that the time for `Insert` can vary. In the best case the new element is correctly positioned initially and no values need to be rearranged. In the worst case the number of executions of the **while** loop is proportional to the number of levels in the heap. Thus if there are n elements in the heap, inserting a new element takes $\Theta(\log n)$ time in the worst case.

```

1  Algorithm Insert( $a, n$ )
2  {
3      // Inserts  $a[n]$  into the heap which is stored in  $a[1 : n - 1]$ .
4       $i := n; item := a[n];$ 
5      while (( $i > 1$ ) and ( $a[\lfloor i/2 \rfloor] < item$ )) do
6      {
7           $a[i] := a[\lfloor i/2 \rfloor]; i := \lfloor i/2 \rfloor;$ 
8      }
9       $a[i] := item; \text{return true};$ 
10 }

```

Algorithm 2.8 Insertion into a heap**Figure 2.14** Action of Insert inserting 90 into an existing heap

To delete the maximum key from the max heap, we use an algorithm called `Adjust`. `Adjust` takes as input the array $a[]$ and the integers i and n . It regards $a[1 : n]$ as a complete binary tree. If the subtrees rooted at $2i$ and $2i + 1$ are already max heaps, then `Adjust` will rearrange elements of $a[]$ such that the tree rooted at i is also a max heap. The maximum element from the max heap $a[1 : n]$ can be deleted by deleting the root of the corresponding complete binary tree. The last element of the array, that is, $a[n]$, is copied to the root, and finally we call $\text{Adjust}(a, 1, n - 1)$. Both `Adjust` and `DelMax` are described in Algorithm 2.9.

```

1  Algorithm Adjust( $a, i, n$ )
2  // The complete binary trees with roots  $2i$  and  $2i + 1$  are
3  // combined with node  $i$  to form a heap rooted at  $i$ . No
4  // node has an address greater than  $n$  or less than 1.
5  {
6       $j := 2i$ ;  $item := a[i]$ ;
7      while ( $j \leq n$ ) do
8      {
9          if (( $j < n$ ) and ( $a[j] < a[j + 1]$ )) then  $j := j + 1$ ;
10         // Compare left and right child
11         // and let  $j$  be the larger child.
12         if ( $item \geq a[j]$ ) then break;
13         // A position for  $item$  is found.
14          $a[\lfloor j/2 \rfloor] := a[j]$ ;  $j := 2j$ ;
15     }
16      $a[\lfloor j/2 \rfloor] := item$ ;
17 }

1  Algorithm DelMax( $a, n, x$ )
2  // Delete the maximum from the heap  $a[1 : n]$  and store it in  $x$ .
3  {
4      if ( $n = 0$ ) then
5      {
6          write ("heap is empty"); return false;
7      }
8       $x := a[1]$ ;  $a[1] := a[n]$ ;
9      Adjust( $a, 1, n - 1$ ); return true;
10 }
```

Algorithm 2.9 Deletion from a heap

Note that the worst-case run time of `Adjust` is also proportional to the height of the tree. Therefore, if there are n elements in a heap, deleting the maximum can be done in $O(\log n)$ time.

To sort n elements, it suffices to make n insertions followed by n deletions from a heap. Algorithm 2.10 has the details. Since insertion and deletion take $O(\log n)$ time each in the worst case, this sorting algorithm has a time complexity of $O(n \log n)$.

```

1  Algorithm Sort( $a, n$ )
2  // Sort the elements  $a[1 : n]$ .
3  {
4      for  $i := 1$  to  $n$  do Insert( $a, i$ );
5      for  $i := n$  to 1 step  $-1$  do
6          {
7              DelMax( $a, i, x$ );  $a[i] := x$ ;
8          }
9  }
```

Algorithm 2.10 A sorting algorithm

It turns out that we can insert n elements into a heap faster than we can apply `Insert` n times. Before getting into the details of the new algorithm, let us consider how the n inserts take place. Figure 2.15 shows how the data (40, 80, 35, 90, 45, 50, and 70) move around until a heap is created when using `Insert`. Trees to the left of any \rightarrow represent the state of the array $a[1 : i]$ before some call of `Insert`. Trees to the right of \rightarrow show how the array was altered by `Insert` to produce a heap. The array is drawn as a complete binary tree for clarity.

The data set that causes the heap creation method using `Insert` to behave in the worst way is a set of elements in ascending order. Each new element rises to become the new root. There are at most 2^{i-1} nodes on level i of a complete binary tree, $1 \leq i \leq \lceil \log_2(n+1) \rceil$. For a node on level i the distance to the root is $i - 1$. Thus the worst-case time for heap creation using `Insert` is

$$\sum_{1 \leq i \leq \lceil \log_2(n+1) \rceil} (i-1)2^{i-1} < \lceil \log_2(n+1) \rceil 2^{\lceil \log_2(n+1) \rceil} = O(n \log n)$$

A surprising fact about `Insert` is that its average behavior on n random inputs is asymptotically faster than its worst case, $O(n)$ rather than $O(n \log n)$.

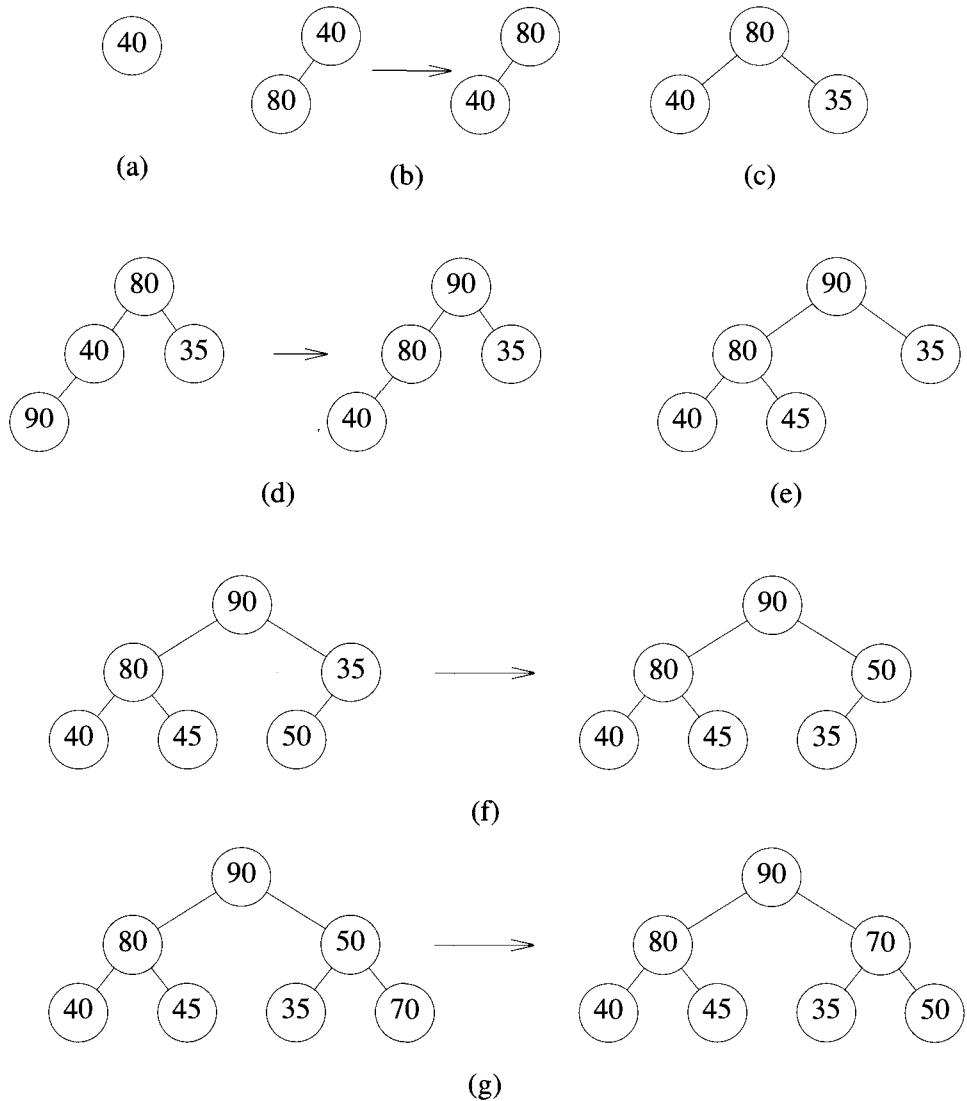


Figure 2.15 Forming a heap from the set $\{40, 80, 35, 90, 45, 50, 70\}$

This implies that on the average each new value only rises a constant number of levels in the tree.

It is possible to devise an algorithm that can perform n inserts in $O(n)$ time rather than $O(n \log n)$. This reduction is achieved by an algorithm that regards any array $a[1 : n]$ as a complete binary tree and works from the leaves up to the root, level by level. At each level, the left and right subtrees of any node are heaps. Only the value in the root node may violate the heap property.

Given n elements in $a[1 : n]$, we can create a heap by applying `Adjust`. It is easy to see that leaf nodes are already heaps. So we can begin by calling `Adjust` for the parents of leaf nodes and then work our way up, level by level, until the root is reached. The resultant algorithm is `Heapify` (Algorithm 2.11). In Figure 2.16 we observe the action of `Heapify` as it creates a heap out of the given seven elements. The initial tree is drawn in Figure 2.16(a). Since $n = 7$, the first call to `Adjust` has $i = 3$. In Figure 2.16(b) the three elements 118, 151, and 132 are rearranged to form a heap. Subsequently `Adjust` is called with $i = 2$ and $i = 1$; this gives the trees in Figure 2.16(c) and (d).

```

1  Algorithm Heapify( $a, n$ )
2  // Readjust the elements in  $a[1 : n]$  to form a heap.
3  {
4      for  $i := \lfloor n/2 \rfloor$  to 1 step -1 do Adjust( $a, i, n$ );
5  }
```

Algorithm 2.11 Creating a heap out of n arbitrary elements

For the worst-case analysis of `Heapify` let $2^{k-1} \leq n < 2^k$, where $k = \lceil \log(n+1) \rceil$, and recall that the levels of the n -node complete binary tree are numbered 1 to k . The worst-case number of iterations for `Adjust` is $k - i$ for a node on level i . The total time for `Heapify` is proportional to

$$\sum_{1 \leq i \leq k} 2^{i-1}(k-i) = \sum_{1 \leq i \leq k-1} i2^{k-i-1} \leq n \sum_{1 \leq i \leq k-1} i/2^i \leq 2n = O(n) \quad (2.1)$$

Comparing `Heapify` with the repeated use of `Insert`, we see that the former is faster in the worst case, requiring $O(n)$ versus $O(n \log n)$ operations. However, `Heapify` requires that all the elements be available before heap creation begins. Using `Insert`, we can add a new element into the heap at any time.

Our discussion on insert, delete, and so on, so far has been with respect to a max heap. It should be easy to see that a parallel discussion could have

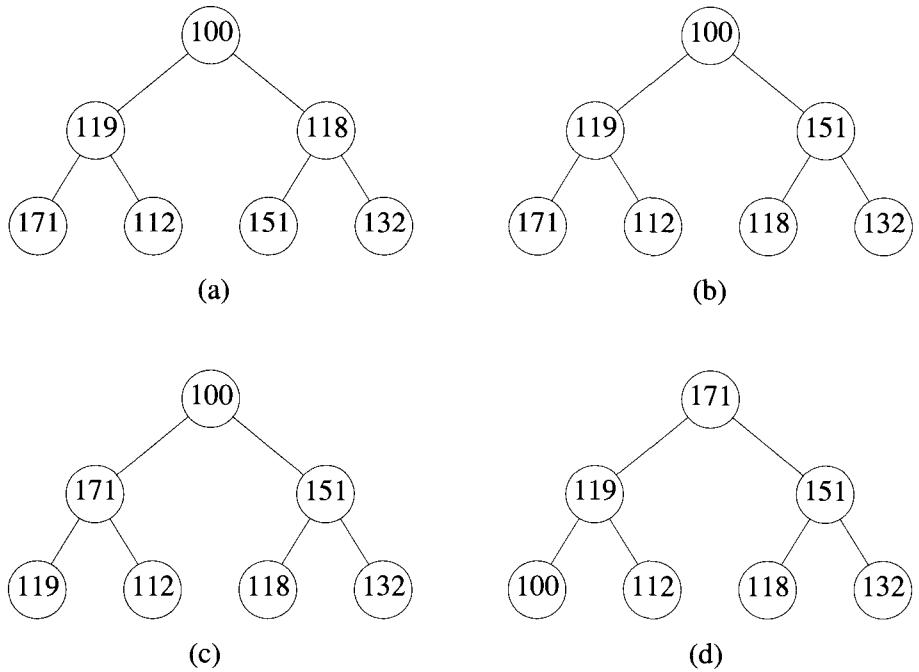


Figure 2.16 Action of `Heapify(a, 7)` on the data (100, 119, 118, 171, 112, 151, and 132)

been carried out with respect to a min heap. For a min heap it is possible to delete the smallest element in $O(\log n)$ time and also to insert an element in $O(\log n)$ time.

2.4.2 Heapsort

The best-known example of the use of a heap arises in its application to sorting. A conceptually simple sorting strategy has been given before, in which the maximum value is continually removed from the remaining unsorted elements. A sorting algorithm that incorporates the fact that n elements can be inserted in $O(n)$ time is given in Algorithm 2.12.

```

1  Algorithm HeapSort( $a, n$ )
2  //  $a[1 : n]$  contains  $n$  elements to be sorted. HeapSort
3  // rearranges them inplace into nondecreasing order.
4  {
5      Heapify( $a, n$ ); // Transform the array into a heap.
6      // Interchange the new maximum with the element
7      // at the end of the array.
8      for  $i := n$  to 2 step -1 do
9      {
10          $t := a[i]; a[i] := a[1]; a[1] := t;$ 
11         Adjust( $a, 1, i - 1$ );
12     }
13 }
```

Algorithm 2.12 Heapsort

Though the call of `Heapify` requires only $O(n)$ operations, `Adjust` possibly requires $O(\log n)$ operations for each invocation. Thus the worst-case time is $O(n \log n)$. Notice that the storage requirements, besides $a[1 : n]$, are only for a few simple variables.

A number of other data structures can also be used to implement a priority queue. Examples include the binomial heap, deap, Fibonacci heap, and so on. A description of these can be found in the book by E. Horowitz, S. Sahni, and D. Mehta. Table 2.2 summarizes the performances of these data structures. Many of these data structures support the operations of deleting and searching for arbitrary elements (Red-Black tree being an example), in addition to the ones needed for a priority queue.

Data structure	insert	delete min
Min heap	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Min-max heap	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Deap	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Leftist tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Binomial heap	$O(\log n)$ (wc) $O(1)$ (am)	$O(\log n)$ (wc) $O(\log n)$ (am)
Fibonacci heap	$O(\log n)$ (wc) $O(1)$ (am)	$O(\log n)$ (wc) $O(\log n)$ (am)
2-3 tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Red-Black tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)

Table 2.2 Performances of different data structures when realizing a priority queue. Here (wc) stands for worst case and (am) denotes amortized cost.

EXERCISES

1. Verify for yourself that algorithm `Insert` (Algorithm 2.8) uses only a constant number of comparisons to insert a random element into a heap by performing an appropriate experiment.
2. (a) Equation 2.1 makes use of the fact that the sum $\sum_{i=1}^{\infty} \frac{i}{2^i}$ converges. Prove this fact.
 (b) Use induction to show that $\sum_{i=1}^k 2^{i-1}(k-i) = 2^k - k - 1$, $k \geq 1$.
3. Program and run algorithm `HeapSort` (Algorithm 2.12) and compare its time with the time of any of the sorting algorithms discussed in Chapter 1.
4. Design a data structure that supports the following operations: INSERT and MIN. The worst-case run time should be $O(1)$ for each of these operations.
5. Notice that a binary search tree can be used to implement a priority queue.
 - (a) Present an algorithm to delete the largest element in a binary search tree. Your procedure should have complexity $O(h)$, where h is the height of the search tree. Since h is $O(\log n)$ on average, you can perform each of the priority queue operations in average time $O(\log n)$.

- (b) Compare the performances of max heaps and binary search trees as data structures for priority queues. For this comparison, generate random sequences of insert and delete max operations and measure the total time taken for each sequence by each of these data structures.
6. Input is a sequence X of n keys with many duplications such that the number of distinct keys is d ($< n$). Present an $O(n \log d)$ -time sorting algorithm for this input. (For example, if $X = 5, 6, 1, 18, 6, 4, 4, 1, 5, 17$, the number of distinct keys in X is six.)

2.5 SETS AND DISJOINT SET UNION

2.5.1 Introduction

In this section we study the use of forests in the representation of sets. We shall assume that the elements of the sets are the numbers $1, 2, 3, \dots, n$. These numbers might, in practice, be indices into a symbol table in which the names of the elements are stored. We assume that the sets being represented are pairwise disjoint (that is, if S_i and S_j , $i \neq j$, are two sets, then there is no element that is in both S_i and S_j). For example, when $n = 10$, the elements can be partitioned into three disjoint sets, $S_1 = \{1, 7, 8, 9\}$, $S_2 = \{2, 5, 10\}$, and $S_3 = \{3, 4, 6\}$. Figure 2.17 shows one possible representation for these sets. In this representation, each set is represented as a tree. Notice that for each set we have linked the nodes from the children to the parent, rather than our usual method of linking from the parent to the children. The reason for this change in linkage becomes apparent when we discuss the implementation of set operations.

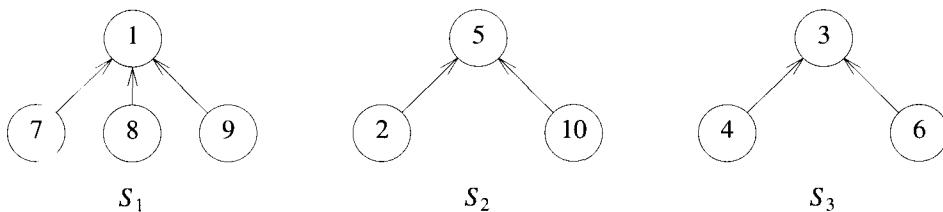


Figure 2.17 Possible tree representation of sets

The operations we wish to perform on these sets are:

1. **Disjoint set union.** If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j =$ all elements x such that x is in S_i or S_j . Thus, $S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$. Since we have assumed that all sets are disjoint, we can assume that following the union of S_i and S_j , the sets S_i and S_j do not exist independently; that is, they are replaced by $S_i \cup S_j$ in the collection of sets.
2. **Find(i).** Given the element i , find the set containing i . Thus, 4 is in set S_3 , and 9 is in set S_1 .

2.5.2 Union and Find Operations

Let us consider the union operation first. Suppose that we wish to obtain the union of S_1 and S_2 (from Figure 2.17). Since we have linked the nodes from children to parent, we simply make one of the trees a subtree of the other. $S_1 \cup S_2$ could then have one of the representations of Figure 2.18.

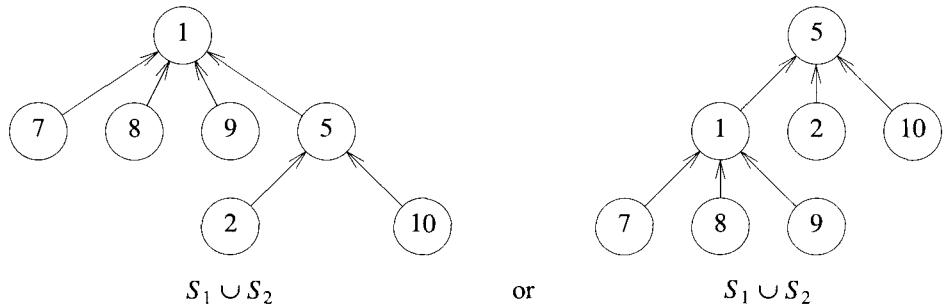


Figure 2.18 Possible representations of $S_1 \cup S_2$

To obtain the union of two sets, all that has to be done is to set the parent field of one of the roots to the other root. This can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set. If, in addition, each root has a pointer to the set name, then to determine which set an element is currently in, we follow parent links to the root of its tree and use the pointer to the set name. The data representation for S_1 , S_2 , and S_3 may then take the form shown in Figure 2.19.

In presenting the union and find algorithms, we ignore the set names and identify sets just by the roots of the trees representing them. This simplifies

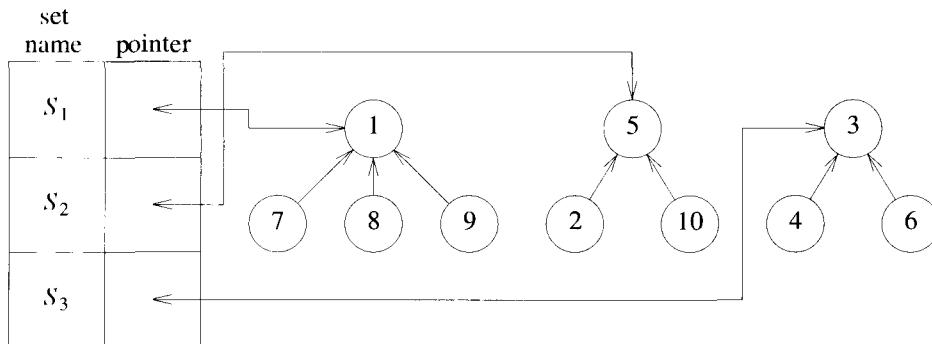


Figure 2.19 Data representation for S_1 , S_2 , and S_3

the discussion. The transition to set names is easy. If we determine that element i is in a tree with root j , and j has a pointer to entry k in the set name table, then the set name is just $name[k]$. If we wish to unite sets S_i and S_j , then we wish to unite the trees with roots $\text{FindPointer}(S_i)$ and $\text{FindPointer}(S_j)$. Here FindPointer is a function that takes a set name and determines the root of the tree that represents it. This is done by an examination of the [set name, pointer] table. In many applications the set name is just the element at the root. The operation of $\text{Find}(i)$ now becomes: Determine the root of the tree containing element i . The function $\text{Union}(i, j)$ requires two trees with roots i and j be joined. Also to simplify, assume that the set elements are the numbers 1 through n .

Since the set elements are numbered 1 through n , we represent the tree nodes using an array $p[1 : n]$, where n is the maximum number of elements. The i th element of this array represents the tree node that contains element i . This array element gives the parent pointer of the corresponding tree node. Figure 2.20 shows this representation of the sets S_1 , S_2 , and S_3 of Figure 2.17. Notice that root nodes have a parent of -1 .

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	5	-1	3	-1	3	1	1	1	5

Figure 2.20 Array representation of S_1 , S_2 , and S_3 of Figure 2.17

We can now implement $Find(i)$ by following the indices, starting at i until we reach a node with parent value -1 . For example, $Find(6)$ starts at 6 and then moves to 6's parent, 3. Since $p[3]$ is negative, we have reached the root. The operation $Union(i, j)$ is equally simple. We pass in two trees with roots i and j . Adopting the convention that the first tree becomes a subtree of the second, the statement $p[i] := j$; accomplishes the union.

```

1  Algorithm SimpleUnion( $i, j$ )
2  {
3       $p[i] := j;$ 
4  }

1  Algorithm SimpleFind( $i$ )
2  {
3      while ( $p[i] \geq 0$ ) do  $i := p[i];$ 
4      return  $i;$ 
5  }

```

Algorithm 2.13 Simple algorithms for union and find

Algorithm 2.13 gives the descriptions of the union and find operations just discussed. Although these two algorithms are very easy to state, their performance characteristics are not very good. For instance, if we start with q elements each in a set of its own (that is, $S_i = \{i\}$, $1 \leq i \leq q$), then the initial configuration consists of a forest with q nodes, and $p[i] = 0$, $1 \leq i \leq q$. Now let us process the following sequence of *union-find* operations:

$$Union(1, 2), \ Union(2, 3), \ Union(3, 4), \ Union(4, 5), \dots, \ Union(n - 1, n)$$

$$Find(1), \ Find(2), \dots, \ Find(n)$$

This sequence results in the degenerate tree of Figure 2.21.

Since the time taken for a union is constant, the $n - 1$ unions can be processed in time $O(n)$. However, each find requires following a sequence of *parent* pointers from the element to be found to the root. Since the time required to process a find for an element at level i of a tree is $O(i)$, the total time needed to process the n finds is $O(\sum_{i=1}^n i) = O(n^2)$.

We can improve the performance of our union and find algorithms by avoiding the creation of degenerate trees. To accomplish this, we make use of a weighting rule for $Union(i, j)$.

**Figure 2.21** Degenerate tree

Definition 2.5 [Weighting rule for $\text{Union}(i,j)$] If the number of nodes in the tree with root i is less than the number in the tree with root j , then make j the parent of i ; otherwise make i the parent of j . \square

When we use the weighting rule to perform the sequence of set unions given before, we obtain the trees of Figure 2.22. In this figure, the unions have been modified so that the input parameter values correspond to the roots of the trees to be combined.

To implement the weighting rule, we need to know how many nodes there are in every tree. To do this easily, we maintain a *count* field in the root of every tree. If i is a root node, then $\text{count}[i]$ equals the number of nodes in that tree. Since all nodes other than the roots of trees have a positive number in the *p* field, we can maintain the count in the *p* field of the roots as a negative number.

Using this convention, we obtain Algorithm 2.14. In this algorithm the time required to perform a union has increased somewhat but is still bounded by a constant (that is, it is $O(1)$). The find algorithm remains unchanged. The maximum time to perform a find is determined by Lemma 2.3.

Lemma 2.3 Assume that we start with a forest of trees, each having one node. Let T be a tree with m nodes created as a result of a sequence of unions each performed using `WeightedUnion`. The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$.

Proof: The lemma is clearly true for $m = 1$. Assume it is true for all trees with i nodes, $i \leq m - 1$. We show that it is also true for $i = m$.

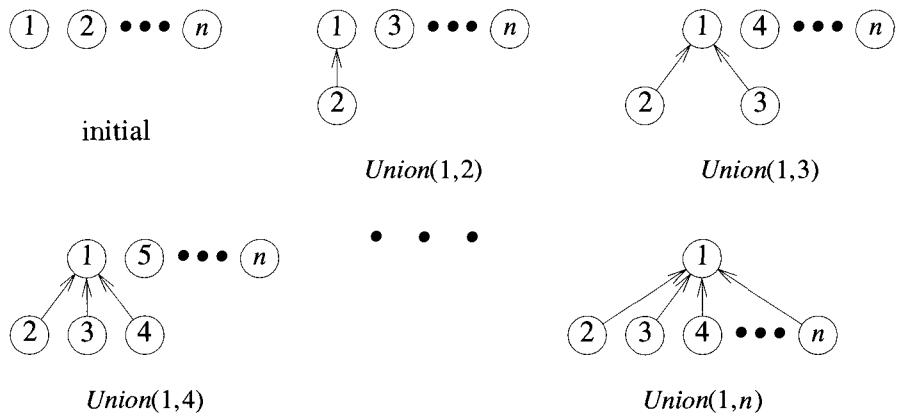


Figure 2.22 Trees obtained using the weighting rule

```

1  Algorithm WeightedUnion( $i, j$ )
2  // Union sets with roots  $i$  and  $j$ ,  $i \neq j$ , using the
3  // weighting rule.  $p[i] = -count[i]$  and  $p[j] = -count[j]$ .
4  {
5      temp :=  $p[i] + p[j]$ ;
6      if ( $p[i] > p[j]$ ) then
7          { //  $i$  has fewer nodes.
8               $p[i] := j$ ;  $p[j] := temp$ ;
9          }
10         else
11             { //  $j$  has fewer or equal nodes.
12                  $p[j] := i$ ;  $p[i] := temp$ ;
13             }
14 }
```

Algorithm 2.14 Union algorithm with weighting rule

Let T be a tree with m nodes created by `WeightedUnion`. Consider the last union operation performed, $\text{Union}(k, j)$. Let a be the number of nodes in tree j , and $m - a$ the number in k . Without loss of generality we can assume $1 \leq a \leq \frac{m}{2}$. Then the height of T is either the same as that of k or is one more than that of j . If the former is the case, the height of T is $\leq \lfloor \log_2(m - a) \rfloor + 1 \leq \lfloor \log_2 m \rfloor + 1$. However, if the latter is the case, the height of T is $\leq \lfloor \log_2 a \rfloor + 2 \leq \lfloor \log_2 \frac{m}{2} \rfloor + 2 \leq \lfloor \log_2 m \rfloor + 1$. \square

Example 2.4 shows that the bound of Lemma 2.3 is achievable for some sequence of unions.

Example 2.4 Consider the behavior of `WeightedUnion` on the following sequence of unions starting from the initial configuration $p[i] = -\text{count}[i] = -1$, $1 \leq i \leq 8 = n$:

$$\begin{aligned} & \text{Union}(1, 2), \quad \text{Union}(3, 4), \quad \text{Union}(5, 6), \quad \text{Union}(7, 8), \\ & \quad \text{Union}(1, 3), \quad \text{Union}(5, 7), \quad \text{Union}(1, 5) \end{aligned}$$

The trees of Figure 2.23 are obtained. As is evident, the height of each tree with m nodes is $\lfloor \log_2 m \rfloor + 1$. \square

From Lemma 2.3, it follows that the time to process a find is $O(\log m)$ if there are m elements in a tree. If an intermixed sequence of $u - 1$ union and f find operations is to be processed, the time becomes $O(u + f \log u)$, as no tree has more than u nodes in it. Of course, we need $O(n)$ additional time to initialize the n -tree forest.

Surprisingly, further improvement is possible. This time the modification is made in the find algorithm using the *collapsing rule*.

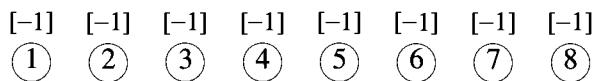
Definition 2.6 [Collapsing rule]: If j is a node on the path from i to its root and $p[i] \neq \text{root}[i]$, then set $p[j]$ to $\text{root}[i]$. \square

`CollapsingFind` (Algorithm 2.15) incorporates the collapsing rule.

Example 2.5 Consider the tree created by `WeightedUnion` on the sequence of unions of Example 2.4. Now process the following eight finds:

$$\text{Find}(8), \quad \text{Find}(8), \dots, \quad \text{Find}(8)$$

If `SimpleFind` is used, each $\text{Find}(8)$ requires going up three parent link fields for a total of 24 moves to process all eight finds. When `CollapsingFind` is used, the first $\text{Find}(8)$ requires going up three links and then resetting two links. Note that even though only two parent links need to be reset, `CollapsingFind` will reset three (the parent of 5 is reset to 1). Each of the remaining seven finds requires going up only one link field. The total cost is now only 13 moves. \square



(a) Initial height-1 trees

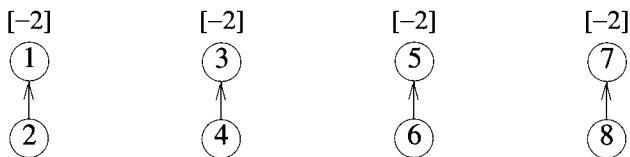
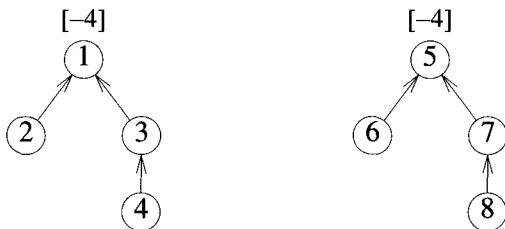
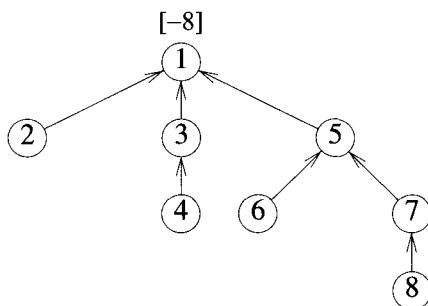
(b) Height-2 trees following *Union*(1,2), (3,4), (5,6), and (7,8)(c) Height-3 trees following *Union*(1,3) and (5,7)(d) Height-4 tree following *Union*(1,5)

Figure 2.23 Trees achieving worst-case bound

```

1  Algorithm CollapsingFind( $i$ )
2  // Find the root of the tree containing element  $i$ . Use the
3  // collapsing rule to collapse all nodes from  $i$  to the root.
4  {
5       $r := i;$ 
6      while ( $p[r] > 0$ ) do  $r := p[r];$  // Find the root.
7      while ( $i \neq r$ ) do // Collapse nodes from  $i$  to root  $r$ .
8      {
9           $s := p[i]; p[i] := r; i := s;$ 
10     }
11     return  $r;$ 
12 }
```

Algorithm 2.15 Find algorithm with collapsing rule

In the algorithms `WeightedUnion` and `CollapsingFind`, use of the collapsing rule roughly doubles the time for an individual find. However, it reduces the worst-case time over a sequence of finds. The worst-case complexity of processing a sequence of unions and finds using `WeightedUnion` and `CollapsingFind` is stated in Lemma 2.4. This lemma makes use of a function $\alpha(p, q)$ that is related to a functional inverse of Ackermann's function $A(i, j)$. These functions are defined as follows:

$$\begin{aligned} A(1, j) &= 2^j && \text{for } j \geq 1 \\ A(i, 1) &= A(i - 1, 2) && \text{for } i \geq 2 \\ A(i, j) &= A(i - 1, A(i, j - 1)) && \text{for } i, j \geq 2 \end{aligned}$$

$$\alpha(p, q) = \min\{z \geq 1 | A(z, \lfloor \frac{p}{q} \rfloor) > \log_2 q\}, \quad p \geq q \geq 1$$

The function $A(i, j)$ is a very rapidly growing function. Consequently, α grows very slowly as p and q are increased. In fact, since $A(3, 1) = 16$, $\alpha(p, q) \leq 3$ for $q < 2^{16} = 65,536$ and $p \geq q$. Since $A(4, 1)$ is a very large number and in our application q is the number n of set elements and p is $n + f$ (f is the number of finds), $\alpha(p, q) \leq 4$ for all practical purposes.

Lemma 2.4 [Tarjan and Van Leeuwen] Assume that we start with a forest of trees, each having one node. Let $T(f, u)$ be the maximum time required to process any intermixed sequence of f finds and u unions. Assume that $u \geq \frac{n}{2}$. Then

$$k_1[n + f\alpha(f + n, n)] \leq T(f, u) \leq k_2[n + f\alpha(f + n, n)]$$

for some positive constants k_1 and k_2 . \square

The requirement that $u \geq \frac{n}{2}$ in Lemma 2.4 is really not significant, as when $u < \frac{n}{2}$, some elements are involved in no union operation. These elements remain in singleton sets throughout the sequence of union and find operations and can be eliminated from consideration, as find operations that involve these can be done in $O(1)$ time each. Even though the function $\alpha(f, u)$ is a very slowly growing function, the complexity of our solution to the set representation problem is not linear in the number of unions and finds. The space requirements are one node for each element.

In the exercises, we explore alternatives to the weight rule and the collapsing rule that preserve the time bounds of Lemma 2.4.

EXERCISES

1. Suppose we start with n sets, each containing a distinct element.
 - (a) Show that if u unions are performed, then no set contains more than $u + 1$ elements.
 - (b) Show that at most $n - 1$ unions can be performed before the number of sets becomes 1.
 - (c) Show that if fewer than $\lceil \frac{n}{2} \rceil$ unions are performed, then at least one set with a single element in it remains.
 - (d) Show that if u unions are performed, then at least $\max\{n - 2u, 0\}$ singleton sets remain.
2. Experimentally compare the performance of `SimpleUnion` and `SimpleFind` (Algorithm 2.13) with `WeightedUnion` (Algorithm 2.14) and `CollapsingFind` (Algorithm 2.15). For this, generate a random sequence of union and find operations.
3. (a) Present an algorithm `HeightUnion` that uses the *height rule* for union operations instead of the weighting rule. This rule is defined below:

Definition 2.7 [Height rule] If the height of tree i is less than that of tree j , then make j the parent of i ; otherwise make i the parent of j . \square

Your algorithm must run in $O(1)$ time and should maintain the height of each tree as a negative number in the p field of the root.

- (b) Show that the height bound of Lemma 2.3 applies to trees constructed using the height rule.
 - (c) Give an example of a sequence of unions that start with n singleton sets and create trees whose heights equal the upper bounds given in Lemma 2.3. Assume that each union is performed using the height rule.
 - (d) Experiment with the algorithms `WeightedUnion` (Algorithm 2.14) and `HeightUnion` to determine which produces better results when used in conjunction with `CollapsingFind` (Algorithm 2.15).
4. (a) Write an algorithm `SplittingFind` that uses *path splitting*, defined below, for the find operations instead of path collapsing.

Definition 2.8 [Path splitting] The parent pointer in each node (except the root and its child) on the path from i to the root is changed to point to the node's grandparent. \square

Note that when path splitting is used, a single pass from i to the root suffices. R. Tarjan and J. Van Leeuwen have shown that Lemma 2.4 holds when path splitting is used in conjunction with either the weight or the height rule for unions.

- (b) Experiment with `CollapsingFind` (Algorithm 2.15) and `SplittingFind` to determine which produces better results when used in conjunction with `WeightedUnion` (Algorithm 2.14).
5. (a) Design an algorithm `HalvingFind` that uses *path halving*, defined below, for the find operations instead of path collapsing.

Definition 2.9 [Path halving] In path halving, the parent pointer of every other node (except the root and its child) on the path from i to the root is changed to point to the node's grandparent. \square

Note that path halving, like path splitting (Exercise 4), can be implemented with a single pass from i to the root. However, in path halving, only half as many pointers are changed as in path splitting. Tarjan and Van Leeuwen have shown that Lemma 2.4 holds when path halving is used in conjunction with either the weight or the height rule for unions.

- (b) Experiment with `CollapsingFind` and `HalvingFind` to determine which produces better results when used in conjunction with `WeightedUnion` (Algorithm 2.14).

2.6 GRAPHS

2.6.1 Introduction

The first recorded evidence of the use of graphs dates back to 1736, when Leonhard Euler used them to solve the now classical Königsberg bridge problem. In the town of Königsberg (now Kaliningrad) the river Pregel (Pre-golya) flows around the island Kneiphof and then divides into two. There are, therefore, four land areas that have this river on their borders (see Figure 2.24(a)). These land areas are interconnected by seven bridges, labeled a to g . The land areas themselves are labeled A to D . The Königsberg bridge problem is to determine whether, starting at one land area, it is possible to walk across all the bridges exactly once in returning to the starting land area. One possible walk: Starting from land area B , walk across bridge a to island A , take bridge e to area D , take bridge g to C , take bridge d to A , take bridge b to B , and take bridge f to D .

This walk does not go across all bridges exactly once, nor does it return to the starting land area B . Euler answered the Königsberg bridge problem in the negative: The people of Königsberg cannot walk across each bridge exactly once and return to the starting point. He solved the problem by representing the land areas as vertices and the bridges as edges in a graph (actually a multigraph) as in Figure 2.24(b). His solution is elegant and applies to all graphs. Defining the *degree* of a vertex to be the number of edges incident to it, Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex if and only if the degree of each vertex is even. A walk that does this is called *Eulerian*. There is no Eulerian walk for the Königsberg bridge problem, as all four vertices are of odd degree.

Since this first application, graphs have been used in a wide variety of applications. Some of these applications are the analysis of electric circuits, finding shortest routes, project planning, identification of chemical compounds, statistical mechanics, genetics, cybernetics, linguistics, social sciences, and so on. Indeed, it might well be said that of all mathematical structures, graphs are the most widely used.

2.6.2 Definitions

A graph G consists of two sets V and E . The set V is a finite, nonempty set of *vertices*. The set E is a set of pairs of vertices; these pairs are called *edges*. The notations $V(G)$ and $E(G)$ represent the sets of vertices and edges, respectively, of graph G . We also write $G = (V, E)$ to represent a graph. In an *undirected graph* the pair of vertices representing any edge is unordered. Thus, the pairs (u, v) and (v, u) represent the same edge. In a *directed graph* each edge is represented by a directed pair $\langle u, v \rangle$; u is the *tail* and v the

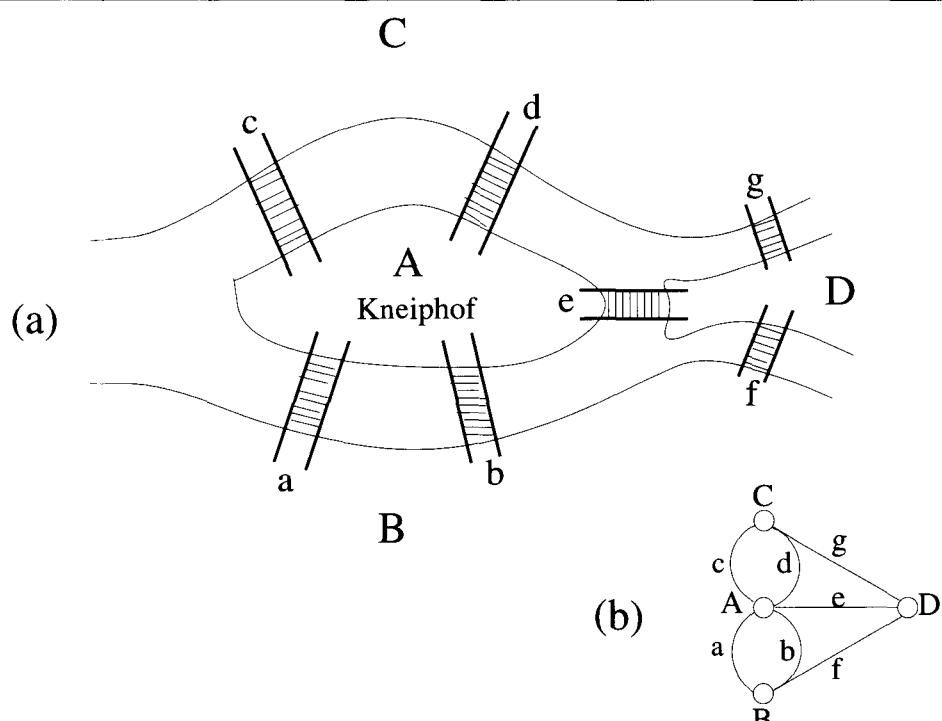


Figure 2.24 Section of the river Pregel in Königsberg and Euler's graph

head of the edge. Therefore, $\langle v, u \rangle$ and $\langle u, v \rangle$ represent two different edges. Figure 2.25 shows three graphs: G_1 , G_2 , and G_3 . The graphs G_1 and G_2 are undirected; G_3 is directed.

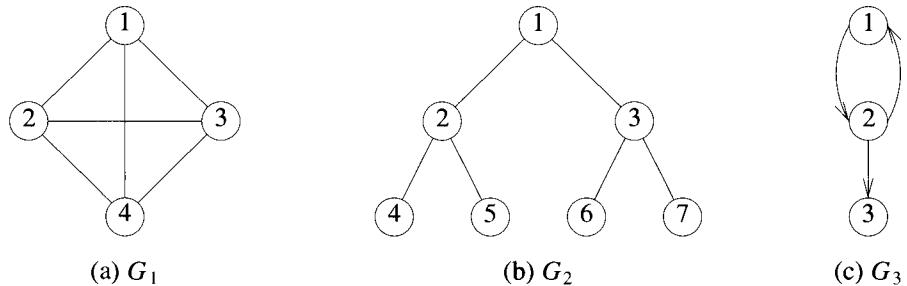


Figure 2.25 Three sample graphs

The set representations of these graphs are

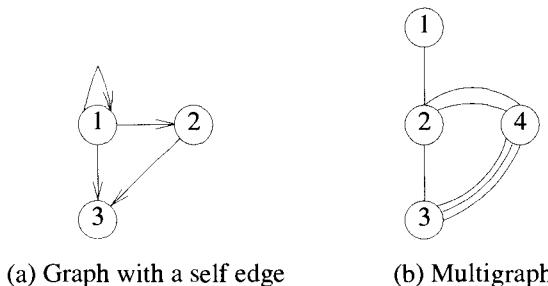
$$\begin{array}{ll} V(G_1) = \{1, 2, 3, 4\} & E(G_1) = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\} \\ V(G_2) = \{1, 2, 3, 4, 5, 6, 7\} & E(G_2) = \{(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7)\} \\ V(G_3) = \{1, 2, 3\} & E(G_3) = \{\langle 1, 2 \rangle, \langle 2, 1 \rangle, \langle 2, 3 \rangle\} \end{array}$$

Notice that the edges of a directed graph are drawn with an arrow from the tail to the head. The graph G_2 is a tree; the graphs G_1 and G_3 are not.

Since we define the edges and vertices of a graph as sets, we impose the following restrictions on graphs:

1. A graph may not have an edge from a vertex v back to itself. That is, edges of the form $\langle v, v \rangle$ and $\langle v, v \rangle$ are not legal. Such edges are known as *self-edges* or *self-loops*. If we permit self-edges, we obtain a data object referred to as a *graph with self-edges*. An example is shown in Figure 2.26(a).
2. A graph may not have multiple occurrences of the same edge. If we remove this restriction, we obtain a data object referred to as a *multi-graph* (see Figure 2.26(b)).

The number of distinct unordered pairs (u, v) with $u \neq v$ in a graph with n vertices is $\frac{n(n-1)}{2}$. This is the maximum number of edges in any n -vertex, undirected graph. An n -vertex, undirected graph with exactly $\frac{n(n-1)}{2}$ edges is said to be *complete*. The graph G_1 of Figure 2.25(a) is the complete graph

**Figure 2.26** Examples of graphlike structures

on four vertices, whereas G_2 and G_3 are not complete graphs. In the case of a directed graph on n vertices, the maximum number of edges is $n(n - 1)$.

If $\langle u, v \rangle$ is an edge in $E(G)$, then we say vertices u and v are *adjacent* and edge $\langle u, v \rangle$ is *incident* on vertices u and v . The vertices adjacent to vertex 2 in G_2 are 4, 5, and 1. The edges incident on vertex 3 in G_2 are $\langle 1, 3 \rangle$, $\langle 3, 6 \rangle$, and $\langle 3, 7 \rangle$. If $\langle u, v \rangle$ is a directed edge, then vertex u is *adjacent to* v , and v is *adjacent from* u . The edge $\langle u, v \rangle$ is incident to u and v . In G_3 , the edges incident to vertex 2 are $\langle 1, 2 \rangle$, $\langle 2, 1 \rangle$, and $\langle 2, 3 \rangle$.

A *subgraph* of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. Figure 2.27 shows some of the subgraphs of G_1 and G_3 .

A *path* from vertex u to vertex v in graph G is a sequence of vertices $u, i_1, i_2, \dots, i_k, v$, such that $\langle u, i_1 \rangle, \langle i_1, i_2 \rangle, \dots, \langle i_k, v \rangle$ are edges in $E(G)$. If G' is directed, then the path consists of the edges $\langle u, i_1 \rangle, \langle i_1, i_2 \rangle, \dots, \langle i_k, v \rangle$ in $E(G')$. The *length* of a path is the number of edges on it. A *simple path* is a path in which all vertices except possibly the first and last are distinct. A path such as $(1, 2), (2, 4), (4, 3)$, is also written as 1, 2, 4, 3. Paths 1, 2, 4, 3 and 1, 2, 4, 2 of G_1 are both of length 3. The first is a simple path; the second is not. The path 1, 2, 3 is a simple directed path in G_3 , but 1, 2, 3, 2 is not a path in G_3 , as the edge $\langle 3, 2 \rangle$ is not in $E(G_3)$.

A *cycle* is a simple path in which the first and last vertices are the same. The path 1, 2, 3, 1 is a cycle in G_1 and 1, 2, 1 is a cycle in G_3 . For directed graphs we normally add the prefix “directed” to the terms cycle and path.

In an undirected graph G , two vertices u and v are said to be *connected* iff there is a path in G from u to v (since G is undirected, this means there must also be a path from v to u). An undirected graph is said to be *connected* iff for every pair of distinct vertices u and v in $V(G)$, there is a path from u to v in G . Graphs G_1 and G_2 are connected, whereas G_4 of Figure 2.28 is not.

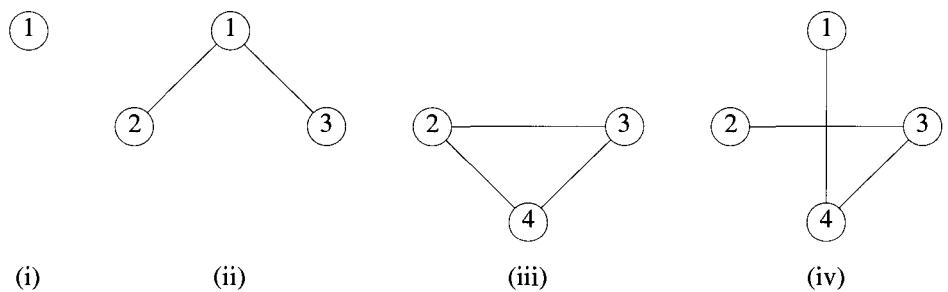
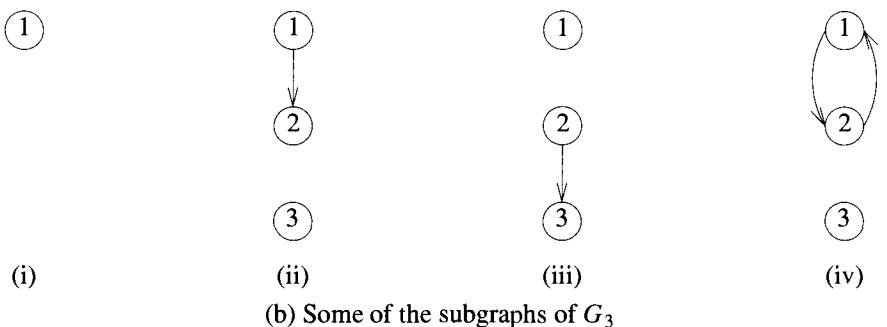
(a) Some of the subgraphs of G_1 (b) Some of the subgraphs of G_3

Figure 2.27 Some subgraphs

A *connected component* (or simply a component) H of an undirected graph is a maximal connected subgraph. By “maximal,” we mean that G contains no other subgraph that is both connected and properly contains H . G_4 has two components, H_1 and H_2 (see Figure 2.28).

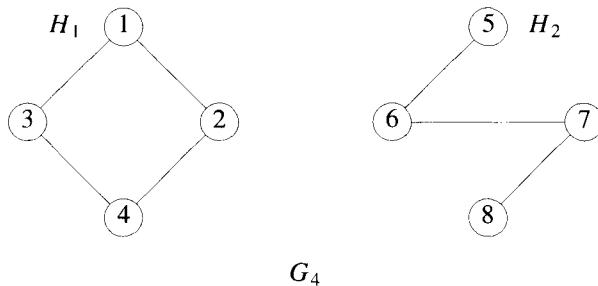


Figure 2.28 A graph with two connected components

A *tree* is a connected acyclic (i.e., has no cycles) graph. A directed graph G is said to be *strongly connected* iff for every pair of distinct vertices u and v in $V(G)$, there is a directed path from u to v and also from v to u . The graph G_3 (repeated in Figure 2.29(a)) is not strongly connected, as there is no path from vertex 3 to 2. A *strongly connected component* is a maximal subgraph that is strongly connected. The graph G_3 has two strongly connected components (see Figure 2.29(b)).

The degree of a vertex is the number of edges incident to that vertex. The degree of vertex 1 in G_1 is 3. If G is a directed graph, we define the *in-degree* of a vertex v to be the number of edges for which v is the head. The *out-degree* is defined to be the number of edges for which v is the tail. Vertex 2 of G_3 has in-degree 1, out-degree 2, and degree 3. If d_i is the degree of vertex i in a graph G with n vertices and e edges, then the number of edges is

$$e = \left(\sum_{i=1}^n d_i \right) / 2$$

In the remainder of this chapter, we refer to a directed graph as a *digraph*. When we use the term graph, we assume that it is an undirected graph.

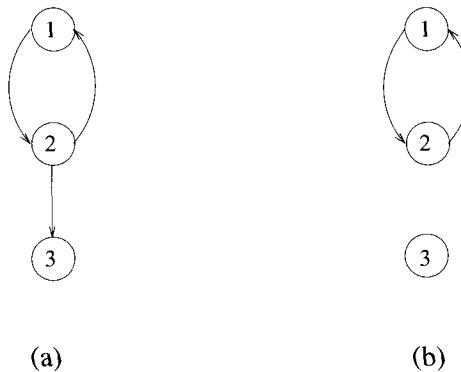


Figure 2.29 A graph and its strongly connected components

2.6.3 Graph Representations

Although several representations for graphs are possible, we study only the three most commonly used: adjacency matrices, adjacency lists, and adjacency multilists. Once again, the choice of a particular representation depends on the application we have in mind and the functions we expect to perform on the graph.

Adjacency Matrix

Let $G = (V, E)$ be a graph with n vertices, $n \geq 1$. The adjacency matrix of G is a two-dimensional $n \times n$ array, say a , with the property that $a[i, j] = 1$ iff the edge (i, j) ($\langle i, j \rangle$ for a directed graph) is in $E(G)$. The element $a[i, j] = 0$ if there is no such edge in G . The adjacency matrices for the graphs G_1 , G_3 , and G_4 are shown in Figure 2.30. The adjacency matrix for an undirected graph is symmetric, as the edge (i, j) is in $E(G)$ iff the edge (j, i) is also in $E(G)$. The adjacency matrix for a directed graph may not be symmetric (as is the case for G_3). The space needed to represent a graph using its adjacency matrix is n^2 bits. About half this space can be saved in the case of an undirected graph by storing only the upper or lower triangle of the matrix.

From the adjacency matrix, we can readily determine whether there is an edge connecting any two vertices i and j . For an undirected graph the degree of any vertex i is its row sum:

$1 \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	$1 \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$	$1 \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$
(a) G_1	(b) G_3	(c) G_4

Figure 2.30 Adjacency matrices

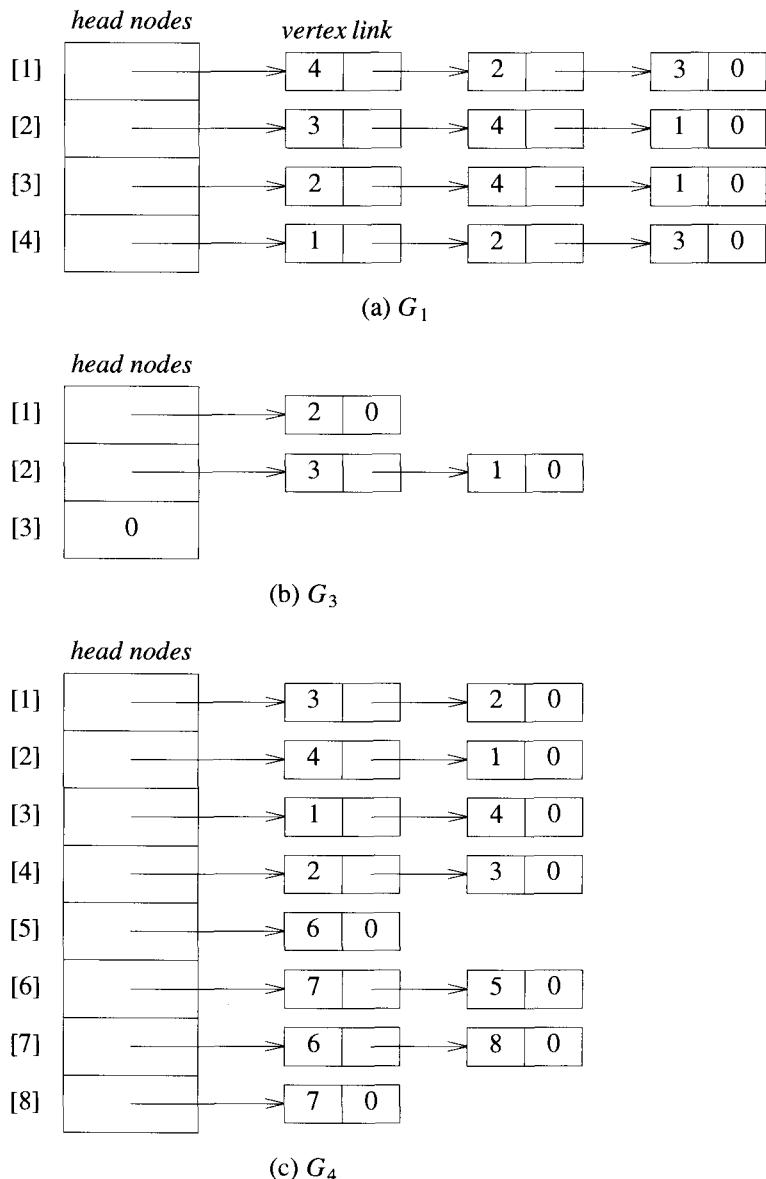
$$\sum_{j=1}^n a[i, j]$$

For a directed graph the row sum is the out-degree, and the column sum is the in-degree.

Suppose we want to answer a nontrivial question about graphs, such as How many edges are there in G ? or Is G connected? Adjacency matrices require at least n^2 time, as $n^2 - n$ entries of the matrix (diagonal entries are zero) have to be examined. When graphs are sparse (i.e., most of the terms in the adjacency matrix are zero), we would expect that the former question could be answered in significantly less time, say $O(e + n)$, where e is the number of edges in G , and $e < \frac{n^2}{2}$. Such a speedup is made possible through the use of a representation in which only the edges that are in G are explicitly stored. This leads to the next representation for graphs, adjacency lists.

Adjacency Lists

In this representation of graphs, the n rows of the adjacency matrix are represented as n linked lists. There is one list for each vertex in G . The nodes in list i represent the vertices that are adjacent from vertex i . Each node has at least two fields: *vertex* and *link*. The *vertex* field contains the indices of the vertices adjacent to vertex i . The adjacency lists for G_1 , G_3 ,

**Figure 2.31** Adjacency lists

and G_4 are shown in Figure 2.31. Notice that the vertices in each list are not required to be ordered. Each list has a head node. The head nodes are sequential, and so provide easy random access to the adjacency list for any particular vertex.

For an undirected graph with n vertices and e edges, this representation requires n head nodes and $2e$ list nodes. Each list node has two fields. In terms of the number of bits of storage needed, this count should be multiplied by $\log n$ for the head nodes and $\log n + \log e$ for the list nodes, as it takes $O(\log m)$ bits to represent a number of value m . Often, you can sequentially pack the nodes on the adjacency lists, and thereby eliminate the use of pointers. In this case, an array $node[1 : n + 2e + 1]$ can be used. The $node[i]$ gives the starting point of the list for vertex i , $1 \leq i \leq n$, and $node[n + 1]$ is set to $n + 2e + 2$. The vertices adjacent from vertex i are stored in $node[i], \dots, node[i + 1] - 1$, $1 \leq i \leq n$. Figure 2.32 shows the sequential representation for the graph G_4 of Figure 2.28.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
10	12	14	16	18	19	21	23	24	3	2	4	1	1	4	2	3	6	7	5	6	8	7

Figure 2.32 Sequential representation of graph G_4 :
Array $node[1 : n + 2e + 1]$

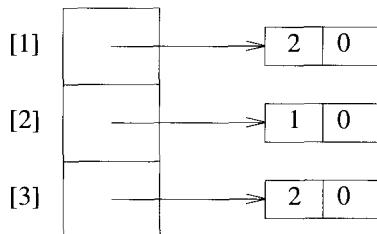
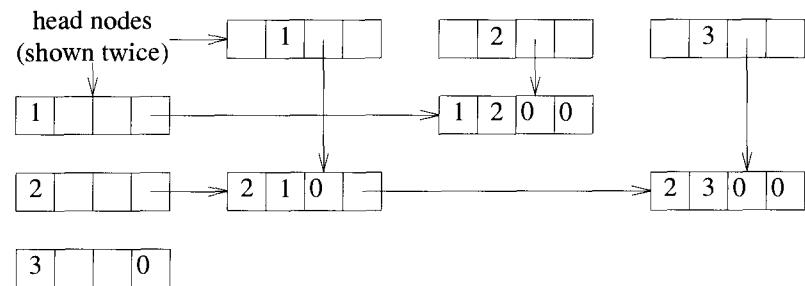
The degree of any vertex in an undirected graph can be determined by just counting the number of nodes in its adjacency list. So, the number of edges in G can be determined in $O(n + e)$ time.

For a digraph, the number of list nodes is only e . The out-degree of any vertex can be determined by counting the number of nodes on its adjacency list. Hence, the total number of edges in G can be determined in $O(n + e)$ time. Determining the in-degree of a vertex is a little more complex. If there is a need to access repeatedly all vertices adjacent to another vertex, then it may be worth the effort to keep another set of lists in addition to the adjacency lists. This set of lists, called *inverse adjacency lists*, contains one list for each vertex. Each list contains a node for each vertex adjacent to the vertex it represents (see Figure 2.33).

One can also adopt a simpler version of the list structure in which each node has four fields and represents one edge. The node structure is

tail	head	column link for head	row link for tail
------	------	----------------------	-------------------

Figure 2.34 shows the resulting structure for the graph G_3 of Figure 2.25(c). The head nodes are stored sequentially.

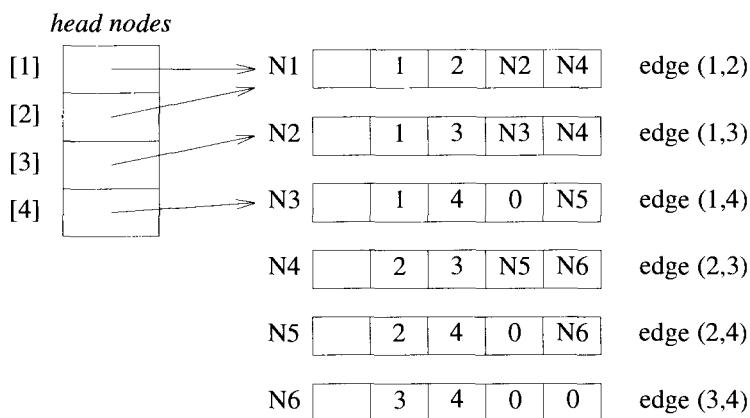
**Figure 2.33** Inverse adjacency lists for G_3 of Figure 2.25(c)**Figure 2.34** Orthogonal list representation for G_3 of Figure 2.25(c)

Adjacency Multilists

In the adjacency-list representation of an undirected graph, each edge (u, v) is represented by two entries, one on the list for u and the other on the list for v . In some applications it is necessary to be able to determine the second entry for a particular edge and mark that edge as having been examined. This can be accomplished easily if the adjacency lists are maintained as multilists (i.e., lists in which nodes can be shared among several lists). For each edge there is exactly one node, but this node is in two lists (i.e., the adjacency lists for each of the two nodes to which it is incident). The new node structure is

m	$vertex1$	$vertex2$	$list1$	$list2$
-----	-----------	-----------	---------	---------

where m is a one-bit mark field that can be used to indicate whether the edge has been examined. The storage requirements are the same as for normal adjacency lists, except for the addition of the mark bit m . Figure 2.35 shows the adjacency multilists for G_1 of Figure 2.25(a).



The lists are vertex 1: N1 → N2 → N3
 vertex 2: N1 → N4 → N5
 vertex 3: N2 → N4 → N6
 vertex 4: N3 → N5 → N6

Figure 2.35 Adjacency multilists for G_1 of Figure 2.25(a)

Weighted Edges

In many applications, the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex. In these applications, the adjacency matrix entries $a[i, j]$ keep this information too. When adjacency lists are used, the weight information can be kept in the list nodes by including an additional field, *weight*. A graph with weighted edges is called a *network*.

EXERCISES

1. Does the multigraph of Figure 2.36 have an Eulerian walk? If so, find one.

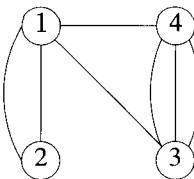
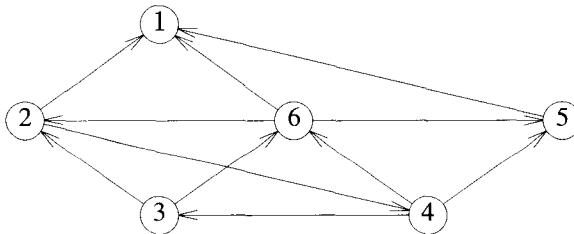
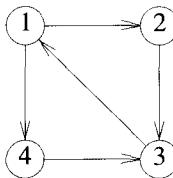


Figure 2.36 A multigraph

2. For the digraph of Figure 2.37 obtain
 - the in-degree and out-degree of each vertex
 - its adjacency-matrix representation
 - its adjacency-list representation
 - its adjacency-multilist representation
 - its strongly connected components
3. Devise a suitable representation for graphs so that they can be stored on disk. Write an algorithm that reads in such a graph and creates its adjacency matrix. Write another algorithm that creates the adjacency lists from the disk input.
4. Draw the complete undirected graphs on one, two, three, four, and five vertices. Prove that the number of edges in an n -vertex complete graph is $\frac{n(n-1)}{2}$.

**Figure 2.37** A digraph

5. Is the directed graph of Figure 2.38 strongly connected? List all the simple paths.

**Figure 2.38** A directed graph

6. Obtain the adjacency-matrix, adjacency-list, and adjacency-multilist representations of the graph of Figure 2.38.
7. Show that the sum of the degrees of the vertices of an undirected graph is twice the number of edges.
8. Prove or disprove:

If $G(V, E)$ is a finite directed graph such that the out-degree of each vertex is at least one, then there is a directed cycle in G .

9. (a) Let G be a connected, undirected graph on n vertices. Show that G must have at least $n - 1$ edges and that all connected, undirected graphs with $n - 1$ edges are trees.

- (b) What is the minimum number of edges in a strongly connected digraph with n vertices? What form do such digraphs have?
10. For an undirected graph G with n vertices, prove that the following are equivalent:
- G is a tree.
 - G is connected, but if any edge is removed, the resulting graph is not connected.
 - For any two distinct vertices $u \in V(G)$ and $v \in V(G)$, there is exactly one simple path from u to v .
 - G contains no cycles and has $n - 1$ edges.
11. Write an algorithm to input the number of vertices in an undirected graph and its edges one by one and to set up the linked adjacency-list representation of the graph. You may assume that no edge is input twice. What is the run time of your procedure as a function of the number of vertices and the number of edges?
12. Do the preceding exercise but now set up the multilist representation.
13. Let G be an undirected, connected graph with at least one vertex of odd degree. Show that G contains no Eulerian walk.

2.7 REFERENCES AND READINGS

A wide-ranging examination of data structures and their efficient implementation can be found in the following:

Fundamentals of Data Structures in C++, by E. Horowitz, S. Sahni, and D. Mehta, Computer Science Press, 1995.

Data Structures and Algorithms 1: Sorting and Searching, by K. Mehlhorn, Springer-Verlag, 1984.

Introduction to Algorithms: A Creative Approach, by U. Manber, Addison-Wesley, 1989.

Handbook of Algorithms and Data Structures, second edition, by G. H. Gonnet and R. Baeza-Yates, Addison-Wesley, 1991.

Proof of Lemma 2.4 can be found in “Worst-case analysis of set union algorithms,” by R. Tarjan and J. Van Leeuwen, *Journal of the ACM* 31; no. 2 (1984): 245–281.

Chapter 6

BASIC TRAVERSAL AND SEARCH TECHNIQUES

The techniques to be discussed in this chapter are divided into two categories. The first category includes techniques applicable only to binary trees. As described, these techniques involve examining every node in the given data object instance. Hence, these techniques are referred to as *traversal methods*. The second category includes techniques applicable to graphs (and hence also to trees and binary trees). These may not examine all vertices and so are referred to only as *search methods*. During a traversal or search the fields of a node may be used several times. It may be necessary to distinguish certain uses of the fields of a node. During these uses, the node is said to be *visited*. Visiting a node may involve printing out its data field, evaluating the operation specified by the node in the case of a binary tree representing an expression, setting a mark bit to one or zero, and so on. Since we are describing traversals and searches of trees and graphs independently of their application, we use the term “visited” rather than the term for the specific function performed on the node at this time.

6.1 TECHNIQUES FOR BINARY TREES

The solution to many problems involves the manipulation of binary trees, trees, or graphs. Often this manipulation requires us to determine a vertex (node) or a subset of vertices in the given data object that satisfies a given property. For example, we may wish to find all vertices in a binary tree with a data value less than x or we may wish to find all vertices in a given graph G that can be reached from another given vertex v . The determination of this subset of vertices satisfying a given property can be carried out by systematically examining the vertices of the given data object. This often takes the form of a search in the data object. When the search necessarily

```
treenode = record
{
    Type data; // Type is the data type of data.
    treenode *lchild; treenode *rchild;
}

1 Algorithm InOrder(t)
2 // t is a binary tree. Each node of t has
3 // three fields: lchild, data, and rchild.
4 {
5     if t ≠ 0 then
6     {
7         InOrder(t → lchild);
8         Visit(t);
9         InOrder(t → rchild);
10    }
11 }
```

Algorithm 6.1 Recursive formulation of inorder traversal

involves the examination of every vertex in the object being searched, it is called a *traversal*.

We have already seen an example of a problem whose solution required a search of a binary tree. In Section 5.5 we presented an algorithm to search a binary search tree for an identifier x . This algorithm is not a traversal algorithm as it does not examine every vertex in the search tree. Sometimes, we may wish to traverse a binary search tree (e.g., when we wish to list out all the identifiers in the tree). Algorithms for this are studied in this chapter.

There are many operations that we want to perform on binary trees. One that arises frequently is traversing a tree, or visiting each node in the tree exactly once. A traversal produces a linear order for the information in a tree. This linear order may be familiar and useful. When traversing a binary tree, we want to treat each node and its subtrees in the same fashion. If we let L , D , and R stand for moving left, printing the data, and moving right when at a node, then there are six possible combinations of traversal: LDR , LRD , DLR , DRL , RDL , and RLD . If we adopt the convention that we traverse left before right, then only three traversals remain: LDR , LRD , and DLR . To these we assign the names *inorder*, *postorder*, and *preorder*. Recursive functions for these three traversals are given in Algorithms 6.1 and 6.2.

```

1  Algorithm PreOrder( $t$ )
2  //  $t$  is a binary tree. Each node of  $t$  has
3  // three fields:  $lchild$ ,  $data$ , and  $rchild$ .
4  {
5      if  $t \neq 0$  then
6      {
7          Visit( $t$ );
8          PreOrder( $t \rightarrow lchild$ );
9          PreOrder( $t \rightarrow rchild$ );
10     }
11 }

1  Algorithm PostOrder( $t$ )
2  //  $t$  is a binary tree. Each node of  $t$  has
3  // three fields:  $lchild$ ,  $data$ , and  $rchild$ .
4  {
5      if  $t \neq 0$  then
6      {
7          PostOrder( $t \rightarrow lchild$ );
8          PostOrder( $t \rightarrow rchild$ );
9          Visit( $t$ );
10     }
11 }

```

Algorithm 6.2 Preorder and postorder traversals

Figure 6.1 shows a binary tree and Figure 6.2 traces how `InOrder` works on it. This trace assumes that visiting a node requires only the printing of its `data` field. The output resulting from this traversal is FDHGIBEAC. With `Visit(t)` replaced by a printing statement, the application of Algorithm 6.2 to the binary tree of Figure 6.1 results in the outputs ABDFGHIEC and FHIGDEBCA, respectively.

Theorem 6.1 Let $T(n)$ and $S(n)$ respectively represent the time and space needed by any one of the traversal algorithms when the input tree t has $n \geq 0$ nodes. If the time and space needed to visit a node are $\Theta(1)$, then $T(n) = \Theta(n)$ and $S(n) = O(n)$.

Proof: Each traversal can be regarded as a walk through the binary tree. During this walk, each node is reached three times: once from its parent (or as the start node in case the node is the root), once on returning from its left

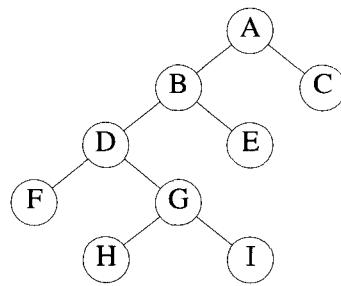


Figure 6.1 A binary tree

call of InOrder	value in root	action
main	A	
1	B	
2	D	
3	F	
4	—	print ('F')
4	—	print ('D')
3	G	
4	H	
5	—	print ('H')
5	—	print ('G')
4	I	
5	—	print ('I')
5	—	print ('B')
2	E	
3	—	print ('E')
3	—	print ('A')
1	C	
2	—	print ('C')
2	—	

Figure 6.2 Inorder traversal of the binary tree of Figure 6.1

subtree, and once on returning from its right subtree. In each of these three times a constant amount of work is done. So, the total time taken by the traversal is $\Theta(n)$. The only additional space needed is that for the recursion stack. If t has depth d , then this space is $\Theta(d)$. For an n -node binary tree, $d \leq n$ and so $S(n) = O(n)$. \square

EXERCISES

Unless otherwise stated, all binary trees are represented using nodes with three fields: *lchild*, *data*, and *rchild*.

1. Give an algorithm to count the number of leaf nodes in a binary tree t . What is its computing time?
 2. Write an algorithm `SwapTree(t)` that takes a binary tree and swaps the left and right children of every node. An example is given in Figure 6.3. Use one of the three traversal methods discussed in Section 6.1.
-

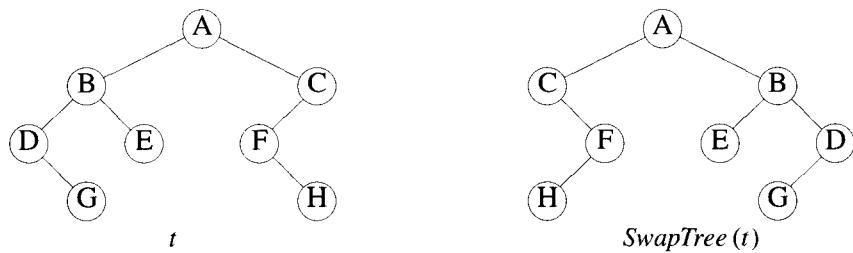


Figure 6.3 Swapping left and right children

3. Use one of the three traversal methods discussed in Section 6.1 to obtain an algorithm `Equiv(t, u)` that determines whether the binary trees t and u are equivalent. Two binary trees t and u are equivalent if and only if they are structurally equivalent and if the data in the corresponding nodes of t and u are the same.
4. Show the following:
 - (a) Inorder and postorder sequences of a binary tree uniquely define the binary tree.
 - (b) Inorder and preorder sequences of a binary tree uniquely define the binary tree.

- (c) Preorder and postorder sequences of a binary tree do not uniquely define the binary tree.
5. In the proof of Theorem 6.1, show, using induction, that $T(n) \leq c_2n + c_1$ (where c_2 is a constant $\geq 2c_1$).
 6. Write a function to construct the binary tree with a given inorder sequence I and a given postorder sequence P . What is the complexity of your function?
 7. Do Exercise 6 for a given inorder and preorder sequence.
 8. Write a nonrecursive algorithm for the preorder traversal of a binary tree t . Your algorithm may use a stack. What are the time and space requirements of your algorithm?
 9. Do Exercise 8 for postorder as well as inorder traversals.
 10. [Triple-order traversal] A triple-order traversal of a binary tree t is defined recursively by Algorithm 6.3. A very simple nonrecursive algorithm for such a traversal is given in Algorithm 6.4. In this algorithm p , q , and r point respectively to the present node, previously visited node, and next node to visit. The algorithm assumes that $t \neq 0$ and that an empty subtree of node p is represented by a link to p rather than a zero. Prove that Algorithm 6.4 is correct. (*Hint:* Three links, $lchild$, $rchild$, and one from its parent, are associated with each node s . Each time s is visited, the links are rotated counterclockwise, and so after three visits they are restored to the original configuration and the algorithm backs up the tree.)
 11. [Level-order traversal] In a level-order traversal of a binary tree t all nodes on level i are visited before any node on level $i + 1$ is visited. Within a level, nodes are visited left to right. In level-order the nodes of the tree of Figure 6.1 are visited in the order ABCDEFGHI. Write an algorithm $\text{Level}(t)$ to traverse the binary tree t in level order. How much time and space are needed by your algorithm?

6.2 TECHNIQUES FOR GRAPHS

A fundamental problem concerning graphs is the reachability problem. In its simplest form it requires us to determine whether there exists a path in the given graph $G = (V, E)$ such that this path starts at vertex v and ends at vertex u . A more general form is to determine for a given starting vertex $v \in V$ all vertices u such that there is a path from v to u . This latter problem can be solved by starting at vertex v and systematically searching the graph G for vertices that can be reached from v . We describe two search methods for this.

```
1  Algorithm Triple( $t$ )
2  {
3      if  $t \neq 0$  then
4      {
5          Visit( $t$ );
6          Triple( $t \rightarrow lchild$ );
7          Visit( $t$ );
8          Triple( $t \rightarrow rchild$ );
9          Visit( $t$ );
10     }
11 }
```

Algorithm 6.3 Triple-order traversal for Exercise 10

```
1  Algorithm Trip( $t$ );
2  // It is assumed that  $lchild$  and  $rchild$  fields are  $> 0$ .
3  {
4       $p := t$ ;  $q := -1$ ;
5      while ( $p \neq -1$ ) do
6      {
7          Visit( $p$ );
8           $r := (p \rightarrow lchild)$ ;  $(p \rightarrow lchild) := (p \rightarrow rchild)$ ;
9           $(p \rightarrow rchild) := q$ ;  $q := p$ ;  $p := r$ ;
10     }
11 }
```

Algorithm 6.4 A nonrecursive algorithm for the triple-order traversal for Exercise 10

6.2.1 Breadth First Search and Traversal

In breadth first search we start at a vertex v and mark it as having been reached (visited). The vertex v is at this time said to be unexplored. A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices adjacent from v are visited next. These are new unexplored vertices. Vertex v has now been explored. The newly visited vertices haven't been explored and are put onto the end of a list of unexplored vertices. The first vertex on this list is the next to be explored. Exploration continues until no unexplored vertex is left. The list of unexplored vertices operates as a queue and can be represented using any of the standard queue representations (see Section 2.1). BFS (Algorithm 6.5) describes, in pseudocode, the details of the search. It makes use of the queue representation given in Section 2.1 (Algorithm 2.3).

Example 6.1 Let us try out the algorithm on the undirected graph of Figure 6.4(a). If the graph is represented by its adjacency lists as in Figure 6.4(c), then the vertices get visited in the order 1, 2, 3, 4, 5, 6, 7, 8. A breadth first search of the directed graph of Figure 6.4(b) starting at vertex 1 results in only the vertices 1, 2, and 3 being visited. Vertex 4 cannot be reached from 1. \square

Theorem 6.2 Algorithm BFS visits all vertices reachable from v .

Proof: Let $G = (V, E)$ be a graph (directed or undirected) and let $v \in V$. We prove the theorem by induction on the length of the shortest path from v to every reachable vertex $w \in V$. The length (i.e., number of edges) of the shortest path from v to a reachable vertex w is denoted by $d(v, w)$.

Basis Step. Clearly, all vertices w with $d(v, w) \leq 1$ get visited.

Induction Hypothesis. Assume that all vertices w with $d(v, w) \leq r$ get visited.

Induction Step. We now show that all vertices w with $d(v, w) = r + 1$ also get visited.

Let w be a vertex in V such that $d(v, w) = r + 1$. Let u be a vertex that immediately precedes w on a shortest v to w path. Then $d(v, u) = r$ and so u gets visited by BFS. We can assume $u \neq v$ and $r \geq 1$. Hence, immediately before u gets visited, it is placed on the queue q of unexplored vertices. The algorithm doesn't terminate until q becomes empty. Hence, u is removed from q at some time and all unvisited vertices adjacent from it get visited in the **for** loop of line 11 of Algorithm 6.5. Hence, w gets visited. \square

```

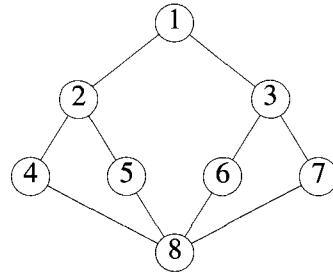
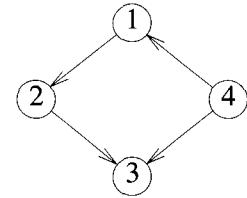
1  Algorithm BFS( $v$ )
2  // A breadth first search of  $G$  is carried out beginning
3  // at vertex  $v$ . For any node  $i$ ,  $visited[i] = 1$  if  $i$  has
4  // already been visited. The graph  $G$  and array  $visited[ ]$ 
5  // are global;  $visited[ ]$  is initialized to zero.
6  {
7       $u := v$ ; //  $q$  is a queue of unexplored vertices.
8       $visited[v] := 1$ ;
9      repeat
10     {
11         for all vertices  $w$  adjacent from  $u$  do
12             {
13                 if ( $visited[w] = 0$ ) then
14                     {
15                         Add  $w$  to  $q$ ; //  $w$  is unexplored.
16                          $visited[w] := 1$ ;
17                     }
18                 }
19                 if  $q$  is empty then return; // No unexplored vertex.
20                 Delete  $u$  from  $q$ ; // Get first unexplored vertex.
21             } until(false);
22     }

```

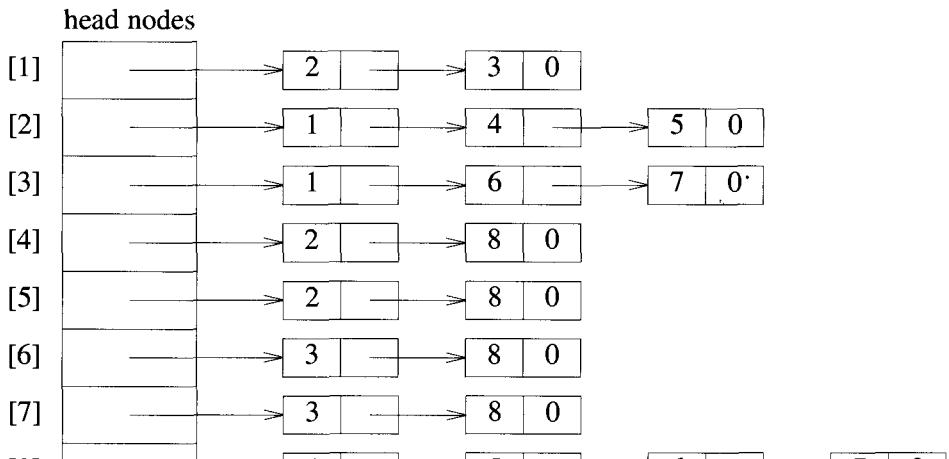
Algorithm 6.5 Pseudocode for breadth first search

Theorem 6.3 Let $T(n, e)$ and $S(n, e)$ be the maximum time and maximum additional space taken by algorithm BFS on any graph G with n vertices and e edges. $T(n, e) = \Theta(n + e)$ and $S(n, e) = \Theta(n)$ if G is represented by its adjacency lists. If G is represented by its adjacency matrix, then $T(n, e) = \Theta(n^2)$ and $S(n, e) = \Theta(n)$.

Proof: Vertices get added to the queue only in line 15 of Algorithm 6.5. A vertex w can get onto the queue only if $visited[w] = 0$. Immediately following w 's addition to the queue, $visited[w]$ is set to 1 (line 16). Hence, each vertex can get onto the queue at most once. Vertex v never gets onto the queue and so at most $n - 1$ additions are made. The queue space needed is at most $n - 1$. The remaining variables take $O(1)$ space. Hence, $S(n, e) = O(n)$. If G is an n -vertex graph with v connected to the remaining $n - 1$ vertices, then all $n - 1$ vertices adjacent from v are on the queue at the same time. Furthermore, $\Theta(n)$ space is needed for the array $visited$. Hence $S(n, e) = \Theta(n)$. This result is independent of whether adjacency matrices or lists are used.

(a) Undirected graph G 

(b) Directed graph

(c) Adjacency list for G **Figure 6.4** Example graphs and adjacency lists

```

1  Algorithm BFT( $G, n$ )
2  // Breadth first traversal of  $G$ 
3  {
4      for  $i := 1$  to  $n$  do // Mark all vertices unvisited.
5           $visited[i] := 0$ ;
6      for  $i := 1$  to  $n$  do
7          if ( $visited[i] = 0$ ) then BFS( $i$ );
8  }

```

Algorithm 6.6 Breadth first graph traversal

If adjacency lists are used, then all vertices adjacent from u can be determined in time $d(u)$, where $d(u)$ is the degree of u if G is undirected and $d(u)$ is the out-degree of u if G is directed. Hence, when vertex u is being explored, the time for the **for** loop of line 11 of Algorithm 6.5 is $\Theta(d(u))$. Since each vertex in G can be explored at most once, the total time for the **repeat** loop of line 9 is $O(\sum d(u)) = O(e)$. Then $visited[i]$ has to be initialized to 0, $1 \leq i \leq n$. This takes $O(n)$ time. The total time is therefore $O(n + e)$. If adjacency matrices are used, then it takes $\Theta(n)$ time to determine all vertices adjacent from u and the time becomes $O(n^2)$. If G is a graph such that all vertices are reachable from v , then all vertices get explored and the time is at least $O(n + e)$ and $O(n^2)$ respectively. Hence, $T(n, e) = \Theta(n + e)$ when adjacency lists are used, and $T(n, e) = \Theta(n^2)$ when adjacency matrices are used. \square

If BFS is used on a connected undirected graph G , then all vertices in G get visited and the graph is traversed. However, if G is not connected, then at least one vertex of G is not visited. A complete traversal of the graph can be made by repeatedly calling BFS each time with a new unvisited starting vertex. The resulting traversal algorithm is known as breadth first traversal (BFT) (see Algorithm 6.6). The proof of Theorem 6.3 can be used for BFT too to show that the time and additional space required by BFT on an n -vertex e -edge graph are $\Theta(n + e)$ and $\Theta(n)$ respectively if adjacency lists are used. If adjacency matrices are used, then the bounds are $\Theta(n^2)$ and $\Theta(n)$ respectively.

6.2.2 Depth First Search and Traversal

A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached. At

```
1  Algorithm DFS( $v$ )
2  // Given an undirected (directed) graph  $G = (V, E)$  with
3  //  $n$  vertices and an array  $visited[ ]$  initially set
4  // to zero, this algorithm visits all vertices
5  // reachable from  $v$ .  $G$  and  $visited[ ]$  are global.
6  {
7       $visited[v] := 1$ ;
8      for each vertex  $w$  adjacent from  $v$  do
9      {
10         if ( $visited[w] = 0$ ) then DFS( $w$ );
11     }
12 }
```

Algorithm 6.7 Depth first search of a graph

this time the exploration of the new vertex u begins. When this new vertex has been explored, the exploration of v continues. The search terminates when all reached vertices have been fully explored. This search process is best described recursively as in Algorithm 6.7.

Example 6.2 A depth first search of the graph of Figure 6.4(a) starting at vertex 1 and using the adjacency lists of Figure 6.4(c) results in the vertices being visited in the order 1, 2, 4, 8, 5, 6, 3, 7. \square

One can easily prove that DFS visits all vertices reachable from vertex v . If $T(n, e)$ and $S(n, e)$ represent the maximum time and maximum additional space taken by DFS for an n -vertex e -edge graph, then $S(n, e) = \Theta(n)$ and $T(n, e) = \Theta(n + e)$ if adjacency lists are used and $T(n, e) = \Theta(n^2)$ if adjacency matrices are used (see the exercises).

A depth first traversal of a graph is carried out by repeatedly calling DFS, with a new unvisited starting vertex each time. The algorithm for this (DFT) differs from BFT only in that the call to $BFS(i)$ is replaced by a call to $DFS(i)$. The exercises contain some problems that are solved best by BFS and others that are solved best by DFS. Later sections of this chapter also discuss graph problems solved best by DFS.

BFS and DFS are two fundamentally different search methods. In BFS a node is fully explored before the exploration of any other node begins. The next node to explore is the first unexplored node remaining. The exercises examine a search technique (D -search) that differs from BFS only in that

the next node to explore is the most recently reached unexplored node. In DFS the exploration of a node is suspended as soon as a new unexplored node is reached. The exploration of this new node is immediately begun.

EXERCISES

1. Devise an algorithm using the idea of BFS to find a shortest (directed) cycle containing a given vertex v . Prove that your algorithm finds a shortest cycle. What are the time and space requirements of your algorithm?
2. Show that DFS visits all vertices in G reachable from v .
3. Prove that the bounds of Theorem 6.3 hold for DFS.
4. It is easy to see that for any graph G , both DFS and BFS will take almost the same amount of time. However, the space requirements may be considerably different.
 - (a) Give an example of an n -vertex graph for which the depth of recursion of DFS starting from a particular vertex v is $n - 1$ whereas the queue of BFS has at most one vertex at any given time if BFS is started from the same vertex v .
 - (b) Give an example of an n -vertex graph for which the queue of BFS has $n - 1$ vertices at one time whereas the depth of recursion of DFS is at most one. Both searches are started from the same vertex.
5. Another way to search a graph is D -search. This method differs from BFS in that the next vertex to explore is the vertex most recently added to the list of unexplored vertices. Hence, this list operates as a stack rather than a queue.
 - (a) Write an algorithm for D -search.
 - (b) Show that D -search starting from vertex v visits all vertices reachable from v .
 - (c) What are the time and space requirements of your algorithm?

6.3 CONNECTED COMPONENTS AND SPANNING TREES

If G is a connected undirected graph, then all vertices of G will get visited on the first call to BFS (Algorithm 6.5). If G is not connected, then at

least two calls to BFS will be needed. Hence, BFS can be used to determine whether G is connected. Furthermore, all newly visited vertices on a call to BFS from BFT represent the vertices in a connected component of G . Hence the connected components of a graph can be obtained using BFT. For this, BFS can be modified so that all newly visited vertices are put onto a list. Then the subgraph formed by the vertices on this list make up a connected component. Hence, if adjacency lists are used, a breadth first traversal will obtain the connected components in $\Theta(n + e)$ time.

BFT can also be used to obtain the reflexive transitive closure matrix of an undirected graph G . If A^* is this matrix, then $A^*(i, j) = 1$ iff either $i = j$ or $i \neq j$ and i and j are in the same connected component. We can set up in $\Theta(n + e)$ time an array *connec* such that *connec*[i] is the index of the connected component containing vertex i , $1 \leq i \leq n$. Hence, we can determine whether $A^*(i, j)$, $i \neq j$, is 1 or 0 by simply seeing whether *connec*[i] = *connec*[j]. The reflexive transitive closure matrix of an undirected graph G with n vertices and e edges can therefore be computed in $\Theta(n^2)$ time and $\Theta(n)$ space using either adjacency lists or matrices (the space count does not include the space needed for A^* itself).

As a final application of breadth first search, consider the problem of obtaining a spanning tree for an undirected graph G . The graph G has a spanning tree iff G is connected. Hence, BFS easily determines the existence of a spanning tree. Furthermore, consider the set of edges (u, w) used in the **for** loop of line 11 of algorithm BFS to reach unvisited vertices w . These edges are called *forward edges*. Let t denote the set of these forward edges. We claim that if G is connected, then t is a spanning tree of G . For the graph of Figure 6.4(a) the set of edges t will be all edges in G except $(5, 8)$, $(6, 8)$, and $(7, 8)$ (see Figure 6.5(b)). Spanning trees obtained using a breadth first search are called *breadth first spanning trees*.

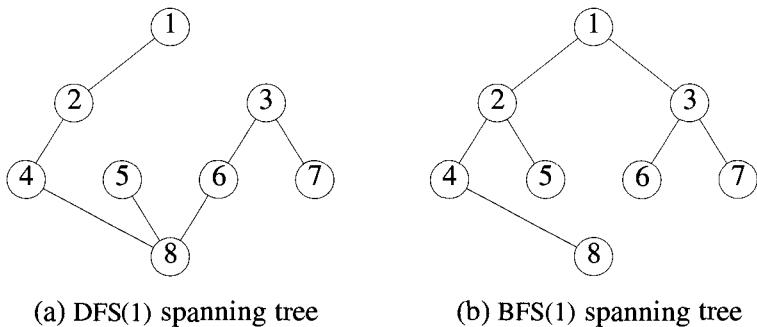


Figure 6.5 DFS and BFS spanning trees for the graph of Figure 6.4(a)

Theorem 6.4 Modify algorithm BFS by adding on the statements $t := \emptyset$; and $t := t \cup \{(u, w)\}$; to lines 8 and 16, respectively. Call the resulting algorithm BFS^* . If BFS^* is called so that v is any vertex in a connected undirected graph G , then on termination, the edges in t form a spanning tree of G .

Proof: We have already seen that if G is a connected graph on n vertices, then all n vertices will get visited. Also, each of these, except the start vertex v , will get onto the queue once (line 15). Hence, t will contain exactly $n - 1$ edges. All these edges are distinct. The $n - 1$ edges in t will therefore define an undirected graph on n vertices. This graph will be connected since it contains a path from the start vertex v to every other vertex (and so there is a path between each two vertices). A simple proof by induction shows that every connected graph on n vertices with exactly $n - 1$ edges is a tree. Hence t is a spanning tree of G . \square

As in the case of BFT, the connected components of a graph can be obtained using DFT. Similarly, the reflexive transitive closure matrix of an undirected graph can be found using DFT. If DFS (Algorithm 6.7) is modified by adding $t := \emptyset$; and $t := t \cup \{(v, w)\}$; to line 7 and the if statement of line 10, respectively, then when DFS terminates, the edges in t define a spanning tree for the undirected graph G if G is connected. A spanning tree obtained in this manner is called a *depth first spanning tree*. For the graph of Figure 6.4(a) the spanning tree obtained will include all edges in G except for $(2, 5)$, $(8, 7)$, and $(1, 3)$ (see Figure 6.5(a)). Hence, DFS and BFS are equally powerful for the search problems discussed so far.

EXERCISES

1. Show that for any undirected graph $G = (V, E)$, a call to $\text{BFS}(v)$ with $v \in V$ results in visiting all the vertices in the connected component containing v .
2. Rewrite BFS and BFT so that all the connected components of the undirected graph G get printed out. Assume that G is input in adjacency list form.
3. Prove that if G is a connected undirected graph with n vertices and $n - 1$ edges, then G is a tree.
4. Present a D -search-based algorithm that produces a spanning tree for an undirected connected graph.
5. (a) The *radius* of a tree is its depth. Show that the forward edges used in $\text{BFS}(v)$ define a spanning tree with root v having minimum radius among all spanning trees, for the undirected connected graph G having root v .

- (b) Using the result of part (a), write an algorithm to find a minimum-radius spanning tree for G . What are the time and space requirements of your algorithm?
6. The *diameter* of a tree is the maximum distance between any two vertices. Let d be the diameter of a minimum-diameter spanning tree for an undirected connected graph G . Let r be the radius of a minimum-radius spanning tree for G .
- Show that $2r - 1 \leq d \leq 2r$.
 - Write an algorithm to find a minimum-diameter spanning tree for G . (*Hint:* Use breadth-first search followed by some local modification.)
 - Prove that your algorithm is correct.
 - What are the time and space requirements of your algorithm?
7. A *bipartite graph* $G = (V, E)$ is an undirected graph whose vertices can be partitioned into two disjoint sets V_1 and $V_2 = V - V_1$ with the properties that no two vertices in V_1 are adjacent in G and no two vertices in V_2 are adjacent in G . The graph G of Figure 6.4(a) is bipartite. A possible partitioning of V is $V_1 = \{1, 4, 5, 6, 7\}$ and $V_2 = \{2, 3, 8\}$. Write an algorithm to determine whether a graph G is bipartite. If G is bipartite, your algorithm should obtain a partitioning of the vertices into two disjoint sets V_1 and V_2 satisfying the properties above. Show that if G is represented by its adjacency lists, then this algorithm can be made to work in time $O(n + e)$, where $n = |V|$ and $e = |E|$.
8. Write an algorithm to find the reflexive transitive closure matrix A^* of a directed graph G . Show that if G has n vertices and e edges and is represented by its adjacency lists, then this can be done in time $\Theta(n^2 + ne)$. How much space does your algorithm take in addition to that needed for G and A^* ? (*Hint:* Use either BFS or DFS.)
9. Input is an undirected connected graph $G(V, E)$ each one of whose edges has the same weight w (w being a real number). Give an $O(|E|)$ time algorithm to find a minimum-cost spanning tree for G . What is the weight of this tree?
10. Given are a directed graph $G(V, E)$ and a node $v \in V$. Write an efficient algorithm to decide whether there is a directed path from v to every other node in the graph. What is the worst-case run time of your algorithm?
11. Design an algorithm to decide whether a given undirected graph $G(V, E)$ contains a cycle of length 4. The running time of the algorithm should be $O(|V|^3)$.

12. Let $G(V, E)$ be a binary tree with n nodes. The *distance* between two vertices in G is the length of the path connecting these two vertices. The problem is to construct an $n \times n$ matrix whose ij th entry is the distance between v_i and v_j . Design an $O(n^2)$ time algorithm to construct such a matrix. Assume that the tree is given in the adjacency-list representation.
13. Present an $O(|V|)$ time algorithm to check whether a given undirected graph $G(V, E)$ is a tree. The graph G is given in the form of an adjacency list.

6.4 BICONNECTED COMPONENTS AND DFS

In this section, by “graph” we always mean an undirected graph. A vertex v in a connected graph G is an *articulation point* if and only if the deletion of vertex v together with all edges incident to v disconnects the graph into two or more nonempty components.

Example 6.3 In the connected graph of Figure 6.6(a) vertex 2 is an articulation point as the deletion of vertex 2 and edges $(1, 2), (2, 3), (2, 5), (2, 7)$, and $(2, 8)$ leaves behind two disconnected nonempty components (Figure 6.6(b)). Graph G of Figure 6.6(a) has only two other articulation points: vertex 5 and vertex 3. Note that if any of the remaining vertices is deleted from G , then exactly one component remains. \square

A graph G is *biconnected* if and only if it contains no articulation points. The graph of Figure 6.6(a) is not biconnected. The graph of Figure 6.7 is biconnected. The presence of articulation points in a connected graph can be an undesirable feature in many cases. For example, if G represents a communication network with the vertices representing communication stations and the edges communication lines, then the failure of a communication station i that is an articulation point would result in the loss of communication to points other than i too. On the other hand, if G has no articulation point, then if any station i fails, we can still communicate between every two stations not including station i .

In this section we develop an efficient algorithm to test whether a connected graph is biconnected. For the case of graphs that are not biconnected, this algorithm will identify all the articulation points. Once it has been determined that a connected graph G is not biconnected, it may be desirable to determine a set of edges whose inclusion makes the graph biconnected. Determining such a set of edges is facilitated if we know the maximal subgraphs of G that are biconnected. $G' = (V', E')$ is a maximal biconnected subgraph of G if and only if G has no biconnected subgraph $G'' = (V'', E'')$

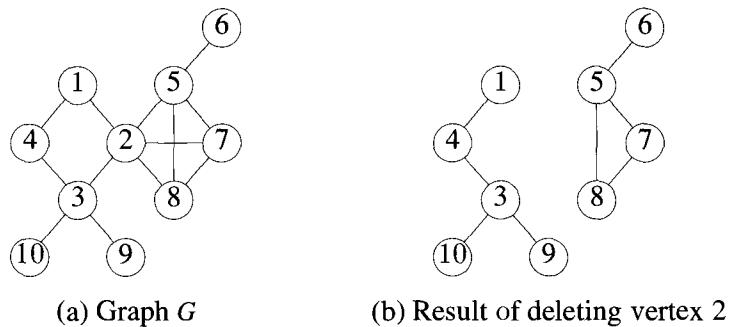


Figure 6.6 An example graph

such that $V' \subseteq V''$ and $E' \subset E''$. A maximal biconnected subgraph is a *biconnected component*.

The graph of Figure 6.7 has only one biconnected component (i.e., the entire graph). The biconnected components of the graph of Figure 6.6(a) are shown in Figure 6.8.

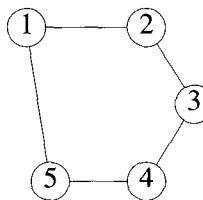


Figure 6.7 A biconnected graph

It is relatively easy to show that

Lemma 6.1 Two biconnected components can have at most one vertex in common and this vertex is an articulation point. \square

Hence, no edge can be in two different biconnected components (as this would require two common vertices). The graph G can be transformed into a biconnected graph by using the edge addition scheme of Algorithm 6.8.

Since every biconnected component of a connected graph G contains at least two vertices (unless G itself has only one vertex), it follows that the v_i of line 5 exists.

Example 6.4 Using the above scheme to transform the graph of Figure 6.6(a) into a biconnected graph requires us to add edges $(4, 10)$ and $(10, 9)$ (corresponding to the articulation point 3), edge $(1, 5)$ (corresponding to the articulation point 2), and edge $(6, 7)$ (corresponding to point 5). \square

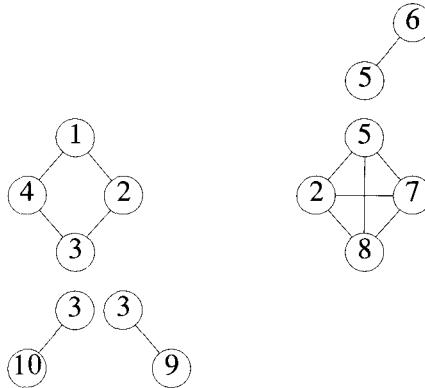


Figure 6.8 Biconnected components of graph of Figure 6.6(a)

Note that once the edges (v_i, v_{i+1}) of line 6 (Algorithm 6.8) are added, vertex a is no longer an articulation point. Hence following the addition

```

1  for each articulation point  $a$  do
2  {
3      Let  $B_1, B_2, \dots, B_k$  be the biconnected
4      components containing vertex  $a$ ;
5      Let  $v_i, v_i \neq a$ , be a vertex in  $B_i$ ,  $1 \leq i \leq k$ ;
6      Add to  $G$  the edges  $(v_i, v_{i+1})$ ,  $1 \leq i < k$ ;
7  }
```

Algorithm 6.8 Scheme to construct a biconnected graph

of the edges corresponding to all articulation points, G has no articulation points and so is biconnected. If G has p articulation points and b biconnected components, then the scheme of Algorithm 6.8 introduces exactly $b - p$ new edges into G . One can show that this scheme may use more than the minimum number of edges needed to make G biconnected (see the exercises).

Now, let us attack the problem of identifying the articulation points and biconnected components of a connected graph G with $n \geq 2$ vertices. The problem is efficiently solved by considering a depth first spanning tree of G .

Figure 6.9(a) and (b) shows a depth first spanning tree of the graph of Figure 6.6(a). In each figure there is a number outside each vertex. These numbers correspond to the order in which a depth first search visits these vertices and are referred to as the *depth first numbers (dfns)* of the vertex. Thus, $dfn[1] = 1$, $dfn[4] = 2$, $dfn[6] = 8$, and so on. In Figure 6.9(b) solid edges form the depth first spanning tree. These edges are called *tree edges*. Broken edges (i.e., all the remaining edges) are called *back edges*.

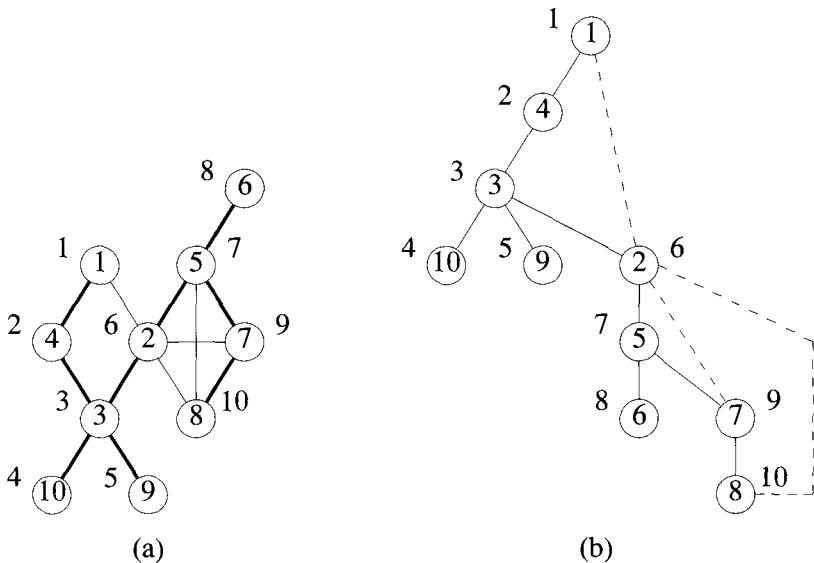


Figure 6.9 A depth first spanning tree of the graph of Figure 6.6(a)

Depth first spanning trees have a property that is very useful in identifying articulation points and biconnected components

Lemma 6.2 If (u, v) is any edge in G , then relative to the depth first spanning tree t , either u is an ancestor of v or v is an ancestor of u . So, there are no cross edges relative to a depth first spanning tree ((u, v) is a *cross edge* relative to t if and only if u is not an ancestor of v and v is not an ancestor of u).

Proof: To see this, assume that $(u, v) \in E(G)$ and (u, v) is a cross edge. Then (u, v) cannot be a tree edge as otherwise u is the parent of v or vice versa. So, (u, v) must be a back edge. Without loss of generality, we can assume $dfn[u] < dfn[v]$. Since vertex u is visited first, its exploration cannot be complete until vertex v is visited. From the definition of depth first search, it follows that u is an ancestor of all the vertices visited until u is completely explored. Hence u is an ancestor of v in t and (u, v) cannot be a cross edge. \square

We make the following observation

Lemma 6.3 The root node of a depth first spanning tree is an articulation point iff it has at least two children. Furthermore, if u is any other vertex, then it is not an articulation point iff from every child w of u it is possible to reach an ancestor of u using only a path made up of descendants of w and a back edge. \square

Note that if this cannot be done for some child w of u , then the deletion of vertex u leaves behind at least two nonempty components (one containing the root and the other containing vertex w). This observation leads to a simple rule to identify articulation points. For each vertex u , define $L[u]$ as follows:

$$L[u] = \min \{dfn[u], \min_{\substack{(u, w) \text{ is a back edge}}} \{L[w] \mid w \text{ is a child of } u\}, \min \{dfn[w] \mid$$

It should be clear that $L[u]$ is the lowest depth first number that can be reached from u using a path of descendants followed by at most one back edge. From the preceding discussion it follows that if u is not the root, then u is an articulation point iff u has a child w such that $L[w] \geq dfn[u]$.

Example 6.5 For the spanning tree of Figure 6.9(b) the L values are $L[1 : 10] = \{1, 1, 1, 1, 6, 8, 6, 6, 5, 4\}$. Vertex 3 is an articulation point as child 10 has $L[10] = 4$ and $dfn[3] = 3$. Vertex 2 is an articulation point as child 5 has $L[5] = 6$ and $dfn[2] = 6$. The only other articulation point is vertex 5; child 6 has $L[6] = 8$ and $dfn[5] = 7$. \square

$L[u]$ can be easily computed if the vertices of the depth first spanning tree are visited in postorder. Thus, to determine the articulation points,

it is necessary to perform a depth first search of the graph G and visit the nodes in the resulting depth first spanning tree in postorder. It is possible to do both these functions in parallel. Pseudocode Art (Algorithm 6.9) carries out a depth first search of G . During this search each newly visited vertex gets assigned its depth first number. At the same time, $L[i]$ is computed for each vertex in the tree. This algorithm assumes that the connected graph G and the arrays dfn and L are global. In addition, it is assumed that the variable num is also global. It is clear from the algorithm that when vertex u has been explored and a return made from the function, then $L[u]$ has been correctly computed. Note that in the `else` clause of line 15, if $w \neq v$, then either (u, w) is a back edge or $dfn[w] > dfn[u] \geq L[u]$. In either case, $L[u]$ is correctly updated. The initial call to Art is $\text{Art}(1, 0)$. Note dfn is initialized to zero before invoking Art.

```

1  Algorithm Art( $u, v$ )
2  //  $u$  is a start vertex for depth first search.  $v$  is its parent if any
3  // in the depth first spanning tree. It is assumed that the global
4  // array  $dfn$  is initialized to zero and that the global variable
5  //  $num$  is initialized to 1.  $n$  is the number of vertices in  $G$ .
6  {
7       $dfn[u] := num$ ;  $L[u] := num$ ;  $num := num + 1$ ;
8      for each vertex  $w$  adjacent from  $u$  do
9      {
10         if ( $dfn[w] = 0$ ) then
11             {
12                 Art( $w, u$ ); //  $w$  is unvisited.
13                  $L[u] := \min(L[u], L[w])$ ;
14             }
15         else if ( $w \neq v$ ) then  $L[u] := \min(L[u], dfn[w])$ ;
16     }
17 }
```

Algorithm 6.9 Pseudocode to compute dfn and L

Once $L[1 : n]$ has been computed, the articulation points can be identified in $O(e)$ time. Since Art has a complexity $O(n + e)$, where e is the number of edges in G , the articulation points of G can be determined in $O(n + e)$ time.

Now, what needs to be done to determine the biconnected components of G ? If following the call to Art (line 12) $L[w] \geq dfn[u]$, then we know that u is either the root or an articulation point. Regardless of whether u is not the root or is the root and has one or more children, the edge (u, w) together with

all edges (both tree and back) encountered during this call to `Art` (except for edges in other biconnected components contained in subtree w) forms a biconnected component. A formal proof of this statement appears in the proof of Theorem 6.5. The modified algorithm appears as Algorithm 6.10.

```

1      Algorithm BiComp( $u, v$ )
2      //  $u$  is a start vertex for depth first search.  $v$  is its parent if
3      // any in the depth first spanning tree. It is assumed that the
4      // global array  $dfn$  is initially zero and that the global variable
5      //  $num$  is initialized to 1.  $n$  is the number of vertices in  $G$ .
6      {
7           $dfn[u] := num; L[u] := num; num := num + 1;$ 
8          for each vertex  $w$  adjacent from  $u$  do
9              {
9.1             if  $((v \neq w) \text{ and } (dfn[w] < dfn[u]))$  then
9.2                 add  $(u, w)$  to the top of a stack  $s$ ;
10            if  $(dfn[w] = 0)$  then
11                {
11.1               if  $(L[w] \geq dfn[u])$  then
11.2                   {
11.3                     write ("New bicomponent");
11.4                     repeat
11.5                         {
11.6                             Delete an edge from the top of stack  $s$ ;
11.7                             Let this edge be  $(x, y)$ ;
11.8                             write  $(x, y)$ ;
11.9                             } until  $((x, y) = (u, w)) \text{ or } ((x, y) = (w, u))$ ;
11.10                        }
12                    BiComp( $w, u$ ); //  $w$  is unvisited.
13                     $L[u] := \min(L[u], L[w]);$ 
14                }
15            else if  $(w \neq v)$  then  $L[u] := \min(L[u], dfn[w]);$ 
16        }
17    }

```

Algorithm 6.10 Pseudocode to determine bicomponents

One can verify that the computing time of Algorithm 6.10 remains $O(n + e)$. The following theorem establishes the correctness of the algorithm. Note that when G has only one vertex, it has no edges so the algorithm generates

no output. In this case G does have a biconnected component, namely its single vertex. This case can be handled separately.

Theorem 6.5 Algorithm 6.10 correctly generates the biconnected components of the connected graph G when G has at least two vertices.

Proof: This can be shown by induction on the number of biconnected components in G . Clearly, for all biconnected graphs G , the root u of the depth first spanning tree has only one child w . Furthermore, w is the only vertex for which $L[w] \geq dfn[u]$ in line 11.1 of Algorithm 6.10. By the time w has been explored, all edges in G have been output as one biconnected component.

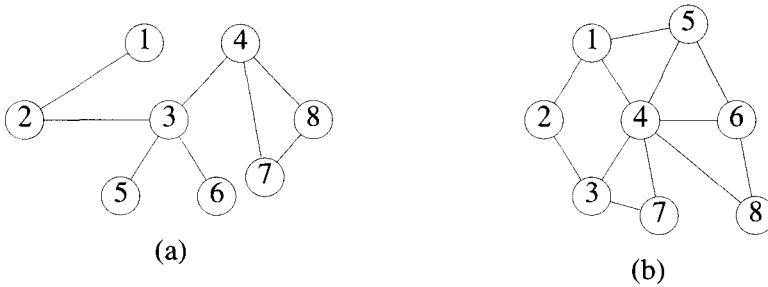
Now assume the algorithm works correctly for all connected graphs G with at most m biconnected components. We show that it also works correctly for all connected graphs with $m + 1$ biconnected components. Let G be any such graph. Consider the first time that $L[w] \geq dfn[u]$ in line 11.1. At this time no edges have been output and so all edges in G incident to the descendants of w are on the stack and are above the edge (u, w) . Since none of the descendants of u is an articulation point and u is, it follows that the set of edges above (u, w) on the stack forms a biconnected component together with the edge (u, w) . Once these edges have been deleted from the stack and output, the algorithm behaves essentially as it would on the graph G' , obtained by deleting from G the biconnected component just output. The behavior of the algorithm on G differs from that on G' only in that during the completion of the exploration of vertex u , some edges (u, r) such that (u, r) is in the component just output may be considered. However, for all such edges, $dfn[r] \neq 0$ and $dfn[r] > dfn[u] \geq L[u]$. Hence, these edges only result in a vacuous iteration of the **for** loop of line 8 and do not materially affect the algorithm.

One can easily establish that G' has at least two vertices. Since in addition G' has exactly m biconnected components, it follows from the induction hypothesis that the remaining components are correctly generated. \square

It should be noted that the algorithm described above will work with any spanning tree relative to which the given graph has no cross edges. Unfortunately, graphs can have cross edges relative to breadth first spanning trees. Hence, algorithm Art cannot be adapted to BFS.

EXERCISES

1. For the graphs of Figure 6.10 identify the articulation points and draw the biconnected components.
2. Show that if G is a connected undirected graph, then no edge of G can be in two different biconnected components.

**Figure 6.10** Graphs for Exercise 1

3. Let $G_i = (V_i, E_i)$, $1 \leq i \leq k$, be the biconnected components of a connected graph G . Show that
 - (a) If $i \neq j$, then $V_i \cap V_j$ contains at most one vertex.
 - (b) Vertex v is an articulation point of G iff $\{v\} = V_i \cap V_j$ for some i and j , $i \neq j$.
4. Show that the scheme of Algorithm 6.8 may use more than the minimum number of edges needed to make G biconnected.
5. Let G be a connected undirected graph. Write an algorithm to find the minimum number of edges that have to be added to G so that G becomes biconnected. Your algorithm should output such a set of edges. What are the time and space requirements of your algorithm?
6. Show that if t is a breadth first spanning tree for an undirected connected graph G , then G may have cross edges relative to t .
7. Prove that a nonroot vertex u is an articulation point iff $L[w] \geq dfn[u]$ for some child w of u .
8. Prove that in BiComp (Algorithm 6.10) if either $v = w$ or $dfn[w] > dfn[u]$, then edge (u, w) is either already on the stack of edges or has been output as part of a biconnected component.
9. Let $G(V, E)$ be any connected undirected graph. A *bridge* of G is defined to be an edge of G which when removed from G , will make it disconnected. Present an $O(|E|)$ time algorithm to find all the bridges of G .
10. Let $S(V, T)$ be any DFS tree for a given connected undirected graph $G(V, E)$. Prove that a leaf of S can not be an articulation point of G .

11. Prove or disprove: “An undirected graph $G(V, E)$ is biconnected if and only if for each pair of distinct vertices v and w in V there are two distinct paths from v to w that have no vertices in common except v and w .”

6.5 REFERENCES AND READINGS

Several applications of depth first search to graph problems are given in “Depth first search and linear graph algorithms,” by R. Tarjan, *SIAM Journal on Computing* 1, no. 2 (1972): 146–160.

The $O(n + e)$ depth first algorithm for biconnected components is due to R. Tarjan and appears in the preceding paper. This paper also contains an $O(n + e)$ algorithm to find the strongly connected components of a directed graph.

An $O(n + e)$ algorithm to find a smallest set of edges that, when added to a graph G , produces a biconnected graph has been given by A. Rosenthal and A. Goldner.

For an extensive coverage on graph algorithms see:

Data Structures and Network Algorithms, by R. E. Tarjan, Society for Industrial and Applied Mathematics, 1983.

Algorithmic Graph Theory, by A. Gibbons, Cambridge University Press, 1985.

Algorithmic Graph Theory and Perfect Graphs, by M. Golumbic, Academic Press, 1980.

UNIT - 2

DIVIDE AND CONQUER

Chapter 3

DIVIDE-AND-CONQUER

3.1 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 can 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 algorithm. Now smaller and smaller subproblems of the same kind are generated until eventually subproblems that are small enough to be solved without splitting are produced.

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 examine here. We can write a control abstraction that mirrors the way an algorithm based on divide-and-conquer will look. By a *control abstraction* we mean a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. DAndC (Algorithm 3.1) is initially invoked as DAndC(P), where P is the problem to be solved.

$\text{Small}(P)$ is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this is so, the function S is invoked. Otherwise the problem P is divided into smaller subproblems. These subproblems P_1, P_2, \dots, P_k are solved by recursive applications of DAndC. Combine is a function that determines the solution to P using the solutions to the k subproblems. If the size of P is n and the sizes of the k subproblems are n_1, n_2, \dots, n_k , respectively, then the

```

1   Algorithm DAndC( $P$ )
2   {
3       if Small( $P$ ) then return  $S(P)$ ;
4       else
5           {
6               divide  $P$  into smaller instances  $P_1, P_2, \dots, P_k$ ,  $k \geq 1$ ;
7               Apply DAndC to each of these subproblems;
8               return Combine(DAndC( $P_1$ ),DAndC( $P_2$ ),...,DAndC( $P_k$ ));
9           }
10      }

```

Algorithm 3.1 Control abstraction for divide-and-conquer

computing time of DAndC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \dots + T(n_k) + f(n) & \text{otherwise} \end{cases} \quad (3.1)$$

where $T(n)$ is the time for DAndC on any input of size n and $g(n)$ is the time to compute the answer directly for small inputs. The function $f(n)$ is the time for dividing P and combining the solutions to subproblems. For divide-and-conquer-based algorithms that produce subproblems of the same type as the original problem, it is very natural to first describe such algorithms using recursion.

The complexity of many divide-and-conquer algorithms is given by recurrences of the form

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases} \quad (3.2)$$

where a and b are known constants. We assume that $T(1)$ is known and n is a power of b (i.e., $n = b^k$).

One of the methods for solving any such recurrence relation is called the *substitution method*. This method repeatedly makes substitution for each occurrence of the function T in the right-hand side until all such occurrences disappear.

Example 3.1 Consider the case in which $a = 2$ and $b = 2$. Let $T(1) = 2$ and $f(n) = n$. We have

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \end{aligned}$$

In general, we see that $T(n) = 2^i T(n/2^i) + in$, for any $\log_2 n \geq i \geq 1$. In particular, then, $T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + n \log_2 n$, corresponding to the choice of $i = \log_2 n$. Thus, $T(n) = nT(1) + n \log_2 n = n \log_2 n + 2n$. \square

Beginning with the recurrence (3.2) and using the substitution method, it can be shown that

$$T(n) = n^{\log_b a} [T(1) + u(n)]$$

where $u(n) = \sum_{j=1}^k h(b^j)$ and $h(n) = f(n)/n^{\log_b a}$. Table 3.1 tabulates the asymptotic value of $u(n)$ for various values of $h(n)$. This table allows one to easily obtain the asymptotic value of $T(n)$ for many of the recurrences one encounters when analyzing divide-and-conquer algorithms. Let us consider some examples using this table.

$h(n)$	$u(n)$
$O(n^r)$, $r < 0$	$O(1)$
$\Theta((\log n)^i)$, $i \geq 0$	$\Theta((\log n)^{i+1}/(i+1))$
$\Omega(n^r)$, $r > 0$	$\Theta(h(n))$

Table 3.1 $u(n)$ values for various $h(n)$ values

Example 3.2 Look at the following recurrence when n is a power of 2:

$$T(n) = \begin{cases} T(1) & n = 1 \\ T(n/2) + c & n > 1 \end{cases}$$

Comparing with (3.2), we see that $a = 1$, $b = 2$, and $f(n) = c$. So, $\log_b(a) = 0$ and $h(n) = f(n)/n^{\log_b a} = c = c(\log n)^0 = \Theta((\log n)^0)$. From Table 3.1, we obtain $u(n) = \Theta(\log n)$. So, $T(n) = n^{\log_b a}[c + \Theta(\log n)] = \Theta(\log n)$. \square

Example 3.3 Next consider the case in which $a = 2$, $b = 2$, and $f(n) = cn$. For this recurrence, $\log_b a = 1$ and $h(n) = f(n)/n = c = \Theta((\log n)^0)$. Hence, $u(n) = \Theta(\log n)$ and $T(n) = n[T(1) + \Theta(\log n)] = \Theta(n \log n)$. \square

Example 3.4 As another example, consider the recurrence $T(n) = 7T(n/2) + 18n^2$, $n \geq 2$ and a power of 2. We obtain $a = 7$, $b = 2$, and $f(n) = 18n^2$. So, $\log_b a = \log_2 7 \approx 2.81$ and $h(n) = 18n^2/n^{\log_2 7} = 18n^{2-\log_2 7} = O(n^r)$, where $r = 2 - \log_2 7 < 0$. So, $u(n) = O(1)$. The expression for $T(n)$ is

$$\begin{aligned} T(n) &= n^{\log_2 7}[T(1) + O(1)] \\ &= \Theta(n^{\log_2 7}) \end{aligned}$$

as $T(1)$ is assumed to be a constant. \square

Example 3.5 As a final example, consider the recurrence $T(n) = 9T(n/3) + 4n^6$, $n \geq 3$ and a power of 3. Comparing with (3.2), we obtain $a = 9$, $b = 3$, and $f(n) = 4n^6$. So, $\log_b a = 2$ and $h(n) = 4n^6/n^2 = 4n^4 = \Omega(n^4)$. From Table 3.1, we see that $u(n) = \Theta(h(n)) = \Theta(n^4)$. So,

$$\begin{aligned} T(n) &= n^2[T(1) + \Theta(n^4)] \\ &= \Theta(n^6) \end{aligned}$$

as $T(1)$ can be assumed constant. \square

EXERCISES

1. Solve the recurrence relation (3.2) for the following choices of a , b , and $f(n)$ (c being a constant):

- (a) $a = 1$, $b = 2$, and $f(n) = cn$
- (b) $a = 5$, $b = 4$, and $f(n) = cn^2$
- (c) $a = 28$, $b = 3$, and $f(n) = cn^3$

2. Solve the following recurrence relations using the substitution method:

- (a) All three recurrences of Exercise 1.
- (b)

$$T(n) = \begin{cases} 1 & n \leq 4 \\ T(\sqrt{n}) + c & n > 4 \end{cases}$$

(c)

$$T(n) = \begin{cases} 1 & n \leq 4 \\ 2T(\sqrt{n}) + \log n & n > 4 \end{cases}$$

(d)

$$T(n) = \begin{cases} 1 & n \leq 4 \\ 2T(\sqrt{n}) + \frac{\log n}{\log \log n} & n > 4 \end{cases}$$

3.2 BINARY SEARCH

Let a_i , $1 \leq i \leq n$, be a list of elements that are sorted in nondecreasing order. Consider the problem of determining whether a given element x is present in the list. If 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. Let $P = (n, a_i, \dots, a_\ell, x)$ denote an arbitrary instance of this search problem (n is the number of elements in the list, a_i, \dots, a_ℓ is the list of elements, and x is the element searched for).

Divide-and-conquer can be used to solve this problem. Let $\text{Small}(P)$ be true if $n = 1$. In this case, $S(P)$ will take the value i if $x = a_i$; otherwise it will take the value 0. Then $g(1) = \Theta(1)$. If P has more than one element, it can be divided (or reduced) into a new subproblem as follows. Pick an index q (in the range $[i, \ell]$) and compare x with a_q . There are three possibilities: (1) $x = a_q$: In this case the problem P is immediately solved. (2) $x < a_q$: In this case x has to be searched for only in the sublist $a_i, a_{i+1}, \dots, a_{q-1}$. Therefore, P reduces to $(q-i, a_i, \dots, a_{q-1}, x)$. (3) $x > a_q$: In this case the sublist to be searched is a_{q+1}, \dots, a_ℓ . P reduces to $(\ell-q, a_{q+1}, \dots, a_\ell, x)$.

In this example, any given problem P gets divided (reduced) into one new subproblem. This division takes only $\Theta(1)$ time. After a comparison with a_q , the instance remaining to be solved (if any) can be solved by using this divide-and-conquer scheme again. If q is always chosen such that a_q is the middle element (that is, $q = \lfloor (n+1)/2 \rfloor$), then the resulting search algorithm is known as binary search. Note that the answer to the new subproblem is also the answer to the original problem P ; there is no need for any combining. Algorithm 3.2 describes this binary search method, where BinSrch has four inputs $a[]$, i , l , and x . It is initially invoked as $\text{BinSrch}(a, 1, n, x)$.

A nonrecursive version of BinSrch is given in Algorithm 3.3. BinSearch has three inputs a , n , and x . The **while** loop continues processing as long as there are more elements left to check. At the conclusion of the procedure 0 is returned if x is not present, or j is returned, such that $a[j] = x$.

Is BinSearch an algorithm? We must be sure that all of the operations such as comparisons between x and $a[mid]$ are well defined. The relational operators carry out the comparisons among elements of a correctly if these operators are appropriately defined. Does BinSearch terminate? We observe

```

1  Algorithm BinSrch( $a, i, l, x$ )
2  // Given an array  $a[i : l]$  of elements in nondecreasing
3  // order,  $1 \leq i \leq l$ , determine whether  $x$  is present, and
4  // if so, return  $j$  such that  $x = a[j]$ ; else return 0.
5  {
6      if ( $l = i$ ) then // If Small( $P$ )
7      {
8          if ( $x = a[i]$ ) then return  $i$ ;
9          else return 0;
10     }
11     else
12     { // Reduce  $P$  into a smaller subproblem.
13          $mid := \lfloor (i + l)/2 \rfloor$ ;
14         if ( $x = a[mid]$ ) then return  $mid$ ;
15         else if ( $x < a[mid]$ ) then
16             return BinSrch( $a, i, mid - 1, x$ );
17         else return BinSrch( $a, mid + 1, l, x$ );
18     }
19 }
```

Algorithm 3.2 Recursive binary search

```

1  Algorithm BinSearch( $a, n, x$ )
2  // Given an array  $a[1 : n]$  of elements in nondecreasing
3  // order,  $n \geq 0$ , determine whether  $x$  is present, and
4  // if so, return  $j$  such that  $x = a[j]$ ; else return 0.
5  {
6       $low := 1; high := n;$ 
7      while ( $low \leq high$ ) do
8      {
9           $mid := \lfloor (low + high)/2 \rfloor$ ;
10         if ( $x < a[mid]$ ) then  $high := mid - 1$ ;
11         else if ( $x > a[mid]$ ) then  $low := mid + 1$ ;
12         else return  $mid$ ;
13     }
14     return 0;
15 }
```

Algorithm 3.3 Iterative binary search

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* becomes greater than *high* and causes termination in a finite number of steps if *x* is not present.

Example 3.6 Let us select the 14 entries

$$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$$

place them in $a[1 : 14]$, and simulate the steps that BinSearch goes through as it searches for different values of *x*. Only the variables *low*, *high*, and *mid* need to be traced as we simulate the algorithm. We try the following values for *x*: 151, -14, and 9 for two successful searches and one unsuccessful search. Table 3.2 shows the traces of BinSearch on these three inputs. \square

$x = 151$	<i>low</i>	<i>high</i>	<i>mid</i>	$x = -14$	<i>low</i>	<i>high</i>	<i>mid</i>
	1	14	7		1	14	7
	8	14	11		1	6	3
	12	14	13		1	2	1
	14	14	14		2	2	2
			found		2	1	not found
$x = 9$	<i>low</i>	<i>high</i>	<i>mid</i>				
	1	14	7				
	1	6	3				
	4	6	5				
			found				

Table 3.2 Three examples of binary search on 14 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 algorithms are very useful because they establish the correctness of the algorithm for *all* possible inputs, whereas testing gives much less in the way of guarantees. Unfortunately, algorithm proving is a very difficult process and the complete proof of an algorithm can be many times longer than the algorithm itself. We content ourselves with an informal “proof” of BinSearch.

Theorem 3.1 Algorithm BinSearch(*a*, *n*, *x*) 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 $a[1] \leq a[2] \leq \dots \leq a[n]$. If *n* = 0, the **while** loop is

not entered and 0 is returned. Otherwise we observe that each time through the loop the possible elements to be checked for equality with x are $a[low]$, $a[low + 1], \dots, a[mid]$, $\dots, a[high]$. If $x = a[mid]$, then the algorithm terminates successfully. Otherwise the range is narrowed by either increasing low to $mid + 1$ or decreasing $high$ to $mid - 1$. Clearly this narrowing of the range does not affect the outcome of the search. If low becomes greater than $high$, then x is not present and hence the loop is exited. \square

Notice that to fully test binary search, we need not concern ourselves with the values of $a[1 : n]$. By varying x sufficiently, we can observe all possible computation sequences of `BinSearch` without devising different values for a . To test all successful searches, x must take on the n values in a . To test all unsuccessful searches, x need only take on $n + 1$ different values. Thus the complexity of testing `BinSearch` is $2n + 1$ for each n .

Now let's analyze the execution profile of `BinSearch`. The two relevant characteristics of this profile are the frequency counts and space required for the algorithm. For `BinSearch`, storage is required for the n elements of the array plus the variables low , $high$, mid , and x , or $n + 4$ locations. As for the time, there are three possibilities to consider: the best, average, and worst cases.

Suppose we begin by determining the time for `BinSearch` on the previous data set. We observe that the only operations in the algorithm are comparisons and some arithmetic and data movements. We concentrate on comparisons between x and the elements in $a[]$, recognizing that the frequency count of all other operations is of the same order as that for these comparisons. Comparisons between x and elements of $a[]$ are referred to as *element comparisons*. We assume that only one comparison is needed to determine which of the three possibilities of the `if` statement holds. The number of element comparisons needed to find each of the 14 elements is

$a:$	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements:	-15	-6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4

No element requires more than 4 comparisons to be found. The average is obtained by summing the comparisons needed to find all 14 items and dividing by 14; this yields $45/14$, or approximately 3.21, comparisons per successful search on the average. There are 15 possible ways that an unsuccessful search may terminate depending on the value of x . If $x < a[1]$, the algorithm requires 3 element comparisons to determine that x is not present. For all the remaining possibilities, `BinSearch` requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is $(3 + 14 * 4)/15 = 59/15 \approx 3.93$.

The analysis just done applies to any sorted sequence containing 14 elements. But the result we would prefer is a formula for n elements. A good

way to derive such a formula plus a better way to understand the algorithm is to consider the sequence of values for mid that are produced by `BinSearch` for all possible values of x . These values are nicely described using a binary decision tree in which the value in each node is the value of mid . For example, if $n = 14$, then Figure 3.1 contains a binary decision tree that traces the way in which these values are produced by `BinSearch`.

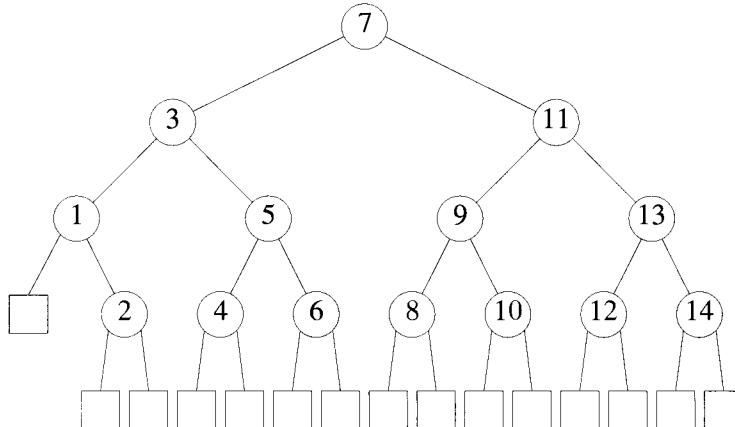


Figure 3.1 Binary decision tree for binary search, $n = 14$

The first comparison is x with $a[7]$. If $x < a[7]$, then the next comparison is with $a[3]$; similarly, if $x > a[7]$, then the next comparison is with $a[11]$. Each path through the tree represents a sequence of comparisons in the binary search method. If x is present, then the algorithm will end at one of the circular nodes that lists the index into the array where x was found. If x is not present, the algorithm will terminate at one of the square nodes. Circular nodes are called *internal nodes*, and square nodes are referred to as *external nodes*.

Theorem 3.2 If n is in the range $[2^{k-1}, 2^k)$, then `BinSearch` makes at most k element comparisons for a successful search and either $k - 1$ or k comparisons for an unsuccessful search. (In other words the time for a successful search is $O(\log n)$ and for an unsuccessful search is $\Theta(\log n)$).

Proof: Consider the binary decision tree describing the action of `BinSearch` on n elements. All successful searches end at a circular node whereas all unsuccessful searches end at a square node. If $2^{k-1} \leq n < 2^k$, then all circular nodes are at levels $1, 2, \dots, k$ whereas all square nodes are at levels

k and $k + 1$ (note that the root is at level 1). The number of element comparisons needed to terminate at a circular node on level i is i whereas the number of element comparisons needed to terminate at a square node at level i is only $i - 1$. The theorem follows. \square

Theorem 3.2 states the worst-case time for binary search. To determine the average behavior, we need to look more closely at the binary decision tree and equate its size to the number of element comparisons in the algorithm. The *distance* of a node from the root is one less than its level. The *internal path length* I is the sum of the distances of all internal nodes from the root. Analogously, the *external path length* E is the sum of the distances of all external nodes from the root. It is easy to show by induction that for any binary tree with n internal nodes, E and I are related by the formula

$$E = I + 2n$$

It turns out that there is a simple relationship between E , I , and the average number of comparisons in binary search. Let $A_s(n)$ be the average number of comparisons in a successful search, and $A_u(n)$ the average number of comparisons in an unsuccessful search. The number of comparisons needed to find an element represented by an internal node is one more than the distance of this node from the root. Hence,

$$A_s(n) = 1 + I/n$$

The number of comparisons on any path from the root to an external node is equal to the distance between the root and the external node. Since every binary tree with n internal nodes has $n + 1$ external nodes, it follows that

$$A_u(n) = E/(n + 1)$$

Using these three formulas for E , $A_s(n)$, and $A_u(n)$, we find that

$$A_s(n) = (1 + 1/n)A_u(n) - 1$$

From this formula we see that $A_s(n)$ and $A_u(n)$ are directly related. The minimum value of $A_s(n)$ (and hence $A_u(n)$) is achieved by an algorithm whose binary decision tree has minimum external and internal path length. This minimum is achieved by the binary tree all of whose external nodes are on adjacent levels, and this is precisely the tree that is produced by binary search. From Theorem 3.2 it follows that E is proportional to $n \log n$. Using this in the preceding formulas, we conclude that $A_s(n)$ and $A_u(n)$ are both proportional to $\log n$. Thus we conclude that the average- and worst-case numbers of comparisons for binary search are the same to within a constant

factor. The best-case analysis is easy. For a successful search only one element comparison is needed. For an unsuccessful search, Theorem 3.2 states that $\lfloor \log n \rfloor$ element comparisons are needed in the best case.

In conclusion we are now able to completely describe the computing time of binary search by giving formulas that describe the best, average, and worst cases:

successful searches	unsuccessful searches
$\Theta(1), \Theta(\log n), \Theta(\log n)$	$\Theta(\log n)$
best, average, worst	best, average, worst

Can we expect another searching algorithm to be significantly better than binary search in the worst case? This question is 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 that depends on comparisons of entire elements. Viewed in this way, we observe that the *longest* path to discover any element is minimized by binary search, and so any alternative algorithm is no better from this point of view.

Before we end this section, there is an interesting variation of binary search that makes only one comparison per iteration of the **while** loop. This variation appears as Algorithm 3.4. The correctness proof of this variation is left as an exercise.

`BinSearch` will sometimes make twice as many element comparisons as `BinSearch1` (for example, when $x > a[n]$). However, for successful searches `BinSearch1` may make $(\log n)/2$ more element comparisons than `BinSearch` (for example, when $x = a[mid]$). The analysis of `BinSearch1` is left as an exercise. It should be easy to see that the best-, average-, and worst-case times for `BinSearch1` are $\Theta(\log n)$ for both successful and unsuccessful searches.

These two algorithms were run on a Sparc 10/30. The first two rows in Table 3.3 represent the average time for a successful search. The second set of two rows give the average times for all possible unsuccessful searches. For both successful and unsuccessful searches `BinSearch1` did marginally better than `BinSearch`.

EXERCISES

1. Run the recursive and iterative versions of binary search and compare the times. For appropriate sizes of n , have each algorithm find every element in the set. Then try all $n + 1$ possible unsuccessful searches.
2. Prove by induction the relationship $E = I + 2n$ for a binary tree with n internal nodes. The variables E and I are the external and internal path length, respectively.

```

1  Algorithm BinSearch1( $a, n, x$ )
2  // Same specifications as BinSearch except  $n > 0$ 
3  {
4       $low := 1; high := n + 1;$ 
5      //  $high$  is one more than possible.
6      while ( $low < (high - 1)$ ) do
7      {
8           $mid := \lfloor (low + high)/2 \rfloor;$ 
9          if ( $x < a[mid]$ ) then  $high := mid;$ 
10         // Only one comparison in the loop.
11         else  $low := mid; // x \geq a[mid]$ 
12     }
13     if ( $x = a[low]$ ) then return  $low; // x$  is present.
14     else return 0; //  $x$  is not present.
15 }
```

Algorithm 3.4 Binary search using one comparison per cycle

Array sizes	5,000	10,000	15,000	20,000	25,000	30,000
successful searches						
BinSearch	51.30	67.95	67.72	73.85	76.77	73.40
BinSearch1	47.68	53.92	61.98	67.46	68.95	71.11
unsuccessful searches						
BinSearch	50.40	66.36	76.78	79.54	78.20	81.15
BinSearch1	41.93	52.65	63.33	66.86	69.22	72.26

Table 3.3 Computing times for two binary search algorithms; times are in microseconds

3. In an infinite array, the first n cells contain integers in sorted order and the rest of the cells are filled with ∞ . Present an algorithm that takes x as input and finds the position of x in the array in $\Theta(\log n)$ time. *You are not given the value of n .*
4. Devise a “binary” search algorithm that splits the set not into two sets of (almost) equal sizes but into two sets, one of which is twice the size of the other. How does this algorithm compare with binary search?
5. Devise a ternary search algorithm that first tests the element at position $n/3$ for equality with some value x , and then checks the element at $2n/3$ and either discovers x or reduces the set size to one-third the size of the original. Compare this with binary search.
6. (a) Prove that BinSearch1 works correctly.
 (b) Verify that the following algorithm segment functions correctly according to the specifications of binary search. Discuss its computing time.

```

low := 1; high := n;
repeat {
    mid := ⌊(low + high)/2⌋;
    if (x  $\geq$  a[mid]) then low := mid;
    else high := mid;
} until ((low + 1) = high)

```

3.3 FINDING THE MAXIMUM AND MINIMUM

Let us consider another simple problem that can be solved by the divide-and-conquer technique. The problem is to find the maximum and minimum items in a set of n elements. Algorithm 3.5 is a straightforward algorithm to accomplish this.

In analyzing the time complexity of this algorithm, we once again concentrate on the number of element comparisons. The justification for this is that the frequency count for other operations in this algorithm is of the same order as that for element comparisons. More importantly, when the elements in $a[1 : n]$ are polynomials, vectors, very large numbers, or strings of characters, the cost of an element comparison is much higher than the cost of the other operations. Hence the time is determined mainly by the total cost of the element comparisons.

StraightMaxMin requires $2(n - 1)$ element comparisons in the best, average, and worst cases. An immediate improvement is possible by realizing

```

1 Algorithm StraightMaxMin( $a, n, max, min$ )
2 // Set  $max$  to the maximum and  $min$  to the minimum of  $a[1 : n]$ .
3 {
4      $max := min := a[1];$ 
5     for  $i := 2$  to  $n$  do
6     {
7         if ( $a[i] > max$ ) then  $max := a[i];$ 
8         if ( $a[i] < min$ ) then  $min := a[i];$ 
9     }
10 }
```

Algorithm 3.5 Straightforward maximum and minimum

that the comparison $a[i] < min$ is necessary only when $a[i] > max$ is false. Hence we can replace the contents of the **for** loop by

```

if ( $a[i] > max$ ) then  $max := a[i];$ 
else if ( $a[i] < min$ ) then  $min := a[i];$ 
```

Now the best case occurs when the elements are in increasing order. The number of element comparisons is $n - 1$. The worst case occurs when the elements are in decreasing order. In this case the number of element comparisons is $2(n - 1)$. The average number of element comparisons is less than $2(n - 1)$. On the average, $a[i]$ is greater than max half the time, and so the average number of comparisons is $3n/2 - 1$.

A divide-and-conquer algorithm for this problem would proceed as follows: Let $P = (n, a[i], \dots, a[j])$ denote an arbitrary instance of the problem. Here n is the number of elements in the list $a[i], \dots, a[j]$ and we are interested in finding the maximum and minimum of this list. Let $\text{Small}(P)$ be true when $n \leq 2$. In this case, the maximum and minimum are $a[i]$ if $n = 1$. If $n = 2$, the problem can be solved by making one comparison.

If the list has more than two elements, P has to be divided into smaller instances. For example, we might divide P into the two instances $P_1 = ([\lfloor n/2 \rfloor], a[1], \dots, a[\lfloor n/2 \rfloor])$ and $P_2 = (n - [\lfloor n/2 \rfloor], a[\lfloor n/2 \rfloor + 1], \dots, a[n])$. After having divided P into two smaller subproblems, we can solve them by recursively invoking the same divide-and-conquer algorithm. How can we combine the solutions for P_1 and P_2 to obtain a solution for P ? If $\text{MAX}(P)$ and $\text{MIN}(P)$ are the maximum and minimum of the elements in P , then $\text{MAX}(P)$ is the larger of $\text{MAX}(P_1)$ and $\text{MAX}(P_2)$. Also, $\text{MIN}(P)$ is the smaller of $\text{MIN}(P_1)$ and $\text{MIN}(P_2)$.

Algorithm 3.6 results from applying the strategy just described. MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements $\{a(i), a(i+1), \dots, a(j)\}$. The situation of set sizes one ($i = j$) and two ($i = j - 1$) are handled separately. For sets containing more than two elements, the midpoint is determined (just as in binary search) and two new subproblems are generated. When the maxima and minima of these subproblems are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire set.

```

1  Algorithm MaxMin( $i, j, max, min$ )
2  //  $a[1 : n]$  is a global array. Parameters  $i$  and  $j$  are integers,
3  //  $1 \leq i \leq j \leq n$ . The effect is to set  $max$  and  $min$  to the
4  // largest and smallest values in  $a[i : j]$ , respectively.
5  {
6      if ( $i = j$ ) then  $max := min := a[i]$ ; // Small( $P$ )
7      else if ( $i = j - 1$ ) then // Another case of Small( $P$ )
8      {
9          if ( $a[i] < a[j]$ ) then
10         {
11              $max := a[j]; min := a[i];$ 
12         }
13         else
14         {
15              $max := a[i]; min := a[j];$ 
16         }
17     }
18     else
19     {
20         // If  $P$  is not small, divide  $P$  into subproblems.
21         // Find where to split the set.
22          $mid := \lfloor (i + j)/2 \rfloor;$ 
23         // Solve the subproblems.
24         MaxMin( $i, mid, max, min$ );
25         MaxMin( $mid + 1, j, max1, min1$ );
26         // Combine the solutions.
27         if ( $max < max1$ ) then  $max := max1;$ 
28         if ( $min > min1$ ) then  $min := min1;$ 
29     }
}

```

Algorithm 3.6 Recursively finding the maximum and minimum

The procedure is initially invoked by the statement

$$\text{MaxMin}(1, n, x, y)$$

Suppose we simulate MaxMin on the following nine elements:

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	-5	-8	15	60	17	31	47

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. For this algorithm each node has four items of information: i , j , \max , and \min . On the array $a[]$ above, the tree of Figure 3.2 is produced.

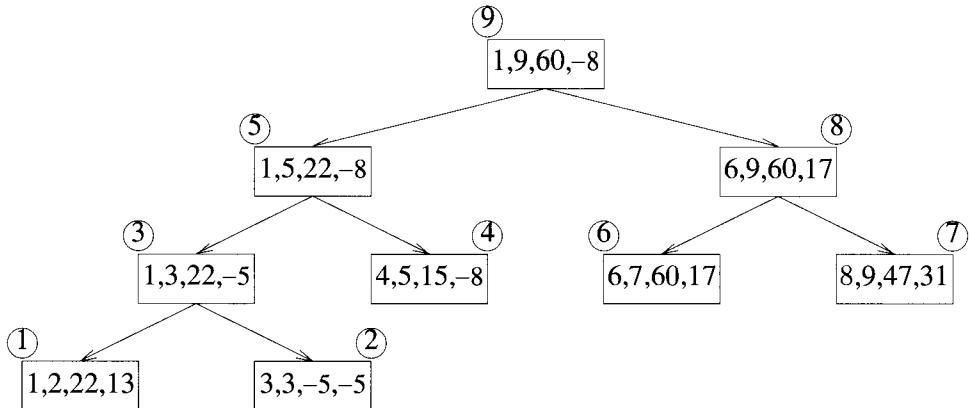


Figure 3.2 Trees of recursive calls of MaxMin

Examining Figure 3.2, we see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new calls to MaxMin, where i and j have the values 1, 5 and 6, 9, respectively, and thus split the set into two subsets of approximately the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call). The circled numbers in the upper left corner of each node represent the orders in which \max and \min are assigned values.

Now what is the number of element comparisons needed for **MaxMin**? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k , then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \sum_{1 \leq i \leq k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned} \tag{3.3}$$

Note that $3n/2 - 2$ is the best-, average-, and worst-case number of comparisons when n is a power of two.

Compared with the $2n - 2$ comparisons for the straightforward method, this is a saving of 25% in comparisons. It can be shown that no algorithm based on comparisons uses less than $3n/2 - 2$ comparisons. So in this sense algorithm **MaxMin** is optimal (see Chapter 10 for more details). But does this imply that **MaxMin** is better in practice? Not necessarily. In terms of storage, **MaxMin** is worse than the straightforward algorithm because it requires stack space for i , j , \max , \min , $\max1$, and $\min1$. Given n elements, there will be $\lfloor \log_2 n \rfloor + 1$ levels of recursion and we need to save seven values for each recursive call (don't forget the return address is also needed).

Let us see what the count is when element comparisons have the same cost as comparisons between i and j . Let $C(n)$ be this number. First, we observe that lines 6 and 7 in Algorithm 3.6 can be replaced with

```
if ( $i \geq j - 1$ ) { // Small( $P$ )
```

to achieve the same effect. Hence, a single comparison between i and $j - 1$ is adequate to implement the modified **if** statement. Assuming $n = 2^k$ for some positive integer k , we get

$$C(n) = \begin{cases} 2C(n/2) + 3 & n > 2 \\ 2 & n = 2 \end{cases}$$

Solving this equation, we obtain

$$\begin{aligned}
 C(n) &= 2C(n/2) + 3 \\
 &= 4C(n/4) + 6 + 3 \\
 &\vdots \\
 &= 2^{k-1}C(2) + 3 \sum_0^{k-2} 2^i \\
 &= 2^k + 3 * 2^{k-1} - 3 \\
 &= 5n/2 - 3
 \end{aligned} \tag{3.4}$$

The comparative figure for `StraightMaxMin` is $3(n - 1)$ (including the comparison needed to implement the `for` loop). This is larger than $5n/2 - 3$. Despite this, `MaxMin` will be slower than `StraightMaxMin` because of the overhead of stacking i, j, max , and min for the recursion.

Algorithm 3.6 makes several points. If comparisons among the elements of $a[]$ are much more costly than comparisons of integer variables, then the divide-and-conquer technique has yielded a more efficient (actually an optimal) algorithm. On the other hand, if this assumption is not true, the technique yields a less-efficient algorithm. Thus the divide-and-conquer strategy is seen to be only a guide to better algorithm design which may not always succeed. Also we see that it is sometimes necessary to work out the constants associated with the computing time bound for an algorithm. Both `MaxMin` and `StraightMaxMin` are $\Theta(n)$, so the use of asymptotic notation is not enough of a discriminator in this situation. Finally, see the exercises for another way to find the maximum and minimum using only $3n/2 - 2$ comparisons.

Note: In the design of any divide-and-conquer algorithm, typically, it is a straightforward task to define $\text{Small}(P)$ and $\text{S}(P)$. So, from now on, we only discuss how to divide any given problem P and how to combine the solutions to subproblems.

EXERCISES

1. Translate algorithm `MaxMin` into a computationally equivalent procedure that uses no recursion.
2. Test your iterative version of `MaxMin` derived above against `StraightMaxMin`. Count all comparisons.
3. There is an iterative algorithm for finding the maximum and minimum which, though not a divide-and-conquer-based algorithm, is probably more efficient than `MaxMin`. It works by comparing consecutive pairs of elements and then comparing the larger one with the current maximum and the smaller one with the current minimum. Write out

the algorithm completely and analyze the number of comparisons it requires.

4. In Algorithm 3.6, what happens if lines 7 to 17 are dropped? Does the resultant function still compute the maximum and minimum elements correctly?

3.4 MERGE SORT

As another example of divide-and-conquer, we investigate a sorting algorithm that has the nice property that in the worst case its complexity is $O(n \log n)$. This algorithm is called *merge sort*. We assume throughout that the elements are to be sorted in nondecreasing order. Given a sequence of n elements (also called keys) $a[1], \dots, a[n]$, the general idea is to imagine them split into two sets $a[1], \dots, a[\lfloor n/2 \rfloor]$ and $a[\lceil n/2 \rceil + 1], \dots, a[n]$. Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of n elements. Thus we have another ideal example of the divide-and-conquer strategy in which the splitting is into two equal-sized sets and the combining operation is the merging of two sorted sets into one.

`MergeSort` (Algorithm 3.7) describes this process very succinctly using recursion and a function `Merge` (Algorithm 3.8) which merges two sorted sets. Before executing `MergeSort`, the n elements should be placed in $a[1 : n]$. Then `MergeSort(1, n)` causes the keys to be rearranged into nondecreasing order in a .

Example 3.7 Consider the array of ten elements $a[1 : 10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)$. Algorithm `MergeSort` begins by splitting $a[]$ into two subarrays each of size five ($a[1 : 5]$ and $a[6 : 10]$). The elements in $a[1 : 5]$ are then split into two subarrays of size three ($a[1 : 3]$) and two ($a[4 : 5]$). Then the items in $a[1 : 3]$ are split into subarrays of size two ($a[1 : 2]$) and one ($a[3 : 3]$). The two values in $a[1 : 2]$ are split a final time into one-element subarrays, and now the merging begins. Note that no movement of data has yet taken place. A record of the subarrays is implicitly maintained by the recursive mechanism. Pictorially the file can now be viewed as

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

where vertical bars indicate the boundaries of subarrays. Elements $a[1]$ and $a[2]$ are merged to yield

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

```

1  Algorithm MergeSort(low, high)
2  // a[low : high] is a global array to be sorted.
3  // Small(P) is true if there is only one element
4  // to sort. In this case the list is already sorted.
5  {
6      if (low < high) then // If there are more than one element
7      {
8          // Divide P into subproblems.
9          // Find where to split the set.
10         mid :=  $\lfloor (\text{low} + \text{high})/2 \rfloor$ ;
11         // Solve the subproblems.
12         MergeSort(low, mid);
13         MergeSort(mid + 1, high);
14         // Combine the solutions.
15         Merge(low, mid, high);
16     }
17 }
```

Algorithm 3.7 Merge sort

Then $a[3]$ is merged with $a[1 : 2]$ and

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

is produced. Next, elements $a[4]$ and $a[5]$ are merged:

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

and then $a[1 : 3]$ and $a[4 : 5]$:

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

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

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

Elements $a[6]$ and $a[7]$ are merged. Then $a[8]$ is merged with $a[6 : 7]$:

```

1  Algorithm Merge(low, mid, high)
2  // a[low : high] is a global array containing two sorted
3  // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4  // is to merge these two sets into a single set residing
5  // in a[low : high]. b[ ] is an auxiliary global array.
6  {
7      h := low; i := low; j := mid + 1;
8      while ((h ≤ mid) and (j ≤ high)) do
9      {
10         if (a[h] ≤ a[j]) then
11             {
12                 b[i] := a[h]; h := h + 1;
13             }
14         else
15             {
16                 b[i] := a[j]; j := j + 1;
17             }
18         i := i + 1;
19     }
20     if (h > mid) then
21         for k := j to high do
22             {
23                 b[i] := a[k]; i := i + 1;
24             }
25     else
26         for k := h to mid do
27             {
28                 b[i] := a[k]; i := i + 1;
29             }
30     for k := low to high do a[k] := b[k];
31 }
```

Algorithm 3.8 Merging two sorted subarrays using auxiliary storage

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

Next $a[9]$ and $a[10]$ are merged, and then $a[6 : 8]$ and $a[9 : 10]$:

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

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

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

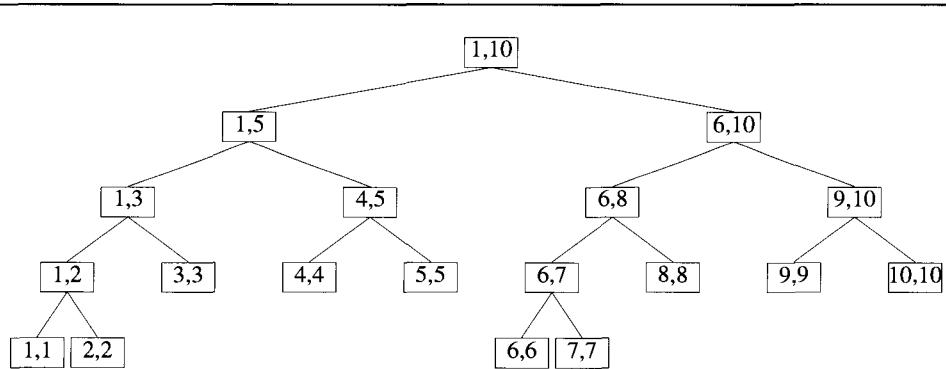


Figure 3.3 Tree of calls of $\text{MergeSort}(1, 10)$

Figure 3.3 is a tree that represents the sequence of recursive calls that are produced by MergeSort when it is applied to ten elements. The pair of values in each node are the values of the parameters *low* and *high*. Notice how the splitting continues until sets containing a single element are produced. Figure 3.4 is a tree representing the calls to procedure Merge by MergeSort . For example, the node containing 1, 2, and 3 represents the merging of $a[1 : 2]$ with $a[3]$. \square

If the time for the merging operation is proportional to n , then the computing time for merge sort is described by the recurrence relation

$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned}
 T(n) &= 2(2T(n/4) + cn/2) + cn \\
 &= 4T(n/4) + 2cn \\
 &= 4(2T(n/8) + cn/4) + 2cn \\
 &\vdots \\
 &= 2^k T(1) + kcn \\
 &= an + cn \log n
 \end{aligned}$$

It is easy to see that if $2^k < n \leq 2^{k+1}$, then $T(n) \leq T(2^{k+1})$. Therefore

$$T(n) = O(n \log n)$$

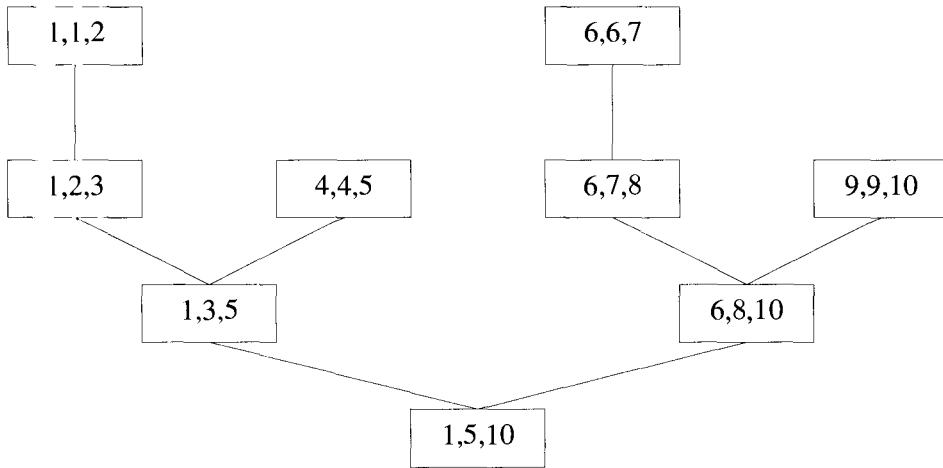


Figure 3.4 Tree of calls of Merge

Though Algorithm 3.7 nicely captures the divide-and-conquer nature of merge sort, there remain several inefficiencies that can and should be eliminated. We present these refinements in an attempt to produce a version of merge sort that is good enough to execute. Despite these improvements the algorithm's complexity remains $O(n \log n)$. We see in Chapter 10 that no sorting algorithm based on comparisons of entire keys can do better.

One complaint we might raise concerning merge sort is its use of $2n$ locations. The additional n locations were needed because we couldn't reasonably merge two sorted sets in place. But despite the use of this space the

algorithm must still work hard and copy the result placed into $b[low : high]$ back into $a[low : high]$ on each call of `Merge`. An alternative to this copying is to associate a new field of information with each key. (The elements in $a[]$ are called *keys*.) This field is used to link the keys and any associated information together in a sorted list (keys and related information are called *records*). Then the merging of the sorted lists proceeds by changing the link values, and no records need be moved at all. A field that contains only a link will generally be smaller than an entire record, so less space will be used.

Along with the original array $a[]$, we define an auxiliary array $link[1 : n]$ that contains integers in the range $[0, n]$. These integers are interpreted as pointers to elements of $a[]$. A list is a sequence of pointers ending with a zero. Below is one set of values for $link$ that contains two lists: Q and R . The integer $Q = 2$ denotes the start of one list and $R = 5$ the start of the other.

<i>link:</i>	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	6	4	7	1	3	0	8	0

The two lists are $Q = (2, 4, 1, 6)$ and $R = (5, 3, 7, 8)$. Interpreting these lists as describing sorted subsets of $a[1 : 8]$, we conclude that $a[2] \leq a[4] \leq a[1] \leq a[6]$ and $a[5] \leq a[3] \leq a[7] \leq a[8]$.

Another complaint we could raise about `MergeSort` is the stack space that is necessitated by the use of recursion. Since merge sort splits each set into two approximately equal-sized subsets, the maximum depth of the stack is proportional to $\log n$. The need for stack space seems indicated by the top-down manner in which this algorithm was devised. The need for stack space can be eliminated if we build an algorithm that works bottom-up; see the exercises for details.

As can be seen from function `MergeSort` and the previous example, even sets of size two will cause two recursive calls to be made. For small set sizes most of the time will be spent processing the recursion instead of sorting. This situation can be improved by not allowing the recursion to go to the lowest level. In terms of the divide-and-conquer control abstraction, we are suggesting that when `Small` is true for merge sort, more work should be done than simply returning with no action. We use a second sorting algorithm that works well on small-sized sets.

Insertion sort works exceedingly fast on arrays of less than, say, 16 elements, though for large n its computing time is $O(n^2)$. Its basic idea for sorting the items in $a[1 : n]$ is as follows:

```

for  $j := 2$  to  $n$  do {
    place  $a[j]$  in its correct position in the sorted set  $a[1 : j - 1]$ ;
}
```

Though all the elements in $a[1 : j - 1]$ may have to be moved to accommodate $a[j]$, for small values of n the algorithm works well. Algorithm 3.9 has the details.

```

1  Algorithm InsertionSort( $a, n$ )
2  // Sort the array  $a[1 : n]$  into nondecreasing order,  $n \geq 1$ .
3  {
4      for  $j := 2$  to  $n$  do
5      {
6          //  $a[1 : j - 1]$  is already sorted.
7          item :=  $a[j]$ ;  $i := j - 1$ ;
8          while (( $i \geq 1$ ) and (item <  $a[i]$ )) do
9          {
10              $a[i + 1] := a[i]$ ;  $i := i - 1$ ;
11         }
12          $a[i + 1] := item$ ;
13     }
14 }
```

Algorithm 3.9 Insertion sort

The statements within the **while** loop can be executed zero up to a maximum of j times. Since j goes from 2 to n , the worst-case time of this procedure is bounded by

$$\sum_{2 \leq j \leq n} j = n(n + 1)/2 - 1 = \Theta(n^2)$$

Its best-case computing time is $\Theta(n)$ under the assumption that the body of the **while** loop is never entered. This will be true when the data is already in sorted order.

We are now ready to present the revised version of merge sort with the inclusion of insertion sort and the links. Function **MergeSort1** (Algorithm 3.10) is initially invoked by placing the keys of the records to be sorted in $a[1 : n]$ and setting $link[1 : n]$ to zero. Then one says **MergeSort1**($1, n$). A pointer to a list of indices that give the elements of $a[]$ in sorted order is returned. Insertion sort is used whenever the number of items to be sorted is less than 16. The version of insertion sort as given by Algorithm 3.9 needs to be altered so that it sorts $a[low : high]$ into a linked list. Call the altered version **InsertionSort1**. The revised merging function, **Merge1**, is given in Algorithm 3.11.

```

1  Algorithm MergeSort1(low, high)
2  // The global array  $a[low : high]$  is sorted in nondecreasing order
3  // using the auxiliary array  $link[low : high]$ . The values in  $link$ 
4  // represent a list of the indices  $low$  through  $high$  giving  $a[ ]$  in
5  // sorted order. A pointer to the beginning of the list is returned.
6  {
7      if ( $(high - low) < 15$ ) then
8          return InsertionSort1( $a, link, low, high$ );
9      else
10     {
11          $mid := \lfloor (low + high)/2 \rfloor$ ;
12          $q := \text{MergeSort1}(low, mid)$ ;
13          $r := \text{MergeSort1}(mid + 1, high)$ ;
14         return Merge1( $q, r$ );
15     }
16 }
```

Algorithm 3.10 Merge sort using links

Example 3.8 As an aid to understanding this new version of merge sort, suppose we simulate the algorithm as it sorts the eight-element sequence (50, 10, 25, 30, 15, 70, 35, 55). We ignore the fact that less than 16 elements would normally be sorted using `InsertionSort`. The $link$ array is initialized to zero. Table 3.4 shows how the $link$ array changes after each call of `MergeSort1` completes. On each row the value of p points to the list in $link$ that was created by the last completion of `Merge1`. To the right are the subsets of sorted elements that are represented by these lists. For example, in the last row $p = 2$ which begins the list of links 2, 5, 3, 4, 7, 1, 8, and 6; this implies $a[2] \leq a[5] \leq a[3] \leq a[4] \leq a[7] \leq a[1] \leq a[8] \leq a[6]$. \square

EXERCISES

1. Why is it necessary to have the auxiliary array $b[low : high]$ in function `Merge`? Give an example that shows why in-place merging is inefficient.
2. The worst-case time of procedure `MergeSort` is $O(n \log n)$. What is its best-case time? Can we say that the time for `MergeSort` is $\Theta(n \log n)$?
3. A sorting method is said to be *stable* if at the end of the method, identical elements occur in the same order as in the original unsorted

```
1  Algorithm Merge1( $q, r$ )
2  //  $q$  and  $r$  are pointers to lists contained in the global array
3  //  $link[0 : n]$ .  $link[0]$  is introduced only for convenience and need
4  // not be initialized. The lists pointed at by  $q$  and  $r$  are merged
5  // and a pointer to the beginning of the merged list is returned.
6  {
7       $i := q; j := r; k := 0;$ 
8      // The new list starts at  $link[0]$ .
9      while (( $i \neq 0$ ) and ( $j \neq 0$ )) do
10     { // While both lists are nonempty do
11         if ( $a[i] \leq a[j]$ ) then
12             { // Find the smaller key.
13                  $link[k] := i; k := i; i := link[i];$ 
14                 // Add a new key to the list.
15             }
16         else
17             {
18                  $link[k] := j; k := j; j := link[j];$ 
19             }
20     }
21     if ( $i = 0$ ) then  $link[k] := j;$ 
22     else  $link[k] := i;$ 
23     return  $link[0];$ 
24 }
```

Algorithm 3.11 Merging linked lists of sorted elements

<i>a:</i>	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
<i>link:</i>	-	50	10	25	30	15	70	35	55
<i>q r p</i>	0	0	0	0	0	0	0	0	0
1 2 2	2	0	1	0	0	0	0	0	(10, 50)
3 4 3	3	0	1	4	0	0	0	0	(10, 50), (25, 30)
2 3 2	2	0	3	4	1	0	0	0	(10, 25, 30, 50)
5 6 5	5	0	3	4	1	6	0	0	(10, 25, 30, 50), (15, 70)
7 8 7	7	0	3	4	1	6	0	8	0
5 7 5	5	0	3	4	1	7	0	8	(10, 25, 30, 50), (15, 70), (35, 55)
2 5 2	2	8	5	4	7	3	0	1	6
									{(10, 15, 25, 30, 35, 50, 55, 70)}

MergeSort1 applied to $a[1 : 8] = (50, 10, 25, 30, 15, 70, 35, 55)$

Table 3.4 Example of *link* array changes

set. Is merge sort a stable sorting method?

4. Suppose $a[1 : m]$ and $b[1 : n]$ both contain sorted elements in non-decreasing order. Write an algorithm that merges these items into $c[1 : m + n]$. Your algorithm should be shorter than Algorithm 3.8 (Merge) since you can now place a large value in $a[m + 1]$ and $b[n + 1]$.
5. Given a file of n records that are partially sorted as $x_1 \leq x_2 \leq \dots \leq x_m$ and $x_{m+1} \leq \dots \leq x_n$, is it possible to sort the entire file in time $O(n)$ using only a small fixed amount of additional storage?
6. Another way to sort a file of n records is to scan the file, merge consecutive pairs of size one, then merge pairs of size two, and so on. Write an algorithm that carries out this process. Show how your algorithm works on the data set (100, 300, 150, 450, 250, 350, 200, 400, 500).
7. A version of insertion sort is used by Algorithm 3.10 to sort small subarrays. However, its parameters and intent are slightly different from the procedure `InsertionSort` of Algorithm 3.9. Write a version of insertion sort that will work as Algorithm 3.10 expects.
8. The sequences X_1, X_2, \dots, X_ℓ are sorted sequences such that $\sum_{i=1}^{\ell} |X_i| = n$. Show how to merge these ℓ sequences in time $O(n \log \ell)$.

3.5 QUICKSORT

The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort. In merge sort, the file $a[1 : n]$ was divided

at its midpoint into subarrays which were independently sorted and later merged. In quicksort, the division into two subarrays is made so that the sorted subarrays do not need to be merged later. 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 $a[m + 1 : n]$ can 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*.

Function `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 all elements in $a[1 : n]$. The assumption that $a[m]$ is the partition element is merely for convenience; other choices for the partitioning element than the first item in the set are better in practice. The function `Interchange`(a, i, j) exchanges $a[i]$ with $a[j]$.

Example 3.9 As an example of how `Partition` works, consider the following array of nine elements. The function is initially invoked as `Partition`($a, 1, 10$). The ends of the horizontal line indicate those elements which were interchanged to produce the next row. The element $a[1] = 65$ is the partitioning element and it is eventually (in the sixth row) determined to be the fifth smallest element of the set. Notice that the remaining elements are unsorted but partitioned about $a[5] = 65$. \square

(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	<u>75</u>	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	<u>80</u>	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	<u>85</u>	60	80	75	70	$+\infty$	5	6
<u>65</u>	45	50	55	<u>60</u>	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 the function `Partition`, two sets S_1 and S_2 are produced. All elements in S_1 are less than or equal

```

1  Algorithm Partition( $a, m, p$ )
2  // Within  $a[m], a[m + 1], \dots, a[p - 1]$  the elements are
3  // rearranged in such a manner that if initially  $t = a[m]$ ,
4  // then after completion  $a[q] = t$  for some  $q$  between  $m$ 
5  // and  $p - 1$ ,  $a[k] \leq t$  for  $m \leq k < q$ , and  $a[k] \geq t$ 
6  // for  $q < k < p$ .  $q$  is returned. Set  $a[p] = \infty$ .
7  {
8       $v := a[m]; i := m; j := p;$ 
9      repeat
10     {
11         repeat
12              $i := i + 1;$ 
13         until ( $a[i] \geq v$ );
14         repeat
15              $j := j - 1;$ 
16         until ( $a[j] \leq v$ );
17         if ( $i < j$ ) then Interchange( $a, i, j$ );
18     } until ( $i \geq j$ );
19      $a[m] := a[j]; a[j] := v; \text{return } j;$ 
20 }

```



```

1  Algorithm Interchange( $a, i, j$ )
2  // Exchange  $a[i]$  with  $a[j]$ .
3  {
4       $p := a[i];$ 
5       $a[i] := a[j]; a[j] := p;$ 
6  }

```

Algorithm 3.12 Partition the array $a[m : p - 1]$ about $a[m]$

to the elements in S_2 . Hence S_1 and S_2 can be sorted independently. Each set is sorted by reusing the function `Partition`. Algorithm 3.13 describes the complete process.

```

1  Algorithm QuickSort( $p, q$ )
2  // Sorts the elements  $a[p], \dots, a[q]$  which reside in the global
3  // array  $a[1 : n]$  into ascending order;  $a[n + 1]$  is considered to
4  // be defined and must be  $\geq$  all the elements in  $a[1 : n]$ .
5  {
6      if ( $p < q$ ) then // If there are more than one element
7      {
8          // divide  $P$  into two subproblems.
9           $j := \text{Partition}(a, p, q + 1);$ 
10         //  $j$  is the position of the partitioning element.
11         // Solve the subproblems.
12         QuickSort( $p, j - 1$ );
13         QuickSort( $j + 1, q$ );
14         // There is no need for combining solutions.
15     }
16 }
```

Algorithm 3.13 Sorting by partitioning

In analyzing `QuickSort`, we 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: the n elements to be sorted are distinct, and the input distribution is such that the partition element $v = a[m]$ in the call to `Partition(a, m, p)` has an equal probability of being the i th smallest element, $1 \leq i \leq p - m$, in $a[m : p - 1]$.

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 $p - m + 1$. Let r be the total number of elements in all the calls to `Partition` at any level of recursion. At level one only one call, `Partition(a, 1, n+1)`, is made and $r = n$; at level two at most two calls are made and $r = n - 1$; and so on. At each level of recursion, $O(r)$ element comparisons are made by `Partition`. At each level, r is at least one less than the r at the previous level as the partitioning elements of the previous level are eliminated. Hence $C_w(n)$ is the sum on r as r varies from 2 to n , or $O(n^2)$. Exercise 7 examines input data on which `QuickSort` uses $\Omega(n^2)$ comparisons.

The average value $C_A(n)$ of $C(n)$ is much less than $C_w(n)$. Under the assumptions made earlier, the partitioning element v has an equal probability

of being the i th-smallest element, $1 \leq i \leq p - m$, in $a[m : p - 1]$. Hence the two subarrays remaining to be sorted are $a[m : j]$ and $a[j + 1 : p - 1]$ with probability $1/(p - m)$, $m \leq j < p$. From this we obtain the recurrence

$$C_A(n) = n + 1 + \frac{1}{n} \sum_{1 \leq k \leq n} [C_A(k - 1)) + C_A(n - k)] \quad (3.5)$$

The number of element comparisons required by `Partition` on its first call is $n + 1$. Note that $C_A(0) = C_A(1) = 0$. Multiplying both sides of (3.5) by n , we obtain

$$nC_A(n) = n(n + 1) + 2[C_A(0) + C_A(1) + \cdots + C_A(n - 1)] \quad (3.6)$$

Replacing n by $n - 1$ in (3.6) gives

$$(n - 1)C_A(n - 1) = n(n - 1) + 2[C_A(0) + \cdots + C_A(n - 2)]$$

Subtracting this from (3.6), we get

$$nC_A(n) - (n - 1)C_A(n - 1) = 2n + 2C_A(n - 1)$$

or

$$C_A(n)/(n + 1) = C_A(n - 1)/n + 2/(n + 1)$$

Repeatedly using this equation to substitute for $C_A(n - 1)$, $C_A(n - 2)$, \dots , we get

$$\begin{aligned} \frac{C_A(n)}{n+1} &= \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &= \frac{C_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &\vdots \\ &= \frac{C_A(1)}{2} + 2 \sum_{3 \leq k \leq n+1} \frac{1}{k} \\ &= 2 \sum_{3 \leq k \leq n+1} \frac{1}{k} \end{aligned} \quad (3.7)$$

Since

$$\sum_{3 \leq k \leq n+1} \frac{1}{k} \leq \int_2^{n+1} \frac{1}{x} dx = \log_e(n + 1) - \log_e 2$$

(3.7) yields

$$C_A(n) \leq 2(n+1)[\log_e(n+2) - \log_e 2] = O(n \log n)$$

Even though the worst-case time is $O(n^2)$, the average time is only $O(n \log n)$. Let us now look at the stack space needed by the recursion. In the worst case the maximum depth of recursion may be $n - 1$. This happens, for example, when the partition element on each call to Partition is the smallest value in $a[m : p - 1]$. The amount of stack space needed can be reduced to $O(\log n)$ by using an iterative version of quicksort in which the smaller of the two subarrays $a[p : j - 1]$ and $a[j + 1 : q]$ is always sorted first. Also, the second recursive call can be replaced by some assignment statements and a jump to the beginning of the algorithm. With these changes, QuickSort takes the form of Algorithm 3.14.

We can now verify that the maximum stack space needed is $O(\log n)$. Let $S(n)$ be the maximum stack space needed. Then it follows that

$$S(n) \leq \begin{cases} 2 + S(\lfloor (n-1)/2 \rfloor) & n > 1 \\ 0 & n \leq 1 \end{cases}$$

which is less than $2 \log n$.

As remarked in Section 3.4, InsertionSort is exceedingly fast for n less than about 16. Hence InsertionSort can be used to speed up QuickSort2 whenever $q - p < 16$. The exercises explore various possibilities for selection of the partition element.

3.5.1 Performance Measurement

QuickSort and MergeSort were evaluated on a SUN workstation 10/30. In both cases the recursive versions were used. For QuickSort the Partition function was altered to carry out the median of three rule (i.e. the partitioning element was the median of $a[m]$, $a[\lfloor (m+p-1)/2 \rfloor]$ and $a[p-1]$). Each data set consisted of random integers in the range (0, 1000). Tables 3.5 and 3.6 record the actual computing times in milliseconds. Table 3.5 displays the average computing times. For each n , 50 random data sets were used. Table 3.6 shows the worst-case computing times for the 50 data sets.

Scanning the tables, we immediately see that QuickSort is faster than MergeSort for all values. Even though both algorithms require $O(n \log n)$ time on the average, QuickSort usually performs well in practice. The exercises discuss other tests that would make useful comparisons.

3.5.2 Randomized Sorting Algorithms

Though algorithm QuickSort has an average time of $O(n \log n)$ on n elements, its worst-case time is $O(n^2)$. On the other hand it does not make use of any

```

1  Algorithm QuickSort2( $p, q$ )
2  // Sorts the elements in  $a[p : q]$ .
3  {
4      // stack is a stack of size  $2 \log(n)$ .
5      repeat
6      {
7          while ( $p < q$ ) do
8          {
9               $j := \text{Partition}(a, p, q + 1);$ 
10             if ( $(j - p) < (q - j)$ ) then
11                 {
12                     Add( $j + 1$ ); // Add  $j + 1$  to stack.
13                     Add( $q$ );  $q := j - 1$ ; // Add  $q$  to stack
14                 }
15             else
16                 {
17                     Add( $p$ ); // Add  $p$  to stack.
18                     Add( $j - 1$ );  $p := j + 1$ ; // Add  $j - 1$  to stack
19                 }
20             } // Sort the smaller subfile.
21             if stack is empty then return;
22             Delete( $q$ ); Delete( $p$ ); // Delete  $q$  and  $p$  from stack.
23         } until (false);
24     }

```

Algorithm 3.14 Iterative version of QuickSort

additional memory as does MergeSort. A possible input on which QuickSort displays worst-case behavior is one in which the elements are already in sorted order. In this case the partition will be such that there will be only one element in one part and the rest of the elements will fall in the other part. The performance of any divide-and-conquer algorithm will be good if the resultant subproblems are as evenly sized as possible. Can QuickSort be modified so that it performs well on every input? The answer is yes. Is the technique of using the median of the three elements $a[p]$, $a[\lfloor (q+p)/2 \rfloor]$, and $a[q]$ the solution? Unfortunately it is possible to construct inputs for which even this method will take $\Omega(n^2)$ time, as is explored in the exercises.

The solution is the use of a randomizer. While sorting the array $a[p : q]$, instead of picking $a[m]$, pick a random element (from among $a[p], \dots, a[q]$) as the partition element. The resultant randomized algorithm (RQuickSort)

n	1000	2000	3000	4000	5000
MergeSort	72.8	167.2	275.1	378.5	500.6
QuickSort	36.6	85.1	138.9	205.7	269.0
n	6000	7000	8000	9000	10000
MergeSort	607.6	723.4	811.5	949.2	1073.6
QuickSort	339.4	411.0	487.7	556.3	645.2

Table 3.5 Average computing times for two sorting algorithms on random inputs

n	1000	2000	3000	4000	5000
MergeSort	105.7	206.4	335.2	422.1	589.9
QuickSort	41.6	97.1	158.6	244.9	397.8
n	6000	7000	8000	9000	10000
MergeSort	691.3	794.8	889.5	1067.2	1167.6
QuickSort	383.8	497.3	569.9	616.2	738.1

Table 3.6 Worst-case computing times for two sorting algorithms on random inputs

works on any input and runs in an expected $O(n \log n)$ time, where the expectation is over the space of all possible outcomes for the randomizer (rather than the space of all possible inputs). The code for RQuickSort is given in Algorithm 3.15. Note that this is a Las Vegas algorithm since it will always output the correct answer. Every call to the randomizer Random takes a certain amount of time. If there are only a very few elements to sort, the time taken by the randomizer may be comparable to the rest of the computation. For this reason, we invoke the randomizer only if $(q - p) > 5$. But 5 is not a magic number; in the machine employed, this seems to give the best results. In general this number should be determined empirically.

```

1  Algorithm RQuickSort( $p, q$ )
2  // Sorts the elements  $a[p], \dots, a[q]$  which reside in the global
3  // array  $a[1 : n]$  into ascending order.  $a[n + 1]$  is considered to
4  // be defined and must be  $\geq$  all the elements in  $a[1 : n]$ .
5  {
6      if ( $p < q$ ) then
7          {
8              if  $((q - p) > 5)$  then
9                  Interchange( $a$ , Random() mod  $(q - p + 1) + p, p$ );
10                  $j := \text{Partition}(a, p, q + 1);$ 
11                 //  $j$  is the position of the partitioning element.
12                 RQuickSort( $p, j - 1$ );
13                 RQuickSort( $j + 1, q$ );
14             }
15     }

```

Algorithm 3.15 Randomized quick sort algorithm

The proof of the fact that RQuickSort has an expected $O(n \log n)$ time is the same as the proof of the average time of QuickSort. Let $A(n)$ be the average time of RQuickSort on *any input* of n elements. Then the number of elements in the second part will be $0, 1, 2, \dots, n - 2$, or $n - 1$, all with an equal probability of $\frac{1}{n}$ (in the probability space of outcomes for the randomizer). Thus the recurrence relation for $A(n)$ will be

$$A(n) = \frac{1}{n} \sum_{1 \leq k \leq n} (A(k - 1) + A(n - k)) + n + 1$$

This is the same as Equation 3.4, and hence its solution is $O(n \log n)$.

RQuickSort and QuickSort (without employing the median of three elements rule) were evaluated on a SUN 10/30 workstation. Table 3.7 displays

the times for the two algorithms in milliseconds averaged over 100 runs. For each n , the input considered was the sequence of numbers $1, 2, \dots, n$. As we can see from the table, RQuickSort performs much better than QuickSort. Note that the times shown in this table for QuickSort are much more than the corresponding entries in Tables 3.5 and 3.6. The reason is that QuickSort makes $\Theta(n^2)$ comparisons on inputs that are already in sorted order. However, on random inputs its average performance is very good.

n	1000	2000	3000	4000	5000
QuickSort	195.5	759.2	1728	3165	4829
RQuickSort	9.4	21.0	30.5	41.6	52.8

Table 3.7 Comparison of QuickSort and RQuickSort on the input $a[i] = i$, $1 \leq i \leq n$; times are in milliseconds.

The performance of RQuickSort can be improved in various ways. For example, we could pick a small number (say 11) of the elements in the array $a[]$ randomly and use the median of these elements as the partition element. These randomly chosen elements form a random sample of the array elements. We would expect that the median of the sample would also be an approximate median of the array and hence result in an approximately even partitioning of the array.

An even more generalized version of the above random sampling technique is shown in Algorithm 3.16. Here we choose a random sample S of s elements (where s is a function of n) from the input sequence X and sort them using HeapSort, MergeSort, or any other sorting algorithm. Let $\ell_1, \ell_2, \dots, \ell_s$ be the sorted sample. We partition X into $s+1$ parts using the sorted sample as partition keys. In particular $X_1 = \{x \in X | x \leq \ell_1\}$; $X_i = \{x \in X | \ell_{i-1} < x \leq \ell_i\}$, for $i = 2, 3, \dots, s$; and $X_{s+1} = \{x \in X | x > \ell_s\}$. After having partitioned X into $s+1$ parts, we sort each part recursively. For a proper choice of s , the number of comparisons made in this algorithm is only $n \log n + \tilde{o}(n \log n)$. Note the constant 1 before $n \log n$. We see in Chapter 10 that this number is very close to the information theoretic lower bound for sorting.

Choose $s = \frac{n}{\log^2 n}$. The sample can be sorted in $O(s \log s) = O(\frac{n}{\log n})$ time and comparisons if we use HeapSort or MergeSort. If we store the sorted sample elements in an array, say $b[]$, for each $x \in X$, we can determine which part X_i it belongs to in $\leq \log n$ comparisons using binary search on $b[]$. Thus the partitioning process takes $n \log n + O(n)$ comparisons. In the exercises you are asked to show that with high probability the cardinality

```

1   Algorithm RSort( $a, n$ )
2   // Sort the elements  $a[1 : n]$ .
3   {
4       Randomly sample  $s$  elements from  $a[ ]$ ;
5       Sort this sample;
6       Partition the input using the sorted sample as partition keys;
7       Sort each part separately;
8   }

```

Algorithm 3.16 A randomized algorithm for sorting

of each X_i is no more than $\tilde{O}(\frac{n}{s} \log n) = \tilde{O}(\log^3 n)$. Using **HeapSort** or **MergeSort** to sort each of the X_i 's (without employing recursion on any of them), the total cost of sorting the X_i 's is

$$\sum_{i=1}^{s+1} O(|X_i| \log |X_i|) = \max_{1 \leq i \leq s+1} \{\log |X_i|\} \sum_{i=1}^{s+1} O(|X_i|)$$

Since each $|X_i|$ is $\tilde{O}(\log^3 n)$, the cost of sorting the $s+1$ parts is $\tilde{O}(n \log \log n) = \tilde{o}(n \log n)$. In summary, the number of comparisons made in this randomized sorting algorithm is $n \log n + \tilde{o}(n \log n)$.

EXERCISES

1. Show how **QuickSort** sorts the following sequences of keys: 1, 1, 1, 1, 1, 1 and 5, 5, 8, 3, 4, 3, 2.
2. **QuickSort** is not a stable sorting algorithm. However, if the key in $a[i]$ is changed to $a[i] * n + i - 1$, then the new keys are all distinct. After sorting, which transformation will restore the keys to their original values?
3. In the function **Partition**, Algorithm 3.12, discuss the merits or demerits of altering the statement **if** ($i < j$) to **if** ($i \leq j$). Simulate both algorithms on the data set (5, 4, 3, 2, 5, 8, 9) to see the difference in how they work.
4. Function **QuickSort** uses the output of function **Partition**, which returns the position where the partition element is placed. If equal keys are present, then two elements can be properly placed instead of one. Show

how you might change Partition so that QuickSort can take advantage of this situation.

5. In addition to Partition, there are many other ways to partition a set. Consider modifying Partition so that i is incremented while $a[i] \leq v$ instead of $a[i] < v$. Rewrite Partition making all of the necessary changes to it and then compare the new version with the original.
6. Compare the sorting methods MergeSort1 and QuickSort2 (Algorithm 3.10 and 3.14, respectively). Devise data sets that compare both the average- and worst-case times for these two algorithms.
7. (a) On which input data does the algorithm QuickSort exhibit its worst-case behavior?
 (b) Answer part (a) for the case in which the partitioning element is selected according to the median of three rule.
8. With MergeSort we included insertion sorting to eliminate the book-keeping for small merges. How would you use this technique to improve QuickSort?
9. Take the iterative versions of MergeSort and QuickSort and compare them for the same-size data sets as used in Section 3.5.1.
10. Let S be a sample of s elements from X . If X is partitioned into $s + 1$ parts as in Algorithm 3.16, show that the size of each part is $\tilde{\mathcal{O}}\left(\frac{n}{s} \log n\right)$.

3.6 SELECTION

The Partition algorithm of Section 3.5 can also be used to obtain an efficient solution for the selection problem. In this problem, we are given n elements $a[1 : n]$ and are required to determine the k th-smallest element. If the partitioning element v is positioned at $a[j]$, then $j - 1$ elements are less than or equal to $a[j]$ and $n - j$ elements are greater than or equal to $a[j]$. Hence if $k < j$, then the k th-smallest element is in $a[1 : j - 1]$; if $k = j$, then $a[j]$ is the k th-smallest element; and if $k > j$, then the k th-smallest element is the $(k - j)$ th-smallest element in $a[j + 1 : n]$. The resulting algorithm is function Select1 (Algorithm 3.17). This function places the k th-smallest element into position $a[k]$ and partitions the remaining elements so that $a[i] \leq a[k]$, $1 \leq i < k$, and $a[i] \geq a[k]$, $k < i \leq n$.

Example 3.10 Let us simulate Select1 as it operates on the same array used to test Partition in Section 3.5. The array has the nine elements 65, 70,

```

1   Algorithm Select1( $a, n, k$ )
2   // Selects the  $k$ th-smallest element in  $a[1 : n]$  and places it
3   // in the  $k$ th position of  $a[ ]$ . The remaining elements are
4   // rearranged such that  $a[m] \leq a[k]$  for  $1 \leq m < k$ , and
5   //  $a[m] \geq a[k]$  for  $k < m \leq n$ .
6   {
7       low := 1; up :=  $n + 1$ ;
8        $a[n + 1] := \infty$ ; //  $a[n + 1]$  is set to infinity.
9       repeat
10      {
11          // Each time the loop is entered,
12          //  $1 \leq low \leq k \leq up \leq n + 1$ .
13           $j := \text{Partition}(a, low, up)$ ;
14          //  $j$  is such that  $a[j]$  is the  $j$ th-smallest value in  $a[ ]$ .
15          if ( $k = j$ ) then return;
16          else if ( $k < j$ ) then up :=  $j$ ; //  $j$  is the new upper limit.
17          else low :=  $j + 1$ ; //  $j + 1$  is the new lower limit.
18      } until (false);
19  }

```

Algorithm 3.17 Finding the k th-smallest element

75, 80, 85, 60, 55, 50, and 45, with $a[10] = \infty$. If $k = 5$, then the first call of Partition will be sufficient since 65 is placed into $a[5]$. Instead, assume that we are looking for the seventh-smallest element of a , that is, $k = 7$. The next invocation of Partition is Partition(6, 10).

$$\begin{array}{ccccccc}
a: & (5) & (6) & (7) & (8) & (9) & (10) \\
& 65 & \underline{85} & 80 & 75 & 70 & +\infty
\end{array}$$

$$\begin{array}{ccccccc}
& 65 & 70 & 80 & 75 & 85 & +\infty
\end{array}$$

This last call of Partition has uncovered the ninth-smallest element of a . The next invocation is Partition(6, 9).

$$\begin{array}{ccccccc}
a: & (5) & (6) & (7) & (8) & (9) & (10) \\
& 65 & \underline{70} & 80 & 75 & 85 & +\infty
\end{array}$$

$$\begin{array}{ccccccc}
& 65 & 70 & 80 & 75 & 85 & +\infty
\end{array}$$

This time, the sixth element has been found. Since $k \neq j$, another call to Partition is made, Partition(7, 9).

$a:$	(5)	(6)	(7)	(8)	(9)	(10)
	65	70	80	75	85	$+\infty$
	65	70	75	80	85	$+\infty$

Now 80 is the partition value and is correctly placed at $a[8]$. However, `Select1` has still not found the seventh-smallest element. It needs one more call to `Partition`, which is `Partition(7, 8)`. This performs only an interchange between $a[7]$ and $a[8]$ and returns, having found the correct value. \square

In analyzing `Select1`, we make the same assumptions that were made for `QuickSort`:

1. The n elements are distinct.
2. The input distribution is such that the partition element can be the i th-smallest element of $a[m : p - 1]$ with an equal probability for each i , $1 \leq i \leq p - m$.

`Partition` requires $O(p - m)$ time. On each successive call to `Partition`, either m increases by at least one or j decreases by at least one. Initially $m = 1$ and $j = n + 1$. Hence, at most n calls to `Partition` can be made. Thus, the worst-case complexity of `Select1` is $O(n^2)$. The time is $\Omega(n^2)$, for example, when the input $a[1 : n]$ is such that the partitioning element on the i th call to `Partition` is the i th-smallest element and $k = n$. In this case, m increases by one following each call to `Partition` and j remains unchanged. Hence, n calls are made for a total cost of $O(\sum_1^n i) = O(n^2)$. The average computing time of `Select1` is, however, only $O(n)$. Before proving this fact, we specify more precisely what we mean by the average time.

Let $T_A^k(n)$ be the average time to find the k th-smallest element in $a[1 : n]$. This average is taken over all $n!$ different permutations of n distinct elements. Now define $T_A(n)$ and $R(n)$ as follows:

$$T_A(n) = \frac{1}{n} \sum_{1 \leq k \leq n} T_A^k(n)$$

and

$$R(n) = \max_k \{T_A^k(n)\}$$

$T_A(n)$ is the average computing time of `Select1`. It is easy to see that $T_A(n) \leq R(n)$. We are now ready to show that $T_A(n) = O(n)$.

Theorem 3.3 The average computing time $T_A(n)$ of `Select1` is $O(n)$.

Proof: On the first call to Partition, the partitioning element v is the i th-smallest element with probability $\frac{1}{n}, 1 \leq i \leq n$ (this follows from the assumption on the input distribution). The time required by Partition and the if statement in Select1 is $O(n)$. Hence, there is a constant $c, c > 0$, such that

$$\begin{aligned} T_A^k(n) &\leq cn + \frac{1}{n} \left[\sum_{1 \leq i < k} T_A^{k-1}(n-i) + \sum_{k < i \leq n} T_A^k(i-1) \right], \quad n \geq 2 \\ \text{So, } R(n) &\leq cn + \frac{1}{n} \max_k \left\{ \sum_{1 \leq i < k} R(n-i) + \sum_{k < i \leq n} R(i-1) \right\} \\ R(n) &\leq cn + \frac{1}{n} \max_k \left\{ \sum_{n-k+1}^{n-1} R(i) + \sum_k^{n-1} R(i) \right\}, \quad n \geq 2 \end{aligned} \quad (3.8)$$

We assume that c is chosen such that $R(1) \leq c$ and show, by induction on n , that $R(n) \leq 4cn$.

Induction Base: For $n = 2$, (3.8) gives

$$\begin{aligned} R(n) &\leq 2c + \frac{1}{2} \max \{R(1), R(1)\} \\ &\leq 2.5c < 4cn \end{aligned}$$

Induction Hypothesis: Assume $R(n) \leq 4cn$ for all $n, 2 \leq n < m$.

Induction Step: For $n = m$, (3.8) gives

$$R(m) \leq cm + \frac{1}{m} \max_k \left\{ \sum_{m-k+1}^{m-1} R(i) + \sum_k^{m-1} R(i) \right\}$$

Since we know that $R(n)$ is a nondecreasing function of n , it follows that

$$\sum_{m-k+1}^{m-1} R(i) + \sum_k^{m-1} R(i)$$

is maximized if $k = \frac{m}{2}$ when m is even and $k = \frac{m+1}{2}$ when m is odd. Thus, if m is even, we obtain

$$R(m) \leq cm + \frac{2}{m} \sum_{m/2}^{m-1} R(i)$$

$$\begin{aligned}
&\leq cm + \frac{8c}{m} \sum_{i=m/2}^{m-1} i \\
&< 4cm \\
\text{If } m \text{ is odd, } R(m) &\leq cm + \frac{2}{m} \sum_{i=(m+1)/2}^{m-1} R(i) \\
&\leq cm + \frac{8c}{m} \sum_{i=(m+1)/2}^{m-1} \\
&< 4cm
\end{aligned}$$

Since $T_A(n) \leq R(n)$, it follows that $T_A(n) \leq 4cn$, and so $T_A(n)$ is $O(n)$. \square

The space needed by `Select1` is $O(1)$.

Algorithm 3.15 is a randomized version of QuickSort in which the partition element is chosen from the array elements randomly with equal probability. The same technique can be applied to `Select1` and the partition element can be chosen to be a random array element. The resulting randomized Las Vegas algorithm (call it `RSelect`) has an expected time of $O(n)$ (where the expectation is over the space of randomizer outputs) on *any input*. The proof of this expected time is the same as in Theorem 3.3.

3.6.1 A Worst-Case Optimal Algorithm

By choosing the partitioning element v more carefully, we can obtain a selection algorithm with worst-case complexity $O(n)$. To obtain such an algorithm, v must be chosen so that at least some fraction of the elements is smaller than v and at least some (other) fraction of elements is greater than v . Such a selection of v can be made using the median of medians (*mm*) rule. In this rule the n elements are divided into $\lfloor n/r \rfloor$ groups of r elements each (for some $r, r > 1$). The remaining $n - r \lfloor n/r \rfloor$ elements are not used. The median m_i of each of these $\lfloor n/r \rfloor$ groups is found. Then, the median mm of the m_i 's, $1 \leq i \leq \lfloor n/r \rfloor$, is found. The median mm is used as the partitioning element. Figure 3.5 illustrates the m_i 's and mm when $n = 35$ and $r = 7$. The five groups of elements are $B_i, 1 \leq i \leq 5$. The seven elements in each group have been arranged into nondecreasing order down the column. The middle elements are the m_i 's. The columns have been arranged in nondecreasing order of m_i . Hence, the m_i corresponding to column 3 is mm .

Since the median of r elements is the $\lceil r/2 \rceil$ th-smallest element, it follows (see Figure 3.5) that at least $\lceil \lfloor n/r \rfloor / 2 \rceil$ of the m_i 's are less than or equal to mm and at least $\lfloor n/r \rfloor - \lceil \lfloor n/r \rfloor / 2 \rceil + 1 \geq \lceil \lfloor n/r \rfloor / 2 \rceil$ of the m_i 's are greater than or equal to mm . Hence, at least $\lceil r/2 \rceil \lceil \lfloor n/r \rfloor / 2 \rceil$ elements are less than

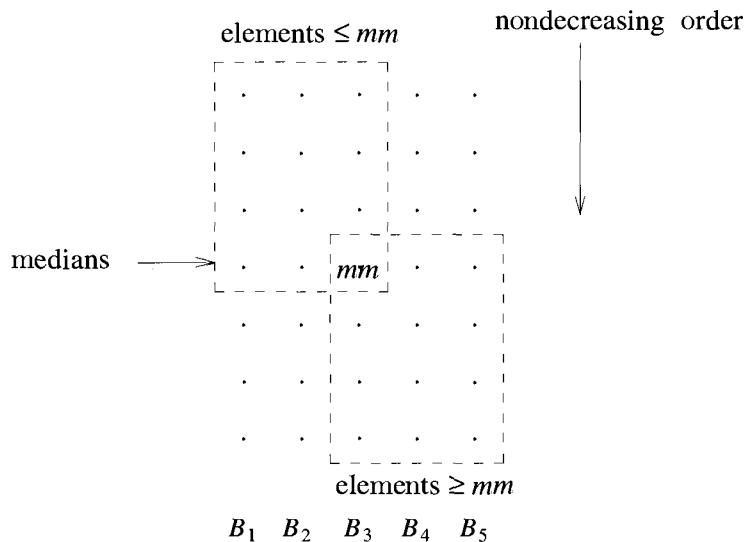


Figure 3.5 The median of medians when $r = 7$, $n = 35$

or equal to (or greater than or equal to) mm . When $r = 5$, this quantity is at least $1.5 \lfloor n/5 \rfloor$. Thus, if we use the median of medians rule with $r = 5$ to select $v = mm$, we are assured that at least $1.5 \lfloor n/5 \rfloor$ elements will be greater than or equal to v . This in turn implies that at most $n - 1.5 \lfloor n/5 \rfloor \leq .7n + 1.2$ elements are less than v . Also, at most $.7n + 1.2$ elements are greater than v . Thus, the median of medians rule satisfies our earlier requirement on v .

The algorithm to select the k th-smallest element uses the median of medians rule to determine a partitioning element. This element is computed by a recursive application of the selection algorithm. A high-level description of the new selection algorithm appears as Select2 (Algorithm 3.18). Select2 can now be analyzed for any given r . First, let us consider the case in which $r = 5$ and all elements in $a[]$ are distinct. Let $T(n)$ be the worst-case time requirement of Select2 when invoked with $up - low + 1 = n$. Lines 4 to 9 and 11 to 12 require at most $O(n)$ time (note that since $r = 5$ is fixed, each $m[i]$ (lines 8 and 9) can be found in $O(1)$ time). The time for line 10 is $T(n/5)$. Let S and R , respectively, denote the elements $a[low : j - 1]$ and $a[j + 1 : up]$. We see that $|S|$ and $|R|$ are at most $.7n + 1.2$, which is no more than $3n/4$ for $n \geq 24$. So, the time for lines 13 to 16 is at most $T(3n/4)$ when $n \geq 24$. Hence, for $n \geq 24$, we obtain

```

1  Algorithm Select2( $a, k, low, up$ )
2  // Find the  $k$ -th smallest in  $a[low : up]$ .
3  {
4       $n := up - low + 1$ ;
5      if ( $n \leq r$ ) then sort  $a[low : up]$  and return the  $k$ -th element;
6      Divide  $a[low : up]$  into  $n/r$  subsets of size  $r$  each;
7      Ignore excess elements;
8      Let  $m[i]$ ,  $1 \leq i \leq (n/r)$  be the set of medians of
9      the above  $n/r$  subsets.
10      $v := \text{Select2}(m, \lceil (n/r)/2 \rceil, 1, n/r)$ ;
11     Partition  $a[low : up]$  using  $v$  as the partition element;
12     Assume that  $v$  is at position  $j$ ;
13     if ( $k = (j - low + 1)$ ) then return  $v$ ;
14     elseif ( $k < (j - low + 1)$ ) then
15         return Select2( $a, k, low, j - 1$ );
16     else return Select2( $a, k - (j - low + 1), j + 1, up$ );
17 }

```

Algorithm 3.18 Selection pseudocode using the median of medians rule

$$T(n) \leq T(n/5) + T(3n/4) + cn \quad (3.9)$$

where c is chosen sufficiently large that

$$T(n) \leq cn \quad \text{for } n \leq 24$$

A proof by induction easily establishes that $T(n) \leq 20cn$ for $n \geq 1$. Algorithm Select2 with $r = 5$ is a linear time algorithm for the selection problem on distinct elements! The exercises examine other values of r that also yield this behavior. Let us now see what happens when the elements of $a[]$ are not all distinct. In this case, following a use of Partition (line 11), the size of S or R may be more than $.7n + 1.2$ as some elements equal to v may appear in both S and R . One way to handle the situation is to partition $a[]$ into three sets U, S , and R such that U contains all elements equal to v , S has all elements smaller than v , and R has the remainder. Lines 11 to 16 become:

Partition $a[]$ into U, S , and R as above.
if ($|S| \geq k$) **then return** Select2($a, k, low, low + |S| - 1$);
else if ($(|S| + |U|) \geq k$) **then return** v ;
else return Select2($a, k - |S| - |U|, low + |S| + |U|, up$);

When this is done, the recurrence (3.9) is still valid as $|S|$ and $|R|$ are $\leq .7n + 1.2$. Hence, the new Select2 will be of linear complexity even when elements are not distinct.

Another way to handle the case of nondistinct elements is to use a different r . To see why a different r is needed, let us analyze Select2 with $r = 5$ and nondistinct elements. Consider the case when $.7n + 1.2$ elements are less than v and the remaining elements are equal to v . An examination of Partition reveals that at most half the remaining elements may be in S . We can verify that this is the worst case. Hence, $|S| \leq .7n + 1.2 + (.3n - 1.2)/2 = .85n + .6$. Similarly, $|R| \leq .85n + .6$. Since, the total number of elements involved in the two recursive calls (in lines 10 and 15 or 16) is now $1.05n + .6 \geq n$, the complexity of Select2 is not $O(n)$. If we try $r = 9$, then at least $2.5 \lfloor n/9 \rfloor$ elements will be less than or equal to v and at least this many will be greater than or equal to v . Hence, the size of S and R will be at most $n - 2.5 \lfloor n/9 \rfloor + 1/2(2.5 \lfloor n/9 \rfloor) = n - 1.25 \lfloor n/9 \rfloor \leq 31/36n + 1.25 \leq 63n/72$ for $n \geq 90$. Hence, we obtain the recurrence

$$T(n) \leq \begin{cases} T(n/9) + T(63n/72) + c_1n & n \geq 90 \\ c_1n & n < 90 \end{cases}$$

where c_1 is a suitable constant. An inductive argument shows that $T(n) \leq 72c_1n$, $n \geq 1$. Other suitable values of r are obtained in the exercises.

As far as the additional space needed by Select2 is concerned, we see that space is needed for the recursion stack. The recursive call from line 15 or 16 is easily eliminated as this call is the last statement executed in Select2. Hence, stack space is needed only for the recursion from line 10. The maximum depth of recursion is $\log n$. The recursion stack should be capable of handling this depth. In addition to this stack space, space is needed only for some simple variables.

3.6.2 Implementation of Select2

Before attempting to write a pseudocode algorithm implementing Select2, we need to decide how the median of a set of size r is to be found and where we are going to store the $\lfloor n/r \rfloor$ medians of lines 8 and 9. Since, we expect to be using a small r (say $r = 5$ or 9), an efficient way to find the median of r elements is to sort them using `InsertionSort(a, i, j)`. This algorithm is a modification of Algorithm 3.9 to sort $a[i : j]$. The median is now the middle element in $a[i : j]$. A convenient place to store these medians is at the front of the array. Thus, if we are finding the k th-smallest element in $a[low : up]$, then the elements can be rearranged so that the medians are $a[low], a[low+1], a[low+2]$, and so on. This makes it easy to implement line 10 as a selection on consecutive elements of $a[]$. Function Select2 (Algorithm 3.19) results from the above discussion and the replacement of the recursive calls of lines 15 and 16 by equivalent code to restart the algorithm.

```

1  Algorithm Select2( $a, k, low, up$ )
2  // Return  $i$  such that  $a[i]$  is the  $k$ th-smallest element in
3  //  $a[low : up]$ ;  $r$  is a global variable as described in the text.
4  {
5      repeat
6      {
7           $n := up - low + 1$ ; // Number of elements
8          if ( $n \leq r$ ) then
9          {
10             InsertionSort( $a, low, up$ );
11             return  $low + k - 1$ ;
12         }
13         for  $i := 1$  to  $\lfloor n/r \rfloor$  do
14         {
15             InsertionSort( $a, low + (i - 1) * r, low + i * r - 1$ );
16             // Collect medians in the front part of  $a[low : up]$ .
17             Interchange( $a, low + i - 1,$ 
18                          $low + (i - 1) * r + \lceil r/2 \rceil - 1$ );
19         }
20          $j := Select2(a, \lceil \lfloor n/r \rfloor / 2 \rceil, low, low + \lfloor n/r \rfloor - 1)$ ; // mm
21         Interchange( $a, low, j$ );
22          $j := Partition(a, low, up + 1)$ ;
23         if ( $k = (j - low + 1)$ ) then return  $j$ ;
24         else if ( $k < (j - low + 1)$ ) then  $up := j - 1$ ;
25         else
26         {
27              $k := k - (j - low + 1)$ ;  $low := j + 1$ ;
28         }
29     } until ( $false$ );
30 }

```

Algorithm 3.19 Algorithm Select2

An alternative to moving the medians to the front of the array $a[low : up]$ (as in the `Interchange` statement within the `for` loop) is to delete this statement and use the fact that the medians are located at $low + (i - 1)r + \lceil r/2 \rceil - 1, 1 \leq i \leq \lfloor n/r \rfloor$. Hence, `Select2`, `Partition`, and `InsertionSort` need to be rewritten to work on arrays for which the interelement distance is $b, b \geq 1$. At the start of the algorithm, all elements are a distance of one apart, i.e., $a[1], a[2], \dots, a[n]$. On the first call of `Select2` we wish to use only elements that are r apart starting with $a[\lceil r/2 \rceil]$. At the next level of recursion, the elements will be r^2 apart and so on. This idea is developed further in the exercises. We refer to arrays with an interelement distance of b as *b-spaced arrays*.

Algorithms `Select1` (Algorithm 3.17) and `Select2` (Algorithm 3.19) were implemented and run on a SUN Sparcstation 10/30. Table 3.8 summarizes the experimental results obtained. Times shown are in milliseconds. These algorithms were tested on random integers in the range $[0, 1000]$ and the average execution times (over 500 input sets) were computed. `Select1` outperforms `Select2` on random inputs. But if the input is already sorted (or nearly sorted), `Select2` can be expected to be superior to `Select1`.

n	1,000	2,000	3,000	4,000	5,000
Select1	7.42	23.50	30.44	39.24	52.36
Select2	49.54	104.02	174.54	233.56	288.64
n	6,000	7,000	8,000	9,000	10,000
Select1	70.88	83.14	95.00	101.32	111.92
Select2	341.34	414.06	476.98	532.30	604.40

Table 3.8 Comparison of `Select1` and `Select2` on random inputs

EXERCISES

1. Rewrite `Select2`, `Partition`, and `InsertionSort` using *b-spaced arrays*.
2. (a) Assume that `Select2` is to be used only when all elements in a are distinct. Which of the following values of r guarantee $O(n)$ worst-case performance: $r = 3, 5, 7, 9$, and 11 ? Prove your answers.
- (b) Do you expect the computing time of `Select2` to increase or decrease if a larger (but still eligible) choice for r is made? Why?

3. Do Exercise 2 for the case in which a is not restricted to distinct elements. Let $r = 7, 9, 11, 13$, and 15 in part (a).
4. Section 3.6 describes an alternative way to handle the situation when $a[]$ is not restricted to distinct elements. Using the partitioning element v , $a[]$ is divided into three subsets. Write algorithms corresponding to **Select1** and **Select2** using this idea. Using your new version of **Select2** show that the worst-case computing time is $O(n)$ even when $r = 5$.
5. Determine optimal r values for worst-case and average performances of function **Select2**.
6. [Shamos] Let $x[1 : n]$ and $y[1 : n]$ contain two sets of integers, each sorted in nondecreasing order. Write an algorithm that finds the median of the $2n$ combined elements. What is the time complexity of your algorithm? (*Hint:* Use binary search.)
7. Let S be a (not necessarily sorted) sequence of n keys. A key k in S is said to be an *approximate median* of S if $|\{k' \in S : k' < k\}| \geq \frac{n}{4}$ and $|\{k' \in S : k' > k\}| \geq \frac{n}{4}$. Devise an $O(n)$ time algorithm to find all the approximate medians of S .
8. Input are a sequence S of n distinct keys, not necessarily in sorted order, and two integers m_1 and m_2 ($1 \leq m_1, m_2 \leq n$). For any x in S , we define the *rank* of x in S to be $|\{k \in S : k \leq x\}|$. Show how to output all the keys of S whose ranks fall in the interval $[m_1, m_2]$ in $O(n)$ time.
9. The k th *quantiles* of an n -element set are the $k - 1$ elements from the set that divide the sorted set into k equal-sized sets. Give an $O(n \log k)$ time algorithm to list the k th quantiles of a set.
10. Input is a (not necessarily sorted) sequence $S = k_1, k_2, \dots, k_n$ of n arbitrary numbers. Consider the collection C of n^2 numbers of the form $\min\{k_i, k_j\}$, for $1 \leq i, j \leq n$. Present an $O(n)$ -time and $O(n)$ -space algorithm to find the median of C .
11. Given two vectors $X = (x_1, \dots, x_n)$ and $Y = (y_1, \dots, y_n)$, $X < Y$ if there exists an i , $1 \leq i \leq n$, such that $x_j = y_j$ for $1 \leq j < i$ and $x_i < y_i$. Given m vectors each of size n , write an algorithm that determines the minimum vector. Analyze the time complexity of your algorithm.
12. Present an $O(1)$ time Monte Carlo algorithm to find the median of an array of n numbers. The answer output should be correct with probability $\geq \frac{1}{n}$.

13. Input is an array $a[]$ of n numbers. Present an $O(\log n)$ time Monte Carlo algorithm to output any member of $a[]$ that is greater than or equal to the median. The answer should be correct with high probability. Provide a probability analysis.
14. Given a set X of n numbers, how will you find an element of X whose rank in X is at most $\frac{n}{f(n)}$, using a Monte Carlo algorithm? Your algorithm should run in time $O(f(n) \log n)$. Prove that the output will be correct with high probability.
15. In addition to Select1 and Select2, we can think of at least two more selection algorithms. The first of these is very straightforward and appears as Algorithm 3.20 (Algorithm Select3). The time complexity of Select3 is

$$O(n \min \{k, n - k + 1\})$$

Hence, it is very fast for values of k close to 1 or close to n . In the worst case, its complexity is $O(n^2)$. Its average complexity is also $O(n^2)$.

Another selection algorithm proceeds by first sorting the n elements into nondecreasing order and then picking out the k th element. A complete sort can be avoided by using a minheap. Now, only k elements need to be removed from the heap. The time to set up the heap is $O(n)$. An additional $O(k \log n)$ time is needed to make k deletions. The total complexity is $O(n + k \log n)$. This basic algorithm can be improved further by using a maxheap when $k > n/2$ and deleting $n - k + 1$ elements. The complexity is now $O(n + \log n \min \{k, n - k + 1\})$. Call the resulting algorithm Select4. Now that we have four plausible selection algorithms, we would like to know which is best. On the basis of the asymptotic analyses of the four selection algorithms, we can make the following qualitative statements about our expectations on the relative performance of the four algorithms.

- Because of the overhead involved in Select1, Select2, and Select4 and the relative simplicity of Select3, Select3 will be fastest both on the average and in the worst case for small values of n . It will also be fastest for large n and very small or very large k , for example, $k = 1, 2, n$, or $n - 1$.
- For larger values of n , Select1 will have the best behavior on the average.
- As far as worst-case behavior is concerned, Select2 will out-perform the others when n is suitably large. However, there will probably be a range of n for which Select4 will be faster than both Select2 and Select3. We expect this because of the relatively large

```

1  Algorithm Select3( $a, n, k$ )
2  // Rearrange  $a[ ]$  such that  $a[k]$  is the  $k$ -th smallest.
3  {
4      if ( $k \leq \lfloor n/2 \rfloor$ ) then
5          for  $i := 1$  to  $k$  do
6              {
7                   $q := i$ ;  $min := a[i]$ ;
8                  for  $j := i + 1$  to  $n$  do
9                      if ( $a[j] < min$ ) then
10                         {
11                              $q := j$ ;  $min := a[j]$ ;
12                         }
13                     Interchange( $a, q, i$ );
14                 }
15             else
16                 for  $i := n$  to  $k$  step  $-1$  do
17                     {
18                          $q := i$ ;  $max := a[i]$ ;
19                         for  $j := (i - 1)$  to  $1$  step  $-1$  do
20                             if ( $a[j] > max$ ) then
21                                 {
22                                      $q := j$ ;  $max := a[j]$ ;
23                                 }
24                         Interchange( $a, q, i$ );
25                     }
26     }

```

Algorithm 3.20 Straightforward selection algorithm

overhead in **Select2** (i.e., the constant term in $O(n)$ is relatively large).

- As a result of the above assertions, it is desirable to obtain composite algorithms for good average and worst-case performances. The composite algorithm for good worst-case performance will have the form of function **Select2** but will include the following after the first **if** statement.

```

if ( $n < c_1$ ) then return Select3( $a, m, p, k$ );
else if ( $n < c_2$ ) then return Select4( $a, m, p, k$ );

```

Since the overheads in **Select1** and **Select4** are about the same, the constants associated with the average computing times will be about

the same. Hence, Select1 may always be better than Select4 or there may be a small c_3 such that Select4 is better than Select1 for $n < c_3$. In any case, we expect there is a $c_4, c_4 > 0$, such that Select3 is faster than Select1 on the average for $n < c_4$.

To verify the preceding statements and determine c_1, c_2, c_3 , and c_4 , it is necessary to program the four algorithms in some programming language and run the four corresponding programs on a computer. Once the programs have been written, test data are needed to determine average and worst-case computing times. So, let us now say something about the data needed to obtain computing times from which $c_i, 1 \leq i \leq 4$, can be determined. Since we would also like information regarding the average and worst-case computing times of the resulting composite algorithms, we need test data for this too. We limit our testing to the case of distinct elements.

To obtain worst-case computing times for Select1, we change the algorithm slightly. This change will not affect its worst-case computing time but will enable us to use a rather simple data set to determine this time for various values of n . We dispense with the random selection rule for Partition and instead use $a[m]$ as the partitioning element. It is easy to see that the worst-case time is obtained with $a[i] = i$, $1 \leq i \leq n$, and $k = n$. As far as the average time for any given n is concerned, it is not easy to arrive at one data set and a k that exhibits this time. On the other hand, trying out all $n!$ different input permutations and $k = 1, 2, \dots, n$ for each of these is not a feasible way to find the average. An approximation to the average computing time can be obtained by trying out a few (say ten) random permutations of the numbers $1, 2, \dots, n$ and for each of these using a few (say five) random values of k . The average of the times obtained can be used as an approximation to the average computing time. Of course, using more permutations and more k values results in a better approximation. However, the number of permutations and k values we can use is limited by the amount of computational resources (in terms of time) we have available.

For Select2, the average time can be obtained in the same way as for Select1. For the worst-case time we can either try to figure out an input permutation for which the number of elements less than the median of medians is always as large as possible and then use $k = 1$. A simpler approach is to find just an approximation to the worst-case time. This can be done by taking the max of the computing times for all the tests used to obtain the average computing time. Since the computing times for Select2 vary with r , it is first necessary to determine an r that yields optimum behavior. Note that the r 's for optimum average and worst-case behaviors may be different.

We can verify that the worst-case data for Select3 are $a[i] = n + 1 - i$, for $1 \leq i \leq n$, and $k = \frac{n}{2}$. The computing time for Select3 is relatively insensitive to the input permutation. This permutation affects only the number of times the second if statement of Algorithm 3.20 is executed. On the average, this will be done about half the time. This can be achieved by using $a[i] = n + 1 - i$, $1 \leq i \leq n/2$, and $a[i] = n + 1$, $n/2 < i \leq n$. The k value needed to obtain the average computing time is readily seen to be $n/4$.

- (a) What test data would you use to determine worst-case and average times for Select4?
 - (b) Use the ideas above to obtain a table of worst-case and average times for Select1, Select2, Select3, and Select4.
16. Program Select1 and Select3. Determine when algorithm Select1 becomes better than Select3 on the average and also when Select2 better than Select3 for worst-case performance.
17. [Project] Program the algorithms of Exercise 4 as well as Select3 and Select4. Carry out a complete test along the lines discussed in Exercise 15. Write a detailed report together with graphs explaining the data sets, test strategies, and determination of c_1, \dots, c_4 . Write the final composite algorithms and give tables of computing times for these algorithms.

3.7 STRASSEN'S MATRIX MULTIPLICATION

Let A and B be two $n \times n$ matrices. The product matrix $C = AB$ is also an $n \times n$ matrix whose i, j th element is formed by taking the elements in the i th row of A and the j th column of B and multiplying them to get

$$C(i, j) = \sum_{1 \leq k \leq n} A(i, k)B(k, j) \quad (3.10)$$

for all i and j between 1 and n . To compute $C(i, j)$ using this formula, we need n multiplications. As the matrix C has n^2 elements, the time for the resulting matrix multiplication algorithm, which we refer to as the conventional method is $\Theta(n^3)$.

The divide-and-conquer strategy suggests another way to compute the product of two $n \times n$ matrices. For simplicity we assume that n is a power of 2, that is, that there exists a nonnegative integer k such that $n = 2^k$. In case n is not a power of two, then enough rows and columns of zeros can be added to both A and B so that the resulting dimensions are a power of two

(see the exercises for more on this subject). Imagine that A and B are each partitioned into four square submatrices, each submatrix having dimensions $\frac{n}{2} \times \frac{n}{2}$. Then the product AB can be computed by using the above formula for the product of 2×2 matrices: if AB is

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \quad (3.11)$$

then

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \quad (3.12)$$

If $n = 2$, then formulas (3.11) and (3.12) are computed using a multiplication operation for the elements of A and B . These elements are typically floating point numbers. For $n > 2$, the elements of C can be computed using *matrix* multiplication and addition operations applied to matrices of size $n/2 \times n/2$. Since n is a power of 2, these matrix products can be recursively computed by the same algorithm we are using for the $n \times n$ case. This algorithm will continue applying itself to smaller-sized submatrices until n becomes suitably small ($n = 2$) so that the product is computed directly.

To compute AB using (3.12), we need to perform eight multiplications of $n/2 \times n/2$ matrices and four additions of $n/2 \times n/2$ matrices. Since two $n/2 \times n/2$ matrices can be added in time cn^2 for some constant c , the overall computing time $T(n)$ of the resulting divide-and-conquer algorithm is given by the recurrence

$$T(n) = \begin{cases} b & n \leq 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

where b and c are constants.

This recurrence can be solved in the same way as earlier recurrences to obtain $T(n) = O(n^3)$. Hence no improvement over the conventional method has been made. Since matrix multiplications are more expensive than matrix additions ($O(n^3)$ versus $O(n^2)$), we can attempt to reformulate the equations for C_{ij} so as to have fewer multiplications and possibly more additions. Volker Strassen has discovered a way to compute the C_{ij} 's of (3.12) using only 7 multiplications and 18 additions or subtractions. His method involves first computing the seven $n/2 \times n/2$ matrices P , Q , R , S , T , U , and V as in (3.13). Then the C_{ij} 's are computed using the formulas in (3.14). As can be seen, P , Q , R , S , T , U , and V can be computed using 7 matrix multiplications and 10 matrix additions or subtractions. The C_{ij} 's require an additional 8 additions or subtractions.

$$\begin{aligned}
 P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
 Q &= (A_{21} + A_{22})B_{11} \\
 R &= A_{11}(B_{12} - B_{22}) \\
 S &= A_{22}(B_{21} - B_{11}) \\
 T &= (A_{11} + A_{12})B_{22} \\
 U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
 V &= (A_{12} - A_{22})(B_{21} + B_{22})
 \end{aligned} \tag{3.13}$$

$$\begin{aligned}
 C_{11} &= P + S - T + V \\
 C_{12} &= R + T \\
 C_{21} &= Q + S \\
 C_{22} &= P + R - Q + U
 \end{aligned} \tag{3.14}$$

The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases} \tag{3.15}$$

where a and b are constants. Working with this formula, we get

$$\begin{aligned}
 T(n) &= an^2[1 + 7/4 + (7/4)^2 + \cdots + (7/4)^{k-1}] + 7^kT(1) \\
 &\leq cn^2(7/4)^{\log_2 n} + 7^{\log_2 n}, \text{ } c \text{ a constant} \\
 &= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7} \\
 &= O(n^{\log_2 7}) \approx O(n^{2.81})
 \end{aligned}$$

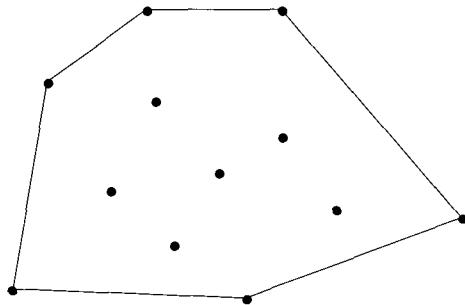
EXERCISES

- Verify by hand that Equations 3.13 and 3.14 yield the correct values for C_{11}, C_{12}, C_{21} , and C_{22} .
- Write an algorithm that multiplies two $n \times n$ matrices using $O(n^3)$ operations. Determine the precise number of multiplications, additions, and array element accesses.
- If k is a nonnegative constant, then prove that the recurrence

$$T(n) = \begin{cases} k & n = 1 \\ 3T(n/2) + kn & n > 1 \end{cases} \tag{3.16}$$

has the following solution (for n a power of 2):

$$T(n) = 3kn^{\log_2 3} - 2kn \tag{3.17}$$

**Figure 3.6** Convex hull: an example

(1) obtain the vertices of the convex hull (these vertices are also called *extreme points*), and (2) obtain the vertices of the convex hull in some order (clockwise, for example).

Here is a simple algorithm for obtaining the extreme points of a given set S of points in the plane. To check whether a particular point $p \in S$ is extreme, look at each possible triplet of points and see whether p lies in the triangle formed by these three points. If p lies in any such triangle, it is not extreme; otherwise it is. Testing whether p lies in a given triangle can be done in $\Theta(1)$ time (using the methods described in Section 3.8.1). Since there are $\Theta(n^3)$ possible triangles, it takes $\Theta(n^3)$ time to determine whether a given point is an extreme point or not. Since there are n points, this algorithm runs in a total of $\Theta(n^4)$ time.

Using divide-and-conquer, we can solve both versions of the convex hull problem in $\Theta(n \log n)$ time. We develop three algorithms for the convex hull in this section. The first has a worst-case time of $\Theta(n^2)$ whereas its average time is $\Theta(n \log n)$. This algorithm has a divide-and-conquer structure similar to that of QuickSort. The second has a worst-case time complexity of $\Theta(n \log n)$ and is not based on divide-and-conquer. The third algorithm is based on divide-and-conquer and has a time complexity of $\Theta(n \log n)$ in the worst case. Before giving further details, we digress to discuss some primitive geometric methods that are used in the convex hull algorithms.

3.8.1 Some Geometric Primitives

Let A be an $n \times n$ matrix whose elements are $\{a_{ij}\}$, $1 \leq i, j \leq n$. The *ijth minor* of A , denoted as A_{ij} , is defined to be the submatrix of A obtained by deleting the i th row and j th column. The *determinant* of A , denoted

$\det(A)$, is given by

$$\det(A) = \begin{cases} a_{11} & n = 1 \\ a_{11} \det(A_{11}) - a_{12} \det(A_{12}) + \cdots + (-1)^{n-1} \det(A_{1n}) & n > 1 \end{cases}$$

Consider the directed line segment $\langle p_1, p_2 \rangle$ from some point $p_1 = (x_1, y_1)$ to some other point $p_2 = (x_2, y_2)$. If $q = (x_3, y_3)$ is another point, we say q is to the left (right) of $\langle p_1, p_2 \rangle$ if the angle $p_1 p_2 q$ is a left (right) turn. [An angle is said to be a left (right) turn if it is less than or equal to (greater than or equal to) 180° .] We can check whether q is to the left (right) of $\langle p_1, p_2 \rangle$ by evaluating the determinant of the following matrix:

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix}$$

If this determinant is positive (negative), then q is to the left (right) of $\langle p_1, p_2 \rangle$. If this determinant is zero, the three points are colinear. This test can be used, for example, to check whether a given point p is within a triangle formed by three points, say p_1, p_2 , and p_3 (in clockwise order). The point p is within the triangle iff p is to the right of the line segments $\langle p_1, p_2 \rangle$, $\langle p_2, p_3 \rangle$, and $\langle p_3, p_1 \rangle$.

Also, for any three points $(x_1, y_1), (x_2, y_2)$, and (x_3, y_3) , the *signed area* formed by the corresponding triangle is given by one-half of the above determinant.

Let p_1, p_2, \dots, p_n be the vertices of the convex polygon Q in clockwise order. Let p be any other point. It is desired to check whether p lies in the interior of Q or outside. Consider a horizontal line h that extends from $-\infty$ to ∞ and goes through p . There are two possibilities: (1) h does not intersect any of the edges of Q , (2) h intersects some of the edges of Q . If case (1) is true, then, p is outside Q . In case (2), there can be at most two points of intersection. If h intersects Q at a single point, it is counted as two. Count the number of points of intersections that are to the left of p . If this number is even, then p is external to Q ; otherwise it is internal to Q . This method of checking whether p is interior to Q takes $\Theta(n)$ time.

3.8.2 The QuickHull Algorithm

An algorithm that is similar to QuickSort can be devised to compute the convex hull of a set X of n points in the plane. This algorithm, called QuickHull, first identifies the two points (call them p_1 and p_2) of X with the smallest and largest x -coordinate values. Assume now that there are no ties. Later we see how to handle ties. Both p_1 and p_2 are extreme points and part of the convex hull. The set X is divided into X_1 and X_2 so that

X_1 has all the points to the left of the line segment $\langle p_1, p_2 \rangle$ and X_2 has all the points to the right of $\langle p_1, p_2 \rangle$. Both X_1 and X_2 include the two points p_1 and p_2 . Then, the convex hulls of X_1 and X_2 (called the *upper hull* and *lower hull*, respectively) are computed using a divide-and-conquer algorithm called **Hull**. The union of these two convex hulls is the overall convex hull.

If there is more than one point with the smallest x -coordinate, let p'_1 and p''_1 be the points from among these with the least and largest y -coordinates, respectively. Similarly define p'_2 and p''_2 for the points with the largest x -coordinate values. Now X_1 will be all the points to the left of $\langle p''_1, p''_2 \rangle$ (including p''_1 and p''_2) and X_2 will be all the points to the right of $\langle p'_1, p'_2 \rangle$ (including p'_1 and p'_2). In the rest of the discussion we assume for simplicity that there are no ties for p_1 and p_2 . Appropriate modifications are needed in the event of ties.

We now describe how **Hull** computes the convex hull of X_1 . We determine a point of X_1 that belongs to the convex hull of X_1 and use it to partition the problem into two independent subproblems. Such a point is obtained by computing the area formed by p_1, p , and p_2 for each p in X_1 and picking the one with the largest (absolute) area. Ties are broken by picking the point p for which the angle pp_1p_2 is maximum. Let p_3 be that point.

Now X_1 is divided into two parts; the first part contains all the points of X_1 that are to the left of $\langle p_1, p_3 \rangle$ (including p_1 and p_3), and the second part contains all the points of X_1 that are to the left of $\langle p_3, p_2 \rangle$ (including p_3 and p_2) (see Figure 3.7). There cannot be any point of X_1 that is to the left of both $\langle p_1, p_3 \rangle$ and $\langle p_3, p_2 \rangle$. Also, all the other points are interior points and can be dropped from future consideration. The convex hull of each part is computed recursively, and the two convex hulls are merged easily by placing one next to the other in the right order.

If there are m points in X_1 , we can identify the point of division p_3 in time $O(m)$. Partitioning X_1 into two parts can also be done in $O(m)$ time. Merging the two convex hulls can be done in time $O(1)$. Let $T(m)$ stand for the run time of **Hull** on a list of m points and let m_1 and m_2 denote the sizes of the two resultant parts. Note that $m_1 + m_2 \leq m$. The recurrence relation for $T(m)$ is $T(m) = T(m_1) + T(m_2) + O(m)$, which is similar to the one for the run time of **QuickSort**. The worst-case run time is thus $O(m^2)$ on an input of m points. This happens when the partitioning at each level of recursion is highly uneven.

If the partitioning is nearly even at each level of recursion, then the run time will equal $O(m \log m)$ as in the case of **QuickSort**. Thus the average run time of **QuickHull** is $O(n \log n)$, on an input of size n , under appropriate assumptions on the input distribution.

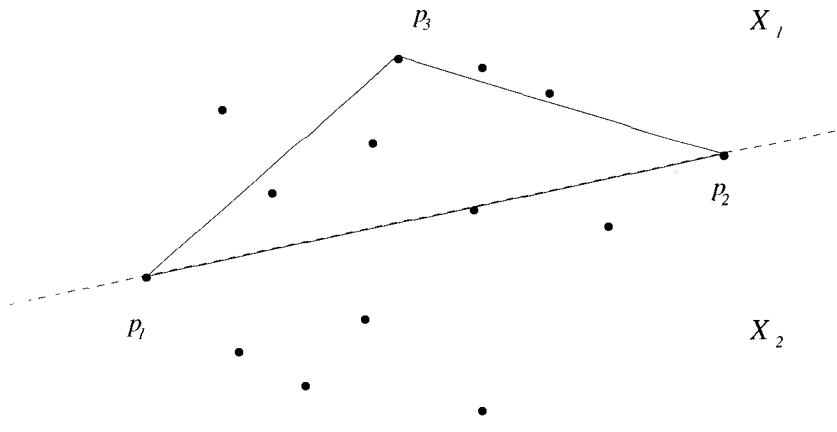


Figure 3.7 Identifying a point on the convex hull of X_1

3.8.3 Graham's Scan

If S is a set of points in the plane, Graham's scan algorithm identifies the point p from S with the lowest y -coordinate value (ties are broken by picking the leftmost among these). It then sorts the points of S according to the angle subtended by the points and p with the positive x -axis. Figure 3.8 gives an example. After having sorted the points, if we scan through the sorted list starting at p , every three successive points will form a left turn if all of these points lie on the hull. On the other hand if there are three successive points, say p_1, p_2 , and p_3 , that form a right turn, then we can immediately eliminate p_2 since it cannot lie on the convex hull. Notice that it will be an internal point because it lies within the triangle formed by p, p_1 , and p_3 .

We can eliminate all the interior points using the above procedure. Starting from p , we consider three successive points p_1, p_2 , and p_3 at a time. To begin with, $p_1 = p$. If these points form a left turn, we move to the next point in the list (that is, we set $p_1 = p_2$, and so on). If these three points form a right turn, then p_2 is deleted since it is an interior point. We move one point behind in the list by setting p_1 equal to its predecessor. This process of scanning ends when we reach the point p again.

Example 3.11 In Figure 3.8, the first three points looked at are $p, 1$, and 2 . Since these form a left turn, we move to $1, 2$, and 3 . These form a right turn and hence 2 is deleted. Next, the three points $p, 1$, and 3 are considered. These form a left turn and hence the pointer is moved to point 1 . The points

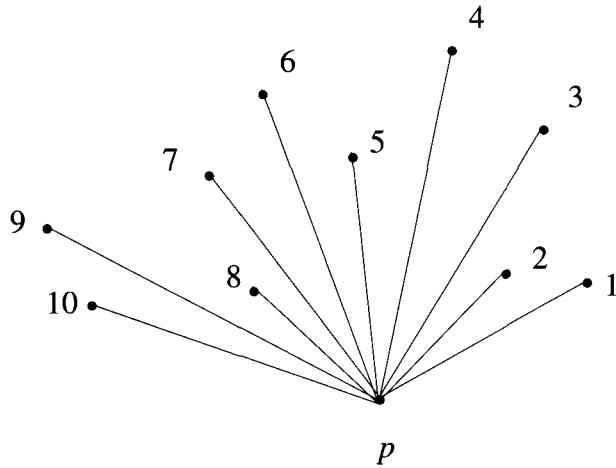


Figure 3.8 Graham's scan algorithm sorts the points first

1, 3, and 4 also form a left turn, and the scan proceeds to 3, 4, and 5 and then to 4, 5, and 6. Now point 5 gets deleted. The triplets 3, 4, 6; 4, 6, 7; and 6, 7, 8 form left turns whereas the next triplet 7, 8, 9 forms a right turn. Therefore, 8 gets deleted and in the next round 7 also gets eliminated. The next three triplets examined are 4, 6, 9; 6, 9, 10; and 9, 10, p , all of which are left turns. The final hull obtained is $p, 1, 3, 4, 6, 9$, and 10, which are points on the hull in counterclockwise (ccw) order. \square

This scan process is given in Algorithm 3.21. In this algorithm the set of points is realized as a doubly linked list $ptslist$. Function `Scan` runs in $O(n)$ time since for each triplet examined, either the scan moves one node ahead or one point gets removed. In the latter case, the scan moves one node back. Also note that for each triplet, the test as to whether a left or right turn is formed can be done in $O(1)$ time. Function `Area` computes the signed area formed by three points. The major work in the algorithm is in sorting the points. Since sorting takes $O(n \log n)$ time, the total time of Graham's scan algorithm is $O(n \log n)$.

3.8.4 An $O(n \log n)$ Divide-and-Conquer Algorithm

In this section we present a simple divide-and-conquer algorithm, called `DCHull`, which also takes $O(n \log n)$ time and computes the convex hull in clockwise order.

```

point = record{
    float x; float y;
    point *prev; point *next;
};

1  Algorithm Scan(list)
2  // list is a pointer to the first node in the input list.
3  {
4      *p := list; *p1 := list;
5      repeat
6      {
7          p2 := (p1 → next);
8          if ((p2 → next) ≠ 0) then p3 := (p2 → next);
9          else return; // End of the list
10         temp := Area((p1 → x), (p1 → y), (p2 → x),
11                         (p2 → y), (p3 → x), (p3 → y));
12         if (temp ≥ 0.0) then p1 := (p1 → next);
13         // If p1, p2, p3 form a left turn, move one point ahead;
14         // If not, delete p2 and move back.
15         else
16         {
17             (p1 → next) := p3; (p3 → prev) := p1; delete p2;
18             p1 := (p1 → prev);
19         }
20     } until (false);
21 }

1  Algorithm ConvexHull(ptslist)
2  {
3      // ptslist is a pointer to the first item of the input list. Find
4      // the point p in ptslist of lowest y-coordinate. Sort the
5      // points according to the angle made with p and the x-axis.
6      Sort(ptslist); Scan(ptslist); PrintList(ptslist);
7  }

```

Algorithm 3.21 Graham's scan algorithm

Given a set X of n points, like that in the case of QuickHull, the problem is reduced to finding the upper hull and the lower hull separately and then putting them together. Since the computations of the upper and lower hulls are very similar, we restrict our discussion to computing the upper hull. The divide-and-conquer algorithm for computing the upper hull partitions X into two nearly equal halves. Partitioning is done according to the x -coordinate values of points using the median x -coordinate as the splitter (see Section 3.6 for a discussion on median finding). Upper hulls are recursively computed for the two halves. These two hulls are then merged by finding the *line of tangent* (i.e., a straight line connecting a point each from the two halves, such that all the points of X are on one side of the line) (see Figure 3.9).

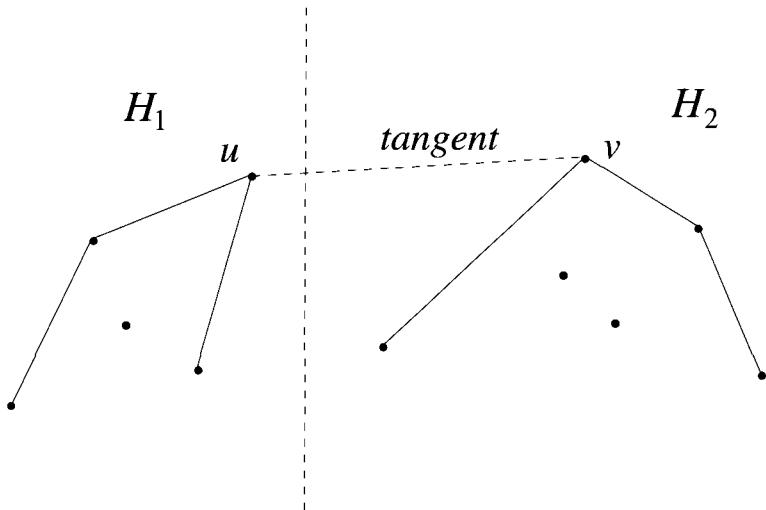


Figure 3.9 Divide and conquer to compute the convex hull

To begin with, the points p_1 and p_2 are identified [where p_1 (p_2) is the point with the least (largest) x -coordinate value]. This can be done in $O(n)$ time. Ties can be handled in exactly the same manner as in QuickHull. So, assume that there are no ties. All the points that are to the left of the line segment $\langle p_1, p_2 \rangle$ are separated from those that are to the right. This separation also can be done in $O(n)$ time. From here on, by "input" and " X " we mean all the points that are to the left of the line segment $\langle p_1, p_2 \rangle$. Also let $|X| = N$.

Sort the input points according to their x -coordinate values. Sorting can be done in $O(N \log N)$ time. This sorting is done only once in the computation of the upper hull. Let q_1, q_2, \dots, q_N be the sorted order of these

points. Now partition the input into two equal halves with $q_1, q_2, \dots, q_{N/2}$ in the first half and $q_{N/2+1}, q_{N/2+2}, \dots, q_N$ in the second half. The upper hull of each half is computed recursively. Let H_1 and H_2 be the upper hulls. Upper hulls are maintained as linked lists in clockwise order. We refer to the first element in the list as the leftmost point and the last element as the rightmost point.

The line of tangent is then found in $O(\log^2 N)$ time. If $\langle u, v \rangle$ is the line of tangent, then all the points of H_1 that are to the right of u are dropped. Similarly, all the points that are to the left of v in H_2 are dropped. The remaining part of H_1 , the line of tangent, and the remaining part of H_2 form the upper hull of the given input set.

If $T(N)$ is the run time of the above recursive algorithm for the upper hull on an input of N points, then we have

$$T(N) = 2T(N/2) + O(\log^2 N)$$

which solves to $T(N) = O(N)$. Thus the run time is dominated by the initial sorting step.

The only part of the algorithm that remains to be specified is how to find the line of tangent $\langle u, v \rangle$ in $O(\log^2 N)$ time. The way to find the tangent is to start from the middle point, call it p , of H_1 . Here the middle point refers to the middle element of the corresponding list. Find the tangent of p with H_2 . Let $\langle p, q \rangle$ be the tangent. Using $\langle p, q \rangle$, we can determine whether u is to the left of, equal to, or to the right of p in H_1 . A binary search in this fashion on the points of H_1 reveals u . Use a similar procedure to isolate v .

Lemma 3.1 Let H_1 and H_2 be two upper hulls with at most m points each. If p is any point of H_1 , its point q of tangency with H_2 can be found in $O(\log m)$ time.

Proof. If q' is any point in H_2 , we can check whether q' is to the left of, equal to, or to the right of q in $O(1)$ time (see Figure 3.10). In Figure 3.10, x and y are the left and right neighbors of q' in H_2 , respectively. If $\angle pq'x$ is a right turn and $\angle pq'y$ is a left turn, then q is to the right of q' (see case 1 of Figure 3.10). If $\angle pq'x$ and $\angle pq'y$ are both right turns, then $q' = q$ (see case 2 of Figure 3.10); otherwise q is to the left of q' (see case 3 of Figure 3.10). Thus we can perform a binary search on the points of H_2 and identify q in $O(\log m)$ time. \square

Lemma 3.2 If H_1 and H_2 are two upper hulls with at most m points each, their common tangent can be computed in $O(\log^2 m)$ time.

Proof. Let $u \in H_1$ and $v \in H_2$ be such that $\langle u, v \rangle$ is the line of tangent. Also let p be an arbitrary point of H_1 and let $q \in H_2$ be such that $\langle p, q \rangle$ is a

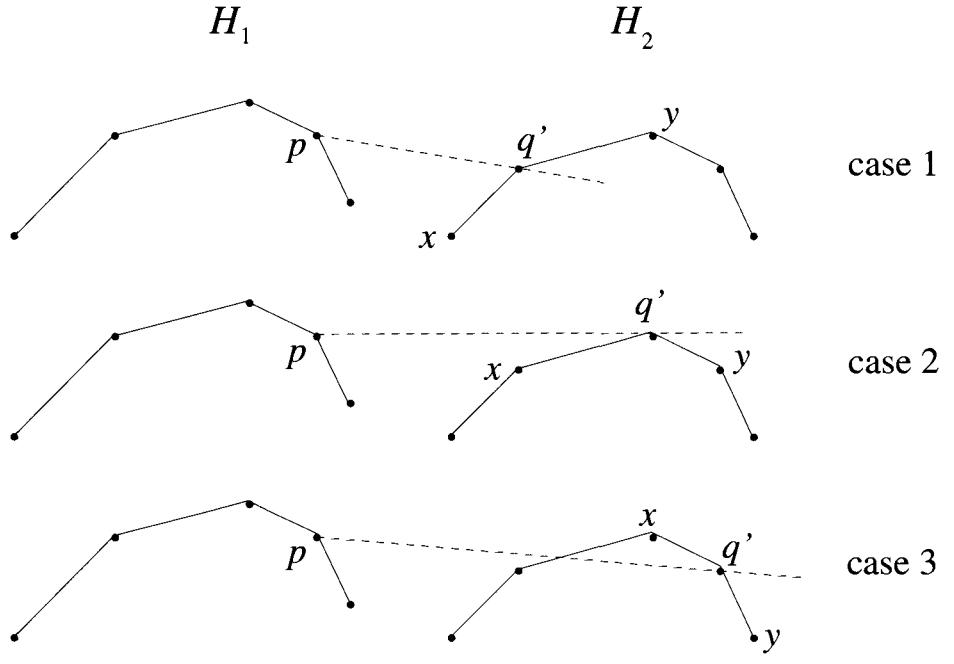
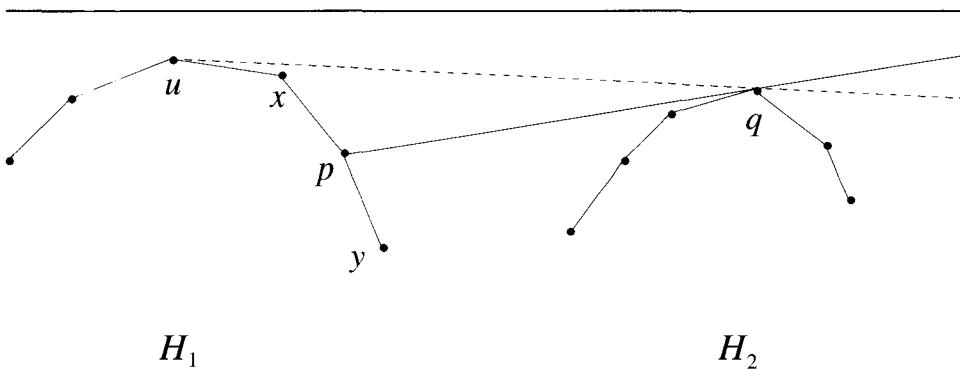


Figure 3.10 Proof of Lemma 3.1

tangent of H_2 . Given p and q , we can check in $O(1)$ time whether u is to the left of, equal to, or to the right of p (see Figure 3.11). Here x and y are left and right neighbors, respectively, of p in H_1 . If $\langle p, q \rangle$ is also tangential to H_1 , then $p = u$. If $\angle xpq$ is a left turn, then u is to the left of p ; otherwise u is to the right of p . This suggests a binary search for u . For each point p of H_1 chosen, we have to determine the tangent from p to H_2 and then decide the relative positioning of p with respect to u . We can do this computation in $O(\log m \times \log m) = O(\log^2 m)$ time. \square

In summary, given two upper hulls with $\frac{N}{2}$ points each, the line of tangent can be computed in $O(\log^2 N)$ time.

Theorem 3.4 A convex hull of n points in the plane can be computed in $O(n \log n)$ time. \square

**Figure 3.11** Proof of Lemma 3.2

EXERCISES

1. Write an algorithm in pseudocode that implements QuickHull and test it using suitable data.
2. Code the divide-and-conquer algorithm DCHull and test it using appropriate data.
3. Run the three algorithms for convex hull discussed in this section on various random inputs and compare their performances.
4. Algorithm DCHull can be modified as follows: Instead of using the median as the splitter, we could use a randomly chosen point as the splitter. Then X is partitioned into two around this point. The rest of the function DCHull is the same. Write code for this modified algorithm and compare it with DCHull empirically.
5. Let S be a set of n points in the plane. It is given that there is only a constant (say c) number of points on the hull of S . Can you devise a convex hull algorithm for S that runs in time $o(n \log n)$? Conceive of special algorithms for $c = 3$ and $c = 4$ first and then generalize.

3.9 REFERENCES AND READINGS

Algorithm MaxMin (Algorithm 3.6) is due to I. Pohl and the quicksort algorithm (Algorithm 3.13) is due to C. A. R. Hoare. The randomized sorting algorithm in Algorithm 3.16 is due to W. D. Frazer and A. C. McKeller and

UNIT - 3

THE GREEDY METHOD

Chapter 4

THE GREEDY METHOD

4.1 THE GENERAL METHOD

The greedy method is perhaps the most straightforward design technique we consider 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 need 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 that works in stages, considering one input at a time. At each stage, a decision is made regarding whether 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. Otherwise, it is added. The selection procedure itself is based on some optimization measure. This measure may 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. This version of the greedy technique is called the *subset paradigm*.

We can describe the subset paradigm abstractly, but more precisely than above, by considering the control abstraction in Algorithm 4.1.

The function `Select` selects an input from $a[]$ and removes it. The selected input's value is assigned to x . `Feasible` is a Boolean-valued function that determines whether x can be included into the solution vector. The function `Union` combines x with the solution and updates the objective function. The

```

1  Algorithm Greedy( $a, n$ )
2  //  $a[1 : n]$  contains the  $n$  inputs.
3  {
4       $solution := \emptyset$ ; // Initialize the solution.
5      for  $i := 1$  to  $n$  do
6          {
7               $x := \text{Select}(a)$ ;
8              if Feasible( $solution, x$ ) then
9                   $solution := \text{Union}(solution, x)$ ;
10             }
11         return  $solution$ ;
12     }

```

Algorithm 4.1 Greedy method control abstraction for the subset paradigm

function **Greedy** describes the essential way that a greedy algorithm will look, once a particular problem is chosen and the functions **Select**, **Feasible**, and **Union** are properly implemented.

For problems that do not call for the selection of an optimal subset, in the greedy method we make decisions by considering the inputs in some order. Each decision is made using an optimization criterion that can be computed using decisions already made. Call this version of the greedy method the *ordering paradigm*. Sections 4.2, 4.3, 4.4, and 4.5 consider problems that fit the subset paradigm, and Sections 4.6, 4.7, and 4.8 consider problems that fit the ordering paradigm.

EXERCISE

1. Write a control abstraction for the ordering paradigm.

4.2 KNAPSACK PROBLEM

Let us try to apply the greedy method to solve the knapsack problem. We are given n objects and a knapsack or bag. 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 can be stated as

$$\text{maximize} \sum_{1 \leq i \leq n} p_i x_i \quad (4.1)$$

$$\text{subject to} \sum_{1 \leq i \leq n} w_i x_i \leq m \quad (4.2)$$

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

The profits and weights are positive numbers.

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

Example 4.1 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)	$\sum w_i x_i$	$\sum p_i x_i$
1.	$(1/2, 1/3, 1/4)$	16.5	24.25
2.	$(1, 2/15, 0)$	20	28.2
3.	$(0, 2/3, 1)$	20	31
4.	$(0, 1, 1/2)$	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit. As we shall soon see, this solution is optimal for the given problem instance. \square

Lemma 4.1 In case the sum of all the weights is $\leq m$, then $x_i = 1, 1 \leq i \leq n$ is an optimal solution. \square

So let us assume the sum of weights exceeds m . Now all the x_i 's cannot be 1. Another observation to make is:

Lemma 4.2 All optimal solutions will fill the knapsack exactly. \square

Lemma 4.2 is true because we can always increase the contribution of some object i by a fractional amount until the total weight is exactly m .

Note that the knapsack problem calls for selecting a subset of the objects and hence fits the subset paradigm. In addition to selecting a subset, the knapsack problem also involves the selection of an x_i for each object. Several simple greedy strategies to obtain feasible solutions whose sums are identically m suggest themselves. First, we can try to fill the knapsack by including next the object with largest profit. If an object under consideration doesn't fit, then a fraction of it is included to fill the knapsack. Thus each time an object is included (except possibly when the last object is included)

into the knapsack, we obtain the largest possible increase in profit value. Note that if only a fraction of the last object is included, then it may be possible to get a bigger increase by using a different object. For example, if we have two units of space left and two objects with $(p_i = 4, w_i = 4)$ and $(p_j = 3, w_j = 2)$ remaining, then using j is better than using half of i . Let us use this selection strategy on the data of Example 4.1.

Object one has the largest profit value ($p_1 = 25$). So it is placed into the knapsack first. Then $x_1 = 1$ and a profit of 25 is earned. Only 2 units of knapsack capacity are left. Object two has the next largest profit ($p_2 = 24$). However, $w_2 = 15$ and it doesn't fit into the knapsack. Using $x_2 = 2/15$ fills the knapsack exactly with part of object 2 and the value of the resulting solution is 28.2. This is solution 2 and it is readily seen to be suboptimal. The method used to obtain this solution is termed a greedy method because at each step (except possibly the last one) we chose to introduce that object which would increase the objective function value the most. However, this greedy method did not yield an optimal solution. Note that even if we change the above strategy so that in the last step the objective function increases by as much as possible, an optimal solution is not obtained for Example 4.1.

We can formulate at least two other greedy approaches attempting to obtain optimal solutions. From the preceding example, we note that considering objects in order of nonincreasing profit values does not yield an optimal solution because even though the objective function value takes on large increases at each step, the number of steps is few as the knapsack capacity is used up at a rapid rate. So, let us try to be greedy with capacity and use it up as slowly as possible. This requires us to consider the objects in order of nondecreasing weights w_i . Using Example 4.1, solution 3 results. This too is suboptimal. This time, even though capacity is used slowly, profits aren't coming in rapidly enough.

Thus, our next attempt is an algorithm that strives to achieve a balance between the rate at which profit increases and the rate at which capacity is used. At each step we include that object which has the maximum profit per unit of capacity used. This means that objects are considered in order of the ratio p_i/w_i . Solution 4 of Example 4.1 is produced by this strategy. If the objects have already been sorted into nonincreasing order of p_i/w_i , then function GreedyKnapsack (Algorithm 4.2) 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.

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 to capacity used. Once an optimization measure has been chosen, the greedy

2. [0/1 Knapsack] Consider the knapsack problem discussed in this section. We add the requirement that $x_i = 1$ or $x_i = 0$, $1 \leq i \leq n$; that is, an object is either included or not included into the knapsack. We wish to solve the problem

$$\begin{aligned} & \max \sum_1^n p_i x_i \\ & \text{subject to } \sum_1^n w_i x_i \leq m \\ & \text{and } x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n \end{aligned}$$

One greedy strategy is to consider the objects in order of nonincreasing density p_i/w_i and add the object into the knapsack if it fits. Show that this strategy doesn't necessarily yield an optimal solution.

4.3 TREE VERTEX SPLITTING

Consider a directed binary tree each edge of which is labeled with a real number (called its *weight*). Trees with edge weights are called *weighted trees*. A weighted tree can be used, for example, to model a distribution network in which electric signals or commodities such as oil are transmitted. Nodes in the tree correspond to receiving stations and edges correspond to transmission lines. It is conceivable that in the process of transmission some loss occurs (drop in voltage in the case of electric signals or drop in pressure in the case of oil). Each edge in the tree is labeled with the loss that occurs in traversing that edge. The network may not be able to tolerate losses beyond a certain level. In places where the loss exceeds the tolerance level, boosters have to be placed. Given a network and a loss tolerance level, the *Tree Vertex Splitting Problem (TVSP)* is to determine an optimal placement of boosters. It is assumed that the boosters can only be placed in the nodes of the tree.

The TVSP can be specified more precisely as follows: Let $T = (V, E, w)$ be a weighted directed tree, where V is the vertex set, E is the edge set, and w is the weight function for the edges. In particular, $w(i, j)$ is the weight of the edge $\langle i, j \rangle \in E$. The weight $w(i, j)$ is undefined for any $\langle i, j \rangle \notin E$. A *source vertex* is a vertex with in-degree zero, and a *sink vertex* is a vertex with out-degree zero. For any path P in the tree, its *delay*, $d(P)$, is defined to be the sum of the weights on that path. The delay of the tree T , $d(T)$, is the maximum of all the path delays.

Let T/X be the forest that results when each vertex u in X is split into two nodes u^i and u^o such that all the edges $\langle u, j \rangle \in E$ ($\langle j, u \rangle \in E$) are

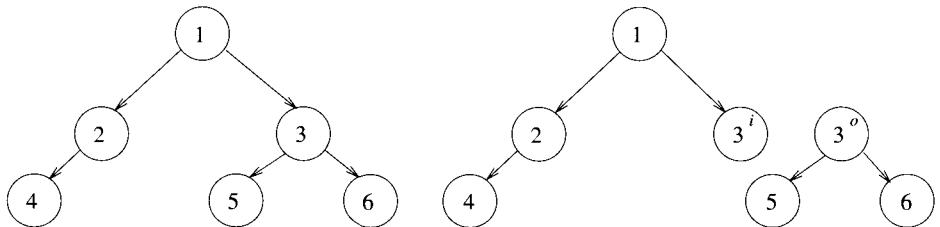


Figure 4.1 A tree before and after splitting the node 3

replaced by edges of the form $\langle u^o, j \rangle$ ($\langle j, u^i \rangle$). In other words, outbound edges from u now leave from u^o and inbound edges to u now enter at u^i . Figure 4.1 shows a tree before and after splitting the node 3. A node that gets split corresponds to a booster station. The TVSP is to identify a set $X \subseteq V$ of minimum cardinality for which $d(T/X) \leq \delta$, for some specified tolerance limit δ . Note that the TVSP has a solution only if the maximum edge weight is $\leq \delta$. Also note that the TVSP naturally fits the subset paradigm.

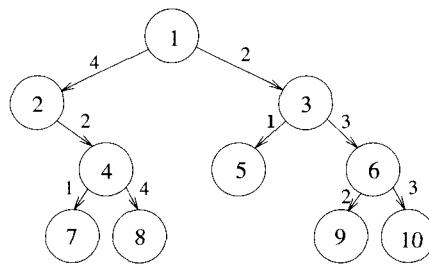
Given a weighted tree $T(V, E, w)$ and a tolerance limit δ , any subset X of V is a feasible solution if $d(T/X) \leq \delta$. Given an X , we can compute $d(T/X)$ in $O(|V|)$ time. A trivial way of solving the TVSP is to compute $d(T/X)$ for each possible subset X of V . But there are $2^{|V|}$ such subsets! A better algorithm can be obtained using the greedy method.

For the TVSP, the quantity that is optimized (minimized) is the number of nodes in X . A greedy approach to solving this problem is to compute for each node $u \in V$, the maximum delay $d(u)$ from u to any other node in its subtree. If u has a parent v such that $d(u) + w(v, u) > \delta$, then the node u gets split and $d(u)$ is set to zero. Computation proceeds from the leaves toward the root.

In the tree of Figure 4.2, let $\delta = 5$. For each of the leaf nodes 7, 8, 5, 9, and 10 the delay is zero. The delay for any node is computed only after the delays for its children have been determined. Let u be any node and $C(u)$ be the set of all children of u . Then $d(u)$ is given by

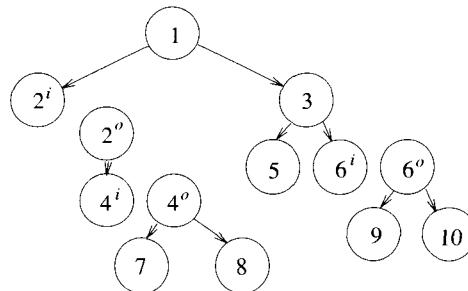
$$d(u) = \max_{v \in C(u)} \{d(v) + w(u, v)\}$$

Using the above formula, for the tree of Figure 4.2, $d(4) = 4$. Since $d(4) + w(2, 4) = 6 > \delta$, node 4 gets split. We set $d(4) = 0$. Now $d(2)$ can be

**Figure 4.2** An example tree

computed and is equal to 2. Since $d(2) + w(1, 2)$ exceeds δ , node 2 gets split and $d(2)$ is set to zero. Then $d(6)$ is equal to 3. Also, since $d(6) + w(3, 6) > \delta$, node 6 has to be split. Set $d(6)$ to zero. Now $d(3)$ is computed as 3. Finally, $d(1)$ is computed as 5.

Figure 4.3 shows the final tree that results after splitting the nodes 2, 4, and 6. This algorithm is described in Algorithm 4.3, which is invoked as $\text{TVS}(root, \delta)$, $root$ being the root of the tree. The order in which TVS visits (i.e., computes the delay values of) the nodes of the tree is called the *post order* and is studied again in Chapter 6.

**Figure 4.3** The final tree after splitting the nodes 2, 4, and 6

```

1   Algorithm TVS( $T, \delta$ )
2   // Determine and output the nodes to be split.
3   //  $w()$  is the weighting function for the edges.
4   {
5       if ( $T \neq 0$ ) then
6           {
7                $d[T] := 0$ ;
8               for each child  $v$  of  $T$  do
9                   {
10                      TVS( $v, \delta$ );
11                       $d[T] := \max\{d[T], d[v] + w(T, v)\}$ ;
12                  }
13                  if (( $T$  is not the root) and
14                       $(d[T] + w(\text{parent}(T), T) > \delta)$ ) then
15                      {
16                          write ( $T$ );  $d[T] := 0$ ;
17                      }
18                  }
19  }

```

Algorithm 4.3 The tree vertex splitting algorithm

Algorithm TVS takes $\Theta(n)$ time, where n is the number of nodes in the tree. This can be seen as follows: When TVS is called on any node T , only a constant number of operations are performed (excluding the time taken for the recursive calls). Also, TVS is called only once on each node T in the tree.

Algorithm 4.4 is a revised version of Algorithm 4.3 for the special case of directed binary trees. A sequential representation of the tree (see Section 2.2) has been employed. The tree is stored in the array $tree[]$ with the root at $tree[1]$. Edge weights are stored in the array $weight[]$. If $tree[i]$ has a tree node, the weight of the incoming edge from its parent is stored in $weight[i]$. The delay of node i is stored in $d[i]$. The array $d[]$ is initialized to zero at the beginning. Entries in the arrays $tree[]$ and $weight[]$ corresponding to nonexistent nodes will be zero. As an example, for the tree of Figure 4.2, $tree[]$ will be set to $\{1, 2, 3, 0, 4, 5, 6, 0, 0, 7, 8, 0, 0, 9, 10\}$ starting at cell 1. Also, $weight[]$ will be set to $\{0, 4, 2, 0, 2, 1, 3, 0, 0, 1, 4, 0, 0, 2, 3\}$ at the beginning, starting from cell 1. The algorithm is invoked as $TVS(1, \delta)$. Now we show that TVS (Algorithm 4.3) will always split a minimal number of nodes.

```

1   Algorithm TVS( $i, \delta$ )
2   // Determine and output a minimum cardinality split set.
3   // The tree is realized using the sequential representation.
4   // Root is at  $tree[1]$ .  $N$  is the largest number such that
5   //  $tree[N]$  has a tree node.
6   {
7       if ( $tree[i] \neq 0$ ) then // If the tree is not empty
8           if ( $2i > N$ ) then  $d[i] := 0$ ; //  $i$  is a leaf.
9           else
10          {
11              TVS( $2i, \delta$ );
12               $d[i] := \max(d[i], d[2i] + weight[2i])$ ;
13              if ( $2i + 1 \leq N$ ) then
14                  {
15                      TVS( $2i + 1, \delta$ );
16                       $d[i] := \max(d[i], d[2i + 1] + weight[2i + 1])$ ;
17                  }
18              }
19              if (( $tree[i] \neq 1$ ) and ( $d[i] + weight[i] > \delta$ )) then
20                  {
21                      write ( $tree[i]$ );  $d[i] := 0$ ;
22                  }
23  }

```

Algorithm 4.4 TVS for the special case of binary trees

Theorem 4.2 Algorithm TVS outputs a minimum cardinality set U such that $d(T/U) \leq \delta$ on any tree T , provided no edge of T has weight $> \delta$.

Proof: The proof is by induction on the number of nodes in the tree. If the tree has a single node, the theorem is true. Assume the theorem for all trees of size $\leq n$. We prove it for trees of size $n + 1$ also.

Let T be any tree of size $n + 1$ and let U be the set of nodes split by TVS. Also let W be a minimum cardinality set such that $d(T/W) \leq \delta$. We have to show that $|U| \leq |W|$. If $|U| = 0$, this is true. Otherwise, let x be the first vertex split by TVS. Let T_x be the subtree rooted at x . Let T' be the tree obtained from T by deleting T_x except for x . Note that W has to have at least one node, say y , from T_x . Let $W' = W - \{y\}$. If there is a W^* such that $|W^*| < |W'|$ and $d(T'/W^*) \leq \delta$, then since $d(T/(W^* + \{x\})) \leq \delta$, W is not a minimum cardinality split set for T . Thus, W' has to be a minimum cardinality split set such that $d(T'/W') \leq \delta$.

If algorithm TVS is run on tree T' , the set of split nodes output is $U - \{x\}$. Since T' has $\leq n$ nodes, $U - \{x\}$ is a minimum cardinality split set for T' . This in turn means that $|W'| \geq |U| - 1$. In other words, $|W| \geq |U|$. \square

EXERCISES

1. For the tree of Figure 4.2 solve the TVSP when (a) $\delta = 4$ and (b) $\delta = 6$.
2. Rewrite TVS (Algorithm 4.3) for general trees. Make use of pointers.

4.4 JOB SEQUENCING WITH DEADLINES

We are given a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in J , or $\sum_{i \in J} p_i$. An optimal solution is a feasible solution with maximum value. Here again, since the problem involves the identification of a subset, it fits the subset paradigm.

Example 4.2 Let $n = 4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. \square

To formulate a greedy algorithm to obtain an optimal solution, we must formulate an optimization measure to determine how the next job is chosen. As a first attempt we can choose the objective function $\sum_{i \in J} p_i$ as our optimization measure. Using this measure, the next job to include is the one that increases $\sum_{i \in J} p_i$ the most, subject to the constraint that the resulting J is a feasible solution. This requires us to consider jobs in nonincreasing order of the p_i 's. Let us apply this criterion to the data of Example 4.2. We begin with $J = \emptyset$ and $\sum_{i \in J} p_i = 0$. Job 1 is added to J as it has the largest profit and $J = \{1\}$ is a feasible solution. Next, job 4 is considered. The solution $J = \{1, 4\}$ is also feasible. Next, job 3 is considered and discarded as $J = \{1, 3, 4\}$ is not feasible. Finally, job 2 is considered for inclusion into J . It is discarded as $J = \{1, 2, 4\}$ is not feasible. Hence, we are left with the solution $J = \{1, 4\}$ with value 127. This is the optimal solution for the given problem instance. Theorem 4.4 proves that the greedy algorithm just described always obtains an optimal solution to this sequencing problem.

Before attempting the proof, let us see how we can determine whether a given J is a feasible solution. One obvious way is to try out all possible permutations of the jobs in J and check whether the jobs in J can be processed in any one of these permutations (sequences) without violating the deadlines. For a given permutation $\sigma = i_1, i_2, i_3, \dots, i_k$, this is easy to do, since the earliest time job $i_q, 1 \leq q \leq k$, will be completed is q . If $q > d_{i_q}$, then using σ , at least job i_q will not be completed by its deadline. However, if $|J| = i$, this requires checking $i!$ permutations. Actually, the feasibility of a set J can be determined by checking only one permutation of the jobs in J . This permutation is any one of the permutations in which jobs are ordered in nondecreasing order of deadlines.

Theorem 4.3 Let J be a set of k jobs and $\sigma = i_1, i_2, \dots, i_k$ a permutation of jobs in J such that $d_{i_1} \leq d_{i_2} \leq \dots \leq d_{i_k}$. Then J is a feasible solution iff the jobs in J can be processed in the order σ without violating any deadline.

Proof: Clearly, if the jobs in J can be processed in the order σ without violating any deadline, then J is a feasible solution. So, we have only to show that if J is feasible, then σ represents a possible order in which the jobs can be processed. If J is feasible, then there exists $\sigma' = r_1, r_2, \dots, r_k$ such that $d_{r_q} \geq q, 1 \leq q \leq k$. Assume $\sigma' \neq \sigma$. Then let a be the least index such that $r_a \neq i_a$. Let $r_b = i_a$. Clearly, $b > a$. In σ' we can interchange r_a and r_b . Since $d_{r_a} \geq d_{r_b}$, the resulting permutation $\sigma'' = s_1, s_2, \dots, s_k$ represents an order in which the jobs can be processed without violating a deadline. Continuing in this way, σ' can be transformed into σ without violating any deadline. Hence, the theorem is proved. \square

Theorem 4.3 is true even if the jobs have different processing times $t_i \geq 0$ (see the exercises).

Theorem 4.4 The greedy method described above always obtains an optimal solution to the job sequencing problem.

Proof: Let $(p_i, d_i), 1 \leq i \leq n$, define any instance of the job sequencing problem. Let I be the set of jobs selected by the greedy method. Let J be the set of jobs in an optimal solution. We now show that both I and J have the same profit values and so I is also optimal. We can assume $I \neq J$ as otherwise we have nothing to prove. Note that if $J \subset I$, then J cannot be optimal. Also, the case $I \subset J$ is ruled out by the greedy method. So, there exist jobs a and b such that $a \in I$, $a \notin J$, $b \in J$, and $b \notin I$. Let a be a highest-profit job such that $a \in I$ and $a \notin J$. It follows from the greedy method that $p_a \geq p_b$ for all jobs b that are in J but not in I . To see this, note that if $p_b > p_a$, then the greedy method would consider job b before job a and include it into I .

Now, consider feasible schedules S_I and S_J for I and J respectively. Let i be a job such that $i \in I$ and $i \in J$. Let i be scheduled from t to $t + 1$ in S_I and t' to $t' + 1$ in S_J . If $t < t'$, then we can interchange the job (if any) scheduled in $[t', t' + 1]$ in S_I with i . If no job is scheduled in $[t', t' + 1]$ in I , then i is moved to $[t', t' + 1]$. The resulting schedule is also feasible. If $t' < t$, then a similar transformation can be made in S_J . In this way, we can obtain schedules S'_I and S'_J with the property that all jobs common to I and J are scheduled at the same time. Consider the interval $[t_a, t_a + 1]$ in S'_I in which the job a (defined above) is scheduled. Let b be the job (if any) scheduled in S'_J in this interval. From the choice of a , $p_a \geq p_b$. Scheduling a from t_a to $t_a + 1$ in S'_J and discarding job b gives us a feasible schedule for job set $J' = J - \{b\} \cup \{a\}$. Clearly, J' has a profit value no less than that of J and differs from I in one less job than J does.

By repeatedly using the transformation just described, J can be transformed into I with no decrease in profit value. So I must be optimal. \square

A high-level description of the greedy algorithm just discussed appears as Algorithm 4.5. This algorithm constructs an optimal set J of jobs that can be processed by their due times. The selected jobs can be processed in the order given by Theorem 4.3.

Now, let us see how to represent the set J and how to carry out the test of lines 7 and 8 in Algorithm 4.5. Theorem 4.3 tells us how to determine whether all jobs in $J \cup \{i\}$ can be completed by their deadlines. We can avoid sorting the jobs in J each time by keeping the jobs in J ordered by deadlines. We can use an array $d[1 : n]$ to store the deadlines of the jobs in the order of their p -values. The set J itself can be represented by a one-dimensional array $J[1 : k]$ such that $J[r], 1 \leq r \leq k$ are the jobs in J and $d[J[1]] \leq d[J[2]] \leq \dots \leq d[J[k]]$. To test whether $J \cup \{i\}$ is feasible, we have just to insert i into J preserving the deadline ordering and then verify that $d[J[r]] \leq r, 1 \leq r \leq k + 1$. The insertion of i into J is simplified by the use of a fictitious job 0 with $d[0] = 0$ and $J[0] = 0$. Note also that if job i is to be inserted at position q , then only the positions of jobs $J[q], J[q + 1],$

```

1  Algorithm GreedyJob( $d, J, n$ )
2  //  $J$  is a set of jobs that can be completed by their deadlines.
3  {
4       $J := \{1\}$ ;
5      for  $i := 2$  to  $n$  do
6      {
7          if (all jobs in  $J \cup \{i\}$  can be completed
8              by their deadlines) then  $J := J \cup \{i\}$ ;
9      }
10 }

```

Algorithm 4.5 High-level description of job sequencing algorithm

$\dots, J[k]$ are changed after the insertion. Hence, it is necessary to verify only that these jobs (and also job i) do not violate their deadlines following the insertion. The algorithm that results from this discussion is function JS (Algorithm 4.6). The algorithm assumes that the jobs are already sorted such that $p_1 \geq p_2 \geq \dots \geq p_n$. Further it assumes that $n \geq 1$ and the deadline $d[i]$ of job i is at least 1. Note that no job with $d[i] < 1$ can ever be finished by its deadline. Theorem 4.5 proves that JS is a correct implementation of the greedy strategy.

Theorem 4.5 Function JS is a correct implementation of the greedy-based method described above.

Proof: Since $d[i] \geq 1$, the job with the largest p_i will always be in the greedy solution. As the jobs are in nonincreasing order of the p_i 's, line 8 in Algorithm 4.6 includes the job with largest p_i . The **for** loop of line 10 considers the remaining jobs in the order required by the greedy method described earlier. At all times, the set of jobs already included in the solution is maintained in J . If $J[i]$, $1 \leq i \leq k$, is the set already included, then J is such that $d[J[i]] \leq d[J[i+1]]$, $1 \leq i < k$. This allows for easy application of the feasibility test of Theorem 4.3. When job i is being considered, the **while** loop of line 15 determines where in J this job has to be inserted. The use of a fictitious job 0 (line 7) allows easy insertion into position 1. Let w be such that $d[J[w]] \leq d[i]$ and $d[J[q]] > d[i]$, $w < q \leq k$. If job i is included into J , then jobs $J[q]$, $w < q \leq k$, have to be moved one position up in J (line 19). From Theorem 4.3, it follows that such a move retains feasibility of J iff $d[J[q]] \neq q$, $w < q \leq k$. This condition is verified in line 15. In addition, i can be inserted at position $w + 1$ iff $d[i] > w$. This is verified in line 16 (note $r = w$ on exit from the **while** loop if $d[J[q]] \neq q$, $w < q \leq k$). The correctness of JS follows from these observations. \square

```

1  Algorithm JS( $d, j, n$ )
2  //  $d[i] \geq 1$ ,  $1 \leq i \leq n$  are the deadlines,  $n \geq 1$ . The jobs
3  // are ordered such that  $p[1] \geq p[2] \geq \dots \geq p[n]$ .  $J[i]$ 
4  // is the  $i$ th job in the optimal solution,  $1 \leq i \leq k$ .
5  // Also, at termination  $d[J[i]] \leq d[J[i+1]]$ ,  $1 \leq i < k$ .
6  {
7       $d[0] := J[0] := 0$ ; // Initialize.
8       $J[1] := 1$ ; // Include job 1.
9       $k := 1$ ;
10     for  $i := 2$  to  $n$  do
11     {
12         // Consider jobs in nonincreasing order of  $p[i]$ . Find
13         // position for  $i$  and check feasibility of insertion.
14          $r := k$ ;
15         while  $((d[J[r]] > d[i]) \text{ and } (d[J[r]] \neq r))$  do  $r := r - 1$ ;
16         if  $((d[J[r]] \leq d[i]) \text{ and } (d[i] > r))$  then
17         {
18             // Insert  $i$  into  $J[ ]$ .
19             for  $q := k$  to  $(r + 1)$  step  $-1$  do  $J[q + 1] := J[q]$ ;
20              $J[r + 1] := i$ ;  $k := k + 1$ ;
21         }
22     }
23     return  $k$ ;
24 }
```

Algorithm 4.6 Greedy algorithm for sequencing unit time jobs with deadlines and profits

For JS there are two possible parameters in terms of which its complexity can be measured. We can use n , the number of jobs, and s , the number of jobs included in the solution J . The **while** loop of line 15 in Algorithm 4.6 is iterated at most k times. Each iteration takes $\Theta(1)$ time. If the conditional of line 16 is true, then lines 19 and 20 are executed. These lines require $\Theta(k - r)$ time to insert job i . Hence, the total time for each iteration of the **for** loop of line 10 is $\Theta(k)$. This loop is iterated $n - 1$ times. If s is the final value of k , that is, s is the number of jobs in the final solution, then the total time needed by algorithm JS is $\Theta(sn)$. Since $s \leq n$, the worst-case time, as a function of n alone is $\Theta(n^2)$. If we consider the job set $p_i = d_i = n - i + 1$, $1 \leq i \leq n$, then algorithm JS takes $\Theta(n^2)$ time to determine J . Hence, the worst-case computing time for JS is $\Theta(n^2)$. In addition to the space needed for d , JS needs $\Theta(s)$ amount of space for J .

Note that the profit values are not needed by JS. It is sufficient to know that $p_i \geq p_{i+1}$, $1 \leq i < n$.

The computing time of JS can be reduced from $O(n^2)$ to nearly $O(n)$ by using the disjoint set union and find algorithms (see Section 2.5) and a different method to determine the feasibility of a partial solution. If J is a feasible subset of jobs, then we can determine the processing times for each of the jobs using the rule: if job i hasn't been assigned a processing time, then assign it to the slot $[\alpha - 1, \alpha]$, where α is the largest integer r such that $1 \leq r \leq d_i$ and the slot $[\alpha - 1, \alpha]$ is free. This rule simply delays the processing of job i as much as possible. Consequently, when J is being built up job by job, jobs already in J do not have to be moved from their assigned slots to accommodate the new job. If for the new job being considered there is no α as defined above, then it cannot be included in J . The proof of the validity of this statement is left as an exercise.

Example 4.3 Let $n = 5$, $(p_1, \dots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \dots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
\emptyset	none	1	assign to $[1, 2]$	0
$\{1\}$	$[1, 2]$	2	assign to $[0, 1]$	20
$\{1, 2\}$	$[0, 1], [1, 2]$	3	cannot fit; reject	35
$\{1, 2\}$	$[0, 1], [1, 2]$	4	assign to $[2, 3]$	35
$\{1, 2, 4\}$	$[0, 1], [1, 2], [2, 3]$	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40. \square

Since there are only n jobs and each job takes one unit of time, it is necessary only to consider the time slots $[i - 1, i]$, $1 \leq i \leq b$, such that $b = \min \{n, \max \{d_i\}\}$. One way to implement the above scheduling rule is to partition the time slots $[i - 1, i]$, $1 \leq i \leq b$, into sets. We use i to represent the time slots $[i - 1, i]$. For any slot i , let n_i be the largest integer such that $n_i \leq i$ and slot n_i is free. To avoid end conditions, we introduce a fictitious slot $[-1, 0]$ which is always free. Two slots i and j are in the same set iff $n_i = n_j$. Clearly, if i and j , $i < j$, are in the same set, then $i, i + 1, i + 2, \dots, j$ are in the same set. Associated with each set k of slots is a value $f(k)$. Then $f(k) = n_i$ for all slots i in set k . Using the set representation of Section 2.5, each set is represented as a tree. The root node identifies the set. The function f is defined only for root nodes. Initially, all slots are free and we have $b + 1$ sets corresponding to the $b + 1$ slots $[i - 1, i]$, $0 \leq i \leq b$. At this time $f(i) = i$, $0 \leq i \leq b$. We use $p(i)$ to link slot i into its set tree. With the conventions for the union and find algorithms of Section 2.5, $p(i) = -1$, $0 \leq i \leq b$, initially. If a job with deadline d is to be scheduled, then we need to find the root of the tree containing the slot $\min\{n, d\}$. If this root is j ,

then $f(j)$ is the nearest free slot, provided $f(j) \neq 0$. Having used this slot, the set with root j should be combined with the set containing slot $f(j) - 1$.

Example 4.4 The trees defined by the $p(i)$'s for the first three iterations in Example 4.3 are shown in Figure 4.4. \square

J	f	trees						job considered	action
		0	1	2	3	4	5		
\emptyset		(-1)	(-1)	(-1)	(-1)	(-1)	(-1)		
		p(0)	p(1)	p(2)	p(3)	p(4)	p(5)		
$\{1\}$	f	0	1		3	4	5	$2, d_1 = 2$	select
		(-1)	(-2) p(1)	(-1)	(-1)	(-1)	(-1)		
		p(0)		(-1)	p(3)	p(4)	p(5)		
				p(2)					
$\{1,2\}$	$f(1)=0$			$f(3)=3$	$f(4)=4$	$f(5)=5$		$3, d_3=1$	reject
				(-1)	(-1)	(-1)			
				p(3)	p(4)	p(5)			

Figure 4.4 Fast job scheduling

The fast algorithm appears as FJS (Algorithm 4.7). Its computing time is readily observed to be $O(n\alpha(2n, n))$ (recall that $\alpha(2n, n)$ is the inverse of Ackermann's function defined in Section 2.5). It needs an additional $2n$ words of space for f and p .

```

1  Algorithm FJS( $d, n, b, j$ )
2  // Find an optimal solution  $J[1 : k]$ . It is assumed that
3  //  $p[1] \geq p[2] \geq \dots \geq p[n]$  and that  $b = \min\{n, \max_i(d[i])\}$ .
4  {
5      // Initially there are  $b + 1$  single node trees.
6      for  $i := 0$  to  $b$  do  $f[i] := i$ ;
7       $k := 0$ ; // Initialize.
8      for  $i := 1$  to  $n$  do
9          { // Use greedy rule.
10              $q := \text{CollapsingFind}(\min(n, d[i]))$ ;
11             if ( $f[q] \neq 0$ ) then
12                 {
13                      $k := k + 1$ ;  $J[k] := i$ ; // Select job  $i$ .
14                      $m := \text{CollapsingFind}(f[q] - 1)$ ;
15                      $\text{WeightedUnion}(m, q)$ ;
16                      $f[q] := f[m]$ ; //  $q$  may be new root.
17                 }
18             }
19         }

```

Algorithm 4.7 Faster algorithm for job sequencing

EXERCISES

1. You are given a set of n jobs. Associated with each job i is a processing time t_i and a deadline d_i by which it must be completed. A feasible schedule is a permutation of the jobs such that if the jobs are processed in that order, then each job finishes by its deadline. Define a greedy schedule to be one in which the jobs are processed in nondecreasing order of deadlines. Show that if there exists a feasible schedule, then all greedy schedules are feasible.
2. [Optimal assignment] Assume there are n workers and n jobs. Let v_{ij} be the value of assigning worker i to job j . An assignment of workers to jobs corresponds to the assignment of 0 or 1 to the variables x_{ij} , $1 \leq i, j \leq n$. Then $x_{ij} = 1$ means worker i is assigned to job j , and $x_{ij} = 0$ means that worker i is not assigned to job j . A valid assignment is one in which each worker is assigned to exactly one job and exactly one worker is assigned to any one job. The value of an assignment is $\sum_i \sum_j v_{ij} x_{ij}$.

For example, assume there are three workers w_1, w_2 , and w_3 and three jobs j_1, j_2 , and j_3 . Let the values of assignment be $v_{11} = 11$, $v_{12} = 5$, $v_{13} = 8$, $v_{21} = 3$, $v_{22} = 7$, $v_{23} = 15$, $v_{31} = 8$, $v_{32} = 12$, and $v_{33} = 9$. Then, a valid assignment is $x_{12} = 1$, $x_{23} = 1$, and $x_{31} = 1$. The rest of the x_{ij} 's are zeros. The value of this assignment is $5 + 15 + 8 = 28$.

An optimal assignment is a valid assignment of maximum value. Write algorithms for two different greedy assignment schemes. One of these assigns a worker to the best possible job. The other assigns to a job the best possible worker. Show that neither of these schemes is guaranteed to yield optimal assignments. Is either scheme always better than the other? Assume $v_{ij} > 0$.

3. (a) What is the solution generated by the function JS when $n = 7$, $(p_1, p_2, \dots, p_7) = (3, 5, 20, 18, 1, 6, 30)$, and $(d_1, d_2, \dots, d_7) = (1, 3, 4, 3, 2, 1, 2)$?
- (b) Show that Theorem 4.3 is true even if jobs have different processing requirements. Associated with job i is a profit $p_i > 0$, a time requirement $t_i > 0$, and a deadline $d_i \geq t_i$.
- (c) Show that for the situation of part (a), the greedy method of this section doesn't necessarily yield an optimal solution.
4. (a) For the job sequencing problem of this section, show that the subset J represents a feasible solution iff the jobs in J can be processed according to the rule: if job i in J hasn't been assigned a processing time, then assign it to the slot $[\alpha - 1, \alpha]$, where α is the least integer r such that $1 \leq r \leq d_i$ and the slot $[\alpha - 1, \alpha]$ is free.
- (b) For the problem instance of Exercise 3(a) draw the trees and give the values of $f(i)$, $0 \leq i \leq n$, after each iteration of the **for** loop of line 8 of Algorithm 4.7.

4.5 MINIMUM-COST SPANNING TREES

Definition 4.1 Let $G = (V, E)$ be an undirected connected graph. A subgraph $t = (V, E')$ of G is a *spanning tree* of G iff t is a tree. \square

Example 4.5 Figure 4.5 shows the complete graph on four nodes together with three of its spanning trees. \square

Spanning trees have many applications. For example, they can be used to obtain an independent set of circuit equations for an electric network. First, a spanning tree for the electric network is obtained. Let B be the set of network edges not in the spanning tree. Adding an edge from B to

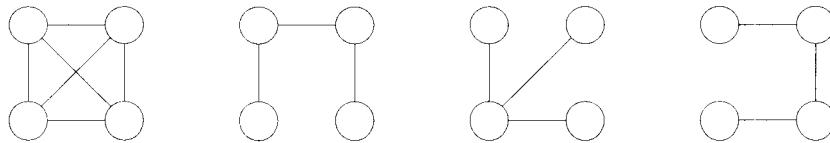


Figure 4.5 An undirected graph and three of its spanning trees

the spanning tree creates a cycle. Kirchoff's second law is used on each cycle to obtain a circuit equation. The cycles obtained in this way are independent (i.e., none of these cycles can be obtained by taking a linear combination of the remaining cycles) as each contains an edge from B that is not contained in any other cycle. Hence, the circuit equations so obtained are also independent. In fact, it can be shown that the cycles obtained by introducing the edges of B one at a time into the resulting spanning tree form a cycle basis, and so all other cycles in the graph can be constructed by taking a linear combination of the cycles in the basis.

Another application of spanning trees arises from the property that a spanning tree is a minimal subgraph G' of G such that $V(G') = V(G)$ and G' is connected. (A minimal subgraph is one with the fewest number of edges.) Any connected graph with n vertices must have at least $n - 1$ edges and all connected graphs with $n - 1$ edges are trees. If the nodes of G represent cities and the edges represent possible communication links connecting two cities, then the minimum number of links needed to connect the n cities is $n - 1$. The spanning trees of G represent all feasible choices.

In practical situations, the edges have weights assigned to them. These weights may represent the cost of construction, the length of the link, and so on. Given such a weighted graph, one would then wish to select cities to have minimum total cost or minimum total length. In either case the links selected have to form a tree (assuming all weights are positive). If this is not so, then the selection of links contains a cycle. Removal of any one of the links on this cycle results in a link selection of less cost connecting all cities. We are therefore interested in finding a spanning tree of G with minimum cost. (The cost of a spanning tree is the sum of the costs of the edges in that tree.) Figure 4.6 shows a graph and one of its minimum-cost spanning trees. Since the identification of a minimum-cost spanning tree involves the selection of a subset of the edges, this problem fits the subset paradigm.

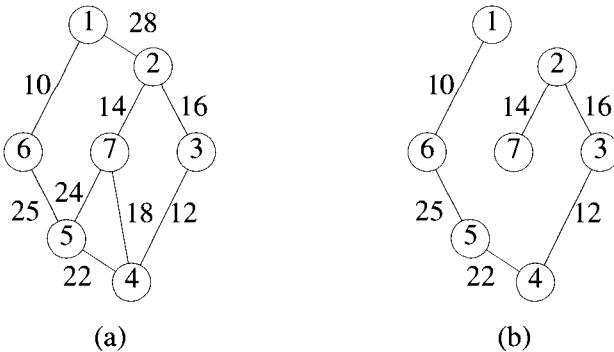


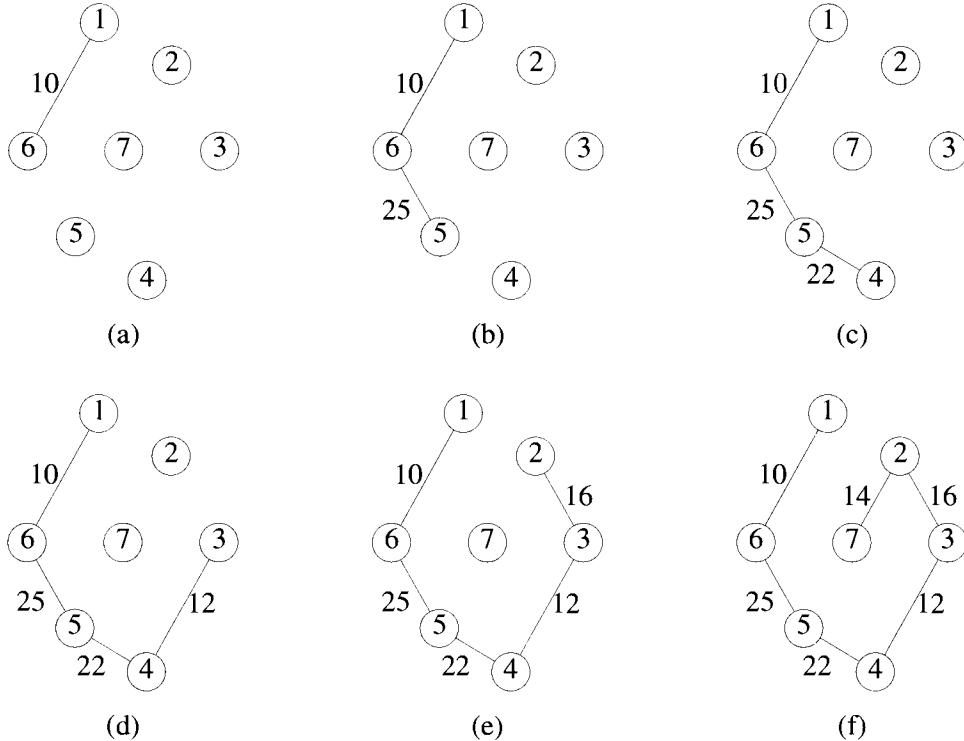
Figure 4.6 A graph and its minimum cost spanning tree

4.5.1 Prim's Algorithm

A greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge. The next edge to include is chosen according to some optimization criterion. The simplest such criterion is to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included. There are two possible ways to interpret this criterion. In the first, the set of edges so far selected form a tree. Thus, if A is the set of edges selected so far, then A forms a tree. The next edge (u, v) to be included in A is a minimum-cost edge not in A with the property that $A \cup \{(u, v)\}$ is also a tree. Exercise 2 shows that this selection criterion results in a minimum-cost spanning tree. The corresponding algorithm is known as Prim's algorithm.

Example 4.6 Figure 4.7 shows the working of Prim's method on the graph of Figure 4.6(a). The spanning tree obtained is shown in Figure 4.6(b) and has a cost of 99. \square

Having seen how Prim's method works, let us obtain a pseudocode algorithm to find a minimum-cost spanning tree using this method. The algorithm will start with a tree that includes only a minimum-cost edge of G . Then, edges are added to this tree one by one. The next edge (i, j) to be added is such that i is a vertex already included in the tree, j is a vertex not yet included, and the cost of (i, j) , $cost[i, j]$, is minimum among all edges (k, l) such that vertex k is in the tree and vertex l is not in the tree. To determine this edge (i, j) efficiently, we associate with each vertex j not yet included in the tree a value $near[j]$. The value $near[j]$ is a vertex in the tree such that $cost[j, near[j]]$ is minimum among all choices for $near[j]$. We define $near[j] = 0$ for all vertices j that are already in the tree. The next edge

**Figure 4.7** Stages in Prim's algorithm

to include is defined by the vertex j such that $\text{near}[j] \neq 0$ (j not already in the tree) and $\text{cost}[j, \text{near}[j]]$ is minimum.

In function `Prim` (Algorithm 4.8), line 9 selects a minimum-cost edge. Lines 10 to 15 initialize the variables so as to represent a tree comprising only the edge (k, l) . In the **for** loop of line 16 the remainder of the spanning tree is built up edge by edge. Lines 18 and 19 select $(j, \text{near}[j])$ as the next edge to include. Lines 23 to 25 update $\text{near}[]$.

The time required by algorithm `Prim` is $O(n^2)$, where n is the number of vertices in the graph G . To see this, note that line 9 takes $O(|E|)$ time and line 10 takes $\Theta(1)$ time. The **for** loop of line 12 takes $\Theta(n)$ time. Lines 18 and 19 and the **for** loop of line 23 require $O(n)$ time. So, each iteration of the **for** loop of line 16 takes $O(n)$ time. The total time for the **for** loop of line 16 is therefore $O(n^2)$. Hence, `Prim` runs in $O(n^2)$ time.

If we store the nodes not yet included in the tree as a red-black tree (see Section 2.4.2), lines 18 and 19 take $O(\log n)$ time. Note that a red-black tree supports the following operations in $O(\log n)$ time: insert, delete (an arbitrary element), find-min, and search (for an arbitrary element). The **for** loop of line 23 has to examine only the nodes adjacent to j . Thus its overall frequency is $O(|E|)$. Updating in lines 24 and 25 also takes $O(\log n)$ time (since an update can be done using a delete and an insertion into the red-black tree). Thus the overall run time is $O((n + |E|) \log n)$.

The algorithm can be speeded a bit by making the observation that a minimum-cost spanning tree includes for each vertex v a minimum-cost edge incident to v . To see this, suppose t is a minimum-cost spanning tree for $G = (V, E)$. Let v be any vertex in t . Let (v, w) be an edge with minimum cost among all edges incident to v . Assume that $(v, w) \notin E(t)$ and $\text{cost}[v, w] < \text{cost}[v, x]$ for all edges $(v, x) \in E(t)$. The inclusion of (v, w) into t creates a unique cycle. This cycle must include an edge (v, x) , $x \neq w$. Removing (v, x) from $E(t) \cup \{(v, w)\}$ breaks this cycle without disconnecting the graph $(V, E(t) \cup \{(v, w)\})$. Hence, $(V, E(t) \cup \{(v, w)\} - \{(v, x)\})$ is also a spanning tree. Since $\text{cost}[v, w] < \text{cost}[v, x]$, this spanning tree has lower cost than t . This contradicts the assumption that t is a minimum-cost spanning tree of G . So, t includes minimum-cost edges as stated above.

From this observation it follows that we can start the algorithm with a tree consisting of any arbitrary vertex and no edge. Then edges can be added one by one. The changes needed are to lines 9 to 17. These lines can be replaced by the lines

```

9'      mincost := 0;
10'     for i := 2 to n do near[i] := 1;
11'           // Vertex 1 is initially in t.
12'     near[1] := 0;
13'-16'    for i := 1 to n - 1 do
17'        { // Find n - 1 edges for t.

```

4.5.2 Kruskal's Algorithm

There is a second possible interpretation of the optimization criteria mentioned earlier in which the edges of the graph are considered in nondecreasing order of cost. This interpretation is that the set t of edges so far selected for the spanning tree be such that it is possible to *complete* t into a tree. Thus t may not be a tree at all stages in the algorithm. In fact, it will generally only be a forest since the set of edges t can be completed into a tree iff there are no cycles in t . We show in Theorem 4.6 that this interpretation of the greedy method also results in a minimum-cost spanning tree. This method is due to Kruskal.

```

1  Algorithm Prim( $E, cost, n, t$ )
2  //  $E$  is the set of edges in  $G$ .  $cost[1 : n, 1 : n]$  is the cost
3  // adjacency matrix of an  $n$  vertex graph such that  $cost[i, j]$  is
4  // either a positive real number or  $\infty$  if no edge  $(i, j)$  exists.
5  // A minimum spanning tree is computed and stored as a set of
6  // edges in the array  $t[1 : n - 1, 1 : 2]$ .  $(t[i, 1], t[i, 2])$  is an edge in
7  // the minimum-cost spanning tree. The final cost is returned.
8  {
9      Let  $(k, l)$  be an edge of minimum cost in  $E$ ;
10      $mincost := cost[k, l];$ 
11      $t[1, 1] := k; t[1, 2] := l;$ 
12     for  $i := 1$  to  $n$  do // Initialize near.
13         if ( $cost[i, l] < cost[i, k]$ ) then  $near[i] := l;$ 
14         else  $near[i] := k;$ 
15      $near[k] := near[l] := 0;$ 
16     for  $i := 2$  to  $n - 1$  do
17         { // Find  $n - 2$  additional edges for  $t$ .
18             Let  $j$  be an index such that  $near[j] \neq 0$  and
19              $cost[j, near[j]]$  is minimum;
20              $t[i, 1] := j; t[i, 2] := near[j];$ 
21              $mincost := mincost + cost[j, near[j]];$ 
22              $near[j] := 0;$ 
23             for  $k := 1$  to  $n$  do // Update near[ ].
24                 if (( $near[k] \neq 0$ ) and ( $cost[k, near[k]] > cost[k, j]$ ))
25                     then  $near[k] := j;$ 
26     }
27     return  $mincost;$ 
28 }
```

Algorithm 4.8 Prim's minimum-cost spanning tree algorithm

Example 4.7 Consider the graph of Figure 4.6(a). We begin with no edges selected. Figure 4.8(a) shows the current graph with no edges selected. Edge (1, 6) is the first edge considered. It is included in the spanning tree being built. This yields the graph of Figure 4.8(b). Next, the edge (3, 4) is selected and included in the tree (Figure 4.8(c)). The next edge to be considered is (2, 7). Its inclusion in the tree being built does not create a cycle, so we get the graph of Figure 4.8(d). Edge (2, 3) is considered next and included in the tree Figure 4.8(e). Of the edges not yet considered, (7, 4) has the least cost. It is considered next. Its inclusion in the tree results in a cycle, so this edge is discarded. Edge (5, 4) is the next edge to be added to the tree being built. This results in the configuration of Figure 4.8(f). The next edge to be considered is the edge (7, 5). It is discarded, as its inclusion creates a cycle. Finally, edge (6, 5) is considered and included in the tree being built. This completes the spanning tree. The resulting tree (Figure 4.6(b)) has cost 99.

□

For clarity, Kruskal's method is written out more formally in Algorithm 4.9. Initially E is the set of all edges in G . The only functions we wish to perform on this set are (1) determine an edge with minimum cost (line 4) and (2) delete this edge (line 5). Both these functions can be performed efficiently if the edges in E are maintained as a sorted sequential list. It is not essential to sort all the edges so long as the next edge for line 4 can be determined easily. If the edges are maintained as a minheap, then the next edge to consider can be obtained in $O(\log |E|)$ time. The construction of the heap itself takes $O(|E|)$ time.

To be able to perform step 6 efficiently, the vertices in G should be grouped together in such a way that one can easily determine whether the vertices v and w are already connected by the earlier selection of edges. If they are, then the edge (v, w) is to be discarded. If they are not, then (v, w) is to be added to t . One possible grouping is to place all vertices in the same connected component of t into a set (all connected components of t will also be trees). Then, two vertices v and w are connected in t iff they are in the same set. For example, when the edge (2, 6) is to be considered, the sets are $\{1, 2\}$, $\{3, 4, 6\}$, and $\{5\}$. Vertices 2 and 6 are in different sets so these sets are combined to give $\{1, 2, 3, 4, 6\}$ and $\{5\}$. The next edge to be considered is (1, 4). Since vertices 1 and 4 are in the same set, the edge is rejected. The edge (3, 5) connects vertices in different sets and results in the final spanning tree. Using the set representation and the union and find algorithms of Section 2.5, we can obtain an efficient (almost linear) implementation of line 6. The computing time is, therefore, determined by the time for lines 4 and 5, which in the worst case is $O(|E| \log |E|)$.

If the representations discussed above are used, then the pseudocode of Algorithm 4.10 results. In line 6 an initial heap of edges is constructed. In line 7 each vertex is assigned to a distinct set (and hence to a distinct tree). The set t is the set of edges to be included in the minimum-cost spanning

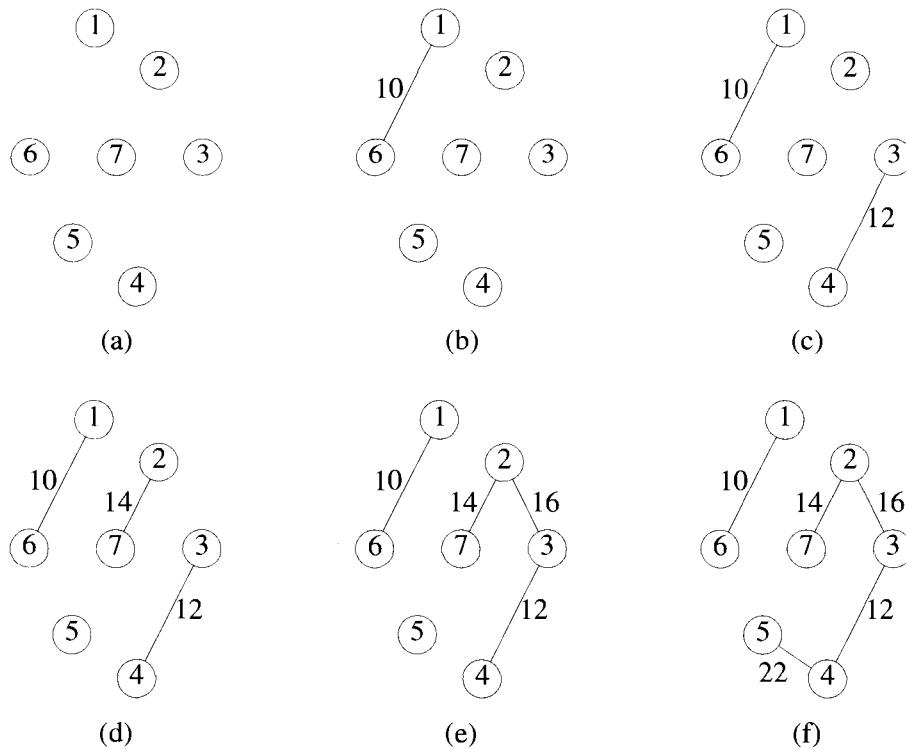


Figure 4.8 Stages in Kruskal's algorithm

tree and i is the number of edges in t . The set t can be represented as a sequential list using a two-dimensional array $t[1 : n - 1, 1 : 2]$. Edge (u, v) can be added to t by the assignments $t[i, 1] := u$; and $t[i, 2] := v$. In the **while** loop of line 10, edges are removed from the heap one by one in nondecreasing order of cost. Line 14 determines the sets containing u and v . If $j \neq k$, then vertices u and v are in different sets (and so in different trees) and edge (u, v) is included into t . The sets containing u and v are combined (line 20). If $u = v$, the edge (u, v) is discarded as its inclusion into t would create a cycle. Line 23 determines whether a spanning tree was found. It follows that $i \neq n - 1$ iff the graph G is not connected. One can verify that the computing time is $O(|E| \log |E|)$, where E is the edge set of G .

Theorem 4.6 Kruskal's algorithm generates a minimum-cost spanning tree for every connected undirected graph G .

```

1    $t := \emptyset;$ 
2   while (( $t$  has less than  $n - 1$  edges) and ( $E \neq \emptyset$ )) do
3   {
4       Choose an edge  $(v, w)$  from  $E$  of lowest cost;
5       Delete  $(v, w)$  from  $E$ ;
6       if  $(v, w)$  does not create a cycle in  $t$  then add  $(v, w)$  to  $t$ ;
7       else discard  $(v, w)$ ;
8   }

```

Algorithm 4.9 Early form of minimum-cost spanning tree algorithm due to Kruskal

```

1   Algorithm Kruskal( $E, cost, n, t$ )
2   //  $E$  is the set of edges in  $G$ .  $G$  has  $n$  vertices.  $cost[u, v]$  is the
3   // cost of edge  $(u, v)$ .  $t$  is the set of edges in the minimum-cost
4   // spanning tree. The final cost is returned.
5   {
6       Construct a heap out of the edge costs using Heapify;
7       for  $i := 1$  to  $n$  do  $parent[i] := -1$ ;
8       // Each vertex is in a different set.
9        $i := 0$ ;  $mincost := 0.0$ ;
10      while  $((i < n - 1) \text{ and } (\text{heap not empty}))$  do
11      {
12          Delete a minimum cost edge  $(u, v)$  from the heap
13          and reheapify using Adjust;
14           $j := \text{Find}(u)$ ;  $k := \text{Find}(v)$ ;
15          if  $(j \neq k)$  then
16          {
17               $i := i + 1$ ;
18               $t[i, 1] := u$ ;  $t[i, 2] := v$ ;
19               $mincost := mincost + cost[u, v]$ ;
20              Union( $j, k$ );
21          }
22      }
23      if  $(i \neq n - 1)$  then write ("No spanning tree");
24      else return  $mincost$ ;
25  }

```

Algorithm 4.10 Kruskal's algorithm

Proof: Let G be any undirected connected graph. Let t be the spanning tree for G generated by Kruskal's algorithm. Let t' be a minimum-cost spanning tree for G . We show that both t and t' have the same cost.

Let $E(t)$ and $E(t')$ respectively be the edges in t and t' . If n is the number of vertices in G , then both t and t' have $n - 1$ edges. If $E(t) = E(t')$, then t is clearly of minimum cost. If $E(t) \neq E(t')$, then let q be a minimum-cost edge such that $q \in E(t)$ and $q \notin E(t')$. Clearly, such a q must exist. The inclusion of q into t' creates a unique cycle (Exercise 5). Let q, e_1, e_2, \dots, e_k be this unique cycle. At least one of the e_i 's, $1 \leq i \leq k$, is not in $E(t)$ as otherwise t would also contain the cycle q, e_1, e_2, \dots, e_k . Let e_j be an edge on this cycle such that $e_j \notin E(t)$. If e_j is of lower cost than q , then Kruskal's algorithm will consider e_j before q and include e_j into t . To see this, note that all edges in $E(t)$ of cost less than the cost of q are also in $E(t')$ and do not form a cycle with e_j . So $\text{cost}(e_j) \geq \text{cost}(q)$.

Now, reconsider the graph with edge set $E(t') \cup \{q\}$. Removal of any edge on the cycle q, e_1, e_2, \dots, e_k will leave behind a tree t'' (Exercise 5). In particular, if we delete the edge e_j , then the resulting tree t'' will have a cost no more than the cost of t' (as $\text{cost}(e_j) \geq \text{cost}(q)$). Hence, t'' is also a minimum-cost tree.

By repeatedly using the transformation described above, tree t' can be transformed into the spanning tree t without any increase in cost. Hence, t is a minimum-cost spanning tree. \square

4.5.3 An Optimal Randomized Algorithm (*)

Any algorithm for finding the minimum-cost spanning tree of a given graph $G(V, E)$ will have to spend $\Omega(|V| + |E|)$ time in the worst case, since it has to examine each node and each edge at least once before determining the correct answer. A randomized Las Vegas algorithm that runs in time $\tilde{O}(|V| + |E|)$ can be devised as follows: (1) Randomly sample m edges from G (for some suitable m). (2) Let G' be the induced subgraph; that is, G' has V as its node set and the sampled edges in its edge set. The subgraph G' need not be connected. Recursively find a minimum-cost spanning tree for each component of G' . Let F be the resultant *minimum-cost spanning forest* of G' . (3) Using F , eliminate certain edges (called the *F-heavy edges*) of G that cannot possibly be in a minimum-cost spanning tree. Let G'' be the graph that results from G after elimination of the *F-heavy edges*. (4) Recursively find a minimum-cost spanning tree for G'' . This will also be a minimum-cost spanning tree for G .

Steps 1 to 3 are useful in reducing the number of edges in G . The algorithm can be speeded up further if we can reduce the number of nodes in the input graph as well. Such a node elimination can be effected using the *Boruvka steps*. In a Boruvka step, for each node, an incident edge with minimum weight is chosen. For example in Figure 4.9(a), the edge (1, 3) is

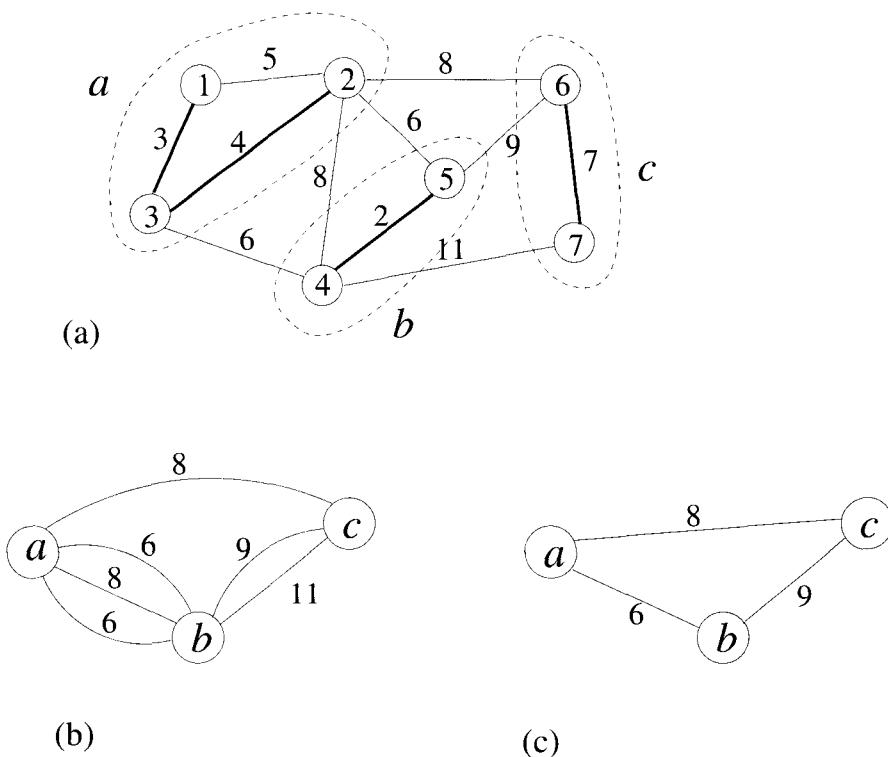
chosen for node 1, the edge $(6, 7)$ is chosen for node 7, and so on. All the chosen edges are shown with thick lines. The connected components of the induced graph are found. In the example of Figure 4.9(a), the nodes 1, 2, and 3 form one component, the nodes 4 and 5 form a second component, and the nodes 6 and 7 form another component. Replace each component with a single node. The component with nodes 1, 2, and 3 is replaced with the node a . The other two components are replaced with the nodes b and c , respectively. Edges within the individual components are thrown away. The resultant graph is shown in Figure 4.9(b). In this graph keep only an edge of minimum weight between any two nodes. Delete any isolated nodes.

Since an edge is chosen for every node, the number of nodes after one Boruvka step reduces by a factor of at least two. A minimum-cost spanning tree for the reduced graph can be extended easily to get a minimum-cost spanning tree for the original graph. If E' is the set of edges in the minimum-cost spanning tree of the reduced graph, we simply include into E' the edges chosen in the Boruvka step to obtain the minimum-cost spanning tree edges for the original graph. In the example of Figure 4.9, a minimum-cost spanning tree for (c) will consist of the edges (a, b) and (b, c) . Thus a minimum-cost spanning tree for the graph of (a) will have the edges: $(1, 3), (3, 2), (4, 5), (6, 7), (3, 4)$, and $(2, 6)$. More details of the algorithms are given below.

Definition 4.2 Let F be a forest that forms a subgraph of a given weighted graph $G(V, E)$. If u and v are any two nodes in F , let $F(u, v)$ denote the path (if any) connecting u and v in F and let $Fcost(u, v)$ denote the maximum weight of any edge in the path $F(u, v)$. If there is no path between u and v in F , $Fcost(u, v)$ is taken to be ∞ . Any edge (x, y) of G is said to be F -heavy if $cost[x, y] > Fcost(x, y)$ and F -light otherwise. \square

Note that all the edges of F are F -light. Also, any F -heavy edge cannot belong to a minimum-cost spanning tree of G . The proof of this is left as an exercise. The randomized algorithm applies two Boruvka steps to reduce the number of nodes in the input graph. Next, it samples the edges of G and processes them to eliminate a constant fraction of them. A minimum-cost spanning tree for the resultant reduced graph is recursively computed. From this tree, a spanning tree for G is obtained. A detailed description of the algorithm appears as Algorithm 4.11.

Lemma 4.3 states that Step 4 can be completed in time $O(|V| + |E|)$. The proof of this can be found in the references supplied at the end of this chapter. Step 1 takes $O(|V| + |E|)$ time and step 2 takes $O(|E|)$ time. Step 6 takes $O(|E|)$ time as well. The time taken in all the recursive calls in steps 3 and 5 can be shown to be $O(|V| + |E|)$. For a proof, see the references at the end of the chapter. A crucial fact that is used in the proof is that both the number of nodes and the number of edges are reduced by a constant factor, with high probability, in each level of recursion.

**Figure 4.9** A Boruvka step

Lemma 4.3 Let $G(V, E)$ be any weighted graph and let F be a subgraph of G that forms a forest. Then, all the F -heavy edges of G can be identified in time $O(|V| + |E|)$. \square

Theorem 4.7 A minimum-weight spanning tree for any given weighted graph can be computed in time $\tilde{O}(|V| + |E|)$. \square

EXERCISES

1. Compute a minimum cost spanning tree for the graph of Figure 4.10 using (a) Prim's algorithm and (b) Kruskal's algorithm.
2. Prove that Prim's method of this section generates minimum-cost spanning trees.

Step 1. Apply two Boruvka steps. At the end, the number of nodes will have decreased by a factor at least 4. Let the resultant graph be $\tilde{G}(\tilde{V}, \tilde{E})$.

Step 2. Form a subgraph $G'(V', E')$ of \tilde{G} , where each edge of \tilde{G} is chosen randomly to be in E' with probability $\frac{1}{2}$. The expected number of edges in E' is $\frac{|\tilde{E}|}{2}$.

Step 3. Recursively find a minimum-cost spanning forest F for G' .

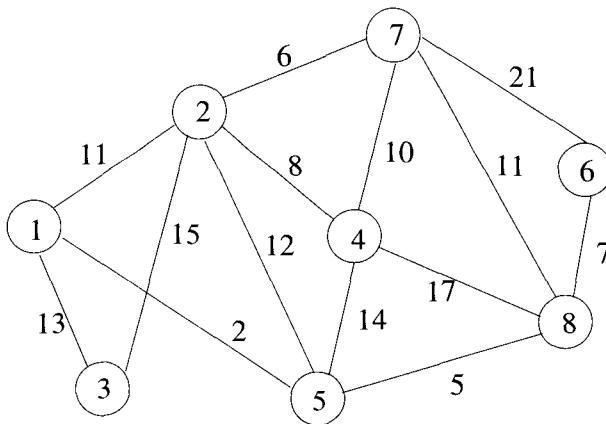
Step 4. Eliminate all the F -heavy edges from \tilde{G} . With high probability, at least a constant fraction of the edges of \tilde{G} will be eliminated. Let G'' be the resultant graph.

Step 5. Compute a minimum-cost spanning tree (call it T'') for G'' recursively. The tree T'' will also be a minimum-cost spanning tree for \tilde{G} .

Step 6. Return the edges of T'' together with the edges chosen in the Boruvka steps of step 1. These are the edges of a minimum-cost spanning tree for G .

Algorithm 4.11 An optimal randomized algorithm

3. (a) Rewrite Prim's algorithm under the assumption that the graphs are represented by adjacency lists.
 (b) Program and run the above version of Prim's algorithm against Algorithm 4.9. Compare the two on a representative set of graphs.
 (c) Analyze precisely the computing time and space requirements of your new version of Prim's algorithm using adjacency lists.
4. Program and run Kruskal's algorithm, described in Algorithm 4.10. You will have to modify functions `Heapify` and `Adjust` of Chapter 2. Use the same test data you devised to test Prim's algorithm in Exercise 3.
5. (a) Show that if t is a spanning tree for the undirected graph G , then the addition of an edge q , $q \notin E(t)$ and $q \in E(G)$, to t creates a unique cycle.

**Figure 4.10** Graph for Exercise 1

- (b) Show that if any of the edges on this unique cycle is deleted from $E(t) \cup \{q\}$, then the remaining edges form a spanning tree of G .
- 6. In Figure 4.9, find a minimum-cost spanning tree for the graph of part (c) and extend the tree to obtain a minimum cost spanning tree for the graph of part (a). Verify the correctness of your answer by applying either Prim's algorithm or Kruskal's algorithm on the graph of part (a).
- 7. Let $G(V, E)$ be any weighted connected graph.
 - (a) If C is any cycle of G , then show that the heaviest edge of C cannot belong to a minimum-cost spanning tree of G .
 - (b) Assume that F is a forest that is a subgraph of G . Show that any F -heavy edge of G cannot belong to a minimum-cost spanning tree of G .
- 8. By considering the complete graph with n vertices, show that the number of spanning trees in an n vertex graph can be greater than $2^{n-1} - 2$.

4.6 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 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 $I = i_1, i_2, \dots, i_n$, 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. This problem fits the ordering paradigm. Minimizing the MRT is equivalent to minimizing $d(I) = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} l_{i_k}$.

Example 4.8 Let $n = 3$ and $(l_1, l_2, l_3) = (5, 10, 3)$. There are $n! = 6$ possible orderings. These orderings and their respective d values are:

ordering I	$d(I)$
1, 2, 3	$5 + 5 + 10 + 5 + 10 + 3 = 38$
1, 3, 2	$5 + 5 + 3 + 5 + 3 + 10 = 31$
2, 1, 3	$10 + 10 + 5 + 10 + 5 + 3 = 43$
2, 3, 1	$10 + 10 + 3 + 10 + 3 + 5 = 41$
3, 1, 2	$3 + 3 + 5 + 3 + 5 + 10 = 29$
3, 2, 1	$3 + 3 + 10 + 3 + 10 + 5 = 34$

The optimal ordering is 3, 1, 2. □

A greedy approach to building the required permutation would choose the next program on the basis of some optimization measure. One possible measure would be the d value of the permutation constructed so far. The next program to be stored on the tape would be one that minimizes the increase in d . If we have already constructed the permutation i_1, i_2, \dots, i_r , then appending program j gives the permutation $i_1, i_2, \dots, i_r, i_{r+1} = j$. This increases the d value by $\sum_{1 \leq k \leq r} l_{i_k} + l_j$. Since $\sum_{1 \leq k \leq r} l_{i_k}$ is fixed and independent of j , we trivially observe that the increase in d is minimized if the next program chosen is the one with the least length from among the remaining programs.

The greedy algorithm resulting from the above discussion is so simple that we won't bother to write it out. The greedy method simply requires us to store the programs in nondecreasing order of their lengths. This ordering can be carried out in $O(n \log n)$ time using an efficient sorting algorithm (e.g., heap sort from Chapter 2). For the programs of Example 4.8, note that the permutation that yields an optimal solution is the one in which the programs are in nondecreasing order of their lengths. Theorem 4.8 shows that the MRT is minimized when programs are stored in this order.

Theorem 4.8 If $l_1 \leq l_2 \leq \dots \leq l_n$, then the ordering $i_j = j, 1 \leq j \leq n$, minimizes

$$\sum_{k=1}^n \sum_{j=1}^k l_{i_j}$$

over all possible permutations of the i_j .

Proof: Let $I = i_1, i_2, \dots, i_n$ be any permutation of the index set $\{1, 2, \dots, n\}$. Then

$$d(I) = \sum_{k=1}^n \sum_{j=1}^k l_{i_j} = \sum_{k=1}^n (n - k + 1)l_{i_k}$$

If there exist a and b such that $a < b$ and $l_{i_a} > l_{i_b}$, then interchanging i_a and i_b results in a permutation I' with

$$d(I') = \left[\sum_{\substack{k \\ k \neq a \\ k \neq b}}^n (n - k + 1)l_{i_k} \right] + (n - a + 1)l_{i_b} + (n - b + 1)l_{i_a}$$

Subtracting $d(I')$ from $d(I)$, we obtain

$$\begin{aligned} d(I) - d(I') &= (n - a + 1)(l_{i_a} - l_{i_b}) + (n - b + 1)(l_{i_b} - l_{i_a}) \\ &= (b - a)(l_{i_a} - l_{i_b}) \\ &> 0 \end{aligned}$$

Hence, no permutation that is not in nondecreasing order of the l_i 's can have minimum d . It is easy to see that all permutations in nondecreasing order of the l_i 's have the same d value. Hence, the ordering defined by $i_j = j, 1 \leq j \leq n$, minimizes the d value. \square

The tape storage problem can be extended to several tapes. If there are $m > 1$ tapes, T_0, \dots, T_{m-1} , then the programs are to be distributed over these tapes. For each tape a storage permutation is to be provided. If I_j is the storage permutation for the subset of programs on tape j , then $d(I_j)$ is as defined earlier. The *total retrieval time* (TD) is $\sum_{0 \leq j \leq m-1} d(I_j)$. The objective is to store the programs in such a way as to minimize TD .

The obvious generalization of the solution for the one-tape case is to consider the programs in nondecreasing order of l_i 's. The program currently

```

1  Algorithm Store( $n, m$ )
2  //  $n$  is the number of programs and  $m$  the number of tapes.
3  {
4       $j := 0$ ; // Next tape to store on
5      for  $i := 1$  to  $n$  do
6      {
7          write ("append program",  $i$ ,
8                  "to permutation for tape",  $j$ );
9           $j := (j + 1) \bmod m$ ;
10     }
11 }

```

Algorithm 4.12 Assigning programs to tapes

being considered is placed on the tape that results in the minimum increase in TD . This tape will be the one with the least amount of tape used so far. If there is more than one tape with this property, then the one with the smallest index can be used. If the jobs are initially ordered so that $l_1 \leq l_2 \leq \dots \leq l_n$, then the first m programs are assigned to tapes T_0, \dots, T_{m-1} respectively. The next m programs will be assigned to tapes T_0, \dots, T_{m-1} respectively. The general rule is that program i is stored on tape $T_{i \bmod m}$. On any given tape the programs are stored in nondecreasing order of their lengths. Algorithm 4.12 presents this rule in pseudocode. It assumes that the programs are ordered as above. It has a computing time of $\Theta(n)$ and does not need to know the program lengths. Theorem 4.9 proves that the resulting storage pattern is optimal.

Theorem 4.9 If $l_1 \leq l_2 \leq \dots \leq l_n$, then Algorithm 4.12 generates an optimal storage pattern for m tapes.

Proof: In any storage pattern for m tapes, let r_i be one greater than the number of programs following program i on its tape. Then the total retrieval time TD is given by

$$TD = \sum_{i=1}^n r_i l_i$$

In any given storage pattern, for any given n , there can be at most m programs for which $r_i = j$. From Theorem 4.8 it follows that TD is minimized if the m longest programs have $r_i = 1$, the next m longest programs have

$r_i = 2$, and so on. When programs are ordered by length, that is, $l_1 \leq l_2 \leq \dots \leq l_n$, then this minimization criteria is satisfied if $r_i = \lceil (n - i + 1)/m \rceil$. Observe that Algorithm 4.12 results in a storage pattern with these r_i 's. \square

The proof of Theorem 4.9 shows that there are many storage patterns that minimize TD . If we compute $r_i = \lceil (n - i + 1)/m \rceil$ for each program i , then so long as all programs with the same r_i are stored on different tapes and have $r_i - 1$ programs following them, TD is the same. If n is a multiple of m , then there are at least $(m!)^{n/m}$ storage patterns that minimize TD . Algorithm 4.12 produces one of these.

EXERCISES

1. Find an optimal placement for 13 programs on three tapes T_0, T_1 , and T_2 , where the programs are of lengths 12, 5, 8, 32, 7, 5, 18, 26, 4, 3, 11, 10, and 6.
2. Show that replacing the code of Algorithm 4.12 by

```
for i := 1 to n do
    write ("append program", i, "to permutation for
tape", (i - 1) mod m);
```

does not affect the output.

3. Let P_1, P_2, \dots, P_n be a set of n programs that are to be stored on a tape of length l . Program P_i requires a_i amount of tape. If $\sum a_i \leq l$, then clearly all the programs can be stored on the tape. So, assume $\sum a_i > l$. The problem is to select a maximum subset Q of the programs for storage on the tape. (A maximum subset is one with the maximum number of programs in it). A greedy algorithm for this problem would build the subset Q by including programs in nondecreasing order of a_i .
 - (a) Assume the P_i are ordered such that $a_1 \leq a_2 \leq \dots \leq a_n$. Write a function for the above strategy. Your function should output an array $s[1 : n]$ such that $s[i] = 1$ if P_i is in Q and $s[i] = 0$ otherwise.
 - (b) Show that this strategy always finds a maximum subset Q such that $\sum_{P_i \in Q} a_i \leq l$.
 - (c) Let Q be the subset obtained using the above greedy strategy. How small can the tape utilization ratio $(\sum_{P_i \in Q} a_i)/l$ get?
 - (d) Suppose the objective now is to determine a subset of programs that maximizes the tape utilization ratio. A greedy approach

would be to consider programs in nonincreasing order of a_i . If there is enough space left on the tape for P_i , then it is included in Q . Assume the programs are ordered so that $a_1 \geq a_2 \geq \dots \geq a_n$. Write a function incorporating this strategy. What is its time and space complexity?

- (e) Show that the strategy of part (d) doesn't necessarily yield a subset that maximizes $(\sum_{P_i \in Q} a_i)/l$. How small can this ratio get? Prove your bound.
- 4. Assume n programs of lengths l_1, l_2, \dots, l_n are to be stored on a tape. Program i is to be retrieved with frequency f_i . If the programs are stored in the order i_1, i_2, \dots, i_n , the *expected retrieval time* (ERT) is

$$\left[\sum_j (f_{i_j} \sum_{k=1}^j l_{i_k}) \right] / \sum f_i$$

- (a) Show that storing the programs in nondecreasing order of l_i does not necessarily minimize the ERT.
- (b) Show that storing the programs in nonincreasing order of f_i does not necessarily minimize the ERT.
- (c) Show that the ERT is minimized when the programs are stored in nonincreasing order of f_i/l_i .
- 5. Consider the tape storage problem of this section. Assume that two tapes T_1 and T_2 , are available and we wish to distribute n given programs of lengths l_1, l_2, \dots, l_n onto these two tapes in such a manner that the maximum retrieval time is minimized. That is, if A and B are the sets of programs on the tapes T_1 and T_2 respectively, then we wish to choose A and B such that $\max \{ \sum_{i \in A} l_i, \sum_{i \in B} l_i \}$ is minimized. A possible greedy approach to obtaining A and B would be to start with A and B initially empty. Then consider the programs one at a time. The program currently being considered is assigned to set A if $\sum_{i \in A} l_i = \min \{ \sum_{i \in A} l_i, \sum_{i \in B} l_i \}$; otherwise it is assigned to B . Show that this does not guarantee optimal solutions even if $l_1 \leq l_2 \leq \dots \leq l_n$. Show that the same is true if we require $l_1 \geq l_2 \geq \dots \geq l_n$.

4.7 OPTIMAL MERGE PATTERNS

In Section 3.4 we saw that two sorted files containing n and m records respectively could be merged together to obtain one sorted file in time $O(n + m)$. When more than two sorted files are to be merged together, the merge can be accomplished by repeatedly merging sorted files in pairs. Thus, if

files x_1, x_2, x_3 , and x_4 are to be merged, we could first merge x_1 and x_2 to get a file y_1 . Then we could merge y_1 and x_3 to get y_2 . Finally, we could merge y_2 and x_4 to get the desired sorted file. Alternatively, we could first merge x_1 and x_2 getting y_1 , then merge x_3 and x_4 and get y_2 , and finally merge y_1 and y_2 and get the desired sorted file. Given n sorted files, there are many ways in which to pairwise merge them into a single sorted file. Different pairings require differing amounts of computing time. The problem we address ourselves to now is that of determining an optimal way (one requiring the fewest comparisons) to pairwise merge n sorted files. Since this problem calls for an ordering among the pairs to be merged, it fits the ordering paradigm.

Example 4.9 The files x_1, x_2 , and x_3 are three sorted files of length 30, 20, and 10 records each. Merging x_1 and x_2 requires 50 record moves. Merging the result with x_3 requires another 60 moves. The total number of record moves required to merge the three files this way is 110. If, instead, we first merge x_2 and x_3 (taking 30 moves) and then x_1 (taking 60 moves), the total record moves made is only 90. Hence, the second merge pattern is faster than the first. \square

A greedy attempt to obtain an optimal merge pattern is easy to formulate. Since merging an n -record file and an m -record file requires possibly $n + m$ record moves, the obvious choice for a selection criterion is: at each step merge the two smallest size files together. Thus, if we have five files (x_1, \dots, x_5) with sizes (20, 30, 10, 5, 30), our greedy rule would generate the following merge pattern: merge x_4 and x_3 to get z_1 ($|z_1| = 15$), merge z_1 and x_1 to get z_2 ($|z_2| = 35$), merge x_2 and x_5 to get z_3 ($|z_3| = 60$), and merge z_2 and z_3 to get the answer z_4 . The total number of record moves is 205. One can verify that this is an optimal merge pattern for the given problem instance.

The merge pattern such as the one just described will be referred to as a *two-way merge pattern* (each merge step involves the merging of two files). The two-way merge patterns can be represented by binary merge trees. Figure 4.11 shows a binary merge tree representing the optimal merge pattern obtained for the above five files. The leaf nodes are drawn as squares and represent the given five files. These nodes are called *external nodes*. The remaining nodes are drawn as circles and are called *internal nodes*. Each internal node has exactly two children, and it represents the file obtained by merging the files represented by its two children. The number in each node is the length (i.e., the number of records) of the file represented by that node.

The external node x_4 is at a distance of 3 from the root node z_4 (a node at level i is at a distance of $i - 1$ from the root). Hence, the records of file x_4 are moved three times, once to get z_1 , once again to get z_2 , and finally one more time to get z_4 . If d_i is the distance from the root to the external

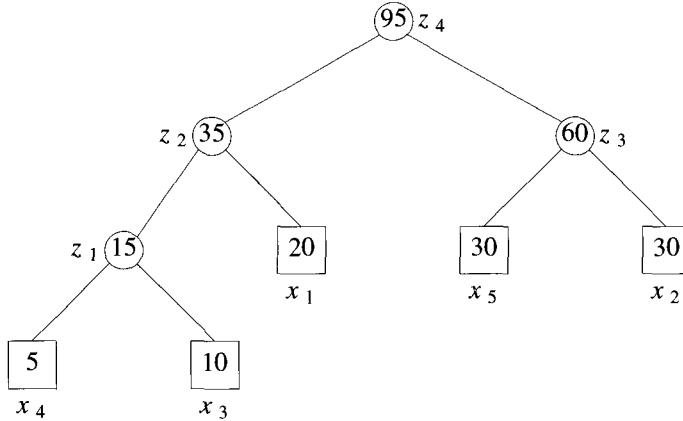


Figure 4.11 Binary merge tree representing a merge pattern

node for file x_i and q_i , the length of x_i is then the total number of record moves for this binary merge tree is

$$\sum_{i=1}^n d_i q_i$$

This sum is called the *weighted external path length* of the tree.

An optimal two-way merge pattern corresponds to a binary merge tree with minimum weighted external path length. The function `Tree` of Algorithm 4.13 uses the greedy rule stated earlier to obtain a two-way merge tree for n files. The algorithm has as input a list `list` of n trees. Each node in a tree has three fields, `lchild`, `rchild`, and `weight`. Initially, each tree in `list` has exactly one node. This node is an external node and has `lchild` and `rchild` fields zero whereas `weight` is the length of one of the n files to be merged. During the course of the algorithm, for any tree in `list` with root node t , $t \rightarrow \text{weight}$ is the length of the merged file it represents ($t \rightarrow \text{weight}$ equals the sum of the lengths of the external nodes in tree t). Function `Tree` uses two functions, `Least(list)` and `Insert(list, t)`. `Least(list)` finds a tree in `list` whose root has least `weight` and returns a pointer to this tree. This tree is removed from `list`. `Insert(list, t)` inserts the tree with root t into `list`. Theorem 4.10 shows that `Tree` (Algorithm 4.13) generates an optimal two-way merge tree.

```

treenode = record {
    treenode * lchild; treenode * rchild;
    integer weight;
};

1 Algorithm Tree( $n$ )
2 //  $list$  is a global list of  $n$  single node
3 // binary trees as described above.
4 {
5     for  $i := 1$  to  $n - 1$  do
6     {
7          $pt := \text{new treenode}$ ; // Get a new tree node.
8         ( $pt \rightarrow lchild$ ) := Least( $list$ ); // Merge two trees with
9         ( $pt \rightarrow rchild$ ) := Least( $list$ ); // smallest lengths.
10        ( $pt \rightarrow weight$ ) := (( $pt \rightarrow lchild$ )  $\rightarrow weight$ )
11            + (( $pt \rightarrow rchild$ )  $\rightarrow weight$ );
12        Insert( $list$ ,  $pt$ );
13    }
14    return Least( $list$ ); // Tree left in  $list$  is the merge tree.
15 }
```

Algorithm 4.13 Algorithm to generate a two-way merge tree

Example 4.10 Let us see how algorithm Tree works when $list$ initially represents six files with lengths $(2, 3, 5, 7, 9, 13)$. Figure 4.12 shows $list$ at the end of each iteration of the **for** loop. The binary merge tree that results at the end of the algorithm can be used to determine which files are merged. Merging is performed on those files which are lowest (have the greatest depth) in the tree. \square

The main **for** loop in Algorithm 4.13 is executed $n - 1$ times. If $list$ is kept in nondecreasing order according to the $weight$ value in the roots, then $\text{Least}(list)$ requires only $O(1)$ time and $\text{Insert}(list, t)$ can be done in $O(n)$ time. Hence the total time taken is $O(n^2)$. In case $list$ is represented as a minheap in which the root value is less than or equal to the values of its children (Section 2.4), then $\text{Least}(list)$ and $\text{Insert}(list, t)$ can be done in $O(\log n)$ time. In this case the computing time for Tree is $O(n \log n)$. Some speedup may be obtained by combining the Insert of line 12 with the Least of line 9.

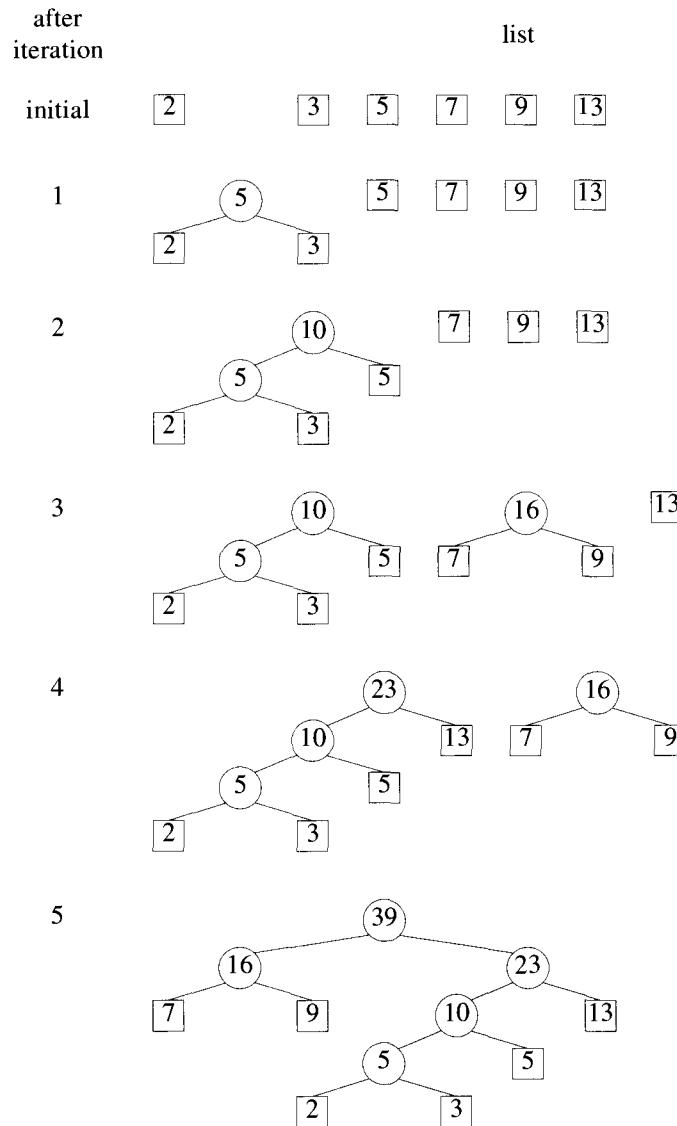
Theorem 4.10 If *list* initially contains $n \geq 1$ single node trees with *weight* values (q_1, q_2, \dots, q_n) , then algorithm `Tree` generates an optimal two-way merge tree for n files with these lengths.

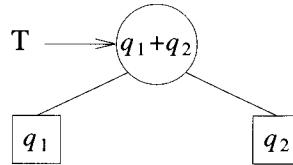
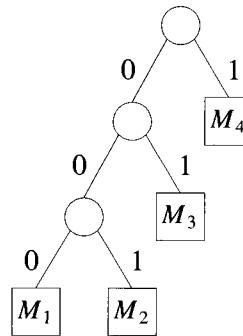
Proof: The proof is by induction on n . For $n = 1$, a tree with no internal nodes is returned and this tree is clearly optimal. For the induction hypothesis, assume the algorithm generates an optimal two-way merge tree for all (q_1, q_2, \dots, q_m) , $1 \leq m < n$. We show that the algorithm also generates optimal trees for all (q_1, q_2, \dots, q_n) . Without loss of generality, we can assume that $q_1 \leq q_2 \leq \dots \leq q_n$ and q_1 and q_2 are the values of the *weight* fields of the trees found by algorithm `Least` in lines 8 and 9 during the first iteration of the `for` loop. Now, the subtree T of Figure 4.13 is created. Let T' be an optimal two-way merge tree for (q_1, q_2, \dots, q_n) . Let p be an internal node of maximum distance from the root. If the children of p are not q_1 and q_2 , then we can interchange the present children with q_1 and q_2 without increasing the weighted external path length of T' . Hence, T is also a subtree in an optimal merge tree. If we replace T in T' by an external node with weight $q_1 + q_2$, then the resulting tree T'' is an optimal merge tree for $(q_1 + q_2, q_3, \dots, q_n)$. From the induction hypothesis, after replacing T by the external node with value $q_1 + q_2$, function `Tree` proceeds to find an optimal merge tree for $(q_1 + q_2, q_3, \dots, q_n)$. Hence, `Tree` generates an optimal merge tree for (q_1, q_2, \dots, q_n) . \square

The greedy method to generate merge trees also works for the case of k -ary merging. In this case the corresponding merge tree is a k -ary tree. Since all internal nodes must have degree k , for certain values of n there is no corresponding k -ary merge tree. For example, when $k = 3$, there is no k -ary merge tree with $n = 2$ external nodes. Hence, it is necessary to introduce a certain number of dummy external nodes. Each dummy node is assigned a q_i of zero. This dummy value does not affect the weighted external path length of the resulting k -ary tree. Exercise 2 shows that a k -ary tree with all internal nodes having degree k exists only when the number of external nodes n satisfies the equality $n \bmod(k-1) = 1$. Hence, at most $k-2$ dummy nodes have to be added. The greedy rule to generate optimal merge trees is: at each step choose k subtrees with least length for merging. Exercise 3 proves the optimality of this rule.

Huffman Codes

Another application of binary trees with minimal weighted external path length is to obtain an optimal set of codes for messages M_1, \dots, M_{n+1} . Each code is a binary string that is used for transmission of the corresponding message. At the receiving end the code is decoded using a decode tree. A decode tree is a binary tree in which external nodes represent messages.

**Figure 4.12** Trees in *list* of Tree for Example 4.10

**Figure 4.13** The simplest binary merge tree**Figure 4.14** Huffman codes

The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node. For example, if we interpret a zero as a left branch and a one as a right branch, then the decode tree of Figure 4.14 corresponds to codes 000, 001, 01, and 1 for messages M_1 , M_2 , M_3 , and M_4 respectively. These codes are called Huffman codes. The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node. If q_i is the relative frequency with which message M_i will be transmitted, then the expected decode time is $\sum_{1 \leq i \leq n+1} q_i d_i$, where d_i is the distance of the external node for message M_i from the root node. The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length! Note that $\sum_{1 \leq i \leq n+1} q_i d_i$ is also the expected length of a transmitted message. Hence the code that minimizes expected decode time also minimizes the expected length of a message.

EXERCISES

1. Find an optimal binary merge pattern for ten files whose lengths are 28, 32, 12, 5, 84, 53, 91, 35, 3, and 11.
2. (a) Show that if all internal nodes in a tree have degree k , then the number n of external nodes is such that $n \bmod (k - 1) = 1$.
 (b) Show that for every n such that $n \bmod (k - 1) = 1$, there exists a k -ary tree T with n external nodes (in a k -ary tree all nodes have degree at most k). Also show that all internal nodes of T have degree k .
3. (a) Show that if $n \bmod (k - 1) = 1$, then the greedy rule described following Theorem 4.10 generates an optimal k -ary merge tree for all (q_1, q_2, \dots, q_n) .
 (b) Draw the optimal three-way merge tree obtained using this rule when $(q_1, q_2, \dots, q_{11}) = (3, 7, 8, 9, 15, 16, 18, 20, 23, 25, 28)$.
4. Obtain a set of optimal Huffman codes for the messages (M_1, \dots, M_7) with relative frequencies $(q_1, \dots, q_7) = (4, 5, 7, 8, 10, 12, 20)$. Draw the decode tree for this set of codes.
5. Let T be a decode tree. An optimal decode tree minimizes $\sum q_i d_i$. For a given set of q 's, let D denote all the optimal decode trees. For any tree $T \in D$, let $L(T) = \max \{d_i\}$ and let $SL(T) = \sum d_i$. Schwartz has shown that there exists a tree $T^* \in D$ such that $L(T^*) = \min_{T \in D} \{L(T)\}$ and $SL(T^*) = \min_{T \in D} \{SL(T)\}$.
 - (a) For $(q_1, \dots, q_8) = (1, 1, 2, 2, 4, 4, 4, 4)$ obtain trees T_1 and T_2 such that $L(T_1) > L(T_2)$.
 - (b) Using the data of a, obtain T_1 and $T_2 \in D$ such that $L(T_1) = L(T_2)$ but $SL(T_1) > SL(T_2)$.
 - (c) Show that if the subalgorithm **Least** used in algorithm **Tree** is such that in case of a tie it returns the tree with least depth, then **Tree** generates a tree with the properties of T^* .

4.8 SINGLE-SOURCE SHORTEST PATHS

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions:

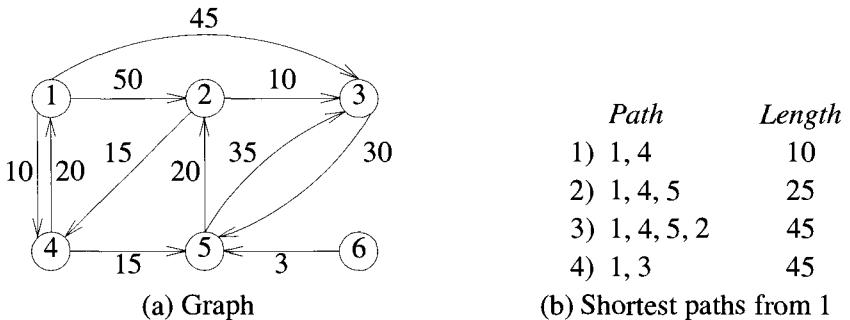


Figure 4.15 Graph and shortest paths from vertex 1 to all destinations

- Is there a path from A to B ?
 - If there is more than one path from A to B , which is the shortest path?

The problems defined by these questions are special cases of the path problem we study in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the *source*, and the last vertex the *destination*. The graphs are digraphs to allow for one-way streets. In the problem we consider, we are given a directed graph $G = (V, E)$, a weighting function $cost$ for the edges of G , and a source vertex v_0 . The problem is to determine the shortest paths from v_0 to *all* the remaining vertices of G . It is assumed that all the weights are positive. The shortest path between v_0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Example 4.11 Consider the directed graph of Figure 4.15(a). The numbers on the edges are the weights. If node 1 is the source vertex, then the shortest path from 1 to 2 is 1, 4, 5, 2. The length of this path is $10 + 15 + 20 = 45$. Even though there are three edges on this path, it is shorter than the path 1, 2 which is of length 50. There is no path from 1 to 6. Figure 4.15(b) lists the shortest paths from node 1 to nodes 4, 5, 2, and 3, respectively. The paths have been listed in nondecreasing order of path length. \square

To formulate a greedy-based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by

one. As an optimization measure we can use the sum of the lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed i shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way (and also a systematic way) to generate the shortest paths from v_0 to the remaining vertices is to generate these paths in nondecreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on. For the graph of Figure 4.15(a) the nearest vertex to $v_0 = 1$ is 4 ($\text{cost}[1, 4] = 10$). The path 1, 4 is the first path generated. The second nearest vertex to node 1 is 5 and the distance between 1 and 5 is 25. The path 1, 4, 5 is the next path generated. In order to generate the shortest paths in this order, we need to be able to determine (1) the next vertex to which a shortest path must be generated and (2) a shortest path to this vertex. Let S denote the set of vertices (including v_0) to which the shortest paths have already been generated. For w not in S , let $\text{dist}[w]$ be the length of the shortest path starting from v_0 , going through only those vertices that are in S , and ending at w . We observe that:

1. If the next shortest path is to vertex u , then the path begins at v_0 , ends at u , and goes through only those vertices that are in S . To prove this, we must show that all the intermediate vertices on the shortest path to u are in S . Assume there is a vertex w on this path that is not in S . Then, the v_0 to u path also contains a path from v_0 to w that is of length less than the v_0 to u path. By assumption the shortest paths are being generated in nondecreasing order of path length, and so the shorter path v_0 to w must already have been generated. Hence, there can be no intermediate vertex that is not in S .
2. The destination of the next path generated must be that of vertex u which has the minimum distance, $\text{dist}[u]$, among all vertices not in S . This follows from the definition of dist and observation 1. In case there are several vertices not in S with the same dist , then any of these may be selected.
3. Having selected a vertex u as in observation 2 and generated the shortest v_0 to u path, vertex u becomes a member of S . At this point the length of the shortest paths starting at v_0 , going though vertices only in S , and ending at a vertex w not in S may decrease; that is, the value of $\text{dist}[w]$ may change. If it does change, then it must be due to a shorter path starting at v_0 and going to u and then to w . The intermediate vertices on the v_0 to u path and the u to w path must all be in S . Further, the v_0 to u path must be the shortest such path; otherwise $\text{dist}[w]$ is not defined properly. Also, the u to w path can be chosen so as not to contain any intermediate vertices. Therefore,

we can conclude that if $dist[w]$ is to change (i.e., decrease), then it is because of a path from v_0 to u to w , where the path from v_0 to u is the shortest such path and the path from u to w is the edge $\langle u, w \rangle$. The length of this path is $dist[u] + cost[u, w]$.

The above observations lead to a simple Algorithm 4.14 for the single-source shortest path problem. This algorithm (known as Dijkstra's algorithm) only determines the lengths of the shortest paths from v_0 to all other vertices in G . The generation of the paths requires a minor extension to this algorithm and is left as an exercise. In the function `ShortestPaths` (Algorithm 4.14) it is assumed that the n vertices of G are numbered 1 through n . The set S is maintained as a bit array with $S[i] = 0$ if vertex i is not in S and $S[i] = 1$ if it is. It is assumed that the graph itself is represented by its cost adjacency matrix with $cost[i, j]$'s being the weight of the edge $\langle i, j \rangle$. The weight $cost[i, j]$ is set to some large number, ∞ , in case the edge $\langle i, j \rangle$ is not in $E(G)$. For $i = j$, $cost[i, j]$ can be set to any nonnegative number without affecting the outcome of the algorithm.

From our earlier discussion, it is easy to see that the algorithm is correct. The time taken by the algorithm on a graph with n vertices is $O(n^2)$. To see this, note that the **for** loop of line 7 in Algorithm 4.14 takes $\Theta(n)$ time. The **for** loop of line 12 is executed $n - 2$ times. Each execution of this loop requires $O(n)$ time at lines 15 and 16 to select the next vertex and again at the **for** loop of line 18 to update $dist$. So the total time for this loop is $O(n^2)$. In case a list t of vertices currently not in s is maintained, then the number of nodes on this list would at any time be $n - num$. This would speed up lines 15 and 16 and the **for** loop of line 18, but the asymptotic time would remain $O(n^2)$. This and other variations of the algorithm are explored in the exercises.

Any shortest path algorithm must examine each edge in the graph at least once since any of the edges could be in a shortest path. Hence, the minimum possible time for such an algorithm would be $\Omega(|E|)$. Since cost adjacency matrices were used to represent the graph, it takes $O(n^2)$ time just to determine which edges are in G , and so any shortest path algorithm using this representation must take $\Omega(n^2)$ time. For this representation then, algorithm `ShortestPaths` is optimal to within a constant factor. If a change to adjacency lists is made, the overall frequency of the **for** loop of line 18 can be brought down to $O(|E|)$ (since $dist$ can change only for vertices adjacent from u). If $V - S$ is maintained as a red-black tree (see Section 2.4.2), each execution of lines 15 and 16 takes $O(\log n)$ time. Note that a red-black tree supports the following operations in $O(\log n)$ time: insert, delete (an arbitrary element), find-min, and search (for an arbitrary element). Each update in line 21 takes $O(\log n)$ time as well (since an update can be done using a delete and an insertion into the red-black tree). Thus the overall run time is $O((n + |E|) \log n)$.

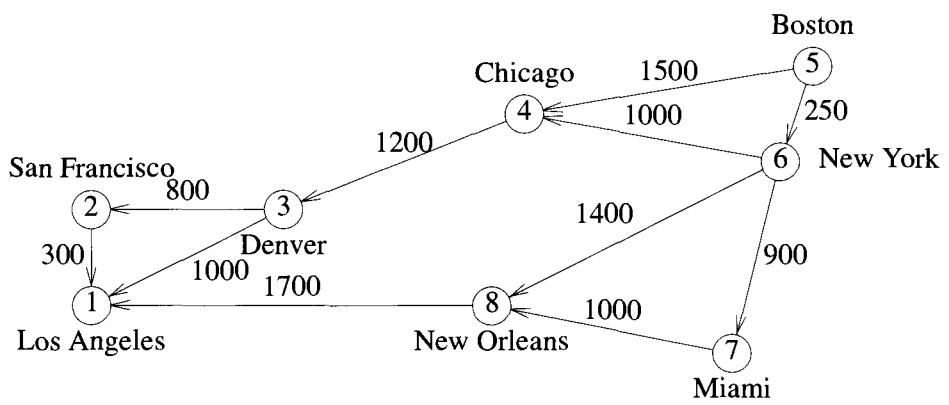
```

1  Algorithm ShortestPaths( $v, cost, dist, n$ )
2  //  $dist[j]$ ,  $1 \leq j \leq n$ , is set to the length of the shortest
3  // path from vertex  $v$  to vertex  $j$  in a digraph  $G$  with  $n$ 
4  // vertices.  $dist[v]$  is set to zero.  $G$  is represented by its
5  // cost adjacency matrix  $cost[1 : n, 1 : n]$ .
6  {
7      for  $i := 1$  to  $n$  do
8          { // Initialize  $S$ .
9               $S[i] := \text{false}$ ;  $dist[i] := cost[v, i]$ ;
10         }
11          $S[v] := \text{true}$ ;  $dist[v] := 0.0$ ; // Put  $v$  in  $S$ .
12         for  $num := 2$  to  $n - 1$  do
13         {
14             // Determine  $n - 1$  paths from  $v$ .
15             Choose  $u$  from among those vertices not
16             in  $S$  such that  $dist[u]$  is minimum;
17              $S[u] := \text{true}$ ; // Put  $u$  in  $S$ .
18             for (each  $w$  adjacent to  $u$  with  $S[w] = \text{false}$ ) do
19                 // Update distances.
20                 if ( $dist[w] > dist[u] + cost[u, w]$ ) then
21                      $dist[w] := dist[u] + cost[u, w]$ ;
22             }
23         }

```

Algorithm 4.14 Greedy algorithm to generate shortest paths

Example 4.12 Consider the eight vertex digraph of Figure 4.16(a) with cost adjacency matrix as in Figure 4.16(b). The values of $dist$ and the vertices selected at each iteration of the **for** loop of line 12 in Algorithm 4.14 for finding all the shortest paths from Boston are shown in Figure 4.17. To begin with, S contains only Boston. In the first iteration of the **for** loop (that is, for $num = 2$), the city u that is not in S and whose $dist[u]$ is minimum is identified to be New York. New York enters the set S . Also the $dist[]$ values of Chicago, Miami, and New Orleans get altered since there are shorter paths to these cities via New York. In the next iteration of the **for** loop, the city that enters S is Miami since it has the smallest $dist[]$ value from among all the nodes not in S . None of the $dist[]$ values are altered. The algorithm continues in a similar fashion and terminates when only seven of the eight vertices are in S . By the definition of $dist$, the distance of the last vertex, in this case Los Angeles, is correct as the shortest path from Boston to Los Angeles can go through only the remaining six vertices. \square



(a) Digraph

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6					1000	0	900	1400
7						0	1000	
8	1700							0

(b) Length-adjacency matrix

Figure 4.16 Figures for Example 4.12

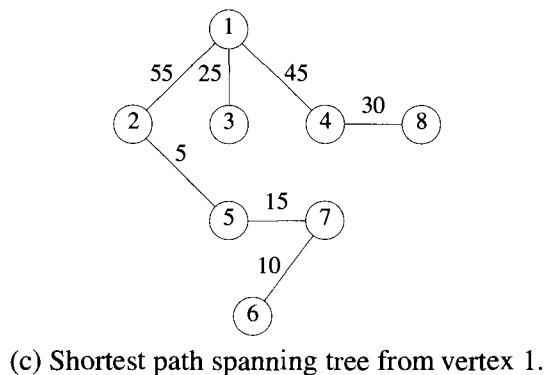
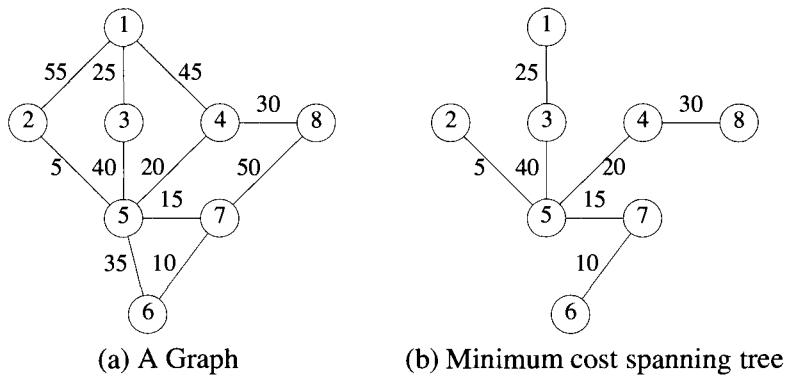
One can easily verify that the edges on the shortest paths from a vertex v to all remaining vertices in a connected undirected graph G form a spanning tree of G . This spanning tree is called a *shortest-path spanning tree*. Clearly, this spanning tree may be different for different root vertices v . Figure 4.18 shows a graph G , its minimum-cost spanning tree, and a shortest-path spanning tree from vertex 1.

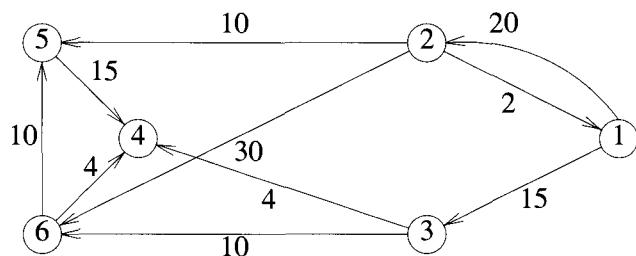
Iteration	S	Vertex selected	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Initial	--	---	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

Figure 4.17 Action of ShortestPaths

EXERCISES

1. Use algorithm `ShortestPaths` to obtain in nondecreasing order the lengths of the shortest paths from vertex 1 to all remaining vertices in the di-graph of Figure 4.19.
2. Using the directed graph of Figure 4.20 explain why `ShortestPaths` will not work properly. What is the shortest path between vertices v_1 and v_7 ?
3. Rewrite algorithm `ShortestPaths` under the following assumptions:
 - (a) G is represented by its adjacency lists. The head nodes are $\text{HEAD}(1), \dots, \text{HEAD}(n)$ and each list node has three fields: VERTEX, COST, and LINK. COST is the length of the corresponding edge and n the number of vertices in G .
 - (b) Instead of representing S , the set of vertices to which the shortest paths have already been found, the set $T = V(G) - S$ is represented using a linked list. What can you say about the computing time of your new algorithm relative to that of `ShortestPaths`?
4. Modify algorithm `ShortestPaths` so that it obtains the shortest paths in addition to the lengths of these paths. What is the computing time of your algorithm?

**Figure 4.18** Graphs and spanning trees

**Figure 4.19** Directed graph

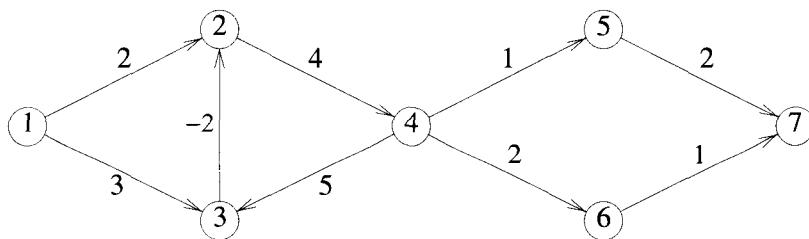


Figure 4.20 Another directed graph

4.9 REFERENCES AND READINGS

The linear time algorithm in Section 4.3 for the tree vertex splitting problem can be found in “Vertex upgrading problems for VLSI,” by D. Paik, Ph.D. thesis, Department of Computer Science, University of Minnesota, October 1991.

The two greedy methods for obtaining minimum-cost spanning trees are due to R. C. Prim and J. B. Kruskal, respectively.

An $O(e \log \log v)$ time spanning tree algorithm has been given by A. C. Yao.

The optimal randomized algorithm for minimum-cost spanning trees presented in this chapter appears in “A randomized linear-time algorithm for finding minimum spanning trees,” by P. N. Klein and R. E. Tarjan, in *Proceedings of the 26th Annual Symposium on Theory of Computing*, 1994, pp. 9–15. See also “A randomized linear-time algorithm to find minimum spanning trees,” by D. R. Karger, P. N. Klein, and R. E. Tarjan, *Journal of the ACM* 42, no. 2 (1995): 321–328.

Proof of Lemma 4.3 can be found in “Verification and sensitivity analysis of minimum spanning trees in linear time,” by B. Dixon, M. Rauch, and R. E. Tarjan, *SIAM Journal on Computing* 21 (1992): 1184–1192, and in “A simple minimum spanning tree verification algorithm,” by V. King, *Proceedings of the Workshop on Algorithms and Data Structures*, 1995.

A very nearly linear time algorithm for minimum-cost spanning trees appears in “Efficient algorithms for finding minimum spanning trees in undirected and directed graphs,” by H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, *Combinatorica* 6 (1986): 109–122.

UNIT - 4

DYNAMIC PROGRAMMING

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 can be viewed as the result of a sequence of decisions. In earlier chapters we saw many problems that can be viewed this way. Here are some examples:

Example 5.1 [Knapsack] The solution to the knapsack problem (Section 4.2) can 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 make a decision on x_1 , then on x_2 , then on x_3 , and so on. An optimal sequence of decisions maximizes the objective function $\sum p_i x_i$. (It also satisfies the constraints $\sum w_i x_i \leq m$ and $0 \leq x_i \leq 1$.) \square

Example 5.2 [Optimal merge patterns] This problem was discussed in Section 4.7. 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, and so on. An optimal sequence of decisions is a least-cost sequence. \square

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, and so on, until vertex j is reached. An optimal sequence of decisions is one that results in a path of least length. \square

For some of the problems that may be viewed in this way, an optimal sequence of decisions can be found by making the decisions one at a time and never making an erroneous decision. This is true for all problems solvable by the greedy method. For many other problems, it is not possible to make stepwise decisions (based only on local information) in such a manner that the sequence of decisions made is optimal.

Example 5.4 [Shortest path] Suppose we wish to find a shortest path from vertex i to vertex j . Let A_i be the vertices adjacent from vertex i . Which of the vertices in A_i should be the second vertex on the path? There is no way to make a decision at this time and guarantee that future decisions leading to an optimal sequence can be made. If on the other hand we wish to find a shortest path from vertex i to all other vertices in G , then at each step, a correct decision can be made (see Section 4.8). \square

One way to solve problems for which it is not possible to make a sequence of stepwise decisions leading to an optimal decision sequence is to try all possible decision sequences. We could enumerate all decision sequences and then pick out the best. But the time and space requirements may be prohibitive. Dynamic programming often drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal. In dynamic programming an optimal sequence of decisions is obtained by making explicit appeal to the *principle of optimality*.

Definition 5.1 [Principle of optimality] The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision. \square

Thus, the essential difference between the greedy method and dynamic programming is that in the greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated. However, sequences containing suboptimal subsequences cannot be optimal (if the principle of optimality holds) and so will not (as far as possible) be generated.

Example 5.5 [Shortest path] Consider the shortest-path problem of Example 5.3. Assume that $i, i_1, i_2, \dots, i_k, j$ is a shortest path from i to j . Starting with the initial vertex i , a decision has been made to go to vertex i_1 . Following this decision, the problem state is defined by vertex i_1 and we need to find a path from i_1 to j . It is clear that the sequence i_1, i_2, \dots, i_k, j must constitute a shortest i_1 to j path. If not, let $i_1, r_1, r_2, \dots, r_q, j$ be a shortest i_1 to j path. Then $i, i_1, r_1, \dots, r_q, j$ is an i to j path that is shorter than the path $i, i_1, i_2, \dots, i_k, j$. Therefore the principle of optimality applies for this problem. \square

Example 5.6 [0/1 knapsack] The 0/1 knapsack problem is similar to the knapsack problem of Section 4.2 except that the x_i 's are restricted to have a value of either 0 or 1. Using KNAP(l, j, y) to represent the problem

$$\begin{aligned} & \text{maximize } \sum_{l \leq i \leq j} p_i x_i \\ & \text{subject to } \sum_{l \leq i \leq j} w_i x_i \leq y \\ & x_i = 0 \text{ or } 1, \quad l \leq i \leq j \end{aligned} \tag{5.1}$$

the knapsack problem is KNAP(1, n, m). Let y_1, y_2, \dots, y_n be an optimal sequence of 0/1 values for x_1, x_2, \dots, x_n , respectively. If $y_1 = 0$, then y_2, y_3, \dots, y_n must constitute an optimal sequence for the problem KNAP(2, n, m). If it does not, then y_1, y_2, \dots, y_n is not an optimal sequence for KNAP(1, n, m). If $y_1 = 1$, then y_2, \dots, y_n must be an optimal sequence for the problem KNAP(2, $n, m - w_1$). If it isn't, then there is another 0/1 sequence z_2, z_3, \dots, z_n such that $\sum_{2 \leq i \leq n} w_i z_i \leq m - w_1$ and $\sum_{2 \leq i \leq n} p_i z_i > \sum_{2 \leq i \leq n} p_i y_i$. Hence, the sequence $y_1, z_2, z_3, \dots, z_n$ is a sequence for (5.1) with greater value. Again the principle of optimality applies. \square

Let S_0 be the initial problem state. Assume that n decisions d_i , $1 \leq i \leq n$, have to be made. Let $D_1 = \{r_1, r_2, \dots, r_j\}$ be the set of possible decision values for d_1 . Let S_i be the problem state following the choice of decision r_i , $1 \leq i \leq j$. Let Γ_i be an optimal sequence of decisions with respect to the problem state S_i . Then, when the principle of optimality holds, an optimal sequence of decisions with respect to S_0 is the best of the decision sequences r_i, Γ_i , $1 \leq i \leq j$.

Example 5.7 [Shortest path] Let A_i be the set of vertices adjacent to vertex i . For each vertex $k \in A_i$, let Γ_k be a shortest path from k to j . Then, a shortest i to j path is the shortest of the paths $\{i, \Gamma_k | k \in A_i\}$. \square

Example 5.8 [0/1 knapsack] Let $g_j(y)$ be the value of an optimal solution to KNAP($j + 1, n, y$). Clearly, $g_0(m)$ is the value of an optimal solution to KNAP(1, n, m). The possible decisions for x_1 are 0 and 1 ($D_1 = \{0, 1\}$). From the principle of optimality it follows that

$$g_0(m) = \max \{g_1(m), g_1(m - w_1) + p_1\} \tag{5.2}$$

\square

While the principle of optimality has been stated only with respect to the initial state and decision, it can be applied equally well to intermediate states and decisions. The next two examples show how this can be done.

Example 5.9 [Shortest path] Let k be an intermediate vertex on a shortest i to j path $i, i_1, i_2, \dots, k, p_1, p_2, \dots, j$. The paths i, i_1, \dots, k and k, p_1, \dots, j must, respectively, be shortest i to k and k to j paths. \square

Example 5.10 [0/1 knapsack] Let y_1, y_2, \dots, y_n be an optimal solution to $\text{KNAP}(1, n, m)$. Then, for each j , $1 \leq j \leq n$, y_1, \dots, y_j , and y_{j+1}, \dots, y_n must be optimal solutions to the problems $\text{KNAP}(1, j, \sum_{1 \leq i \leq j} w_i y_i)$ and $\text{KNAP}(j+1, n, m - \sum_{1 \leq i \leq j} w_i y_i)$ respectively. This observation allows us to generalize (5.2) to

$$g_i(y) = \max \{g_{i+1}(y), g_{i+1}(y - w_{i+1}) + p_{i+1}\} \quad (5.3)$$

□

The recursive application of the optimality principle results in a recurrence equation of type (5.3). Dynamic programming algorithms solve this recurrence to obtain a solution to the given problem instance. The recurrence (5.3) can be solved using the knowledge $g_n(y) = 0$ for all $y \geq 0$ and $g_n(y) = -\infty$ for $y < 0$. From $g_n(y)$, one can obtain $g_{n-1}(y)$ using (5.3) with $i = n - 1$. Then, using $g_{n-1}(y)$, one can obtain $g_{n-2}(y)$. Repeating in this way, one can determine $g_1(y)$ and finally $g_0(m)$ using (5.3) with $i = 0$.

Example 5.11 [0/1 knapsack] Consider the case in which $n = 3$, $w_1 = 2$, $w_2 = 3$, $w_3 = 4$, $p_1 = 1$, $p_2 = 2$, $p_3 = 5$, and $m = 6$. We have to compute $g_0(6)$. The value of $g_0(6) = \max \{g_1(6), g_1(4) + 1\}$.

In turn, $g_1(6) = \max \{g_2(6), g_2(3) + 2\}$. But $g_2(6) = \max \{g_3(6), g_3(2) + 5\} = \max \{0, 5\} = 5$. Also, $g_2(3) = \max \{g_3(3), g_3(3 - 4) + 5\} = \max \{0, -\infty\} = 0$. Thus, $g_1(6) = \max \{5, 2\} = 5$.

Similarly, $g_1(4) = \max \{g_2(4), g_2(4 - 3) + 2\}$. But $g_2(4) = \max \{g_3(4), g_3(4 - 4) + 5\} = \max \{0, 5\} = 5$. The value of $g_2(1) = \max \{g_3(1), g_3(1 - 4) + 5\} = \max \{0, -\infty\} = 0$. Thus, $g_1(4) = \max \{5, 0\} = 5$.

Therefore, $g_0(6) = \max \{5, 5 + 1\} = 6$. □

Example 5.12 [Shortest path] Let P_j be the set of vertices adjacent to vertex j (that is, $k \in P_j$ iff $\langle k, j \rangle \in E(G)$). For each $k \in P_j$, let Γ_k be a shortest i to k path. The principle of optimality holds and a shortest i to j path is the shortest of the paths $\{\Gamma_k, j | k \in P_j\}$.

To obtain this formulation, we started at vertex j and looked at the last decision made. The last decision was to use one of the edges $\langle k, j \rangle$, $k \in P_j$. In a sense, we are looking backward on the i to j path. □

Example 5.13 [0/1 knapsack] Looking backward on the sequence of decisions x_1, x_2, \dots, x_n , we see that

$$f_j(y) = \max \{f_{j-1}(y), f_{j-1}(y - w_j) + p_j\} \quad (5.4)$$

where $f_j(y)$ is the value of an optimal solution to $\text{KNAP}(1, j, y)$. □

The value of an optimal solution to $\text{KNAP}(1, n, m)$ is $f_n(m)$. Equation 5.4 can be solved by beginning with $f_0(y) = 0$ for all y , $y \geq 0$, and $f_0(y) = -\infty$, for all y , $y < 0$. From this, f_1, f_2, \dots, f_n can be successively obtained. \square

The solution method outlined in Examples 5.12 and 5.13 may indicate that one has to look at all possible decision sequences to obtain an optimal decision sequence using dynamic programming. This is not the case. Because of the use of the principle of optimality, decision sequences containing subsequences that are suboptimal are *not* considered. Although the total number of different decision sequences is exponential in the number of decisions (if there are d choices for each of the n decisions to be made then there are d^n possible decision sequences), dynamic programming algorithms often have a polynomial complexity.

Another important feature of the dynamic programming approach is that optimal solutions to subproblems are retained so as to avoid recomputing their values. The use of these tabulated values makes it natural to recast the recursive equations into an iterative algorithm. Most of the dynamic programming algorithms in this chapter are expressed in this way.

The remaining sections of this chapter apply dynamic programming to a variety of problems. These examples should help you understand the method better and also realize the advantage of dynamic programming over explicitly enumerating all decision sequences.

EXERCISES

1. The principle of optimality does not hold for every problem whose solution can be viewed as the result of a sequence of decisions. Find two problems for which the principle does not hold. Explain why the principle does not hold for these problems.
2. For the graph of Figure 5.1, find the shortest path between the nodes 1 and 2. Use the recurrence relations derived in Examples 5.10 and 5.13.

5.2 MULTISTAGE GRAPHS

A multistage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i , $1 \leq i \leq k$. In addition, if $\langle u, v \rangle$ is an edge in E , then $u \in V_i$ and $v \in V_{i+1}$ for some i , $1 \leq i < k$. The sets V_1 and V_k are such that $|V_1| = |V_k| = 1$. Let s and t , respectively, be the vertices in V_1 and V_k . The vertex s is the *source*, and t the *sink*. Let $c(i, j)$ be the cost of edge $\langle i, j \rangle$. The cost of a path from s to t is the sum of the costs of the edges on the path. The *multistage graph problem* is to find a minimum-cost

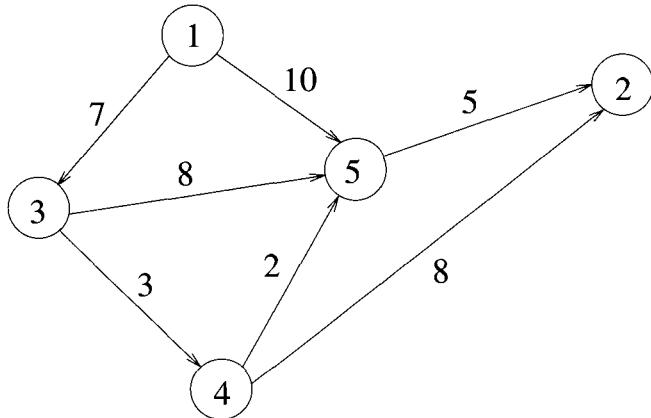


Figure 5.1 Graph for Exercise 2 (Section 5.1)

path from s to t . Each set V_i defines a stage in the graph. Because of the constraints on E , every path from s to t starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k . Figure 5.2 shows a five-stage graph. A minimum-cost s to t path is indicated by the broken edges.

Many problems can be formulated as multistage graph problems. We give only one example. Consider a resource allocation problem in which n units of resource are to be allocated to r projects. If j , $0 \leq j \leq n$, units of the resource are allocated to project i , then the resulting net profit is $N(i, j)$. The problem is to allocate the resource to the r projects in such a way as to maximize total net profit. This problem can be formulated as an $r + 1$ stage graph problem as follows. Stage i , $1 \leq i \leq r$, represents project i . There are $n + 1$ vertices $V(i, j)$, $0 \leq j \leq n$, associated with stage i , $2 \leq i \leq r$. Stages 1 and $r + 1$ each have one vertex, $V(1, 0) = s$ and $V(r + 1, n) = t$, respectively. Vertex $V(i, j)$, $2 \leq i \leq r$, represents the state in which a total of j units of resource have been allocated to projects $1, 2, \dots, i - 1$. The edges in G are of the form $\langle V(i, j), V(i + 1, l) \rangle$ for all $j \leq l$ and $1 \leq i < r$. The edge $\langle V(i, j), V(i + 1, l) \rangle$, $j \leq l$, is assigned a weight or cost of $N(i, l - j)$ and corresponds to allocating $l - j$ units of resource to project i , $1 \leq i < r$. In addition, G has edges of the type $\langle V(r, j), V(r + 1, n) \rangle$. Each such edge is assigned a weight of $\max_{0 \leq p \leq n-j} \{N(r, p)\}$. The resulting graph for a three-project problem with $n = 4$ is shown in Figure 5.3. It should be easy to see that an optimal allocation of resources is defined by a maximum cost s to t path. This is easily converted into a minimum-cost problem by changing the sign of all the edge costs.

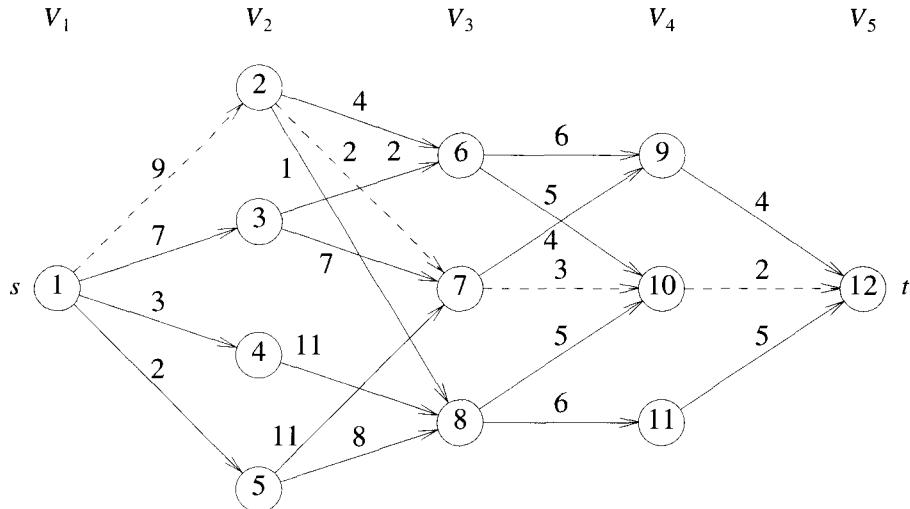


Figure 5.2 Five-stage graph

A dynamic programming formulation for a k -stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of $k - 2$ decisions. The i th decision involves determining which vertex in V_{i+1} , $1 \leq i \leq k - 2$, is to be on the path. It is easy to see that the principle of optimality holds. Let $p(i, j)$ be a minimum-cost path from vertex j in V_i to vertex t . Let $cost(i, j)$ be the cost of this path. Then, using the forward approach, we obtain

$$cost(i, j) = \min_{\substack{l \in V_{i+1} \\ (j, l) \in E}} \{c(j, l) + cost(i + 1, l)\} \quad (5.5)$$

Since, $cost(k - 1, j) = c(j, t)$ if $\langle j, t \rangle \in E$ and $cost(k - 1, j) = \infty$ if $\langle j, t \rangle \notin E$, (5.5) may be solved for $cost(1, s)$ by first computing $cost(k - 2, j)$ for all $j \in V_{k-2}$, then $cost(k - 3, j)$ for all $j \in V_{k-3}$, and so on, and finally $cost(1, s)$. Trying this out on the graph of Figure 5.2, we obtain

$$\begin{aligned} cost(3, 6) &= \min \{6 + cost(4, 9), 5 + cost(4, 10)\} \\ &= 7 \\ cost(3, 7) &= \min \{4 + cost(4, 9), 3 + cost(4, 10)\} \\ &= 5 \end{aligned}$$

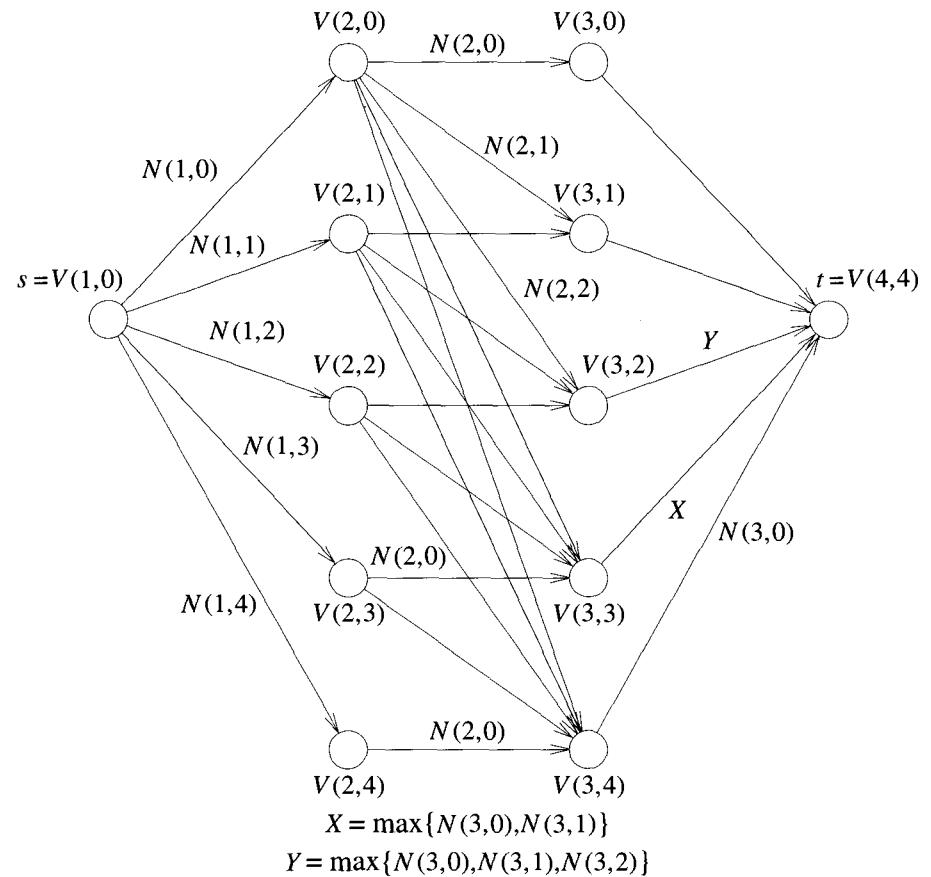


Figure 5.3 Four-stage graph corresponding to a three-project problem

$$\begin{aligned}
cost(3, 8) &= 7 \\
cost(2, 2) &= \min \{4 + cost(3, 6), 2 + cost(3, 7), 1 + cost(3, 8)\} \\
&= 7 \\
cost(2, 3) &= 9 \\
cost(2, 4) &= 18 \\
cost(2, 5) &= 15 \\
cost(1, 1) &= \min \{9 + cost(2, 2), 7 + cost(2, 3), 3 + cost(2, 4), \\
&\quad 2 + cost(2, 5)\} \\
&= 16
\end{aligned}$$

Note that in the calculation of $cost(2, 2)$, we have reused the values of $cost(3, 6)$, $cost(3, 7)$, and $cost(3, 8)$ and so avoided their recomputation. A minimum cost s to t path has a cost of 16. This path can be determined easily if we record the decision made at each state (vertex). Let $d(i, j)$ be the value of l (where l is a node) that minimizes $c(j, l) + cost(i + 1, l)$ (see Equation 5.5). For Figure 5.2 we obtain

$$\begin{aligned}
d(3, 6) &= 10; & d(3, 7) &= 10; & d(3, 8) &= 10; \\
d(2, 2) &= 7; & d(2, 3) &= 6; & d(2, 4) &= 8; & d(2, 5) &= 8; \\
d(1, 1) &= 2
\end{aligned}$$

Let the minimum-cost path be $s = 1, v_2, v_3, \dots, v_{k-1}, t$. It is easy to see that $v_2 = d(1, 1) = 2$, $v_3 = d(2, d(1, 1)) = 7$, and $v_4 = d(3, d(2, d(1, 1))) = d(3, 7) = 10$.

Before writing an algorithm to solve (5.5) for a general k -stage graph, let us impose an ordering on the vertices in V . This ordering makes it easier to write the algorithm. We require that the n vertices in V are indexed 1 through n . Indices are assigned in order of stages. First, s is assigned index 1, then vertices in V_2 are assigned indices, then vertices from V_3 , and so on. Vertex t has index n . Hence, indices assigned to vertices in V_{i+1} are bigger than those assigned to vertices in V_i (see Figure 5.2). As a result of this indexing scheme, $cost$ and d can be computed in the order $n - 1, n - 2, \dots, 1$. The first subscript in $cost$, p , and d only identifies the stage number and is omitted in the algorithm. The resulting algorithm, in pseudocode, is **FGraph** (Algorithm 5.1).

The complexity analysis of the function **FGraph** is fairly straightforward. If G is represented by its adjacency lists, then r in line 9 of Algorithm 5.1 can be found in time proportional to the degree of vertex j . Hence, if G has $|E|$ edges, then the time for the **for** loop of line 7 is $\Theta(|V| + |E|)$. The time for the **for** loop of line 16 is $\Theta(k)$. Hence, the total time is $\Theta(|V| + |E|)$. In addition to the space needed for the input, space is needed for $cost[]$, $d[]$, and $p[]$.

```

1  Algorithm FGraph( $G, k, n, p$ )
2  // The input is a  $k$ -stage graph  $G = (V, E)$  with  $n$  vertices
3  // indexed in order of stages.  $E$  is a set of edges and  $c[i, j]$ 
4  // is the cost of  $\langle i, j \rangle$ .  $p[1 : k]$  is a minimum-cost path.
5  {
6       $cost[n] := 0.0;$ 
7      for  $j := n - 1$  to 1 step  $-1$  do
8          { // Compute  $cost[j]$ .
9              Let  $r$  be a vertex such that  $\langle j, r \rangle$  is an edge
10             of  $G$  and  $c[j, r] + cost[r]$  is minimum;
11              $cost[j] := c[j, r] + cost[r];$ 
12              $d[j] := r;$ 
13         }
14         // Find a minimum-cost path.
15          $p[1] := 1; p[k] := n;$ 
16         for  $j := 2$  to  $k - 1$  do  $p[j] := d[p[j - 1]];$ 
17     }

```

Algorithm 5.1 Multistage graph pseudocode corresponding to the forward approach

The multistage graph problem can also be solved using the backward approach. Let $bp(i, j)$ be a minimum-cost path from vertex s to a vertex j in V_i . Let $bcost(i, j)$ be the cost of $bp(i, j)$. From the backward approach we obtain

$$bcost(i, j) = \min_{\substack{l \in V_{i-1} \\ \langle l, j \rangle \in E}} \{bcost(i - 1, l) + c(l, j)\} \quad (5.6)$$

Since $bcost(2, j) = c(1, j)$ if $\langle 1, j \rangle \in E$ and $bcost(2, j) = \infty$ if $\langle 1, j \rangle \notin E$, $bcost(i, j)$ can be computed using (5.6) by first computing $bcost$ for $i = 3$, then for $i = 4$, and so on. For the graph of Figure 5.2, we obtain

$$\begin{aligned} bcost(3, 6) &= \min \{bcost(2, 2) + c(2, 6), bcost(2, 3) + c(3, 6)\} \\ &= \min \{9 + 4, 7 + 2\} \\ &= 9 \\ bcost(3, 7) &= 11 \\ bcost(3, 8) &= 10 \\ bcost(4, 9) &= 15 \end{aligned}$$

$$\begin{aligned}bcost(4, 10) &= 14 \\bcost(4, 11) &= 16 \\bcost(5, 12) &= 16\end{aligned}$$

The corresponding algorithm, in pseudocode, to obtain a minimum-cost $s - t$ path is **BGraph** (Algorithm 5.2). The first subscript on $bcost$, p , and d are omitted for the same reasons as before. This algorithm has the same complexity as **FGraph** provided G is now represented by its inverse adjacency lists (i.e., for each vertex v we have a list of vertices w such that $\langle w, v \rangle \in E$).

```

1  Algorithm BGraph( $G, k, n, p$ )
2  // Same function as FGraph
3  {
4       $bcost[1] := 0.0;$ 
5      for  $j := 2$  to  $n$  do
6          { // Compute  $bcost[j]$ .
7              Let  $r$  be such that  $\langle r, j \rangle$  is an edge of
8               $G$  and  $bcost[r] + c[r, j]$  is minimum;
9               $bcost[j] := bcost[r] + c[r, j];$ 
10              $d[j] := r;$ 
11         }
12         // Find a minimum-cost path.
13          $p[1] := 1$ ;  $p[k] := n$ ;
14         for  $j := k - 1$  to  $2$  do  $p[j] := d[p[j + 1]]$ ;
15     }
```

Algorithm 5.2 Multistage graph pseudocode corresponding to backward approach

It should be easy to see that both **FGraph** and **BGraph** work correctly even on a more generalized version of multistage graphs. In this generalization, the graph is permitted to have edges $\langle u, v \rangle$ such that $u \in V_i, v \in V_j$, and $i < j$.

Note: In the pseudocodes **FGraph** and **BGraph**, $bcost(i, j)$ is set to ∞ for any $\langle i, j \rangle \notin E$. When programming these pseudocodes, one could use the maximum allowable floating point number for ∞ . If the weight of any such edge is added to some other costs, a floating point overflow might occur. Care should be taken to avoid such overflows.

EXERCISES

- Find a minimum-cost path from s to t in the multistage graph of Figure 5.4. Do this first using the forward approach and then using the backward approach.

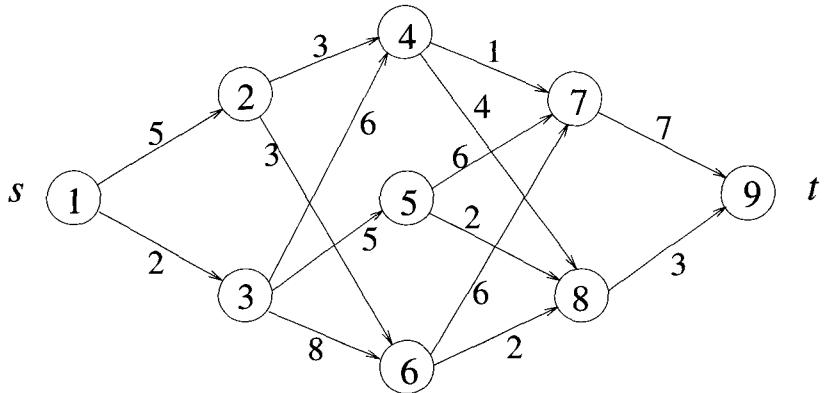


Figure 5.4 Multistage graph for Exercise 1

- Refine Algorithm 5.1 into a program. Assume that G is represented by its adjacency lists. Test the correctness of your code using suitable graphs.
- Program Algorithm 5.1. Assume that G is an array $G[1 : e, 1 : 3]$. Each edge $\langle i, j \rangle$, $i < j$, of G is stored in $G[q]$, for some q and $G[q, 1] = i$, $G[q, 2] = j$, and $G[q, 3] = \text{cost of edge } \langle i, j \rangle$. Assume that $G[q, 1] \leq G[q + 1, 1]$ for $1 \leq q < e$, where e is the number of edges in the multistage graph. Test the correctness of your function using suitable multistage graphs. What is the time complexity of your function?
- Program Algorithm 5.2 for the multistage graph problem using the backward approach. Assume that the graph is represented using inverse adjacency lists. Test its correctness. What is its complexity?
- Do Exercise 4 using the graph representation of Exercise 3. This time, however, assume that $G[q, 2] \leq G[q + 1, 2]$ for $1 \leq q < e$.
- Extend the discussion of this section to directed acyclic graphs (dags). Suppose the vertices of a dag are numbered so that all edges have the form $\langle i, j \rangle$, $i < j$. What changes, if any, need to be made to Algorithm 5.1 to find the length of the longest path from vertex 1 to vertex n ?

7. [W. Miller] Show that `BGraph1` computes shortest paths for directed acyclic graphs represented by adjacency lists (instead of inverse adjacency lists as in `BGraph`).

```

1  Algorithm BGraph1( $G, n$ )
2  {
3       $bcost[1] := 0.0;$ 
4      for  $j := 2$  to  $n$  do  $bcost[j] := \infty;$ 
5      for  $j := 1$  to  $n - 1$  do
6          for each  $r$  such that  $\langle j, r \rangle$  is an edge of  $G$  do
7               $bcost[r] := \min(bcost[r], bcost[j] + c[j, r]);$ 
8  }
```

Note: There is a possibility of a floating point overflow in this function. In such cases the program should be suitably modified.

5.3 ALL-PAIRS SHORTEST PATHS

Let $G = (V, E)$ be a directed graph with n vertices. Let $cost$ be a cost adjacency matrix for G such that $cost(i, i) = 0$, $1 \leq i \leq n$. Then $cost(i, j)$ is the length (or cost) of edge $\langle i, j \rangle$ if $\langle i, j \rangle \in E(G)$ and $cost(i, j) = \infty$ if $i \neq j$ and $\langle i, j \rangle \notin E(G)$. The *all-pairs shortest-path problem* is to determine a matrix A such that $A(i, j)$ is the length of a shortest path from i to j . The matrix A can be obtained by solving n single-source problems using the algorithm `ShortestPaths` of Section 4.8. Since each application of this procedure requires $O(n^2)$ time, the matrix A can be obtained in $O(n^3)$ time. We obtain an alternate $O(n^3)$ solution to this problem using the principle of optimality. Our alternate solution requires a weaker restriction on edge costs than required by `ShortestPaths`. Rather than require $cost(i, j) \geq 0$, for every edge $\langle i, j \rangle$, we only require that G have no cycles with negative length. Note that if we allow G to contain a cycle of negative length, then the shortest path between any two vertices on this cycle has length $-\infty$.

Let us examine a shortest i to j path in G , $i \neq j$. This path originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j . We can assume that this path contains no cycles for if there is a cycle, then this can be deleted without increasing the path length (no cycle has negative length). If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j , respectively. Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds. This alerts us to the prospect of using dynamic programming. If k is the intermediate vertex with highest index, then the i to k path is a shortest i to k path in G going through no vertex with index greater than $k - 1$. Similarly the k to j path is a shortest k to j path in G going through no vertex of index greater than

$k - 1$. We can regard the construction of a shortest i to j path as first requiring a decision as to which is the highest indexed intermediate vertex k . Once this decision has been made, we need to find two shortest paths, one from i to k and the other from k to j . Neither of these may go through a vertex with index greater than $k - 1$. Using $A^k(i, j)$ to represent the length of a shortest path from i to j going through no vertex of index greater than k , we obtain

$$A(i, j) = \min \left\{ \min_{1 \leq k \leq n} \{A^{k-1}(i, k) + A^{k-1}(k, j)\}, cost(i, j) \right\} \quad (5.7)$$

Clearly, $A^0(i, j) = cost(i, j)$, $1 \leq i \leq n$, $1 \leq j \leq n$. We can obtain a recurrence for $A^k(i, j)$ using an argument similar to that used before. A shortest path from i to j going through no vertex higher than k either goes through vertex k or it does not. If it does, $A^k(i, j) = A^{k-1}(i, k) + A^{k-1}(k, j)$. If it does not, then no intermediate vertex has index greater than $k - 1$. Hence $A^k(i, j) = A^{k-1}(i, j)$. Combining, we get

$$A^k(i, j) = \min \{A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j)\}, \quad k \geq 1 \quad (5.8)$$

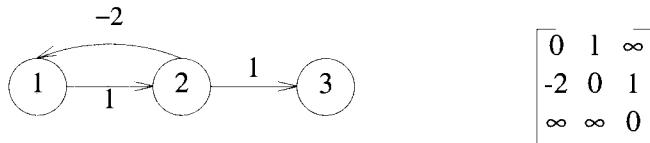
The following example shows that (5.8) is not true for graphs with cycles of negative length.

Example 5.14 Figure 5.5 shows a digraph together with its matrix A^0 . For this graph $A^2(1, 3) \neq \min\{A^1(1, 3), A^1(1, 2) + A^1(2, 3)\} = 2$. Instead we see that $A^2(1, 3) = -\infty$. The length of the path

$$1, 2, 1, 2, 1, 2, \dots, 1, 2, 3$$

can be made arbitrarily small. This is so because of the presence of the cycle 1 2 1 which has a length of -1 . \square

Recurrence (5.8) can be solved for A^n by first computing A^1 , then A^2 , then A^3 , and so on. Since there is no vertex in G with index greater than n , $A(i, j) = A^n(i, j)$. Function `AllPaths` computes $A^n(i, j)$. The computation is done inplace so the superscript on A is not needed. The reason this computation can be carried out in-place is that $A^k(i, k) = A^{k-1}(i, k)$ and $A^k(k, j) = A^{k-1}(k, j)$. Hence, when A^k is formed, the k th column and row do not change. Consequently, when $A^k(i, j)$ is computed in line 11 of Algorithm 5.3, $A(i, k) = A^{k-1}(i, k) = A^k(i, k)$ and $A(k, j) = A^{k-1}(k, j) = A^k(k, j)$. So, the old values on which the new values are based do not change on this iteration.

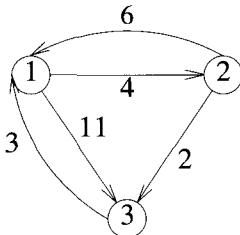
**Figure 5.5** Graph with negative cycle

```

0 Algorithm AllPaths(cost, A, n)
1 // cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
2 // n vertices; A[i, j] is the cost of a shortest path from vertex
3 // i to vertex j. cost[i, i] = 0.0, for  $1 \leq i \leq n$ .
4 {
5     for i := 1 to n do
6         for j := 1 to n do
7             A[i, j] := cost[i, j]; // Copy cost into A.
8         for k := 1 to n do
9             for i := 1 to n do
10                for j := 1 to n do
11                    A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
12 }
```

Algorithm 5.3 Function to compute lengths of shortest paths

Example 5.15 The graph of Figure 5.6(a) has the cost matrix of Figure 5.6(b). The initial A matrix, $A^{(0)}$, plus its values after 3 iterations $A^{(1)}$, $A^{(2)}$, and $A^{(3)}$ are given in Figure 5.6. \square



(a) Example digraph

A^0	1	2	3	A^1	1	2	3
1	0	4	11	1	0	4	11
2	6	0	2	2	6	0	2
3	3	∞	0	3	3	7	0

(b) A^0

A^2	1	2	3	A^3	1	2	3
1	0	4	6	1	0	4	6
2	6	0	2	2	5	0	2
3	3	7	0	3	3	7	0

(d) A^2 (e) A^3 **Figure 5.6** Directed graph and associated matrices

Let $M = \max \{cost(i, j) | \langle i, j \rangle \in E(G)\}$. It is easy to see that $A^n(ij) \leq (n - 1)M$. From the working of AllPaths, it is clear that if $\langle i, j \rangle \notin E(G)$ and $i \neq j$, then we can initialize $cost(i, j)$ to any number greater than $(n - 1)M$ (rather than the maximum allowable floating point number). If, at termination, $A(i, j) > (n - 1)M$, then there is no directed path from i to j in G . Even for this choice of ∞ , care should be taken to avoid any floating point overflows.

The time needed by AllPaths (Algorithm 5.3) is especially easy to determine because the looping is independent of the data in the matrix A . Line 11 is iterated n^3 times, and so the time for AllPaths is $\Theta(n^3)$. An exercise examines the extensions needed to obtain the i to j paths with these lengths. Some speedup 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 ∞ .

EXERCISES

1. (a) Does the recurrence (5.8) hold for the graph of Figure 5.7? Why?
-

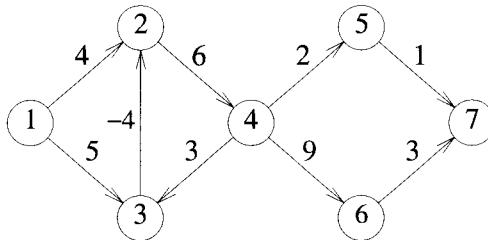


Figure 5.7 Graph for Exercise 1

- (b) Why does Equation 5.8 not hold for graphs with cycles of negative length?
2. Modify the function AllPaths so that a shortest path is output for each pair of vertices (i, j) . What are the time and space complexities of the new algorithm?
3. Let A be the adjacency matrix of a directed graph G . Define the transitive closure A^+ of A to be a matrix with the property $A^+(i, j) = 1$ iff G has a directed path, containing at least one edge, from vertex i to vertex j . $A^+(i, j) = 0$ otherwise. The reflexive transitive closure A^* is a matrix with the property $A^*(i, j) = 1$ iff G has a path, containing zero or more edges, from i to j . $A^*(i, j) = 0$ otherwise.
- (a) Obtain A^+ and A^* for the directed graph of Figure 5.8.
-

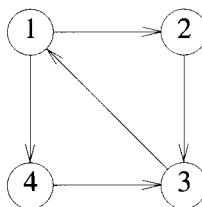


Figure 5.8 Graph for Exercise 3

- (b) Let $A^k(i, j) = 1$ iff there is a path with zero or more edges from i to j going through no vertex of index greater than k . Define A^0 in terms of the adjacency matrix A .

- (c) Obtain a recurrence between A^k and A^{k-1} similar to (5.8). Use the logical operators **or** and **and** rather than **min** and **+**.
- (d) Write an algorithm, using the recurrence of part (c), to find A^* . Your algorithm can use only $O(n^2)$ space. What is its time complexity?
- (e) Show that $A^+ = A \times A^*$, where matrix multiplication is defined as $A^+(i, j) = \bigvee_{k=1}^n (A(i, k) \wedge A^*(k, j))$. The operation \vee is the logical **or** operation, and \wedge the logical **and** operation. Hence A^+ may be computed from A^* .

5.4 SINGLE-SOURCE SHORTEST PATHS: GENERAL WEIGHTS

We now consider the single-source shortest path problem discussed in Section 4.8 when some or all of the edges of the directed graph G may have negative length. **ShortestPaths** (Algorithm 4.14) does not necessarily give the correct results on such graphs. To see this, consider the graph of Figure 5.9. Let $v = 1$ be the source vertex. Referring back to Algorithm 4.14, since $n = 3$, the loop of lines 12 to 22 is iterated just once. Also $u = 3$ in lines 15 and 16, and so no changes are made to $dist[]$. The algorithm terminates with $dist[2] = 7$ and $dist[3] = 5$. The shortest path from 1 to 3 is 1, 2, 3. This path has length 2, which is less than the computed value of $dist[3]$.

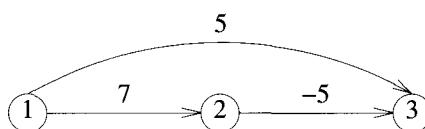


Figure 5.9 Directed graph with a negative-length edge

When negative edge lengths are permitted, we require that the graph have no cycles of negative length. This is necessary to ensure that shortest paths consist of a finite number of edges. For example, in the graph of Figure 5.5, the length of the shortest path from vertex 1 to vertex 3 is $-\infty$. The length of the path

$$1, 2, 1, 2, 1, 2, \dots, 1, 2, 3$$

can be made arbitrarily small as was shown in Example 5.14.

When there are no cycles of negative length, there is a shortest path between any two vertices of an n -vertex graph that has at most $n - 1$ edges

on it. To see this, note that a path that has more than $n - 1$ edges must repeat at least one vertex and hence must contain a cycle. Elimination of the cycles from the path results in another path with the same source and destination. This path is cycle-free and has a length that is no more than that of the original path, as the length of the eliminated cycles was at least zero. We can use this observation on the maximum number of edges on a cycle-free shortest path to obtain an algorithm to determine a shortest path from a source vertex to all remaining vertices in the graph. As in the case of `ShortestPaths` (Algorithm 4.14), we compute only the length, $dist[u]$, of the shortest path from the source vertex v to u . An exercise examines the extension needed to construct the shortest paths.

Let $dist^\ell[u]$ be the length of a shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most ℓ edges. Then, $dist^1[u] = cost[v, u]$, $1 \leq u \leq n$. As noted earlier, when there are no cycles of negative length, we can limit our search for shortest paths to paths with at most $n - 1$ edges. Hence, $dist^{n-1}[u]$ is the length of an unrestricted shortest path from v to u .

Our goal then is to compute $dist^{n-1}[u]$ for all u . This can be done using the dynamic programming methodology. First, we make the following observations:

1. If the shortest path from v to u with at most k , $k > 1$, edges has no more than $k - 1$ edges, then $dist^k[u] = dist^{k-1}[u]$.
2. If the shortest path from v to u with at most k , $k > 1$, edges has exactly k edges, then it is made up of a shortest path from v to some vertex j followed by the edge $\langle j, u \rangle$. The path from v to j has $k - 1$ edges, and its length is $dist^{k-1}[j]$. All vertices i such that the edge $\langle i, u \rangle$ is in the graph are candidates for j . Since we are interested in a shortest path, the i that minimizes $dist^{k-1}[i] + cost[i, u]$ is the correct value for j .

These observations result in the following recurrence for $dist$:

$$dist^k[u] = \min \{dist^{k-1}[u], \min_i \{dist^{k-1}[i] + cost[i, u]\}\}$$

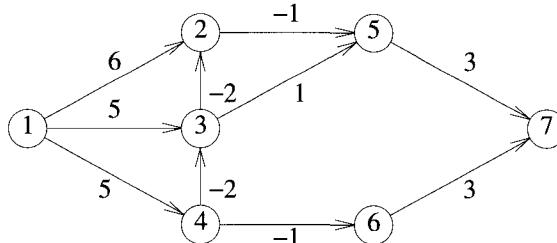
This recurrence can be used to compute $dist^k$ from $dist^{k-1}$, for $k = 2, 3, \dots, n - 1$.

Example 5.16 Figure 5.10 gives a seven-vertex graph, together with the arrays $dist^k$, $k = 1, \dots, 6$. These arrays were computed using the equation just given. For instance, $dist^k[1] = 0$ for all k since 1 is the source node. Also, $dist^1[2] = 6$, $dist^1[3] = 5$, and $dist^1[4] = 5$, since there are edges from

1 to these nodes. The distance $dist^1[]$ is ∞ for the nodes 5, 6, and 7 since there are no edges to these from 1.

$$\begin{aligned} dist^2[2] &= \min \{dist^1[2], \min_i dist^1[i] + cost[i, 2]\} \\ &= \min \{6, 0 + 6, 5 - 2, 5 + \infty, \infty + \infty, \infty + \infty, \infty + \infty\} = 3 \end{aligned}$$

Here the terms $0 + 6, 5 - 2, 5 + \infty, \infty + \infty, \infty + \infty$, and $\infty + \infty$ correspond to a choice of $i = 1, 3, 4, 5, 6$, and 7, respectively. The rest of the entries are computed in an analogous manner. \square



(a) A directed graph

k	$dist^k[1..7]$						
	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Figure 5.10 Shortest paths with negative edge lengths

An exercise shows that if we use the same memory location $dist[u]$ for $dist^k[u]$, $k = 1, \dots, n - 1$, then the final value of $dist[u]$ is still $dist^{n-1}[u]$. Using this fact and the recurrence for $dist$ shown above, we arrive at the pseudocode of Algorithm 5.4 to compute the length of the shortest path from vertex v to each other vertex of the graph. This algorithm is referred to as the Bellman and Ford algorithm.

Each iteration of the **for** loop of lines 7 to 12 takes $O(n^2)$ time if adjacency matrices are used and $O(e)$ time if adjacency lists are used. Here e is the number of edges in the graph. The overall complexity is $O(n^3)$ when adjacency matrices are used and $O(ne)$ when adjacency lists are used. The observed complexity of the shortest-path algorithm can be reduced by noting that if none of the $dist$ values change on one iteration of the **for** loop of lines 7 to 12, then none will change on successive iterations. So, this loop can be rewritten to terminate either after $n - 1$ iterations or after the

```
1  Algorithm BellmanFord( $v, cost, dist, n$ )
2  // Single-source/all-destinations shortest
3  // paths with negative edge costs
4  {
5      for  $i := 1$  to  $n$  do // Initialize  $dist$ .
6           $dist[i] := cost[v, i]$ ;
7      for  $k := 2$  to  $n - 1$  do
8          for each  $u$  such that  $u \neq v$  and  $u$  has
9              at least one incoming edge do
10             for each  $\langle i, u \rangle$  in the graph do
11                 if  $dist[u] > dist[i] + cost[i, u]$  then
12                      $dist[u] := dist[i] + cost[i, u]$ ;
13 }
```

Algorithm 5.4 Bellman and Ford algorithm to compute shortest paths

first iteration in which no $dist$ values are changed, whichever occurs first. Another possibility is to maintain a queue of vertices i whose $dist$ values changed on the previous iteration of the **for** loop. These are the only values for i that need to be considered in line 10 during the next iteration. When a queue of these values is maintained, we can rewrite the loop of lines 7 to 12 so that on each iteration, a vertex i is removed from the queue, and the $dist$ values of all vertices adjacent from i are updated as in lines 11 and 12. Vertices whose $dist$ values decrease as a result of this are added to the end of the queue unless they are already on it. The loop terminates when the queue becomes empty. These two strategies to improve the performance of BellmanFord are considered in the exercises. Other strategies for improving performance are discussed in References and Readings. \square

EXERCISES

1. Find the shortest paths from node 1 to every other node in the graph of Figure 5.11 using the Bellman and Ford algorithm.
2. Prove the correctness of BellmanFord (Algorithm 5.4). Note that this algorithm does not faithfully implement the computation of the recurrence for $dist^k$. In fact, for $k < n - 1$, the $dist$ values following iteration k of the **for** loop of lines 7 to 12 may not be $dist^k$.
3. Transform BellmanFord into a program. Assume that graphs are represented using adjacency lists in which each node has an additional field

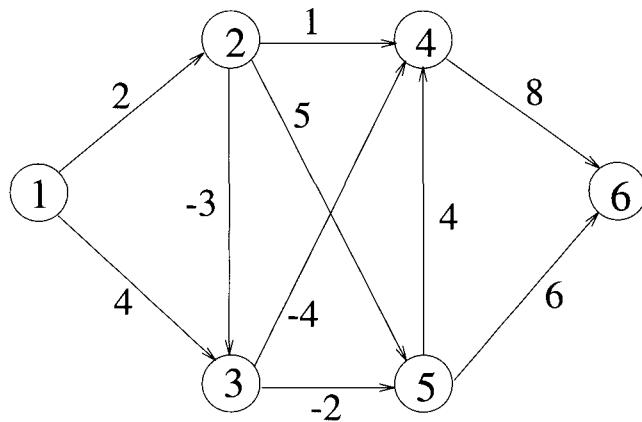


Figure 5.11 Graph for Exercise 1

called *cost* that gives the length of the edge represented by that node. As a result of this, there is no cost adjacency matrix. Generate some test graphs and test the correctness of your program.

4. Rewrite the algorithm `BellmanFord` so that the loop of lines 7 to 12 terminates either after $n - 1$ iterations or after the first iteration in which no *dist* values are changed, whichever occurs first.
5. Rewrite `BellmanFord` by replacing the loop of lines 7 to 12 with code that uses a queue of vertices that may potentially result in a reduction of other *dist* vertices. This queue initially contains all vertices that are adjacent from the source vertex v . On each successive iteration of the new loop, a vertex i is removed from the queue (unless the queue is empty), and the *dist* values to vertices adjacent from i are updated as in lines 11 and 12 of Algorithm 5.4. When the *dist* value of a vertex is reduced because of this, it is added to the queue unless it is already on the queue.
 - (a) Prove that the new algorithm produces the same results as the original one.
 - (b) Show that the complexity of the new algorithm is no more than that of the original one.
6. Compare the run-time performance of the Bellman and Ford algorithms of the preceding two exercises and that of Algorithm 5.4. For this, generate test graphs that will expose the relative performances of the three algorithms.

7. Modify algorithm BellmanFord so that it obtains the shortest paths, in addition to the lengths of these paths. What is the computing time of your algorithm?

5.5 OPTIMAL BINARY SEARCH TREES (*)

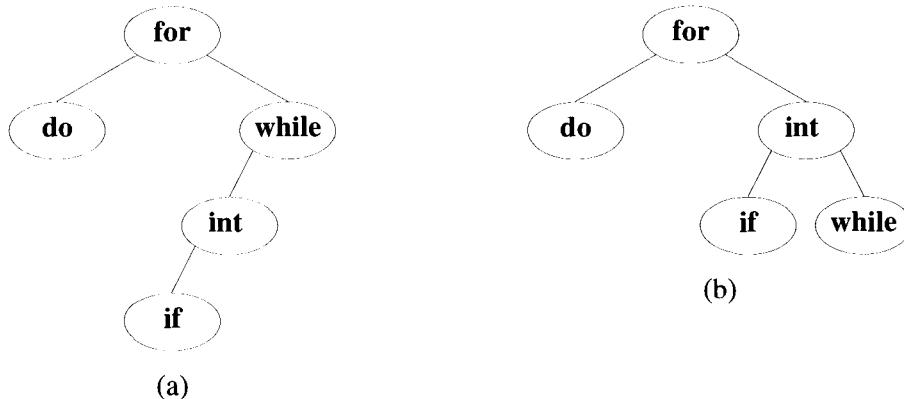


Figure 5.12 Two possible binary search trees

Given a fixed set of identifiers, we wish to create a binary search tree (see Section 2.3) organization. We may expect different binary search trees for the same identifier set to have different performance characteristics. The tree of Figure 5.12(a), in the worst case, requires four comparisons to find an identifier, whereas the tree of Figure 5.12(b) requires only three. On the average the two trees need $12/5$ and $11/5$ comparisons, respectively. For example, in the case of tree (a), it takes 1, 2, 2, 3, and 4 comparisons, respectively, to find the identifiers **for**, **do**, **while**, **int**, and **if**. Thus the average number of comparisons is $\frac{1+2+2+3+4}{5} = \frac{12}{5}$. This calculation assumes that each identifier is searched for with equal probability and that no unsuccessful searches (i.e., searches for identifiers not in the tree) are made.

In a general situation, we can expect different identifiers to be searched for with different frequencies (or probabilities). In addition, we can expect unsuccessful searches also to be made. Let us assume that the given set of identifiers is $\{a_1, a_2, \dots, a_n\}$ with $a_1 < a_2 < \dots < a_n$. Let $p(i)$ be the probability with which we search for a_i . Let $q(i)$ be the probability that the identifier x being searched for is such that $a_i < x < a_{i+1}$, $0 \leq i \leq n$ (assume $a_0 = -\infty$ and $a_{n+1} = +\infty$). Then, $\sum_{0 \leq i \leq n} q(i)$ is the probability of

an unsuccessful search. Clearly, $\sum_{1 \leq i \leq n} p(i) + \sum_{0 \leq i \leq n} q(i) = 1$. Given this data, we wish to construct an optimal binary search tree for $\{a_1, a_2, \dots, a_n\}$. First, of course, we must be precise about what we mean by an optimal binary search tree.

In obtaining a cost function for binary search trees, it is useful to add a fictitious node in place of every empty subtree in the search tree. Such nodes, called external nodes, are drawn square in Figure 5.13. All other nodes are internal nodes. If a binary search tree represents n identifiers, then there will be exactly n internal nodes and $n + 1$ (fictitious) external nodes. Every internal node represents a point where a successful search may terminate. Every external node represents a point where an unsuccessful search may terminate.

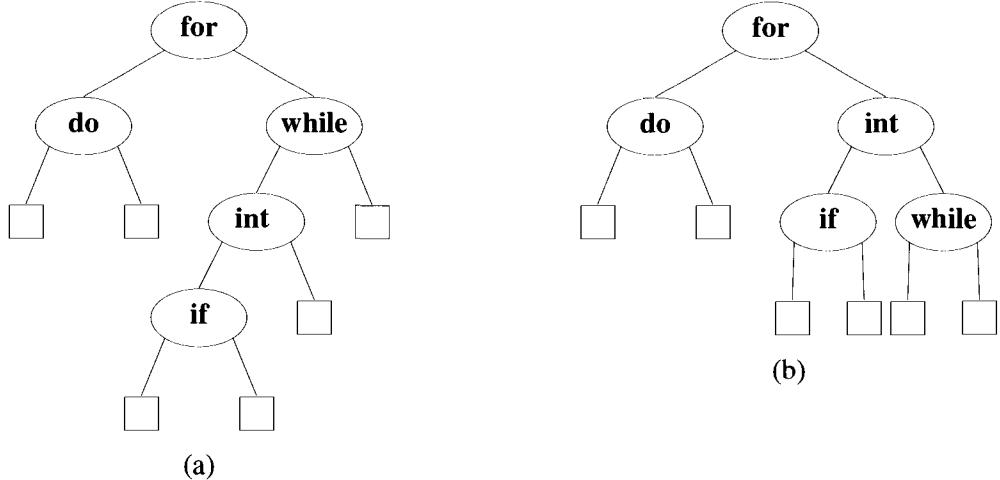


Figure 5.13 Binary search trees of Figure 5.12 with external nodes added

If a successful search terminates at an internal node at level l , then l iterations of the **while** loop of Algorithm 2.5 are needed. Hence, the expected cost contribution from the internal node for a_i is $p(i) * \text{level}(a_i)$.

Unsuccessful searches terminate with $t = 0$ (i.e., at an external node) in algorithm `ISearch` (Algorithm 2.5). The identifiers not in the binary search tree can be partitioned into $n + 1$ equivalence classes $E_i, 0 \leq i \leq n$. The class E_0 contains all identifiers x such that $x < a_1$. The class E_i contains all identifiers x such that $a_i < x < a_{i+1}, 1 \leq i < n$. The class E_n contains all identifiers $x, x > a_n$. It is easy to see that for all identifiers in the same class E_i , the search terminates at the same external node. For identifiers in different E_i the search terminates at different external nodes. If the failure

node for E_i is at level l , then only $l - 1$ iterations of the **while** loop are made. Hence, the cost contribution of this node is $q(i) * (\text{level}(E_i) - 1)$.

The preceding discussion leads to the following formula for the expected cost of a binary search tree:

$$\sum_{1 \leq i \leq n} p(i) * \text{level}(a_i) + \sum_{0 \leq i \leq n} q(i) * (\text{level}(E_i) - 1) \quad (5.9)$$

We define an optimal binary search tree for the identifier set $\{a_1, a_2, \dots, a_n\}$ to be a binary search tree for which (5.9) is minimum.

Example 5.17 The possible binary search trees for the identifier set $(a_1, a_2, a_3) = (\text{do}, \text{if}, \text{while})$ are given if Figure 5.14. With equal probabilities $p(i) = q(i) = 1/7$ for all i , we have

$$\begin{array}{llll} \text{cost(tree a)} & = & 15/7 & \text{cost(tree b)} = 13/7 \\ \text{cost(tree c)} & = & 15/7 & \text{cost(tree d)} = 15/7 \\ \text{cost(tree e)} & = & 15/7 & \end{array}$$

As expected, tree b is optimal. With $p(1) = .5$, $p(2) = .1$, $p(3) = .05$, $q(0) = .15$, $q(1) = .1$, $q(2) = .05$ and $q(3) = .05$ we have

$$\begin{array}{llll} \text{cost(tree a)} & = & 2.65 & \text{cost(tree b)} = 1.9 \\ \text{cost(tree c)} & = & 1.5 & \text{cost(tree d)} = 2.05 \\ \text{cost(tree e)} & = & 1.6 & \end{array}$$

For instance, cost(tree a) can be computed as follows. The contribution from successful searches is $3 * 0.5 + 2 * 0.1 + 0.05 = 1.75$ and the contribution from unsuccessful searches is $3 * 0.15 + 3 * 0.1 + 2 * 0.05 + 0.05 = 0.90$. All the other costs can also be calculated in a similar manner. Tree c is optimal with this assignment of p 's and q 's. \square

To apply dynamic programming to the problem of obtaining an optimal binary search tree, we need to view the construction of such a tree as the result of a sequence of decisions and then observe that the principle of optimality holds when applied to the problem state resulting from a decision. A possible approach to this would be to make a decision as to which of the a_i 's should be assigned to the root node of the tree. If we choose a_k , then it is clear that the internal nodes for a_1, a_2, \dots, a_{k-1} as well as the external nodes for the classes E_0, E_1, \dots, E_{k-1} will lie in the left subtree l of the root. The remaining nodes will be in the right subtree r . Define

$$\text{cost}(l) = \sum_{1 \leq i < k} p(i) * \text{level}(a_i) + \sum_{0 \leq i < k} q(i) * (\text{level}(E_i) - 1)$$

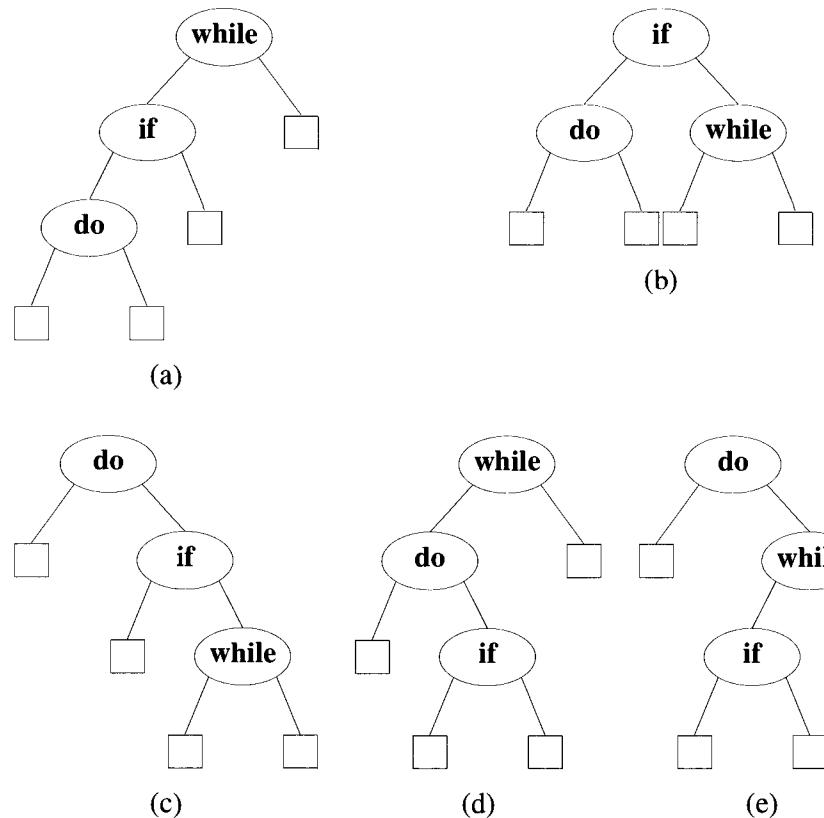


Figure 5.14 Possible binary search trees for the identifier set {do, if, while}

and

$$\text{cost}(r) = \sum_{k < i \leq n} p(i) * \text{level}(a_i) + \sum_{k < i \leq n} q(i) * (\text{level}(E_i) - 1)$$

In both cases the level is measured by regarding the root of the respective subtree to be at level 1.

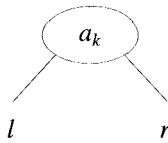


Figure 5.15 An optimal binary search tree with root a_k

Using $w(i, j)$ to represent the sum $q(i) + \sum_{l=i+1}^j (q(l) + p(l))$, we obtain the following as the expected cost of the search tree (Figure 5.15):

$$p(k) + \text{cost}(l) + \text{cost}(r) + w(0, k - 1) + w(k, n) \quad (5.10)$$

If the tree is optimal, then (5.10) must be minimum. Hence, $\text{cost}(l)$ must be minimum over all binary search trees containing a_1, a_2, \dots, a_{k-1} and E_0, E_1, \dots, E_{k-1} . Similarly $\text{cost}(r)$ must be minimum. If we use $c(i, j)$ to represent the cost of an optimal binary search tree t_{ij} containing a_{i+1}, \dots, a_j and E_i, \dots, E_j , then for the tree to be optimal, we must have $\text{cost}(l) = c(0, k - 1)$ and $\text{cost}(r) = c(k, n)$. In addition, k must be chosen such that

$$p(k) + c(0, k - 1) + c(k, n) + w(0, k - 1) + w(k, n)$$

is minimum. Hence, for $c(0, n)$ we obtain

$$c(0, n) = \min_{1 \leq k \leq n} \{c(0, k - 1) + c(k, n) + p(k) + w(0, k - 1) + w(k, n)\} \quad (5.11)$$

We can generalize (5.11) to obtain for any $c(i, j)$

$$c(i, j) = \min_{i < k \leq j} \{c(i, k - 1) + c(k, j) + p(k) + w(i, k - 1) + w(k, j)\}$$

$$c(i, j) = \min_{i < k \leq j} \{c(i, k - 1) + c(k, j)\} + w(i, j) \quad (5.12)$$

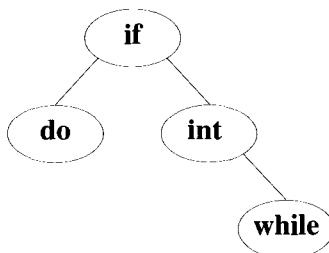
Equation 5.12 can be solved for $c(0, n)$ by first computing all $c(i, j)$ such that $j - i = 1$ (note $c(i, i) = 0$ and $w(i, i) = q(i)$, $0 \leq i \leq n$). Next we can compute all $c(i, j)$ such that $j - i = 2$, then all $c(i, j)$ with $j - i = 3$, and so on. If during this computation we record the root $r(i, j)$ of each tree t_{ij} , then an optimal binary search tree can be constructed from these $r(i, j)$. Note that $r(i, j)$ is the value of k that minimizes (5.12).

Example 5.18 Let $n = 4$ and $(a_1, a_2, a_3, a_4) = (\text{do, if, int, while})$. Let $p(1 : 4) = (3, 3, 1, 1)$ and $q(0 : 4) = (2, 3, 1, 1, 1)$. The p 's and q 's have been multiplied by 16 for convenience. Initially, we have $w(i, i) = q(i)$, $c(i, i) = 0$ and $r(i, i) = 0$, $0 \leq i \leq 4$. Using Equation 5.12 and the observation $w(i, j) = p(j) + q(j) + w(i, j - 1)$, we get

$$\begin{aligned} w(0, 1) &= p(1) + q(1) + w(0, 0) = 8 \\ c(0, 1) &= w(0, 1) + \min\{c(0, 0) + c(1, 1)\} = 8 \\ r(0, 1) &= 1 \\ w(1, 2) &= p(2) + q(2) + w(1, 1) = 7 \\ c(1, 2) &= w(1, 2) + \min\{c(1, 1) + c(2, 2)\} = 7 \\ r(0, 2) &= 2 \\ w(2, 3) &= p(3) + q(3) + w(2, 2) = 3 \\ c(2, 3) &= w(2, 3) + \min\{c(2, 2) + c(3, 3)\} = 3 \\ r(2, 3) &= 3 \\ w(3, 4) &= p(4) + q(4) + w(3, 3) = 3 \\ c(3, 4) &= w(3, 4) + \min\{c(3, 3) + c(4, 4)\} = 3 \\ r(3, 4) &= 4 \end{aligned}$$

Knowing $w(i, i + 1)$ and $c(i, i + 1)$, $0 \leq i < 4$, we can again use Equation 5.12 to compute $w(i, i + 2)$, $c(i, i + 2)$, and $r(i, i + 2)$, $0 \leq i < 3$. This process can be repeated until $w(0, 4)$, $c(0, 4)$, and $r(0, 4)$ are obtained. The table of Figure 5.16 shows the results of this computation. The box in row i and column j shows the values of $w(j, j + i)$, $c(j, j + i)$ and $r(j, j + i)$ respectively. The computation is carried out by row from row 0 to row 4. From the table we see that $c(0, 4) = 32$ is the minimum cost of a binary search tree for (a_1, a_2, a_3, a_4) . The root of tree t_{04} is a_2 . Hence, the left subtree is t_{01} and the right subtree t_{24} . Tree t_{01} has root a_1 and subtrees t_{00} and t_{11} . Tree t_{24} has root a_3 ; its left subtree is t_{22} and its right subtree t_{34} . Thus, with the data in the table it is possible to reconstruct t_{04} . Figure 5.17 shows t_{04} . \square

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

Figure 5.16 Computation of $c(0,4)$, $w(0,4)$, and $r(0,4)$ **Figure 5.17** Optimal search tree for Example 5.18

The above example illustrates how Equation 5.12 can be used to determine the c 's and r 's and also how to reconstruct t_{0n} knowing the r 's. Let us examine the complexity of this procedure to evaluate the c 's and r 's. The evaluation procedure described in the above example requires us to compute $c(i, j)$ for $(j - i) = 1, 2, \dots, n$ in that order. When $j - i = m$, there are $n - m + 1$ $c(i, j)$'s to compute. The computation of each of these $c(i, j)$'s requires us to find the minimum of m quantities (see Equation 5.12). Hence, each such $c(i, j)$ can be computed in time $O(m)$. The total time for all $c(i, j)$'s with $j - i = m$ is therefore $O(nm - m^2)$. The total time to evaluate all the $c(i, j)$'s and $r(i, j)$'s is therefore

$$\sum_{1 \leq m \leq n} (nm - m^2) = O(n^3)$$

We can do better than this using a result due to D. E. Knuth which shows that the optimal k in Equation 5.12 can be found by limiting the search to the range $r(i, j - 1) \leq k \leq r(i + 1, j)$. In this case the computing time becomes $O(n^2)$ (see the exercises). The function OBST (Algorithm 5.5) uses this result to obtain the values of $w(i, j)$, $r(i, j)$, and $c(i, j)$, $0 \leq i \leq j \leq n$, in $O(n^2)$ time. The tree t_{0n} can be constructed from the values of $r(i, j)$ in $O(n)$ time. The algorithm for this is left as an exercise.

EXERCISES

1. Use function OBST (Algorithm 5.5) to compute $w(i, j)$, $r(i, j)$, and $c(i, j)$, $0 \leq i < j \leq 4$, for the identifier set $(a_1, a_2, a_3, a_4) = (\text{cout}, \text{float}, \text{if}, \text{while})$ with $p(1) = 1/20$, $p(2) = 1/5$, $p(3) = 1/10$, $p(4) = 1/20$, $q(0) = 1/5$, $q(1) = 1/10$, $q(2) = 1/5$, $q(3) = 1/20$, and $q(4) = 1/20$. Using the $r(i, j)$'s, construct the optimal binary search tree.
2. (a) Show that the computing time of function OBST (Algorithm 5.5) is $O(n^2)$.
(b) Write an algorithm to construct the optimal binary search tree given the roots $r(i, j)$, $0 \leq i < j \leq n$. Show that this can be done in time $O(n)$.
3. Since often only the approximate values of the p 's and q 's are known, it is perhaps just as meaningful to find a binary search tree that is nearly optimal. That is, its cost, Equation 5.9, is almost minimal for the given p 's and q 's. This exercise explores an $O(n \log n)$ algorithm that results in nearly optimal binary search trees. The search tree heuristic we use is

```

1  Algorithm OBST( $p, q, n$ )
2  // Given  $n$  distinct identifiers  $a_1 < a_2 < \dots < a_n$  and probabilities
3  //  $p[i]$ ,  $1 \leq i \leq n$ , and  $q[i]$ ,  $0 \leq i \leq n$ , this algorithm computes
4  // the cost  $c[i, j]$  of optimal binary search trees  $t_{ij}$  for identifiers
5  //  $a_{i+1}, \dots, a_j$ . It also computes  $r[i, j]$ , the root of  $t_{ij}$ .
6  //  $w[i, j]$  is the weight of  $t_{ij}$ .
7  {
8      for  $i := 0$  to  $n - 1$  do
9      {
10         // Initialize.
11          $w[i, i] := q[i]; r[i, i] := 0; c[i, i] := 0.0;$ 
12         // Optimal trees with one node
13          $w[i, i + 1] := q[i] + q[i + 1] + p[i + 1];$ 
14          $r[i, i + 1] := i + 1;$ 
15          $c[i, i + 1] := q[i] + q[i + 1] + p[i + 1];$ 
16     }
17      $w[n, n] := q[n]; r[n, n] := 0; c[n, n] := 0.0;$ 
18     for  $m := 2$  to  $n$  do // Find optimal trees with  $m$  nodes.
19     {
20         for  $i := 0$  to  $n - m$  do
21         {
22              $j := i + m;$ 
23              $w[i, j] := w[i, j - 1] + p[j] + q[j];$ 
24             // Solve 5.12 using Knuth's result.
25              $k := \text{Find}(c, r, i, j);$ 
26             // A value of  $l$  in the range  $r[i, j - 1] \leq l \leq r[i + 1, j]$  that minimizes  $c[i, l - 1] + c[l, j];$ 
27              $c[i, j] := w[i, j] + c[i, k - 1] + c[k, j];$ 
28              $r[i, j] := k;$ 
29         }
30         write ( $c[0, n], w[0, n], r[0, n]$ );
31     }
32 }

1  Algorithm Find( $c, r, i, j$ )
2  {
3       $min := \infty;$ 
4      for  $m := r[i, j - 1]$  to  $r[i + 1, j]$  do
5          if  $(c[i, m - 1] + c[m, j]) < min$  then
6          {
7               $min := c[i, m - 1] + c[m, j]; l := m;$ 
8          }
9      return  $l;$ 
10 }

```

Choose the root k such that $|w(0, k - 1) - w(k, n)|$ is as small as possible. Repeat this procedure to find the left and right subtrees of the root.

- (a) Using this heuristic, obtain the resulting binary search tree for the data of Exercise 1. What is its cost?
- (b) Write an algorithm implementing the above heuristic. Your algorithm should have time complexity $O(n \log n)$.

5.6 STRING EDITING

We are given two strings $X = x_1, x_2, \dots, x_n$ and $Y = y_1, y_2, \dots, y_m$, where x_i , $1 \leq i \leq n$, and y_j , $1 \leq j \leq m$, are members of a finite set of symbols known as the *alphabet*. We want to transform X into Y using a sequence of *edit operations* on X . The permissible edit operations are insert, delete, and change (a symbol of X into another), and there is a cost associated with performing each. The cost of a sequence of operations is the sum of the costs of the individual operations in the sequence. The problem of string editing is to identify a minimum-cost sequence of edit operations that will transform X into Y .

Let $D(x_i)$ be the cost of deleting the symbol x_i from X , $I(y_j)$ be the cost of inserting the symbol y_j into X , and $C(x_i, y_j)$ be the cost of changing the symbol x_i of X into y_j .

Example 5.19 Consider the sequences $X = x_1, x_2, x_3, x_4, x_5 = a, a, b, a, b$ and $Y = y_1, y_2, y_3, y_4 = b, a, b, b$. Let the cost associated with each insertion and deletion be 1 (for any symbol). Also let the cost of changing any symbol to any other symbol be 2. One possible way of transforming X into Y is delete each x_i , $1 \leq i \leq 5$, and insert each y_j , $1 \leq j \leq 4$. The total cost of this edit sequence is 9. Another possible edit sequence is delete x_1 and x_2 and insert y_4 at the end of string X . The total cost is only 3. \square

A solution to the string editing problem consists of a sequence of decisions, one for each edit operation. Let \mathcal{E} be a minimum-cost edit sequence for transforming X into Y . The first operation, O , in \mathcal{E} is delete, insert, or change. If $\mathcal{E}' = \mathcal{E} - \{O\}$ and X' is the result of applying O on X , then \mathcal{E}' should be a minimum-cost edit sequence that transforms X' into Y . Thus the principle of optimality holds for this problem. A dynamic programming solution for this problem can be obtained as follows. Define $cost(i, j)$ to be the minimum cost of any edit sequence for transforming x_1, x_2, \dots, x_i into y_1, y_2, \dots, y_j (for $0 \leq i \leq n$ and $0 \leq j \leq m$). Compute $cost(i, j)$ for each i and j . Then $cost(n, m)$ is the cost of an optimal edit sequence.

For $i = j = 0$, $cost(i, j) = 0$, since the two sequences are identical (and empty). Also, if $j = 0$ and $i > 0$, we can transform X into Y by a sequence of

deletes. Thus, $\text{cost}(i, 0) = \text{cost}(i-1, 0) + D(x_i)$. Similarly, if $i = 0$ and $j > 0$, we get $\text{cost}(0, j) = \text{cost}(0, j-1) + I(y_j)$. If $i \neq 0$ and $j \neq 0$, x_1, x_2, \dots, x_i can be transformed into y_1, y_2, \dots, y_j in one of three ways:

1. Transform x_1, x_2, \dots, x_{i-1} into y_1, y_2, \dots, y_j using a minimum-cost edit sequence and then delete x_i . The corresponding cost is $\text{cost}(i-1, j) + D(x_i)$.
2. Transform x_1, x_2, \dots, x_{i-1} into y_1, y_2, \dots, y_{j-1} using a minimum-cost edit sequence and then change the symbol x_i to y_j . The associated cost is $\text{cost}(i-1, j-1) + C(x_i, y_j)$.
3. Transform x_1, x_2, \dots, x_i into y_1, y_2, \dots, y_{j-1} using a minimum-cost edit sequence and then insert y_j . This corresponds to a cost of $\text{cost}(i, j-1) + I(y_j)$.

The minimum cost of any edit sequence that transforms x_1, x_2, \dots, x_i into y_1, y_2, \dots, y_j (for $i > 0$ and $j > 0$) is the minimum of the above three costs, according to the principle of optimality. Therefore, we arrive at the following recurrence equation for $\text{cost}(i, j)$:

$$\text{cost}(i, j) = \begin{cases} 0 & i = j = 0 \\ \text{cost}(i-1, 0) + D(x_i) & j = 0, i > 0 \\ \text{cost}(0, j-1) + I(y_j) & i = 0, j > 0 \\ \text{cost}'(i, j) & i > 0, j > 0 \end{cases} \quad (5.13)$$

$$\text{where } \text{cost}'(i, j) = \min \{ \text{cost}(i-1, j) + D(x_i), \\ \text{cost}(i-1, j-1) + C(x_i, y_j), \\ \text{cost}(i, j-1) + I(y_j) \}$$

We have to compute $\text{cost}(i, j)$ for all possible values of i and j ($0 \leq i \leq n$ and $0 \leq j \leq m$). There are $(n+1)(m+1)$ such values. These values can be computed in the form of a table, M , where each row of M corresponds to a particular value of i and each column of M corresponds to a specific value of j . $M(i, j)$ stores the value $\text{cost}(i, j)$. The zeroth row can be computed first since it corresponds to performing a series of insertions. Likewise the zeroth column can also be computed. After this, one could compute the entries of M in row-major order, starting from the first row. Rows should be processed in the order $1, 2, \dots, n$. Entries in any row are computed in increasing order of column number.

The entries of M can also be computed in column-major order, starting from the first column. Looking at Equation 5.13, we see that each entry of M takes only $O(1)$ time to compute. Therefore the whole algorithm takes $O(mn)$ time. The value $\text{cost}(n, m)$ is the final answer we are interested in. Having computed all the entries of M , a minimum edit sequence can be

obtained by a simple backward trace from $\text{cost}(n, m)$. This backward trace is enabled by recording which of the three options for $i > 0, j > 0$ yielded the minimum cost for each i and j .

Example 5.20 Consider the string editing problem of Example 5.19. $X = a, a, b, a, b$ and $Y = b, a, b, b$. Each insertion and deletion has a unit cost and a change costs 2 units. For the cases $i = 0, j > 1$, and $j = 0, i > 1$, $\text{cost}(i, j)$ can be computed first (Figure 5.18). Let us compute the rest of the entries in row-major order. The next entry to be computed is $\text{cost}(1, 1)$.

$$\begin{aligned}\text{cost}(1, 1) &= \min \{\text{cost}(0, 1) + D(x_1), \text{cost}(0, 0) + C(x_1, y_1), \text{cost}(1, 0) + I(y_1)\} \\ &= \min \{2, 2, 2\} = 2\end{aligned}$$

Next is computed $\text{cost}(1, 2)$.

$$\begin{aligned}\text{cost}(1, 2) &= \min \{\text{cost}(0, 2) + D(x_1), \text{cost}(0, 1) + C(x_1, y_2), \text{cost}(1, 1) + I(y_2)\} \\ &= \min \{3, 1, 3\} = 1\end{aligned}$$

The rest of the entries are computed similarly. Figure 5.18 displays the whole table. The value $\text{cost}(5, 4) = 3$. One possible minimum-cost edit sequence is delete x_1 , delete x_2 , and insert y_4 . Another possible minimum cost edit sequence is change x_1 to y_2 and delete x_4 . \square

		0	1	2	3	4
		0	1	2	3	4
i	0	0	1	2	3	4
	1	1	2	1	2	3
2	2	3	2	3	4	
3	3	2	3	2	3	
4	4	3	2	3	4	
5	5	4	3	2	3	

Figure 5.18 Cost table for Example 5.20

EXERCISES

- Let $X = a, a, b, a, a, b, a, b, a, a$ and $Y = b, a, b, a, a, b, a, b$. Find a minimum-cost edit sequence that transforms X into Y .
- Present a pseudocode algorithm that implements the string editing algorithm discussed in this section. Program it and test its correctness using suitable data.
- Modify the above program not only to compute $\text{cost}(n, m)$ but also to output a minimum-cost edit sequence. What is the time complexity of your program?
- Given a sequence X of symbols, a subsequence of X is defined to be any contiguous portion of X . For example, if $X = x_1, x_2, x_3, x_4, x_5$, x_2, x_3 and x_1, x_2, x_3 are subsequences of X . Given two sequences X and Y , present an algorithm that will identify the longest subsequence that is common to both X and Y . This problem is known as *the longest common subsequence problem*. What is the time complexity of your algorithm?

5.7 0/1 KNAPSACK

The terminology and notation used in this section is the same as that in Section 5.1. A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables x_1, x_2, \dots, x_n . A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that decisions on the x_i are made in the order x_n, x_{n-1}, \dots, x_1 . Following a decision on x_n , we may be in one of two possible states: the capacity remaining in the knapsack is m and no profit has accrued or the capacity remaining is $m - w_n$ and a profit of p_n has accrued. It is clear that the remaining decisions x_{n-1}, \dots, x_1 must be optimal with respect to the problem state resulting from the decision on x_n . Otherwise, x_n, \dots, x_1 will not be optimal. Hence, the principle of optimality holds.

Let $f_j(y)$ be the value of an optimal solution to $\text{KNAP}(1, j, y)$. Since the principle of optimality holds, we obtain

$$f_n(m) = \max \{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\} \quad (5.14)$$

For arbitrary $f_i(y)$, $i > 0$, Equation 5.14 generalizes to

$$f_i(y) = \max \{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \quad (5.15)$$

Equation 5.15 can be solved for $f_n(m)$ by beginning with the knowledge $f_0(y) = 0$ for all y and $f_i(y) = -\infty$, $y < 0$. Then f_1, f_2, \dots, f_n can be successively computed using (5.15).

When the w_i 's are integer, we need to compute $f_i(y)$ for integer y , $0 \leq y \leq m$. Since $f_i(y) = -\infty$ for $y < 0$, these function values need not be computed explicitly. Since each f_i can be computed from f_{i-1} in $\Theta(m)$ time, it takes $\Theta(mn)$ time to compute f_n . When the w_i 's are real numbers, $f_i(y)$ is needed for real numbers y such that $0 \leq y \leq m$. So, f_i cannot be explicitly computed for all y in this range. Even when the w_i 's are integer, the explicit $\Theta(mn)$ computation of f_n may not be the most efficient computation. So, we explore an alternative method for both cases.

Notice that $f_i(y)$ is an ascending step function; i.e., there are a finite number of y 's, $0 = y_1 < y_2 < \dots < y_k$, such that $f_i(y_1) < f_i(y_2) < \dots < f_i(y_k)$; $f_i(y) = -\infty$, $y < y_1$; $f_i(y) = f(y_k)$, $y \geq y_k$; and $f_i(y) = f_i(y_j)$, $y_j \leq y < y_{j+1}$. So, we need to compute only $f_i(y_j)$, $1 \leq j \leq k$. We use the ordered set $S^i = \{(f(y_j), y_j) | 1 \leq j \leq k\}$ to represent $f_i(y)$. Each member of S^i is a pair (P, W) , where $P = f_i(y_j)$ and $W = y_j$. Notice that $S^0 = \{(0, 0)\}$. We can compute S^{i+1} from S^i by first computing

$$S_1^i = \{(P, W) | (P - p_i, W - w_i) \in S^i\} \quad (5.16)$$

Now, S^{i+1} can be computed by merging the pairs in S^i and S_1^i together. Note that if S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$, then the pair (P_j, W_j) can be discarded because of (5.15). Discarding or purging rules such as this one are also known as *dominance rules*. Dominated tuples get purged. In the above, (P_k, W_k) dominates (P_j, W_j) .

Interestingly, the strategy we have come up with can also be derived by attempting to solve the knapsack problem via a systematic examination of the up to 2^n possibilities for x_1, x_2, \dots, x_n . Let S^i represent the possible states resulting from the 2^i decision sequences for x_1, \dots, x_i . A state refers to a pair (P_j, W_j) , W_j being the total weight of objects included in the knapsack and P_j being the corresponding profit. To obtain S^{i+1} , we note that the possibilities for x_{i+1} are $x_{i+1} = 0$ or $x_{i+1} = 1$. When $x_{i+1} = 0$, the resulting states are the same as for S^i . When $x_{i+1} = 1$, the resulting states are obtained by adding (p_{i+1}, w_{i+1}) to each state in S^i . Call the set of these additional states S_1^i . The S_1^i is the same as in Equation 5.16. Now, S^{i+1} can be computed by merging the states in S^i and S_1^i together.

Example 5.21 Consider the knapsack instance $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$, and $m = 6$. For these data we have

$$\begin{aligned} S^0 &= \{(0, 0)\}; S_1^0 = \{(1, 2)\} \\ S^1 &= \{(0, 0), (1, 2)\}; S_1^1 = \{(2, 3), (3, 5)\} \\ S^2 &= \{(0, 0), (1, 2), (2, 3), (3, 5)\}; S_1^2 = \{(5, 4), (6, 6), (7, 7), (8, 9)\} \\ S^3 &= \{(0, 0), (1, 2), (2, 3), (5, 4), (6, 6), (7, 7), (8, 9)\} \end{aligned}$$

Note that the pair (3, 5) has been eliminated from S^3 as a result of the purging rule stated above. \square

When generating the S^i 's, we can also purge all pairs (P, W) with $W > m$ as these pairs determine the value of $f_n(x)$ only for $x > m$. Since the knapsack capacity is m , we are not interested in the behavior of f_n for $x > m$. When all pairs (P_j, W_j) with $W_j > m$ are purged from the S^i 's, $f_n(m)$ is given by the P value of the last pair in S^n (note that the S^i 's are ordered sets). Note also that by computing S^n , we can find the solutions to all the knapsack problems $\text{KNAP}(1, n, x)$, $0 \leq x \leq m$, and not just $\text{KNAP}(1, n, m)$. Since, we want only a solution to $\text{KNAP}(1, n, m)$, we can dispense with the computation of S^n . The last pair in S^n is either the last one in S^{n-1} or it is $(P_j + p_n, W_j + w_n)$, where $(P_j, W_j) \in S^{n-1}$ such that $W_j + w_n \leq m$ and W_j is maximum.

If $(P1, W1)$ is the last tuple in S^n , a set of 0/1 values for the x_i 's such that $\sum p_i x_i = P1$ and $\sum w_i x_i = W1$ can be determined by carrying out a search through the S^i 's. We can set $x_n = 0$ if $(P1, W1) \in S^{n-1}$. If $(P1, W1) \notin S^{n-1}$, then $(P1 - p_n, W1 - w_n) \in S^{n-1}$ and we can set $x_n = 1$. This leaves us to determine how either $(P1, W1)$ or $(P1 - p_n, W1 - w_n)$ was obtained in S^{n-1} . This can be done recursively.

Example 5.22 With $m = 6$, the value of $f_3(6)$ is given by the tuple (6, 6) in S^3 (Example 5.21). The tuple (6, 6) $\notin S^2$, and so we must set $x_3 = 1$. The pair (6, 6) came from the pair $(6 - p_3, 6 - w_3) = (1, 2)$. Hence (1, 2) $\in S^2$. Since (1, 2) $\in S^1$, we can set $x_2 = 0$. Since (1, 2) $\notin S^0$, we obtain $x_1 = 1$. Hence an optimal solution is $(x_1, x_2, x_3) = (1, 0, 1)$. \square

We can sum up all we have said so far in the form of an informal algorithm DKP (Algorithm 5.6). To evaluate the complexity of the algorithm, we need to specify how the sets S^i and S_1^i are to be represented; provide an algorithm to merge S^i and S_1^i ; and specify an algorithm that will trace through S^{n-1}, \dots, S^1 and determine a set of 0/1 values for x_n, \dots, x_1 .

We can use an array $\text{pair}[]$ to represent all the pairs (P, W) . The P values are stored in $\text{pair}[].p$ and the W values in $\text{pair}[].w$. Sets S^0, S^1, \dots, S^{n-1} can be stored adjacent to each other. This requires the use of pointers $b[i]$, $0 \leq i \leq n$, where $b[i]$ is the location of the first element in S^i , $0 \leq i < n$, and $b[n]$ is one more than the location of the last element in S^{n-1} .

Example 5.23 Using the representation above, the sets S^0, S^1 , and S^2 of Example 5.21 appear as

```

1  Algorithm DKP( $p, w, n, m$ )
2  {
3       $S^0 := \{(0, 0)\}$ ;
4      for  $i := 1$  to  $n - 1$  do
5      {
6           $S_1^{i-1} := \{(P, W) | (P - p_i, W - w_i) \in S^{i-1} \text{ and } W \leq m\}$ ;
7           $S^i := \text{MergePurge}(S^{i-1}, S_1^{i-1})$ ;
8      }
9      ( $PX, WX$ ) := last pair in  $S^{n-1}$ ;
10     ( $PY, WY$ ) :=  $(P' + p_n, W' + w_n)$  where  $W'$  is the largest  $W$  in
11         any pair in  $S^{n-1}$  such that  $W + w_n \leq m$ ;
12     // Trace back for  $x_n, x_{n-1}, \dots, x_1$ .
13     if ( $PX > PY$ ) then  $x_n := 0$ ;
14     else  $x_n := 1$ ;
15     TraceBackFor( $x_{n-1}, \dots, x_1$ );
16 }

```

Algorithm 5.6 Informal knapsack algorithm

	1	2	3	4	5	6	7
$pair[].p$	0	0	1	0	1	2	3
$pair[].w$	0	0	2	0	2	3	5

\uparrow \uparrow \uparrow \uparrow \uparrow \square
 $b[0]$ $b[1]$ $b[2]$ $b[3]$

The merging and purging of S^{i-1} and S_1^{i-1} can be carried out at the same time that S_1^{i-1} is generated. Since the pairs in S^{i-1} are in increasing order of P and W , the pairs for S^i are generated in this order. If the next pair generated for S_1^{i-1} is (PQ, WQ) , then we can merge into S^i all pairs from S^{i-1} with W value $\leq WQ$. The purging rule can be used to decide whether any pairs get purged. Hence, no additional space is needed in which to store S_1^{i-1} .

DKnap (Algorithm 5.7) generates S^i from S^{i-1} in this way. The S^i 's are generated in the **for** loop of lines 7 to 42 of Algorithm 5.7. At the start of each iteration $t = b[i - 1]$ and h is the index of the last pair in S^{i-1} . The variable k points to the next tuple in S^{i-1} that has to be merged into S^i . In line 10, the function **Largest** determines the largest q , $t \leq q \leq h$,

for which $\text{pair}[q].w + w[i] \leq m$. This can be done by performing a binary search. The code for this function is left as an exercise. Since u is set such that for all $W_j, h \geq j > u$, $W_j + w_i > m$, the pairs for S_1^{i-1} are $(P(j) + p_i, W(j) + w_i)$, $1 \leq j \leq u$. The **for** loop of lines 11 to 33 generates these pairs. Each time a pair (pp, ww) is generated, all pairs (P, W) in S^{i-1} with $W < ww$ not yet purged or merged into S^i are merged into S^i . Note that none of these may be purged. Lines 21 to 25 handle the case when the next pair in S^{i-1} has a W value equal to ww . In this case the pair with lesser P value gets purged. In case $pp > P(\text{next} - 1)$, then the pair (pp, ww) gets purged. Otherwise, (pp, ww) is added to S^i . The **while** loop of lines 31 and 32 purges all unmerged pairs in S^{i-1} that can be purged at this time. Finally, following the merging of S_1^{i-1} into S^i , there may be pairs remaining in S^{i-1} to be merged into S^i . This is taken care of in the **while** loop of lines 35 to 39. Note that because of lines 31 and 32, none of these pairs can be purged. Function `TraceBack` (line 43) implements the **if** statement and trace-back step of the function `DKP` (Algorithm 5.6). This is left as an exercise.

If $|S^i|$ is the number of pairs in S^i , then the array `pair` should have a minimum dimension of $d = \sum_{0 \leq i \leq n-1} |S^i|$. Since it is not possible to predict the exact space needed, it is necessary to test for $\text{next} > d$ each time next is incremented. Since each S^i , $i > 0$, is obtained by merging S^{i-1} and S_1^{i-1} and $|S_1^{i-1}| \leq |S^{i-1}|$, it follows that $|S^i| \leq 2|S^{i-1}|$. In the worst case no pairs will get purged and

$$\sum_{0 \leq i \leq n-1} |S^i| = \sum_{0 \leq i \leq n-1} 2^i = 2^n - 1$$

The time needed to generate S^i from S^{i-1} is $\Theta(|S^{i-1}|)$. Hence, the time needed to compute all the S^i 's, $0 \leq i < n$, is $\Theta(\sum |S^{i-1}|)$. Since $|S^i| \leq 2^i$, the time needed to compute all the S^i 's is $O(2^n)$. If the p_j 's are integers, then each pair (P, W) in S^i has an integer P and $P \leq \sum_{1 \leq j \leq i} p_j$. Similarly, if the w_j 's are integers, each W is an integer and $W \leq m$. In any S^i the pairs have distinct W values and also distinct P values. Hence,

$$|S^i| \leq 1 + \sum_{1 \leq j \leq i} p_j$$

when the p_j 's are integers and

$$|S^i| \leq 1 + \min \left\{ \sum_{1 \leq j \leq i} w_j, m \right\}$$

```

1  PW = record {float p; float w; }
2  {
3      // pair[ ] is an array of PW's.
4      b[0] := 1; pair[1].p := pair[1].w := 0.0; // S0
5      t := 1; h := 1; // Start and end of S0
6      b[1] := next := 2; // Next free spot in pair[ ]
7      for i := 1 to n - 1 do
8          { // Generate Si.
9              k := t;
10             u := Largest(pair, w, t, h, i, m);
11             for j := t to u do
12                 { // Generate S1i-1 and merge.
13                     pp := pair[j].p + p[i]; ww := pair[j].w + w[i];
14                     // (pp, ww) is the next element in S1i-1.
15                     while ((k ≤ h) and (pair[k].w ≤ ww)) do
16                         {
17                             pair[next].p := pair[k].p;
18                             pair[next].w := pair[k].w;
19                             next := next + 1; k := k + 1;
20                         }
21                         if ((k ≤ h) and (pair[k].w = ww)) then
22                             {
23                                 if pp < pair[k].p then pp := pair[k].p;
24                                 k := k + 1;
25                             }
26                         if pp > pair[next - 1].p then
27                             {
28                                 pair[next].p := pp; pair[next].w := ww;
29                                 next := next + 1 ;
30                             }
31                         while ((k ≤ h) and (pair[k].p ≤ pair[next - 1].p))
32                             do k := k + 1;
33                     }
34                     // Merge in remaining terms from Si-1.
35                     while (k ≤ h) do
36                         {
37                             pair[next].p := pair[k].p; pair[next].w := pair[k].w;
38                             next := next + 1; k := k + 1;
39                         }
40                     // Initialize for Si+1.
41                     t := h + 1; h := next - 1; b[i + 1] := next;
42                 }
43                 TraceBack(p, w, pair, x, m, n);
44             }

```

when the w_j 's are integers. When both the p_j 's and w_j 's are integers, the time and space complexity of DKnap (excluding the time for TraceBack) is $O(\min\{2^n, n \sum_{1 \leq i \leq n} p_i, nm\})$. In this bound $\sum_{1 \leq i \leq n} p_i$ can be replaced by $\sum_{1 \leq i \leq n} p_i / \gcd(p_1, \dots, p_n)$ and m by $\gcd(w_1, w_2, \dots, w_n, m)$ (see the exercises). The exercises indicate how TraceBack may be implemented so as to have a space complexity $O(1)$ and a time complexity $O(n^2)$.

Although the above analysis may seem to indicate that DKnap requires too much computational resource to be practical for large n , in practice many instances of this problem can be solved in a reasonable amount of time. This happens because usually, all the p 's and w 's are integers and m is much smaller than 2^n . The purging rule is effective in purging most of the pairs that would otherwise remain in the S^i 's.

Algorithm DKnap can be speeded up by the use of heuristics. Let L be an estimate on the value of an optimal solution such that $f_n(m) \geq L$. Let $\text{PLEFT}(i) = \sum_{i < j \leq n} p_j$. If S^i contains a tuple (P, W) such that $P + \text{PLEFT}(i) < L$, then (P, W) can be purged from S^i . To see this, observe that (P, W) can contribute at best the pair $(P + \sum_{i < j \leq n} p_j, W + \sum_{i < j \leq n} w)$ to S^{n-1} . Since $P + \sum_{i < j \leq n} p_j = P + \text{PLEFT}(i) < L$, it follows that this pair cannot lead to a pair with value at least L and so cannot determine an optimal solution. A simple way to estimate L such that $L \leq f_n(m)$ is to consider the last pair (P, W) in S^i . Then, $P \leq f_n(m)$. A better estimate is obtained by adding some of the remaining objects to (P, W) . Example 5.24 illustrates this. Heuristics for the knapsack problem are discussed in greater detail in the chapter on branch-and-bound. The exercises explore a divide-and-conquer approach to speed up DKnap so that the worst case time is $O(2^{n/2})$.

Example 5.24 Consider the following instance of the knapsack problem: $n = 6$, $(p_1, p_2, p_3, p_4, p_5, p_6) = (w_1, w_2, w_3, w_4, w_5, w_6) = (100, 50, 20, 10, 10, 7, 3)$, and $m = 165$. Attempting to fill the knapsack using objects in the order 1, 2, 3, 4, 5, and 6, we see that objects 1, 2, 4, and 6 fit in and yield a profit of 163 and a capacity utilization of 163. We can thus begin with $L = 163$ as a value with the property $L \leq f_n(m)$. Since $p_i = w_i$, every pair $(P, W) \in S^i$, $0 \leq i \leq 6$ has $P = W$. Hence, each pair can be replaced by the singleton P or W . $\text{PLEFT}(0) = 190$, $\text{PLEFT}(1) = 90$, $\text{PLEFT}(2) = 40$, $\text{PLEFT}(3) = 20$, $\text{PLEFT}(4) = 10$, $\text{PLEFT}(5) = 3$, and $\text{PLEFT}(6) = 0$. Eliminating from each S^i any singleton P such that $P + \text{PLEFT}(i) < L$, we obtain

$$\begin{aligned} S^0 &= \{0\}; & S_1^0 &= \{100\} \\ S^1 &= \{100\}; & S_1^1 &= \{150\} \\ S^2 &= \{150\}; & S_1^2 &= \emptyset \end{aligned}$$

$$\begin{aligned} S^3 &= \{150\}; \quad S_1^3 = \{160\} \\ S^4 &= \{160\}; \quad S_1^4 = \emptyset \\ S^5 &= \{160\} \end{aligned}$$

The singleton 0 is deleted from S^1 as $0 + \text{PLEFT}(1) < 163$. The set S_1^2 does not contain the singleton $150 + 20 = 170$ as $m < 170$. S^3 does not contain the 100 or the 120 as each is less than $L - \text{PLEFT}(3)$. And so on. The value $f_6(165)$ can be determined from S^5 . In this example, the value of L did not change. In general, L will change if a better estimate is obtained as a result of the computation of some S^i . If the heuristic wasn't used, then the computation would have proceeded as

$$\begin{aligned} S^0 &= \{0\} \\ S^1 &= \{0, 100\} \\ S^2 &= \{0, 50, 100, 150\} \\ S^3 &= \{0, 20, 50, 70, 100, 120, 150\} \\ S^4 &= \{0, 10, 20, 30, 50, 60, 70, 80, 100, 110, 120, 130, 150, 160\} \\ S^5 &= \{0, 7, 10, 17, 20, 27, 30, 37, 50, 57, 60, 67, 70, 77, 80, 87, 100, \\ &\quad 107, 110, 117, 120, 127, 130, 137, 150, 157, 160\} \end{aligned}$$

The value $f_6(165)$ can now be determined from S^5 , using the knowledge $(p_6, w_6) = (3, 3)$. \square

EXERCISES

1. Generate the sets S^i , $0 \leq i \leq 4$ (Equation 5.16), when $(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$ and $(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$.
2. Write a function `Largest(pair, w, t, h, i, m)` that uses binary search to determine the largest q , $t \leq q \leq h$, such that $\text{pair}[q].w + w[i] \leq m$.
3. Write a function `TraceBack` to determine an optimal solution x_1, x_2, \dots, x_n to the knapsack problem. Assume that S^i , $0 \leq i < n$, have already been computed as in function `DKnap`. Knowing $b(i)$ and $b(i+1)$, you can use a binary search to determine whether $(P', W') \in S^i$. Hence, the time complexity of your algorithm should be no more than $O(n \max_i \{\log |S^i|\}) = O(n^2)$.
4. Give an example of a set of knapsack instances for which $|S^i| = 2^i$, $0 \leq i \leq n$. Your set should include one instance for each n .

5. (a) Show that if the p_j 's are integers, then the size of each S^i , $|S^i|$, in the knapsack problem is no more than $1 + \sum_{1 \leq i \leq j} p_j / \gcd(p_1, p_2, \dots, p_n)$, where $\gcd(p_1, p_2, \dots, p_n)$ is the greatest common divisor of the p_i 's.
 (b) Show that when the w_j 's are integer, then $|S^i| \leq 1 + \min\{\sum_{1 \leq j \leq i} w_j, m\} / \gcd(w_1, w_2, \dots, w_n, m)$.
6. (a) Using a divide-and-conquer approach coupled with the set generation approach of the text, show how to obtain an $O(2^{n/2})$ algorithm for the 0/1 knapsack problem.
 (b) Develop an algorithm that uses this approach to solve the 0/1 knapsack problem.
 (c) Compare the run time and storage requirements of this approach with those of Algorithm 5.7. Use suitable test data.
7. Consider the integer knapsack problem obtained by replacing the 0/1 constraint in (5.2) by $x_i \geq 0$ and integer. Generalize $f_i(x)$ to this problem in the obvious way.
 - (a) Obtain the dynamic programming recurrence relation corresponding to (5.15).
 - (b) Show how to transform this problem into a 0/1 knapsack problem. (*Hint:* Introduce new 0/1 variables for each x_i . If $0 \leq x_i < 2^j$, then introduce j variables, one for each bit in the binary representation of x_i .)

5.8 RELIABILITY DESIGN

In this section we look at an example of how to use dynamic programming to solve a problem with a multiplicative optimization function. The problem is to design a system that is composed of several devices connected in series (Figure 5.19). Let r_i be the reliability of device D_i (that is, r_i is the probability that device i will function properly). Then, the reliability of the entire system is $\prod r_i$. Even if the individual devices are very reliable (the r_i 's are very close to one), the reliability of the system may not be very good. For example, if $n = 10$ and $r_i = .99$, $1 \leq i \leq 10$, then $\prod r_i = .904$. Hence, it is desirable to duplicate devices. Multiple copies of the same device type are connected in parallel (Figure 5.20) through the use of switching circuits. The switching circuits determine which devices in any given group are functioning properly. They then make use of one such device at each stage.

If stage i contains m_i copies of device D_i , then the probability that all m_i have a malfunction is $(1 - r_i)^{m_i}$. Hence the reliability of stage i becomes

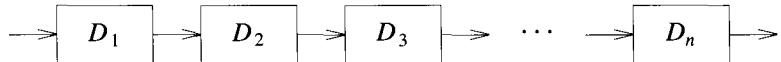


Figure 5.19 n devices D_i , $1 \leq i \leq n$, connected in series

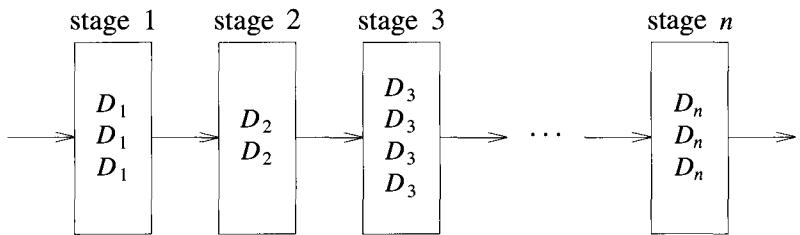


Figure 5.20 Multiple devices connected in parallel in each stage

$1 - (1 - r_i)^{m_i}$. Thus, if $r_i = .99$ and $m_i = 2$, the stage reliability becomes .9999. In any practical situation, the stage reliability is a little less than $1 - (1 - r_i)^{m_i}$ because the switching circuits themselves are not fully reliable. Also, failures of copies of the same device may not be fully independent (e.g., if failure is due to design defect). Let us assume that the reliability of stage i is given by a function $\phi_i(m_i)$, $1 \leq i \leq n$. (It is quite conceivable that $\phi_i(m_i)$ may decrease after a certain value of m_i .) The reliability of the system of stages is $\prod_{1 \leq i \leq n} \phi_i(m_i)$.

Our problem is to use device duplication to maximize reliability. This maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed. We wish to solve the following maximization problem:

$$\text{maximize } \prod_{1 \leq i \leq n} \phi_i(m_i)$$

$$\text{subject to } \sum_{1 \leq i \leq n} c_i m_i \leq c \quad (5.17)$$

$$m_i \geq 1 \text{ and integer, } 1 \leq i \leq n$$

A dynamic programming solution can be obtained in a manner similar to that used for the knapsack problem. Since, we can assume each $c_i > 0$, each m_i must be in the range $1 \leq m_i \leq u_i$, where

$$u_i = \left\lfloor (c + c_i - \sum_1^n c_j)/c_i \right\rfloor$$

The upper bound u_i follows from the observation that $m_j \geq 1$. An optimal solution m_1, m_2, \dots, m_n is the result of a sequence of decisions, one decision for each m_i . Let $f_i(x)$ represent the maximum value of $\Pi_{1 \leq j \leq i} \phi(m_j)$ subject to the constraints $\sum_{1 \leq j \leq i} c_j m_j \leq x$ and $1 \leq m_j \leq u_j$, $1 \leq j \leq i$. Then, the value of an optimal solution is $f_n(c)$. The last decision made requires one to choose m_n from $\{1, 2, 3, \dots, u_n\}$. Once a value for m_n has been chosen, the remaining decisions must be such as to use the remaining funds $c - c_n m_n$ in an optimal way. The principle of optimality holds and

$$f_n(c) = \max_{1 \leq m_n \leq u_n} \{\phi_n(m_n) f_{n-1}(c - c_n m_n)\} \quad (5.18)$$

For any $f_i(x)$, $i \geq 1$, this equation generalizes to

$$f_i(x) = \max_{1 \leq m_i \leq u_i} \{\phi_i(m_i) f_{i-1}(x - c_i m_i)\} \quad (5.19)$$

Clearly, $f_0(x) = 1$ for all x , $0 \leq x \leq c$. Hence, (5.19) can be solved using an approach similar to that used for the knapsack problem. Let S^i consist of tuples of the form (f, x) , where $f = f_i(x)$. There is at most one tuple for each different x that results from a sequence of decisions on m_1, m_2, \dots, m_n . The dominance rule (f_1, x_1) dominates (f_2, x_2) iff $f_1 \geq f_2$ and $x_1 \leq x_2$ holds for this problem too. Hence, dominated tuples can be discarded from S^i .

Example 5.25 We are to design a three stage system with device types D_1, D_2 , and D_3 . The costs are \$30, \$15, and \$20 respectively. The cost of the system is to be no more than \$105. The reliability of each device type is .9, .8 and .5 respectively. We assume that if stage i has m_i devices of type i in parallel, then $\phi_i(m_i) = 1 - (1 - r_i)^{m_i}$. In terms of the notation used earlier, $c_1 = 30$, $c_2 = 15$, $c_3 = 20$, $c = 105$, $r_1 = .9$, $r_2 = .8$, $r_3 = .5$, $u_1 = 2$, $u_2 = 3$, and $u_3 = 3$.

We use S^i to represent the set of all undominated tuples (f, x) that may result from the various decision sequences for m_1, m_2, \dots, m_i . Hence, $f(x) = f_i(x)$. Beginning with $S^0 = \{(1, 0)\}$, we can obtain each S^i from S^{i-1} by trying out all possible values for m_i and combining the resulting tuples together. Using S_j^i to represent all tuples obtainable from S^{i-1} by choosing $m_i = j$, we obtain $S_1^1 = \{(.9, 30)\}$ and $S_2^1 = \{ (.9, 30), (.99, 60)\}$. The set

$S_1^2 = \{(.72, 45), (.792, 75)\}$; $S_2^2 = \{(.864, 60)\}$. Note that the tuple $(.9504, 90)$ which comes from $(.99, 60)$ has been eliminated from S_2^2 as this leaves only \$10. This is not enough to allow $m_3 = 1$. The set $S_3^2 = \{(.8928, 75)\}$. Combining, we get $S^2 = \{(.72, 45), (.864, 60), (.8928, 75)\}$ as the tuple $(.792, 75)$ is dominated by $(.864, 60)$. The set $S_1^3 = \{(.36, 65), (.432, 80), (.4464, 95)\}$, $S_2^3 = \{(.54, 85), (.648, 100)\}$, and $S_3^3 = \{(.63, 105)\}$. Combining, we get $S^3 = \{(.36, 65), (.432, 80), (.54, 85), (.648, 100)\}$.

The best design has a reliability of .648 and a cost of 100. Tracing back through the S^i 's, we determine that $m_1 = 1$, $m_2 = 2$, and $m_3 = 2$. \square

As in the case of the knapsack problem, a complete dynamic programming algorithm for the reliability problem will use heuristics to reduce the size of the S^i 's. There is no need to retain any tuple (f, x) in S^i with x value greater than $c - \sum_{i \leq j \leq n} c_j$ as such a tuple will not leave adequate funds to complete the system. In addition, we can devise a simple heuristic to determine the best reliability obtainable by completing a tuple (f, x) in S^i . If this is less than a heuristically determined lower bound on the optimal system reliability, then (f, x) can be eliminated from S^i .

EXERCISE

1. (a) Present an algorithm similar to DKnap to solve the recurrence (5.19).
 (b) What are the time and space requirements of your algorithm?
 (c) Test the correctness of your algorithm using suitable test data.

5.9 THE TRAVELING SALESPERSON PROBLEM

We have seen how to apply dynamic programming to a subset selection problem (0/1 knapsack). Now we turn our attention to a permutation problem. Note that permutation problems usually are much harder to solve than subset problems as there are $n!$ different permutations of n objects whereas there are only 2^n different subsets of n objects ($n! > 2^n$). Let $G = (V, E)$ be a directed graph with edge costs c_{ij} . The variable c_{ij} is defined such that $c_{ij} > 0$ for all i and j and $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$. Let $|V| = n$ and assume $n > 1$. A *tour* of G is a directed simple cycle that includes every vertex in V . The cost of a tour is the sum of the cost of the edges on the tour. The *traveling salesperson problem* is to find a tour of minimum cost.

The traveling salesperson problem finds application in a variety of situations. Suppose we have to route a postal van to pick up mail from mail

boxes located at n different sites. An $n + 1$ vertex graph can be used to represent the situation. One vertex represents the post office from which the postal van starts and to which it must return. Edge $\langle i, j \rangle$ is assigned a cost equal to the distance from site i to site j . The route taken by the postal van is a tour, and we are interested in finding a tour of minimum length.

As a second example, suppose we wish to use a robot arm to tighten the nuts on some piece of machinery on an assembly line. The arm will start from its initial position (which is over the first nut to be tightened), successively move to each of the remaining nuts, and return to the initial position. The path of the arm is clearly a tour on a graph in which vertices represent the nuts. A minimum-cost tour will minimize the time needed for the arm to complete its task (note that only the total arm movement time is variable; the nut tightening time is independent of the tour).

Our final example is from a production environment in which several commodities are manufactured on the same set of machines. The manufacture proceeds in cycles. In each production cycle, n different commodities are produced. When the machines are changed from production of commodity i to commodity j , a change over cost c_{ij} is incurred. It is desired to find a sequence in which to manufacture these commodities. This sequence should minimize the sum of change over costs (the remaining production costs are sequence independent). Since the manufacture proceeds cyclically, it is necessary to include the cost of starting the next cycle. This is just the change over cost from the last to the first commodity. Hence, this problem can be regarded as a traveling salesperson problem on an n vertex graph with edge cost c_{ij} 's being the changeover cost from commodity i to commodity j .

In the following discussion we shall, without loss of generality, regard a tour to be a simple path that starts and ends at vertex 1. Every tour consists of an edge $\langle 1, k \rangle$ for some $k \in V - \{1\}$ and a path from vertex k to vertex 1. The path from vertex k to vertex 1 goes through each vertex in $V - \{1, k\}$ exactly once. It is easy to see that if the tour is optimal, then the path from k to 1 must be a shortest k to 1 path going through all vertices in $V - \{1, k\}$. Hence, the principle of optimality holds. Let $g(i, S)$ be the length of a shortest path starting at vertex i , going through all vertices in S , and terminating at vertex 1. The function $g(1, V - \{1\})$ is the length of an optimal salesperson tour. From the principal of optimality it follows that

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, k\})\} \quad (5.20)$$

Generalizing (5.20), we obtain (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \quad (5.21)$$

Equation 5.20 can be solved for $g(1, V - \{1\})$ if we know $g(k, V - \{1, k\})$ for all choices of k . The g values can be obtained by using (5.21). Clearly,

$g(i, \phi) = c_{i1}$, $1 \leq i \leq n$. Hence, we can use (5.21) to obtain $g(i, S)$ for all S of size 1. Then we can obtain $g(i, S)$ for S with $|S| = 2$, and so on. When $|S| < n - 1$, the values of i and S for which $g(i, S)$ is needed are such that $i \neq 1$, $1 \notin S$, and $i \notin S$.

Example 5.26 Consider the directed graph of Figure 5.21(a). The edge lengths are given by matrix c of Figure 5.21(b).

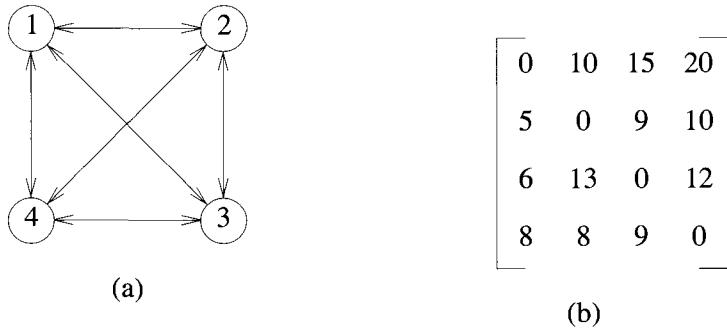


Figure 5.21 Directed graph and edge length matrix c

Thus $g(2, \phi) = c_{21} = 5$, $g(3, \phi) = c_{31} = 6$, and $g(4, \phi) = c_{41} = 8$. Using (5.21), we obtain

$$\begin{array}{rclcrcl} g(2, \{3\}) & = & c_{23} + g(3, \phi) & = & 15 & & g(2, \{4\}) & = & 18 \\ g(3, \{2\}) & = & 18 & & & & g(3, \{4\}) & = & 20 \\ g(4, \{2\}) & = & 13 & & & & g(4, \{3\}) & = & 15 \end{array}$$

Next, we compute $g(i, S)$ with $|S| = 2$, $i \neq 1$, $1 \notin S$ and $i \notin S$.

$$\begin{array}{rclcrcl} g(2, \{3, 4\}) & = & \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} & = & 25 \\ g(3, \{2, 4\}) & = & \min \{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\} & = & 25 \\ g(4, \{2, 3\}) & = & \min \{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\} & = & 23 \end{array}$$

Finally, from (5.20) we obtain

$$\begin{aligned} g(1, \{2, 3, 4\}) & = \min \{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\} \\ & = \min \{35, 40, 43\} \\ & = 35 \end{aligned}$$

An optimal tour of the graph of Figure 5.21(a) has length 35. A tour of this length can be constructed if we retain with each $g(i, S)$ the value of j that minimizes the right-hand side of (5.21). Let $J(i, S)$ be this value. Then, $J(1, \{2, 3, 4\}) = 2$. Thus the tour starts from 1 and goes to 2. The remaining tour can be obtained from $g(2, \{3, 4\})$. So $J(2, \{3, 4\}) = 4$. Thus the next edge is $\langle 2, 4 \rangle$. The remaining tour is for $g(4, \{3\})$. So $J(4, \{3\}) = 3$. The optimal tour is 1, 2, 4, 3, 1. \square

Let N be the number of $g(i, S)$'s that have to be computed before (5.20) can be used to compute $g(1, V - \{1\})$. For each value of $|S|$ there are $n - 1$ choices for i . The number of distinct sets S of size k not including 1 and i is $\binom{n-2}{k}$. Hence

$$N = \sum_{k=0}^{n-2} (n-1) \binom{n-2}{k} = (n-1)2^{n-2}$$

An algorithm that proceeds to find an optimal tour by using (5.20) and (5.21) will require $\Theta(n^2 2^n)$ time as the computation of $g(i, S)$ with $|S| = k$ requires $k - 1$ comparisons when solving (5.21). This is better than enumerating all $n!$ different tours to find the best one. The most serious drawback of this dynamic programming solution is the space needed, $O(n2^n)$. This is too large even for modest values of n .

EXERCISE

1. (a) Obtain a data representation for the values $g(i, S)$ of the traveling salesperson problem. Your representation should allow for easy access to the value of $g(i, S)$, given i and S . (i) How much space does your representation need for an n vertex graph? (ii) How much time is needed to retrieve or update the value of $g(i, S)$?
- (b) Using the representation of (a), develop an algorithm corresponding to the dynamic programming solution of the traveling salesperson problem.
- (c) Test the correctness of your algorithm using suitable test data.

5.10 FLOW SHOP SCHEDULING

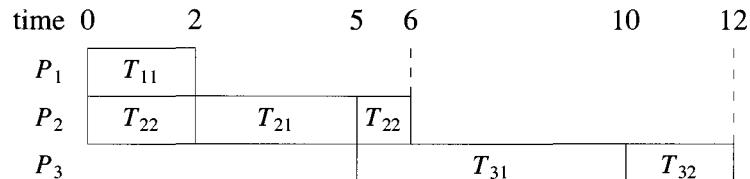
Often the processing of a job requires the performance of several distinct tasks. Computer programs run in a multiprogramming environment are input and then executed. Following the execution, the job is queued for output

and the output eventually printed. In a general flow shop we may have n jobs each requiring m tasks $T_{1i}, T_{2i}, \dots, T_{mi}$, $1 \leq i \leq n$, to be performed. Task T_{ji} is to be performed on processor P_j , $1 \leq j \leq m$. The time required to complete task T_{ji} is t_{ji} . A schedule for the n jobs is an assignment of tasks to time intervals on the processors. Task T_{ji} must be assigned to processor P_j . No processor may have more than one task assigned to it in any time interval. Additionally, for any job i the processing of task T_{ji} , $j > 1$, cannot be started until task $T_{j-1,i}$ has been completed.

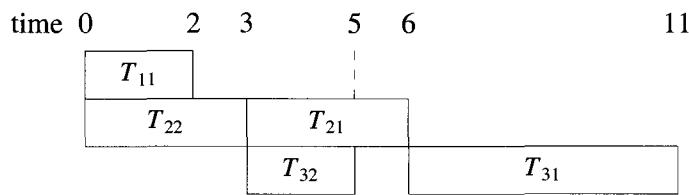
Example 5.27 Two jobs have to be scheduled on three processors. The task times are given by the matrix \mathcal{J}

$$\mathcal{J} = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$$

Two possible schedules for the jobs are shown in Figure 5.22. □



(a)



(b)

Figure 5.22 Two possible schedules for Example 5.27

A *nonpreemptive* schedule is a schedule in which the processing of a task on any processor is not terminated until the task is complete. A schedule for which this need not be true is called *preemptive*. The schedule of Figure 5.22(a) is a preemptive schedule. Figure 5.22(b) shows a nonpreemptive schedule. The *finish time* $f_i(S)$ of job i is the time at which all tasks of job i have been completed in schedule S . In Figure 5.22(a), $f_1(S) = 10$ and $f_2(S) = 12$. In Figure 5.22(b), $f_1(S) = 11$ and $f_2(S) = 5$. The finish time $F(S)$ of a schedule S is given by

$$F(S) = \max_{1 \leq i \leq n} \{f_i(S)\} \quad (5.22)$$

The *mean flow time* MFT(S) is defined to be

$$\text{MFT}(S) = \frac{1}{n} \sum_{1 \leq i \leq n} f_i(S) \quad (5.23)$$

An optimal finish time (OFT) schedule for a given set of jobs is a non-preemptive schedule S for which $F(S)$ is minimum over all nonpreemptive schedules S . A preemptive optimal finish time (POFT) schedule, optimal mean finish time schedule (OMFT), and preemptive optimal mean finish (POMFT) schedule are defined in the obvious way.

Although the general problem of obtaining OFT and POFT schedules for $m > 2$ and of obtaining OMFT schedules is computationally difficult (see Chapter 11), dynamic programming leads to an efficient algorithm to obtain OFT schedules for the case $m = 2$. In this section we consider this special case.

For convenience, we shall use a_i to represent t_{1i} , and b_i to represent t_{2i} . For the two-processor case, one can readily verify that nothing is to be gained by using different processing orders on the two processors (this is not true for $m > 2$). Hence, a schedule is completely specified by providing a permutation of the jobs. Jobs will be executed on each processor in this order. Each task will be started at the earliest possible time. The schedule of Figure 5.23 is completely specified by the permutation (5, 1, 3, 2, 4). We make the simplifying assumption that $a_i \neq 0$, $1 \leq i \leq n$. Note that if jobs with $a_i = 0$ are allowed, then an optimal schedule can be constructed by first finding an optimal permutation for all jobs with $a_i \neq 0$ and then adding all jobs with $a_i = 0$ (in any order) in front of this permutation (see the exercises).

It is easy to see that an optimal permutation (schedule) has the property that given the first job in the permutation, the remaining permutation is optimal with respect to the state the two processors are in following the completion of the first job. Let $\sigma_1, \sigma_2, \dots, \sigma_k$ be a permutation prefix defining a schedule for jobs T_1, T_2, \dots, T_k . For this schedule let f_1 and f_2 be the times at which the processing of jobs T_1, T_2, \dots, T_k is completed on processors P_1

P_1	a_5	a_1		a_3	a_2	a_4	
P_2		b_5		b_1	b_3	b_2	

Figure 5.23 A schedule

and P_2 respectively. Let $t = f_2 - f_1$. The state of the processors following the sequence of decisions T_1, T_2, \dots, T_k is completely characterized by t . Let $g(S, t)$ be the length of an optimal schedule for the subset of jobs S under the assumption that processor 2 is not available until time t . The length of an optimal schedule for the job set $\{1, 2, \dots, n\}$ is $g(\{1, 2, \dots, n\}, 0)$.

Since the principle of optimality holds, we obtain

$$g(\{1, 2, \dots, n\}, 0) = \min_{1 \leq i \leq n} \{a_i + g(\{1, 2, \dots, n\} - \{i\}, b_i)\} \quad (5.24)$$

Equation 5.24 generalizes to (5.25) for arbitrary S and t . This generalization requires that $g(\phi, t) = \max\{t, 0\}$ and that $a_i \neq 0$, $1 \leq i \leq n$.

$$g(S, t) = \min_{i \in S} \{a_i + g(S - \{i\}, b_i + \max\{t - a_i, 0\})\} \quad (5.25)$$

The term $\max\{t - a_i, 0\}$ comes into (5.25) as task T_{2i} cannot start until $\max\{a_i, t\}$ (P_2 is not available until time t). Hence $f_2 - f_1 = b_i + \max\{a_i, t\} - a_i = b_i + \max\{t - a_i, 0\}$. We can solve for $g(S, t)$ using an approach similar to that used to solve (5.21). However, it turns out that (5.25) can be solved algebraically and a very simple rule to generate an optimal schedule obtained.

Consider any schedule R for a subset of jobs S . Assume that P_2 is not available until time t . Let i and j be the first two jobs in this schedule. Then, from (5.25) we obtain

$$\begin{aligned} g(S, t) &= a_i + g(S - \{i\}, b_i + \max\{t - a_i, 0\}) \\ g(S, t) &= a_i + a_j + g(S - \{i, j\}, b_j + \max\{b_i + \max\{t - a_i, 0\} - a_j, 0\}) \end{aligned} \quad (5.26)$$

Equation 5.26 can be simplified using the following result:

$$\begin{aligned} t_{ij} &= b_j + \max \{b_i + \max \{t - a_i, 0\} - a_j, 0\} \\ &= b_j + b_i - a_j + \max \{\max \{t - a_i, 0\}, a_j - b_i\} \\ &= b_j + b_i - a_j + \max \{t - a_i, a_j - b_i, 0\} \\ t_{ij} &= b_j + b_i - a_j - a_i + \max \{t, a_i + a_j - b_i, a_i\} \end{aligned} \quad (5.27)$$

If jobs i and j are interchanged in R , then the finish time $g'(S, t)$ is

$$g'(S, t) = a_i + a_j + g(S - \{i, j\}, t_{ji})$$

$$\text{where } t_{ji} = b_j + b_i - a_j - a_i + \max \{t, a_i + a_j - b_j, a_j\}$$

Comparing $g(S, t)$ and $g'(S, t)$, we see that if (5.28) below holds, then $g(S, t) \leq g'(S, t)$.

$$\max \{t, a_i + a_j - b_i, a_i\} \leq \max \{t, a_i + a_j - b_j, a_j\} \quad (5.28)$$

In order for (5.28) to hold for all values of t , we need

$$\max \{a_i + a_j - b_i, a_i\} \leq \max \{a_i + a_j - b_j, a_j\}$$

$$\text{or } a_i + a_j + \max \{-b_i, -a_j\} \leq a_i + a_j + \max \{-b_j, -a_i\}$$

$$\text{or } \min \{b_i, a_j\} \geq \min \{b_j, a_i\} \quad (5.29)$$

From (5.29) we can conclude that there exists an optimal schedule in which for every pair (i, j) of adjacent jobs, $\min\{b_i, a_j\} \geq \min\{b_j, a_i\}$. Exercise 4 shows that all schedules with this property have the same length. Hence, it suffices to generate any schedule for which (5.29) holds for every pair of adjacent jobs. We can obtain a schedule with this property by making the following observations from (5.29). If $\min\{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n\}$ is a_i , then job i should be the first job in an optimal schedule. If $\min\{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n\}$ is b_j , then job j should be the last job in an optimal schedule. This enables us to make a decision as to the positioning of one of the n jobs. Equation 5.29 can now be used on the remaining $n - 1$ jobs to correctly position another job, and so on. The scheduling rule resulting from (5.29) is therefore:

1. Sort all the a_i 's and b_j 's into nondecreasing order.
2. Consider this sequence in this order. If the next number in the sequence is a_j and job j hasn't yet been scheduled, schedule job j at the leftmost available spot. If the next number is b_j and job j hasn't yet been scheduled, schedule job j at the rightmost available spot. If j has already been scheduled, go to the next number in the sequence.

Note that the above rule also correctly positions jobs with $a_i = 0$. Hence, these jobs need not be considered separately.

Example 5.28 Let $n = 4$, $(a_1, a_2, a_3, a_4) = (3, 4, 8, 10)$, and $(b_1, b_2, b_3, b_4) = (6, 2, 9, 15)$. The sorted sequence of a 's and b 's is $(b_2, a_1, a_2, b_1, a_3, b_3, a_4, b_4) = (2, 3, 4, 6, 8, 9, 10, 15)$. Let $\sigma_1, \sigma_2, \sigma_3$, and σ_4 be the optimal schedule. Since the smallest number is b_2 , we set $\sigma_4 = 2$. The next number is a_1 and we set $\sigma_1 = a_1$. The next smallest number is a_2 . Job 2 has already been scheduled. The next number is b_1 . Job 1 has already been scheduled. The next is a_3 and we set σ_3 . This leaves σ_2 free and job 4 unscheduled. Thus, $\sigma_3 = 4$. \square

The scheduling rule above can be implemented to run in time $O(n \log n)$ (see exercises). Solving (5.24) and (5.25) directly for $g(1, 2, \dots, n, 0)$ for the optimal schedule will take $\Omega(2^n)$ time as there are these many different S 's for which $g(S, t)$ will be computed.

EXERCISES

1. N jobs are to be processed. Two machines A and B are available. If job i is processed on machine A , then a_i units of processing time are needed. If it is processed on machine B , then b_i units of processing time are needed. Because of the peculiarities of the jobs and the machines, it is quite possible that $a_i \geq b_i$ for some i while $a_j < b_j$ for some j , $j \neq i$. Obtain a dynamic programming formulation to determine the minimum time needed to process all the jobs. Note that jobs cannot be split between machines. Indicate how you would go about solving the recurrence relation obtained. Do this on an example of your choice. Also indicate how you would determine an optimal assignment of jobs to machines.
2. N jobs have to be scheduled for processing on one machine. Associated with job i is a 3-tuple (p_i, t_i, d_i) . The variable t_i is the processing time needed to complete job i . If job i is completed by its deadline d_i , then a profit p_i is earned. If not, then nothing is earned. From Section 4.4 we know that J is a subset of jobs that can all be completed by their

deadlines iff the jobs in J can be processed in nondecreasing order of deadlines without violating any deadline. Assume $d_i \leq d_{i+1}$, $1 \leq i < n$. Let $f_i(x)$ be the maximum profit that can be earned from a subset J of jobs when $n = i$. Here $f_n(d_n)$ is the value of an optimal selection of jobs J . Let $f_0(x) = 0$. Show that for $x \leq t_i$,

$$f_i(x) = \max \{f_{i-1}(x), f_{i-1}(x - t_i) + p_i\}$$

3. Let I be any instance of the two-processor flow shop problem.
 - (a) Show that the length of every POFT schedule for I is the same as the length of every OFT schedule for I . Hence, the algorithm of Section 5.10 also generates a POFT schedule.
 - (b) Show that there exists an OFT schedule for I in which jobs are processed in the same order on both processors.
 - (c) Show that there exists an OFT schedule for I defined by some permutation σ of the jobs (see part (b)) such that all jobs with $a_i = 0$ are at the front of this permutation. Further, show that the order in which these jobs appear at the front of the permutation is not important.
4. Let I be any instance of the two-processor flow shop problem. Let $\sigma = \sigma_1 \sigma_2 \cdots \sigma_n$ be a permutation defining an OFT schedule for I .
 - (a) Use (5.29) to argue that there exists an OFT σ such that $\min \{b_i, a_j\} \geq \min \{b_j, a_i\}$ for every i and j such that $i = \sigma_k$ and $j = \sigma_{k+1}$ (that is, i and j are adjacent).
 - (b) For a σ satisfying the conditions of part (a), show that $\min \{b_i, a_j\} \geq \min \{b_j, a_i\}$ for every i and j such that $i = \sigma_k$ and $j = \sigma_r$, $k < r$.
 - (c) Show that all schedules corresponding to σ 's satisfying the conditions of part (a) have the same finish time. (*Hint:* use part (b) to transform one of two different schedules satisfying (a) into the other without increasing the finish time.)

5.11 REFERENCES AND READINGS

Two classic references on dynamic programming are:

Introduction to Dynamic Programming, by G. Nemhauser, John Wiley and Sons, 1966.

Applied Dynamic Programming by R. E. Bellman and S. E. Dreyfus, Princeton University Press, 1962.

UNIT - 5

BACK TRACKING

Chapter 7

BACKTRACKING

7.1 THE GENERAL METHOD

In the search for fundamental principles of algorithm design, backtracking represents one of the most general techniques. Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation. The name backtrack was first coined by D. H. Lehmer in the 1950s. Early workers who studied the process were R. J. Walker, who gave an algorithmic account of it in 1960, and S. Golomb and L. Baumert who presented a very general description of it as well as a variety of applications.

In many applications of the backtrack method, the desired solution is expressible as an n -tuple (x_1, \dots, x_n) , where the x_i are chosen from some finite set S_i . Often the problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a *criterion function* $P(x_1, \dots, x_n)$. Sometimes it seeks all vectors that satisfy P . For example, sorting the array of integers in $a[1 : n]$ is a problem whose solution is expressible by an n -tuple, where x_i is the index in a of the i th smallest element. The criterion function P is the inequality $a[x_i] \leq a[x_{i+1}]$ for $1 \leq i < n$. The set S_i is finite and includes the integers 1 through n . Though sorting is not usually one of the problems solved by backtracking, it is one example of a familiar problem whose solution can be formulated as an n -tuple. In this chapter we study a collection of problems whose solutions are best done using backtracking.

Suppose m_i is the size of set S_i . Then there are $m = m_1 m_2 \cdots m_n$ n -tuples that are possible candidates for satisfying the function P . The *brute force approach* would be to form all these n -tuples, evaluate each one with P , and save those which yield the optimum. The backtrack algorithm has as its virtue the ability to yield the same answer with far fewer than m trials. Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions $P_i(x_1, \dots, x_i)$ (sometimes called

bounding functions) to test whether the vector being formed has any chance of success. The major advantage of this method is this: if it is realized that the partial vector (x_1, x_2, \dots, x_i) can in no way lead to an optimal solution, then $m_{i+1} \cdots m_n$ possible test vectors can be ignored entirely.

Many of the problems we solve using backtracking require that all the solutions satisfy a complex set of constraints. For any problem these constraints can be divided into two categories: *explicit* and *implicit*.

Definition 7.1 Explicit constraints are rules that restrict each x_i to take on values only from a given set. \square

Common examples of explicit constraints are

$$\begin{array}{lll} x_i \geq 0 & \text{or} & S_i = \{\text{all nonnegative real numbers}\} \\ x_i = 0 \text{ or } 1 & \text{or} & S_i = \{0, 1\} \\ l_i \leq x_i \leq u_i & \text{or} & S_i = \{a : l_i \leq a \leq u_i\} \end{array}$$

The explicit constraints depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible *solution space* for I .

Definition 7.2 The implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus implicit constraints describe the way in which the x_i must relate to each other. \square

Example 7.1 [8-queens] A classic combinatorial problem is to place eight queens on an 8×8 chessboard so that no two “attack,” that is, so that no two of them are on the same row, column, or diagonal. Let us number the rows and columns of the chessboard 1 through 8 (Figure 7.1). The queens can also be numbered 1 through 8. Since each queen must be on a different row, we can without loss of generality assume queen i is to be placed on row i . All solutions to the 8-queens problem can therefore be represented as 8-tuples (x_1, \dots, x_8) , where x_i is the column on which queen i is placed. The explicit constraints using this formulation are $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq 8$. Therefore the solution space consists of 8^8 8-tuples. The implicit constraints for this problem are that no two x_i ’s can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal. The first of these two constraints implies that all solutions are permutations of the 8-tuple $(1, 2, 3, 4, 5, 6, 7, 8)$. This realization reduces the size of the solution space from 8^8 tuples to $8!$ tuples. We see later how to formulate the second constraint in terms of the x_i . Expressed as an 8-tuple, the solution in Figure 7.1 is $(4, 6, 8, 2, 7, 1, 3, 5)$. \square

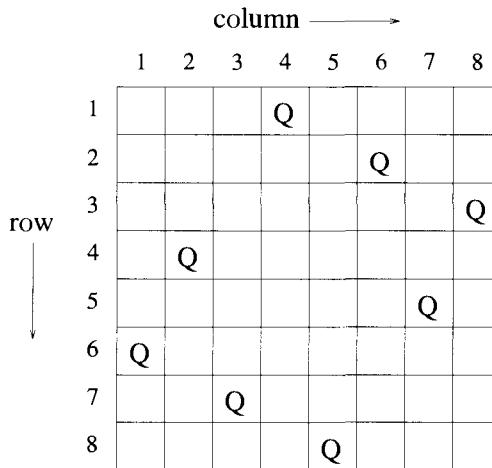


Figure 7.1 One solution to the 8-queens problem

Example 7.2 [Sum of subsets] Given positive numbers w_i , $1 \leq i \leq n$, and m , this problem calls for finding all subsets of the w_i whose sums are m . For example, if $n = 4$, $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$, and $m = 31$, then the desired subsets are $(11, 13, 7)$ and $(24, 7)$. Rather than represent the solution vector by the w_i which sum to m , we could represent the solution vector by giving the indices of these w_i . Now the two solutions are described by the vectors $(1, 2, 4)$ and $(3, 4)$. In general, all solutions are k -tuples (x_1, x_2, \dots, x_k) , $1 \leq k \leq n$, and different solutions may have different-sized tuples. The explicit constraints require $x_i \in \{j \mid j \text{ is an integer and } 1 \leq j \leq n\}$. The implicit constraints require that no two be the same and that the sum of the corresponding w_i 's be m . Since we wish to avoid generating multiple instances of the same subset (e.g., $(1, 2, 4)$ and $(1, 4, 2)$ represent the same subset), another implicit constraint that is imposed is that $x_i < x_{i+1}$, $1 \leq i < k$.

In another formulation of the sum of subsets problem, each solution subset is represented by an n -tuple (x_1, x_2, \dots, x_n) such that $x_i \in \{0, 1\}$, $1 \leq i \leq n$. Then $x_i = 0$ if w_i is not chosen and $x_i = 1$ if w_i is chosen. The solutions to the above instance are $(1, 1, 0, 1)$ and $(0, 0, 1, 1)$. This formulation expresses all solutions using a fixed-sized tuple. Thus we conclude that there may be several ways to formulate a problem so that all solutions are tuples that satisfy some constraints. One can verify that for both of the above formulations, the solution space consists of 2^n distinct tuples. \square

Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a *tree organization* for the solution space. For a given solution space many tree organizations may be possible. The next two examples examine some of the ways to organize a solution into a tree.

Example 7.3 [n -queens] The n -queens problem is a generalization of the 8-queens problem of Example 7.1. Now n queens are to be placed on an $n \times n$ chessboard so that no two attack; that is, no two queens are on the same row, column, or diagonal. Generalizing our earlier discussion, the solution space consists of all $n!$ permutations of the n -tuple $(1, 2, \dots, n)$. Figure 7.2 shows a possible tree organization for the case $n = 4$. A tree such as this is called a *permutation tree*. The edges are labeled by possible values of x_i . Edges from level 1 to level 2 nodes specify the values for x_1 . Thus, the leftmost subtree contains all solutions with $x_1 = 1$; its leftmost subtree contains all solutions with $x_1 = 2$, and so on. Edges from level i to level $i+1$ are labeled with the values of x_i . The solution space is defined by all paths from the root node to a leaf node. There are $4! = 24$ leaf nodes in the tree of Figure 7.2. \square

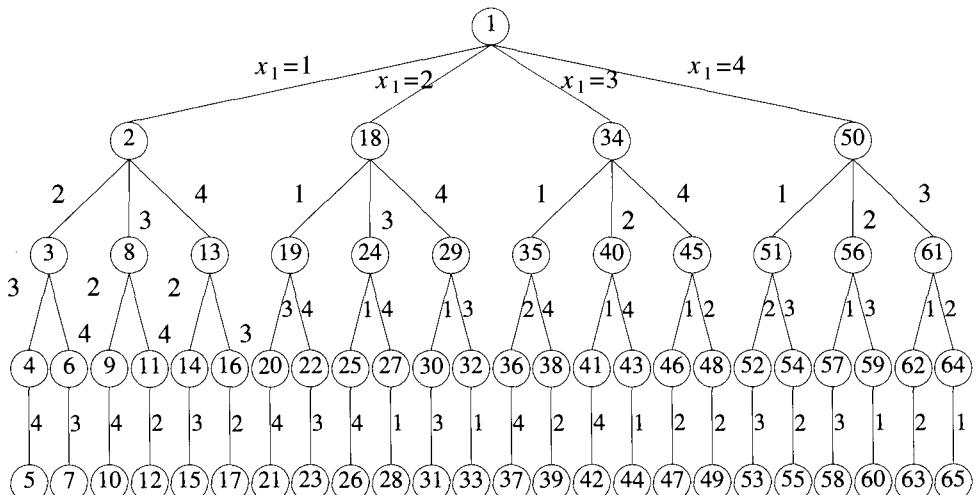


Figure 7.2 Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

Example 7.4 [Sum of subsets] In Example 7.2 we gave two possible formulations of the solution space for the sum of subsets problem. Figures 7.3 and 7.4 show a possible tree organization for each of these formulations for the case $n = 4$. The tree of Figure 7.3 corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level $i + 1$ node represents a value for x_i . At each node, the solution space is partitioned into subsolution spaces. The solution space is defined by all paths from the root node to any node in the tree, since any such path corresponds to a subset satisfying the explicit constraints. The possible paths are $()$ (this corresponds to the empty path from the root to itself), (1) , $(1, 2)$, $(1, 2, 3)$, $(1, 2, 3, 4)$, $(1, 2, 4)$, $(1, 3, 4)$, (2) , $(2, 3)$, and so on. Thus, the left-most subtree defines all subsets containing w_1 , the next subtree defines all subsets containing w_2 but not w_1 , and so on.

The tree of Figure 7.4 corresponds to the fixed tuple size formulation. Edges from level i nodes to level $i + 1$ nodes are labeled with the value of x_i , which is either zero or one. All paths from the root to a leaf node define the solution space. The left subtree of the root defines all subsets containing w_1 , the right subtree defines all subsets not containing w_1 , and so on. Now there are 2^4 leaf nodes which represent 16 possible tuples. \square

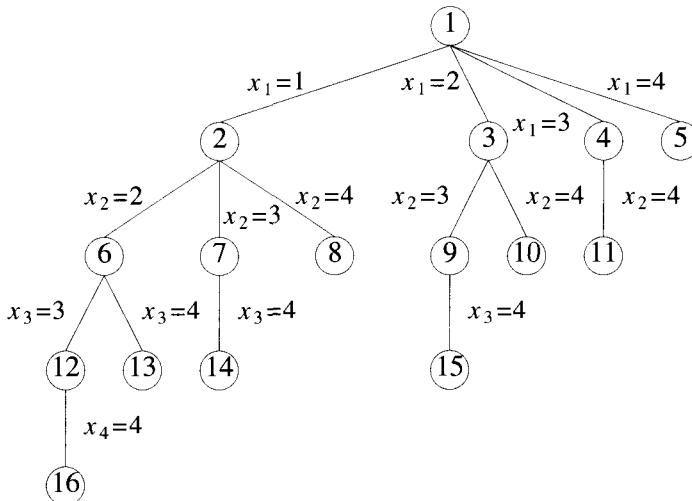


Figure 7.3 A possible solution space organization for the sum of subsets problem. Nodes are numbered as in breadth-first search.

At this point it is useful to develop some terminology regarding tree organizations of solution spaces. Each node in this tree defines a *problem*

state. All paths from the root to other nodes define the *state space* of the problem. *Solution states* are those problem states s for which the path from the root to s defines a tuple in the solution space. In the tree of Figure 7.3 all nodes are solution states whereas in the tree of Figure 7.4 only leaf nodes are solution states. *Answer states* are those solution states s for which the path from the root to s defines a tuple that is a member of the set of solutions (i.e., it satisfies the implicit constraints) of the problem. The tree organization of the solution space is referred to as the *state space tree*.

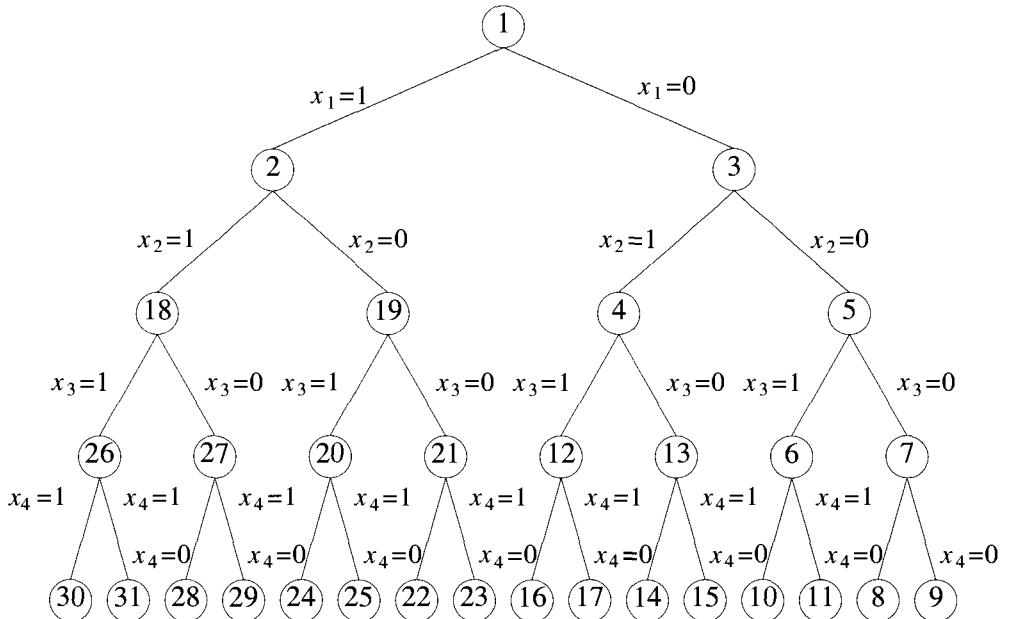


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in D-search.

At each internal node in the space tree of Examples 7.3 and 7.4 the solution space is partitioned into disjoint sub-solution spaces. For example, at node 1 of Figure 7.2 the solution space is partitioned into four disjoint sets. Subtrees 2, 18, 34, and 50 respectively represent all elements of the solution space with $x_1 = 1, 2, 3$, and 4. At node 2 the sub-solution space with $x_1 = 1$ is further partitioned into three disjoint sets. Subtree 3 represents all solution space elements with $x_1 = 1$ and $x_2 = 2$. For all the state space trees we study in this chapter, the solution space is partitioned into disjoint sub-solution spaces at each internal node. It should be noted that this is

not a requirement on a state space tree. The only requirement is that every element of the solution space be represented by at least one node in the state space tree.

The state space tree organizations described in Example 7.4 are called *static trees*. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instances. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called *dynamic trees*. As an example, consider the fixed tuple size formulation for the sum of subsets problem (Example 7.4). Using a dynamic tree organization, one problem instance with $n = 4$ can be solved by means of the organization given in Figure 7.4. Another problem instance with $n = 4$ can be solved by means of a tree in which at level 1 the partitioning corresponds to $x_2 = 1$ and $x_2 = 0$. At level 2 the partitioning could correspond to $x_1 = 1$ and $x_1 = 0$, at level 3 it could correspond to $x_3 = 1$ and $x_3 = 0$, and so on. We see more of dynamic trees in Sections 7.6 and 8.3.

Once a state space tree has been conceived of for any problem, this problem can be solved by systematically generating the problem states, determining which of these are solution states, and finally determining which solution states are answer states. There are two fundamentally different ways to generate the problem states. Both of these begin with the root node and generate other nodes. A node which has been generated and all of whose children have not yet been generated is called a *live node*. The live node whose children are currently being generated is called the *E-node* (node being expanded). A *dead node* is a generated node which is not to be expanded further or all of whose children have been generated. In both methods of generating problem states, we have a list of live nodes. In the first of these two methods as soon as a new child C of the current *E-node* R is generated, this child will become the new *E-node*. Then R will become the *E-node* again when the subtree C has been fully explored. This corresponds to a depth first generation of the problem states. In the second state generation method, the *E-node* remains the *E-node* until it is dead. In both methods, *bounding functions* are used to kill live nodes without generating all their children. This is done carefully enough that at the conclusion of the process at least one answer node is always generated or all answer nodes are generated if the problem requires us to find all solutions. Depth first node generation with bounding functions is called *backtracking*. State generation methods in which the *E-node* remains the *E-node* until it is dead lead to *branch-and-bound* methods. The branch-and-bound technique is discussed in Chapter 8.

The nodes of Figure 7.2 have been numbered in the order they would be generated in a depth first generation process. The nodes in Figures 7.3 and

7.4 have been numbered according to two generation methods in which the E -node remains the E -node until it is dead. In Figure 7.3 each new node is placed into a queue. When all the children of the current E -node have been generated, the next node at the front of the queue becomes the new E -node. In Figure 7.4 new nodes are placed into a stack instead of a queue. Current terminology is not uniform in referring to these two alternatives. Typically the queue method is called breadth first generation and the stack method is called D -search (depth search).

Example 7.5 [4-queens] Let us see how backtracking works on the 4-queens problem of Example 7.3. As a bounding function, we use the obvious criteria that if (x_1, x_2, \dots, x_i) is the path to the current E -node, then all children nodes with parent-child labelings x_{i+1} are such that (x_1, \dots, x_{i+1}) represents a chessboard configuration in which no two queens are attacking. We start with the root node as the only live node. This becomes the E -node and the path is $()$. We generate one child. Let us assume that the children are generated in ascending order. Thus, node number 2 of Figure 7.2 is generated and the path is now (1) . This corresponds to placing queen 1 on column 1. Node 2 becomes the E -node. Node 3 is generated and immediately killed. The next node generated is node 8 and the path becomes $(1, 3)$. Node 8 becomes the E -node. However, it gets killed as all its children represent board configurations that cannot lead to an answer node. We backtrack to node 2 and generate another child, node 13. The path is now $(1, 4)$. Figure 7.5 shows the board configurations as backtracking proceeds. Figure 7.5 shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen which were tried and rejected because another queen was attacking. In Figure 7.5(b) the second queen is placed on columns 1 and 2 and finally settles on column 3. In Figure 7.5(c) the algorithm tries all four columns and is unable to place the next queen on a square. Backtracking now takes place. In Figure 7.5(d) the second queen is moved to the next possible column, column 4 and the third queen is placed on column 2. The boards in Figure 7.5 (e), (f), (g), and (h) show the remaining steps that the algorithm goes through until a solution is found.

Figure 7.6 shows the part of the tree of Figure 7.2 that is generated. Nodes are numbered in the order in which they are generated. A node that gets killed as a result of the bounding function has a B under it. Contrast this tree with Figure 7.2 which contains 31 nodes. \square

With this example completed, we are now ready to present a precise formulation of the backtracking process. We continue to treat backtracking in a general way. We assume that all answer nodes are to be found and not just one. Let (x_1, x_2, \dots, x_i) be a path from the root to a node in a state space tree. Let $T(x_1, x_2, \dots, x_i)$ be the set of all possible values for x_{i+1} such that $(x_1, x_2, \dots, x_{i+1})$ is also a path to a problem state. $T(x_1, x_2, \dots, x_n) = \emptyset$.

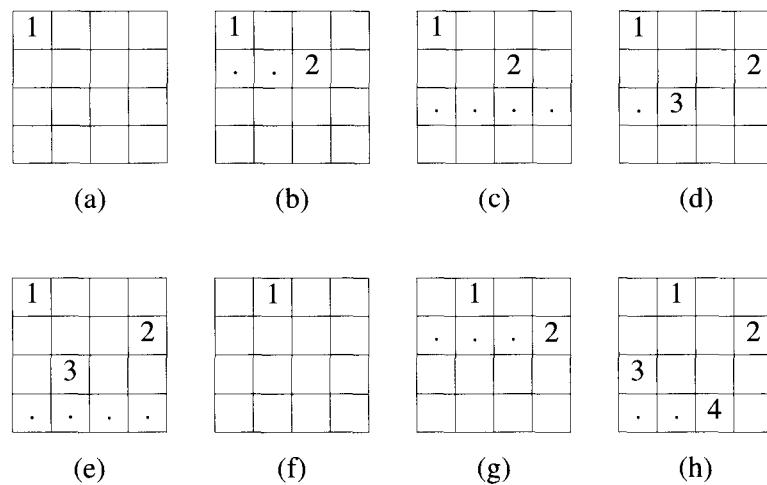


Figure 7.5 Example of a backtrack solution to the 4-queens problem

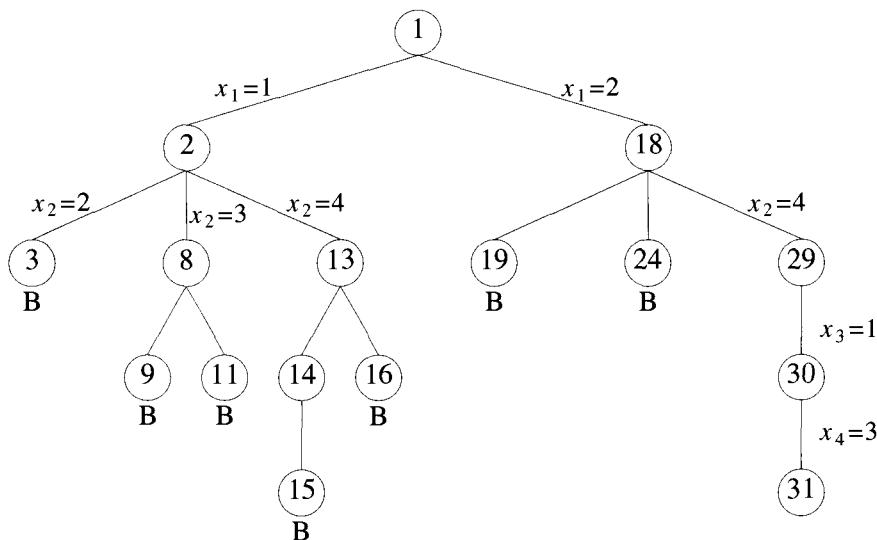


Figure 7.6 Portion of the tree of Figure 7.2 that is generated during backtracking

We assume the existence of bounding function B_{i+1} (expressed as predicates) such that if $B_{i+1}(x_1, x_2, \dots, x_{i+1})$ is false for a path $(x_1, x_2, \dots, x_{i+1})$ from the root node to a problem state, then the path cannot be extended to reach an answer node. Thus the candidates for position $i + 1$ of the solution vector (x_1, \dots, x_n) are those values which are generated by T and satisfy B_{i+1} . Algorithm 7.1 presents a recursive formulation of the backtracking technique. It is natural to describe backtracking in this way since it is essentially a postorder traversal of a tree (see Section 6.1). This recursive version is initially invoked by

Backtrack(1);

```

1  Algorithm Backtrack( $k$ )
2  // This schema describes the backtracking process using
3  // recursion. On entering, the first  $k - 1$  values
4  //  $x[1], x[2], \dots, x[k - 1]$  of the solution vector
5  //  $x[1 : n]$  have been assigned.  $x[ ]$  and  $n$  are global.
6  {
7      for (each  $x[k] \in T(x[1], \dots, x[k - 1])$ ) do
8      {
9          if ( $B_k(x[1], x[2], \dots, x[k]) \neq 0$ ) then
10         {
11             if ( $x[1], x[2], \dots, x[k]$  is a path to an answer node)
12                 then write ( $x[1 : k]$ );
13             if ( $k < n$ ) then Backtrack( $k + 1$ );
14         }
15     }
16 }
```

Algorithm 7.1 Recursive backtracking algorithm

The solution vector (x_1, \dots, x_n) , is treated as a global array $x[1 : n]$. All the possible elements for the k th position of the tuple that satisfy B_k are generated, one by one, and adjoined to the current vector (x_1, \dots, x_{k-1}) . Each time x_k is attached, a check is made to determine whether a solution has been found. Then the algorithm is recursively invoked. When the **for** loop of line 7 is exited, no more values for x_k exist and the current copy of Backtrack ends. The last unresolved call now resumes, namely, the one that continues to examine the remaining elements assuming only $k - 2$ values have been set.

Note that this algorithm causes *all* solutions to be printed and assumes that tuples of various sizes may make up a solution. If only a single solution is desired, then a flag can be added as a parameter to indicate the first occurrence of success.

```

1  Algorithm |Backtrack( $n$ )
2  // This schema describes the backtracking process.
3  // All solutions are generated in  $x[1 : n]$  and printed
4  // as soon as they are determined.
5  {
6       $k := 1$ ;
7      while ( $k \neq 0$ ) do
8      {
9          if (there remains an untried  $x[k] \in T(x[1], x[2], \dots,$ 
10          $x[k - 1])$  and  $B_k(x[1], \dots, x[k])$  is true) then
11          {
12              if ( $x[1], \dots, x[k]$  is a path to an answer node)
13                  then write ( $x[1 : k]$ );
14               $k := k + 1$ ; // Consider the next set.
15          }
16          else  $k := k - 1$ ; // Backtrack to the previous set.
17      }
18  }
```

Algorithm 7.2 General iterative backtracking method

An iterative version of Algorithm 7.1 appears in Algorithm 7.2. Note that $T()$ will yield the set of all possible values that can be placed as the first component x_1 of the solution vector. The component x_1 will take on those values for which the bounding function $B_1(x_1)$ is true. Also note how the elements are generated in a depth first manner. The variable k is continually incremented and a solution vector is grown until either a solution is found or no untried value of x_k remains. When k is decremented, the algorithm must resume the generation of possible elements for the k th position that have not yet been tried. Therefore one must develop a procedure that generates these values in some order. If only one solution is desired, replacing **write** ($x[1 : k]$); with {**write** ($x[1 : k]$); **return**;} suffices.

The efficiency of both the backtracking algorithms we've just seen depends very much on four factors: (1) the time to generate the next x_k , (2) the number of x_k satisfying the explicit constraints, (3) the time for the bounding functions B_k , and (4) the number of x_k satisfying the B_k . Bound-

```

1  Algorithm Place( $k, i$ )
2  // Returns true if a queen can be placed in  $k$ th row and
3  //  $i$ th column. Otherwise it returns false.  $x[ ]$  is a
4  // global array whose first  $(k - 1)$  values have been set.
5  // Abs( $r$ ) returns the absolute value of  $r$ .
6  {
7      for  $j := 1$  to  $k - 1$  do
8          if  $((x[j] = i) \text{ // Two in the same column}$ 
9              or  $(\text{Abs}(x[j] - i) = \text{Abs}(j - k)) \text{ // or in the same diagonal}$ 
10             then return false;
11         return true;
12     }
13 }
```

Algorithm 7.4 Can a new queen be placed?

```

1  Algorithm NQueens( $k, n$ )
2  // Using backtracking, this procedure prints all
3  // possible placements of  $n$  queens on an  $n \times n$ 
4  // chessboard so that they are nonattacking.
5  {
6      for  $i := 1$  to  $n$  do
7          {
8              if Place( $k, i$ ) then
9                  {
10                       $x[k] := i;$ 
11                      if  $(k = n)$  then write ( $x[1 : n]$ );
12                      else NQueens( $k + 1, n$ );
13                  }
14          }
15    }
```

Algorithm 7.5 All solutions to the n -queens problem

At this point we might wonder how effective function `NQueens` is over the brute force approach. For an 8×8 chessboard there are $\binom{64}{8}$ possible ways to place 8 pieces, or approximately 4.4 billion 8-tuples to examine. However, by allowing only placements of queens on distinct rows and columns, we require the examination of at most $8!$, or only 40,320 8-tuples.

We can use `Estimate` to estimate the number of nodes that will be generated by `NQueens`. Note that the assumptions that are needed for `Estimate` do hold for `NQueens`. The bounding function is static. No change is made to the function as the search proceeds. In addition, all nodes on the same level of the state space tree have the same degree. In Figure 7.8 we see five 8×8 chessboards that were created using `Estimate`.

As required, the placement of each queen on the chessboard was chosen randomly. With each choice we kept track of the number of columns a queen could legitimately be placed on. These numbers are listed in the vector beneath each chessboard. The number following the vector represents the value that function `Estimate` would produce from these sizes. The average of these five trials is 1625. The total number of nodes in the 8-queens state space tree is

$$1 + \sum_{j=0}^7 \left[\prod_{i=0}^j (8-i) \right] = 69,281$$

So the estimated number of unbounded nodes is only about 2.34% of the total number of nodes in the 8-queens state space tree. (See the exercises for more ideas about the efficiency of `NQueens`.)

EXERCISES

- Algorithm `NQueens` can be made more efficient by redefining the function `Place(k, i)` so that it either returns the next legitimate column on which to place the k th queen or an illegal value. Rewrite both functions (Algorithms 7.4 and 7.5) so they implement this alternate strategy.
- For the n -queens problem we observe that some solutions are simply reflections or rotations of others. For example, when $n = 4$, the two solutions given in Figure 7.9 are equivalent under reflection.

Observe that for finding inequivalent solutions the algorithm need only set $x[1] = 2, 3, \dots, \lceil n/2 \rceil$.

- Modify `NQueens` so that only inequivalent solutions are computed.
- Run the n -queens program devised above for $n = 8, 9$, and 10. Tabulate the number of solutions your program finds for each value of n .

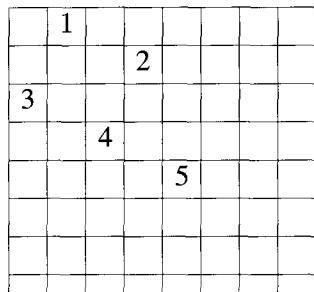
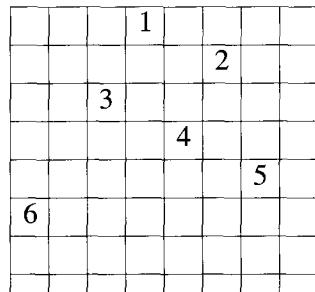
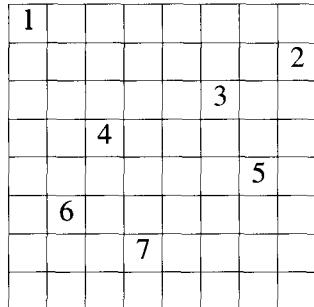
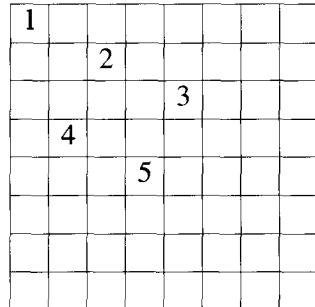
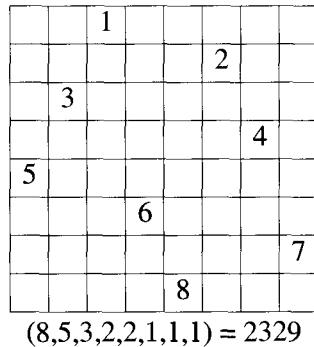

 $(8,5,4,3,2) = 1649$

 $(8,5,3,1,2,1) = 769$

 $(8,6,4,2,1,1,1) = 1401$

 $(8,6,4,3,2) = 1977$

 $(8,5,3,2,2,1,1,1) = 2329$

Figure 7.8 Five walks through the 8-queens problem plus estimates of the tree size

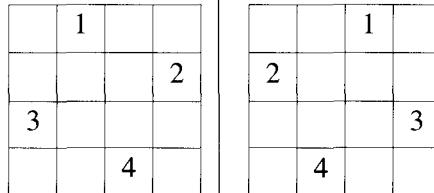


Figure 7.9 Equivalent solutions to the 4-queens problem

- Given an $n \times n$ chessboard, a knight is placed on an arbitrary square with coordinates (x, y) . The problem is to determine $n^2 - 1$ knight moves such that every square of the board is visited once if such a sequence of moves exists. Present an algorithm to solve this problem.

7.3 SUM OF SUBSETS

Suppose we are given n distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sums are m . This is called the *sum of subsets* problem. Examples 7.2 and 7.4 showed how we could formulate this problem using either fixed- or variable-sized tuples. We consider a backtracking solution using the fixed tuple size strategy. In this case the element x_i of the solution vector is either one or zero depending on whether the weight w_i is included or not.

The children of any node in Figure 7.4 are easily generated. For a node at level i the left child corresponds to $x_i = 1$ and the right to $x_i = 0$.

A simple choice for the bounding functions is $B_k(x_1, \dots, x_k) = \text{true}$ iff

$$\sum_{i=1}^k w_i x_i + \sum_{i=k+1}^n w_i \geq m$$

Clearly x_1, \dots, x_k cannot lead to an answer node if this condition is not satisfied. The bounding functions can be strengthened if we assume the w_i 's are initially in nondecreasing order. In this case x_1, \dots, x_k cannot lead to an answer node if

$$\sum_{i=1}^k w_i x_i + w_{k+1} > m$$

The bounding functions we use are therefore

$$\begin{aligned}
 B_k(x_1, \dots, x_k) = \text{true} \text{ iff } & \sum_{i=1}^k w_i x_i + \sum_{i=k+1}^n w_i \geq m \\
 \text{and } & \sum_{i=1}^k w_i x_i + w_{k+1} \leq m
 \end{aligned} \tag{7.1}$$

Since our algorithm will not make use of B_n , we need not be concerned by the appearance of w_{n+1} in this function. Although we have now specified all that is needed to directly use either of the backtracking schemas, a simpler algorithm results if we tailor either of these schemas to the problem at hand. This simplification results from the realization that if $x_k = 1$, then

$$\sum_{i=1}^k w_i x_i + \sum_{i=k+1}^n w_i > m$$

For simplicity we refine the recursive schema. The resulting algorithm is `SumOfSub` (Algorithm 7.6).

Algorithm `SumOfSub` avoids computing $\sum_{i=1}^k w_i x_i$ and $\sum_{i=k+1}^n w_i$ each time by keeping these values in variables s and r respectively. *The algorithm assumes $w_1 \leq m$ and $\sum_{i=1}^n w_i \geq m$.* The initial call is `SumOfSub(0, 1, $\sum_{i=1}^n w_i$)`. It is interesting to note that the algorithm does not explicitly use the test $k > n$ to terminate the recursion. This test is not needed as on entry to the algorithm, $s \neq m$ and $s + r \geq m$. Hence, $r \neq 0$ and so k can be no greater than n . Also note that in the `else if` statement (line 11), since $s + w_k < m$ and $s + r \geq m$, it follows that $r \neq w_k$ and hence $k + 1 \leq n$. Observe also that if $s + w_k = m$ (line 9), then x_{k+1}, \dots, x_n must be zero. These zeros are omitted from the output of line 9. In line 11 we do not test for $\sum_{i=1}^k w_i x_i + \sum_{i=k+1}^n w_i \geq m$, as we already know $s + r \geq m$ and $x_k = 1$.

Example 7.6 Figure 7.10 shows the portion of the state space tree generated by function `SumOfSub` while working on the instance $n = 6$, $m = 30$, and $w[1 : 6] = \{5, 10, 12, 13, 15, 18\}$. The rectangular nodes list the values of s , k , and r on each of the calls to `SumOfSub`. Circular nodes represent points at which subsets with sums m are printed out. At nodes A , B , and C the output is respectively $(1, 1, 0, 0, 1)$, $(1, 0, 1, 1)$, and $(0, 0, 1, 0, 0, 1)$. Note that the tree of Figure 7.10 contains only 23 rectangular nodes. The full state space tree for $n = 6$ contains $2^6 - 1 = 63$ nodes from which calls could be made (this count excludes the 64 leaf nodes as no call need be made from a leaf). \square

```

1  Algorithm SumOfSub( $s, k, r$ )
2  // Find all subsets of  $w[1 : n]$  that sum to  $m$ . The values of  $x[j]$ ,
3  //  $1 \leq j < k$ , have already been determined.  $s = \sum_{j=1}^{k-1} w[j] * x[j]$ 
4  // and  $r = \sum_{j=k}^n w[j]$ . The  $w[j]$ 's are in nondecreasing order.
5  // It is assumed that  $w[1] \leq m$  and  $\sum_{i=1}^n w[i] \geq m$ .
6  {
7      // Generate left child. Note:  $s + w[k] \leq m$  since  $B_{k-1}$  is true.
8       $x[k] := 1$ ;
9      if ( $s + w[k] = m$ ) then write ( $x[1 : k]$ ); // Subset found
10     // There is no recursive call here as  $w[j] > 0$ ,  $1 \leq j \leq n$ .
11     else if ( $s + w[k] + w[k+1] \leq m$ )
12         then SumOfSub( $s + w[k], k + 1, r - w[k]$ );
13     // Generate right child and evaluate  $B_k$ .
14     if (( $s + r - w[k] \geq m$ ) and ( $s + w[k+1] \leq m$ )) then
15     {
16          $x[k] := 0$ ;
17         SumOfSub( $s, k + 1, r - w[k]$ );
18     }
19 }
```

Algorithm 7.6 Recursive backtracking algorithm for sum of subsets problem

EXERCISES

1. Prove that the size of the set of all subsets of n elements is 2^n .
2. Let $w = \{5, 7, 10, 12, 15, 18, 20\}$ and $m = 35$. Find all possible subsets of w that sum to m . Do this using SumOfSub. Draw the portion of the state space tree that is generated.
3. With $m = 35$, run SumOfSub on the data (a) $w = \{5, 7, 10, 12, 15, 18, 20\}$, (b) $w = \{20, 18, 15, 12, 10, 7, 5\}$, and (c) $w = \{15, 7, 20, 5, 18, 10, 12\}$. Are there any discernible differences in the computing times?
4. Write a backtracking algorithm for the sum of subsets problem using the state space tree corresponding to the variable tuple size formulation.

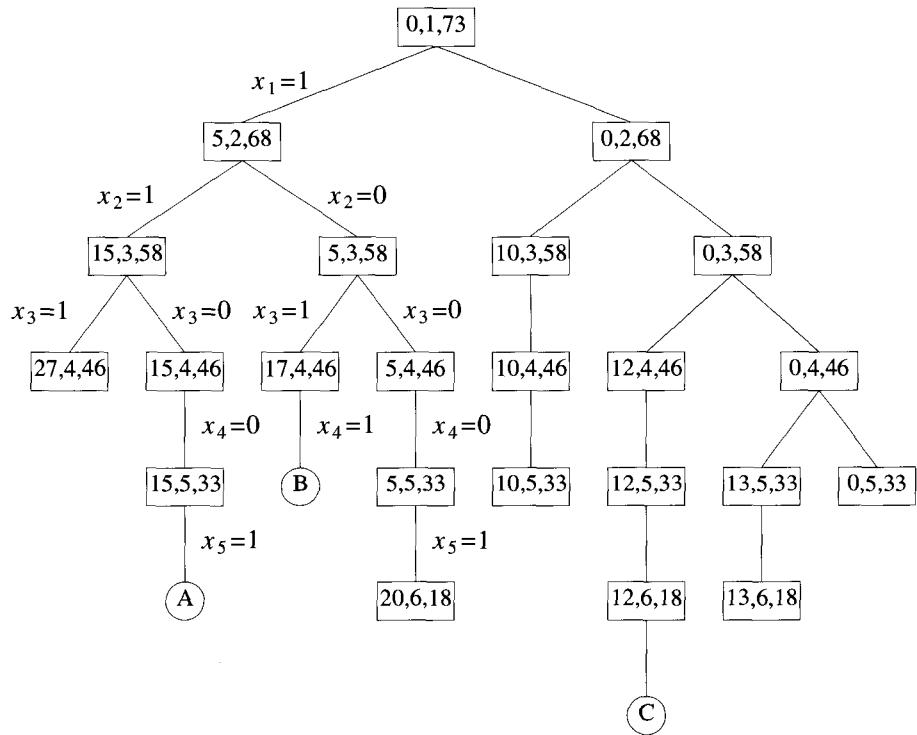


Figure 7.10 Portion of state space tree generated by SumOfSub

7.4 GRAPH COLORING

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used. This is termed the *m -colorability decision* problem and it is discussed again in Chapter 11. Note that if d is the degree of the given graph, then it can be colored with $d + 1$ colors. The *m -colorability optimization* problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the *chromatic number* of the graph. For example, the graph of Figure 7.11 can be colored with three colors 1, 2, and 3. The color of each node is indicated next to it. It can also be seen that three colors are needed to color this graph and hence this graph's chromatic number is 3.

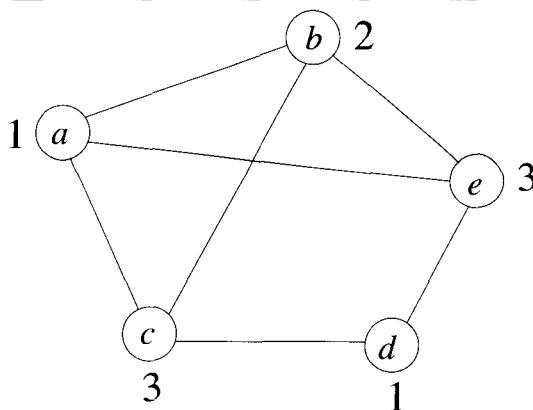


Figure 7.11 An example graph and its coloring

A graph is said to be *planar* iff it can be drawn in a plane in such a way that no two edges cross each other. A famous special case of the m -colorability decision problem is the 4-color problem for planar graphs. This problem asks the following question: given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only four colors are needed? This turns out to be a problem for which graphs are very useful, because a map can easily be transformed into a graph. Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge. Figure 7.12 shows a map with five regions and its corresponding graph. This map requires four colors. For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient. In this section we consider not only graphs that are produced from maps but all graphs. We are interested in determining all the different ways in which a given graph can be colored using at most m colors.

Suppose we represent a graph by its adjacency matrix $G[1 : n, 1 : n]$, where $G[i, j] = 1$ if (i, j) is an edge of G , and $G[i, j] = 0$ otherwise. The colors are represented by the integers $1, 2, \dots, m$ and the solutions are given by the n -tuple (x_1, \dots, x_n) , where x_i is the color of node i . Using the recursive backtracking formulation as given in Algorithm 7.1, the resulting algorithm is **mColoring** (Algorithm 7.7). The underlying state space tree used is a tree of degree m and height $n + 1$. Each node at level i has m children corresponding to the m possible assignments to x_i , $1 \leq i \leq n$. Nodes at

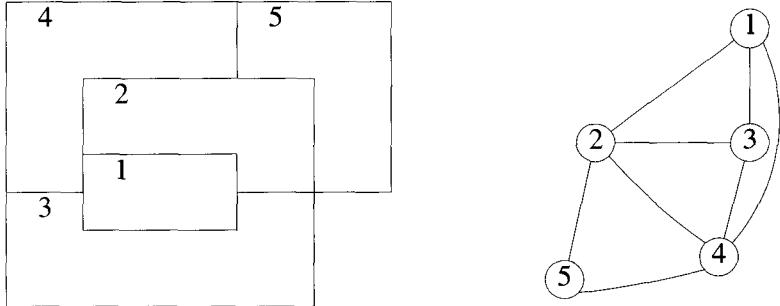


Figure 7.12 A map and its planar graph representation

level $n + 1$ are leaf nodes. Figure 7.13 shows the state space tree when $n = 3$ and $m = 3$.

Function `mColoring` is begun by first assigning the graph to its adjacency matrix, *setting the array $x[]$ to zero*, and then invoking the statement `mColoring(1);`

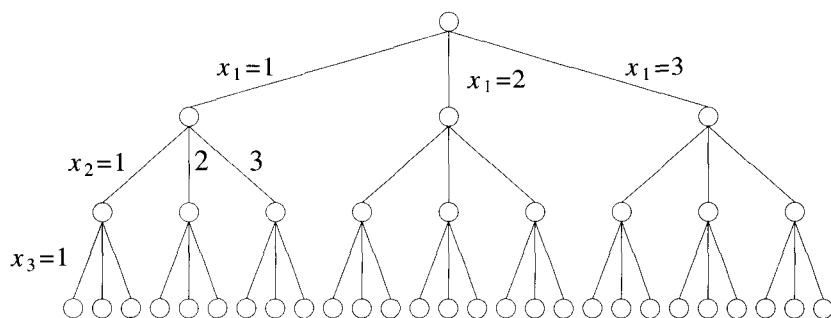
Notice the similarity between this algorithm and the general form of the recursive backtracking schema of Algorithm 7.1. Function `NextValue` (Algorithm 7.8) produces the possible colors for x_k after x_1 through x_{k-1} have been defined. The main loop of `mColoring` repeatedly picks an element from the set of possibilities, assigns it to x_k , and then calls `mColoring` recursively. For instance, Figure 7.14 shows a simple graph containing four nodes. Below that is the tree that is generated by `mColoring`. Each path to a leaf represents a coloring using at most three colors. Note that only 12 solutions exist with *exactly* three colors. In this tree, after choosing $x_1 = 2$ and $x_2 = 1$, the possible choices for x_3 are 2 and 3. After choosing $x_1 = 2$, $x_2 = 1$, and $x_3 = 2$, possible values for x_4 are 1 and 3. And so on.

An upper bound on the computing time of `mColoring` can be arrived at by noticing that the number of internal nodes in the state space tree is $\sum_{i=0}^{n-1} m^i$. At each internal node, $O(mn)$ time is spent by `NextValue` to determine the children corresponding to legal colorings. Hence the total time is bounded by $\sum_{i=0}^{n-1} m^{i+1} n = \sum_{i=1}^n m^i n = n(m^{n+1} - 2)/(m - 1) = O(nm^n)$.

```

1  Algorithm mColoring( $k$ )
2  // This algorithm was formed using the recursive backtracking
3  // schema. The graph is represented by its boolean adjacency
4  // matrix  $G[1 : n, 1 : n]$ . All assignments of  $1, 2, \dots, m$  to the
5  // vertices of the graph such that adjacent vertices are
6  // assigned distinct integers are printed.  $k$  is the index
7  // of the next vertex to color.
8  {
9    repeat
10   { // Generate all legal assignments for  $x[k]$ .
11     NextValue( $k$ ); // Assign to  $x[k]$  a legal color.
12     if ( $x[k] = 0$ ) then return; // No new color possible
13     if ( $k = n$ ) then // At most  $m$  colors have been
14       // used to color the  $n$  vertices.
15     write ( $x[1 : n]$ );
16     else mColoring( $k + 1$ );
17   } until (false);
18 }

```

Algorithm 7.7 Finding all m -colorings of a graph**Figure 7.13** State space tree for mColoring when $n = 3$ and $m = 3$

```

1  Algorithm NextValue( $k$ )
2  //  $x[1], \dots, x[k - 1]$  have been assigned integer values in
3  // the range  $[1, m]$  such that adjacent vertices have distinct
4  // integers. A value for  $x[k]$  is determined in the range
5  //  $[0, m]$ .  $x[k]$  is assigned the next highest numbered color
6  // while maintaining distinctness from the adjacent vertices
7  // of vertex  $k$ . If no such color exists, then  $x[k]$  is 0.
8  {
9    repeat
10   {
11      $x[k] := (x[k] + 1) \bmod (m + 1)$ ; // Next highest color.
12     if ( $x[k] = 0$ ) then return; // All colors have been used.
13     for  $j := 1$  to  $n$  do
14       {
15         // Check if this color is
16         // distinct from adjacent colors.
17         if (( $G[k, j] \neq 0$ ) and ( $x[k] = x[j]$ ))
18         // If  $(k, j)$  is and edge and if adj.
19         // vertices have the same color.
20         then break;
21       }
22       if ( $j = n + 1$ ) then return; // New color found
23   } until (false); // Otherwise try to find another color.

```

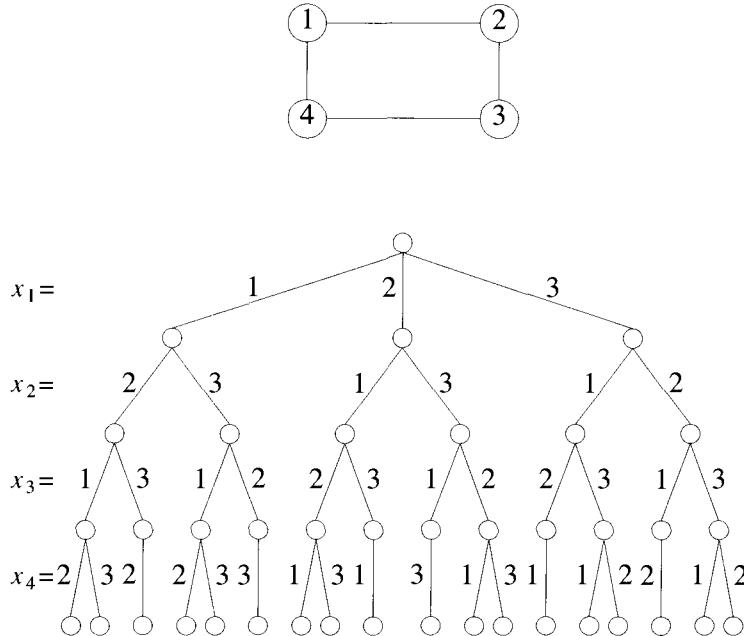
Algorithm 7.8 Generating a next color

EXERCISE

1. Program and run `mColoring` (Algorithm 7.7) using as data the complete graphs of size $n = 2, 3, 4, 5, 6$, and 7 . Let the desired number of colors be $k = n$ and $k = n/2$. Tabulate the computing times for each value of n and k .

7.5 HAMILTONIAN CYCLES

Let $G = (V, E)$ be a connected graph with n vertices. A Hamiltonian cycle (suggested by Sir William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other words if a Hamiltonian cycle begins at some vertex $v_1 \in G$ and the vertices

**Figure 7.14** A 4-node graph and all possible 3-colorings

of G are visited in the order v_1, v_2, \dots, v_{n+1} , then the edges (v_i, v_{i+1}) are in E , $1 \leq i \leq n$, and the v_i are distinct except for v_1 and v_{n+1} , which are equal.

The graph $G1$ of Figure 7.15 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph $G2$ of Figure 7.15 contains no Hamiltonian cycle. There is no known easy way to determine whether a given graph contains a Hamiltonian cycle. We now look at a backtracking algorithm that finds all the Hamiltonian cycles in a graph. The graph may be directed or undirected. Only distinct cycles are output.

The backtracking solution vector (x_1, \dots, x_n) is defined so that x_i represents the i th visited vertex of the proposed cycle. Now all we need do is determine how to compute the set of possible vertices for x_k if x_1, \dots, x_{k-1} have already been chosen. If $k = 1$, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from x_1, x_2, \dots, x_{k-1} and v is connected by an edge to x_{k-1} . The vertex x_n can only be the one remaining vertex and it must be connected to both x_{n-1} and x_1 . We begin by presenting function `NextValue(k)` (Algorithm 7.9), which determines a possible next

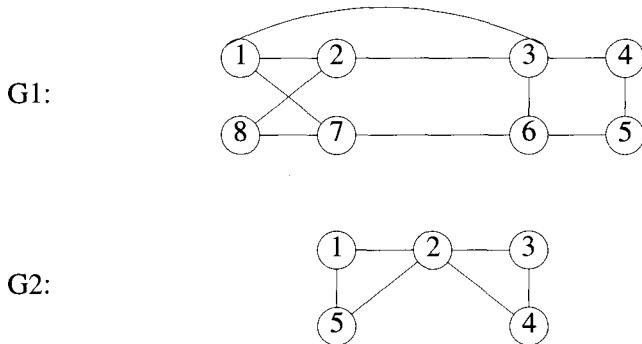


Figure 7.15 Two graphs, one containing a Hamiltonian cycle

vertex for the proposed cycle.

Using `NextValue` we can particularize the recursive backtracking schema to find all Hamiltonian cycles (Algorithm 7.10). This algorithm is started by first initializing the adjacency matrix $G[1 : n, 1 : n]$, then setting $x[2 : n]$ to zero and $x[1]$ to 1, and then executing `Hamiltonian(2)`.

Recall from Section 5.9 the traveling salesperson problem which asked for a tour that has minimum cost. This tour is a Hamiltonian cycle. For the simple case of a graph all of whose edge costs are identical, `Hamiltonian` will find a minimum-cost tour if a tour exists. If the common edge cost is c , the cost of a tour is cn since there are n edges in a Hamiltonian cycle.

EXERCISES

1. Determine the order of magnitude of the worst-case computing time for the backtracking procedure that finds all Hamiltonian cycles.
2. Draw the portion of the state space tree generated by Algorithm 7.10 for the graph $G1$ of Figure 7.15.
3. Generalize `Hamiltonian` so that it processes a graph whose edges have costs associated with them and finds a Hamiltonian cycle with minimum cost. You can assume that all edge costs are positive.

```

1  Algorithm NextValue( $k$ )
2  //  $x[1 : k - 1]$  is a path of  $k - 1$  distinct vertices. If  $x[k] = 0$ , then
3  // no vertex has as yet been assigned to  $x[k]$ . After execution,
4  //  $x[k]$  is assigned to the next highest numbered vertex which
5  // does not already appear in  $x[1 : k - 1]$  and is connected by
6  // an edge to  $x[k - 1]$ . Otherwise  $x[k] = 0$ . If  $k = n$ , then
7  // in addition  $x[k]$  is connected to  $x[1]$ .
8  {
9    repeat
10   {
11      $x[k] := (x[k] + 1) \bmod (n + 1)$ ; // Next vertex.
12     if ( $x[k] = 0$ ) then return;
13     if ( $G[x[k - 1], x[k]] \neq 0$ ) then
14       { // Is there an edge?
15         for  $j := 1$  to  $k - 1$  do if ( $x[j] = x[k]$ ) then break;
16           // Check for distinctness.
17         if ( $j = k$ ) then // If true, then the vertex is distinct.
18           if (( $k < n$ ) or (( $k = n$ ) and  $G[x[n], x[1]] \neq 0$ ))
19             then return;
20       }
21     } until (false);
22   }

```

Algorithm 7.9 Generating a next vertex

```

1  Algorithm Hamiltonian( $k$ )
2  // This algorithm uses the recursive formulation of
3  // backtracking to find all the Hamiltonian cycles
4  // of a graph. The graph is stored as an adjacency
5  // matrix  $G[1 : n, 1 : n]$ . All cycles begin at node 1.
6  {
7      repeat
8          { // Generate values for  $x[k]$ .
9              NextValue( $k$ ); // Assign a legal next value to  $x[k]$ .
10             if ( $x[k] = 0$ ) then return;
11             if ( $k = n$ ) then write ( $x[1 : n]$ );
12             else Hamiltonian( $k + 1$ );
13         } until ( $\text{false}$ );
14     }

```

Algorithm 7.10 Finding all Hamiltonian cycles

7.6 KNAPSACK PROBLEM

In this section we reconsider a problem that was defined and solved by a dynamic programming algorithm in Chapter 5, the 0/1 knapsack optimization problem. Given n positive weights w_i , n positive profits p_i , and a positive number m that is the knapsack capacity, this problem calls for choosing a subset of the weights such that

$$\sum_{1 \leq i \leq n} w_i x_i \leq m \quad \text{and} \quad \sum_{1 \leq i \leq n} p_i x_i \text{ is maximized} \quad (7.2)$$

The x_i 's constitute a zero-one-valued vector.

The solution space for this problem consists of the 2^n distinct ways to assign zero or one values to the x_i 's. Thus the solution space is the same as that for the sum of subsets problem. Two possible tree organizations are possible. One corresponds to the fixed tuple size formulation (Figure 7.4) and the other to the variable tuple size formulation (Figure 7.3). Backtracking algorithms for the knapsack problem can be arrived at using either of these two state space trees. Regardless of which is used, bounding functions are needed to help kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upper bound is not higher than

the value of the best solution determined so far, then that live node can be killed.

We continue the discussion using the fixed tuple size formulation. If at node Z the values of x_i , $1 \leq i \leq k$, have already been determined, then an upper bound for Z can be obtained by relaxing the requirement $x_i = 0$ or 1 to $0 \leq x_i \leq 1$ for $k+1 \leq i \leq n$ and using the greedy algorithm of Section 4.2 to solve the relaxed problem. Function $\text{Bound}(cp, cw, k)$ (Algorithm 7.11) determines an upper bound on the best solution obtainable by expanding any node Z at level $k+1$ of the state space tree. The object weights and profits are $w[i]$ and $p[i]$. It is assumed that $p[i]/w[i] \geq p[i+1]/w[i+1]$, $1 \leq i < n$.

```

1  Algorithm Bound( $cp, cw, k$ )
2  //  $cp$  is the current profit total,  $cw$  is the current
3  // weight total;  $k$  is the index of the last removed
4  // item; and  $m$  is the knapsack size.
5  {
6       $b := cp$ ;  $c := cw$ ;
7      for  $i := k + 1$  to  $n$  do
8      {
9           $c := c + w[i]$ ;
10         if ( $c < m$ ) then  $b := b + p[i]$ ;
11         else return  $b + (1 - (c - m)/w[i]) * p[i]$ ;
12     }
13 } return  $b$ ;

```

Algorithm 7.11 A bounding function

From Bound it follows that the bound for a feasible left child of a node Z is the same as that for Z . Hence, the bounding function need not be used whenever the backtracking algorithm makes a move to the left child of a node. The resulting algorithm is BKnap (Algorithm 7.12). It was obtained from the recursive backtracking schema. Initially set $fp := -1$. This algorithm is invoked as

$\text{BKnap}(1, 0, 0);$

When $fp \neq -1$, $x[i]$, $1 \leq i \leq n$, is such that $\sum_{i=1}^n p[i]x[i] = fp$. In lines 8 to 18 left children are generated. In line 20, Bound is used to test whether a

```

1  Algorithm BKnap( $k, cp, cw$ )
2  //  $m$  is the size of the knapsack;  $n$  is the number of weights
3  // and profits.  $w[ ]$  and  $p[ ]$  are the weights and profits.
4  //  $p[i]/w[i] \geq p[i+1]/w[i+1]$ .  $fw$  is the final weight of
5  // knapsack;  $fp$  is the final maximum profit.  $x[k] = 0$  if  $w[k]$ 
6  // is not in the knapsack; else  $x[k] = 1$ .
7  {
8      // Generate left child.
9      if ( $cw + w[k] \leq m$ ) then
10     {
11          $y[k] := 1$ ;
12         if ( $k < n$ ) then BKnap( $k + 1, cp + p[k], cw + w[k]$ );
13         if (( $cp + p[k] > fp$ ) and ( $k = n$ )) then
14             {
15                  $fp := cp + p[k]; fw := cw + w[k];$ 
16                 for  $j := 1$  to  $k$  do  $x[j] := y[j];$ 
17             }
18         }
19         // Generate right child.
20         if (Bound( $cp, cw, k$ )  $\geq fp$ ) then
21         {
22              $y[k] := 0$ ; if ( $k < n$ ) then BKnap( $k + 1, cp, cw$ );
23             if (( $cp > fp$ ) and ( $k = n$ )) then
24                 {
25                      $fp := cp; fw := cw;$ 
26                     for  $j := 1$  to  $k$  do  $x[j] := y[j];$ 
27                 }
28             }
29     }

```

Algorithm 7.12 Backtracking solution to the 0/1 knapsack problem

right child should be generated. The path $y[i]$, $1 \leq i \leq k$, is the path to the current node. The current weight $cw = \sum_{i=1}^{k-1} w[i]y[i]$ and $cp = \sum_{i=1}^{k-1} p[i]y[i]$. In lines 13 to 17 and 23 to 27 the solution vector is updated if need be.

So far, all our backtracking algorithms have worked on a static state space tree. We now see how a dynamic state space tree can be used for the knapsack problem. One method for dynamically partitioning the solution space is based on trying to obtain an optimal solution using the greedy algorithm of Section 4.2. We first replace the integer constraint $x_i = 0$ or 1 by the constraint $0 \leq x_i \leq 1$. This yields the relaxed problem

$$\max \sum_{1 \leq i \leq n} p_i x_i \text{ subject to } \sum_{1 \leq i \leq n} w_i x_i \leq m \quad (7.3)$$

$$0 \leq x_i \leq 1, \quad 1 \leq i \leq n$$

If the solution generated by the greedy method has all x_i 's equal to zero or one, then it is also an optimal solution to the original 0/1 knapsack problem. If this is not the case, then exactly one x_i will be such that $0 < x_i < 1$. We partition the solution space of (7.2) into two subspaces. In one $x_i = 0$ and in the other $x_i = 1$. Thus the left subtree of the state space tree will correspond to $x_i = 0$ and the right to $x_i = 1$. In general, at each node Z of the state space tree the greedy algorithm is used to solve (7.3) under the added restrictions corresponding to the assignments already made along the path from the root to this node. In case the solution is all integer, then an optimal solution for this node has been found. If not, then there is exactly one x_i such that $0 < x_i < 1$. The left child of Z corresponds to $x_i = 0$, and the right to $x_i = 1$.

The justification for this partitioning scheme is that the noninteger x_i is what prevents the greedy solution from being a feasible solution to the 0/1 knapsack problem. So, we would expect to reach a feasible greedy solution quickly by forcing this x_i to be integer. Choosing left branches to correspond to $x_i = 0$ rather than $x_i = 1$ is also justifiable. Since the greedy algorithm requires $p_j/w_j \geq p_{j+1}/w_{j+1}$, we would expect most objects with low index (i.e., small j and hence high density) to be in an optimal filling of the knapsack. When x_i is set to zero, we are not preventing the greedy algorithm from using any of the objects with $j < i$ (unless x_j has already been set to zero). On the other hand, when x_i is set to one, some of the x_j 's with $j < i$ will not be able to get into the knapsack. Therefore we expect to arrive at an optimal solution with $x_i = 0$. So we wish the backtracking algorithm to try this alternative first. Hence the left subtree corresponds to $x_i = 0$.

Example 7.7 Let us try out a backtracking algorithm and the above dynamic partitioning scheme on the following data: $p = \{11, 21, 31, 33, 43, 53, 55, 65\}$, $w = \{1, 11, 21, 23, 33, 43, 45, 55\}$, $m = 110$, and $n = 8$. The greedy

solution corresponding to the root node (i.e., Equation (7.3)) is $x = \{1, 1, 1, 1, 21/45, 0, 0\}$. Its value is 164.88. The two subtrees of the root correspond to $x_6 = 0$ and $x_6 = 1$, respectively (Figure 7.16). The greedy solution at node 2 is $x = \{1, 1, 1, 1, 1, 0, 21/45, 0\}$. Its value is 164.66. The solution space at node 2 is partitioned using $x_7 = 0$ and $x_7 = 1$. The next E -node is node 3. The solution here has $x_8 = 21/55$. The partitioning now is with $x_8 = 0$ and $x_8 = 1$. The solution at node 4 is all integer so there is no need to expand this node further. The best solution found so far has value 139 and $x = \{1, 1, 1, 1, 1, 0, 0, 0\}$. Node 5 is the next E -node. The greedy solution for this node is $x = \{1, 1, 1, 22/23, 0, 0, 0, 1\}$. Its value is 159.56. The partitioning is now with $x_4 = 0$ and $x_4 = 1$. The greedy solution at node 6 has value 156.66 and $x_5 = 2/3$. Next, node 7 becomes the E -node. The solution here is $\{1, 1, 1, 0, 0, 0, 0, 1\}$. Its value is 128. Node 7 is not expanded as the greedy solution here is all integer. At node 8 the greedy solution has value 157.71 and $x_3 = 4/7$. The solution at node 9 is all integer and has value 140. The greedy solution at node 10 is $\{1, 0, 1, 0, 1, 0, 0, 1\}$. Its value is 150. The next E -node is 11. Its value is 159.52 and $x_3 = 20/21$. The partitioning is now on $x_3 = 0$ and $x_3 = 1$. The remainder of the backtracking process on this knapsack instance is left as an exercise. \square

Experimental work due to E. Horowitz and S. Sahni, cited in the references, indicates that backtracking algorithms for the knapsack problem generally work in less time when using a static tree than when using a dynamic tree. The dynamic partitioning scheme is, however, useful in the solution of integer linear programs. The general integer linear program is mathematically stated in (7.4).

$$\begin{aligned} & \text{minimize} && \sum_{1 \leq j \leq n} c_j x_j \\ & \text{subject to} && \sum_{1 \leq j \leq n} a_{ij} x_j \leq b_i, \quad 1 \leq i \leq m \\ & && x_j \text{'s are nonnegative integers} \end{aligned} \tag{7.4}$$

If the integer constraints on the x_i 's in (7.4) are replaced by the constraint $x_i \geq 0$, then we obtain a linear program whose optimal solution has a value at least as large as the value of an optimal solution to (7.4). Linear programs can be solved using the simplex methods (see the references). If the solution is not all integer, then a noninteger x_i is chosen to partition the solution space. Let us assume that the value of x_i in the optimal solution to the linear program corresponding to any node Z in the state space is v and v is not an integer. The left child of Z corresponds to $x_i \leq \lfloor v \rfloor$ whereas the right child of Z corresponds to $x_i \geq \lceil v \rceil$. Since the resulting state space tree has a potentially infinite depth (note that on the path from the root to a node Z

the solution space can be partitioned on one x_i many times as each x_i can have as value any nonnegative integer), it is almost always searched using a branch-and-bound method (see Chapter 8).

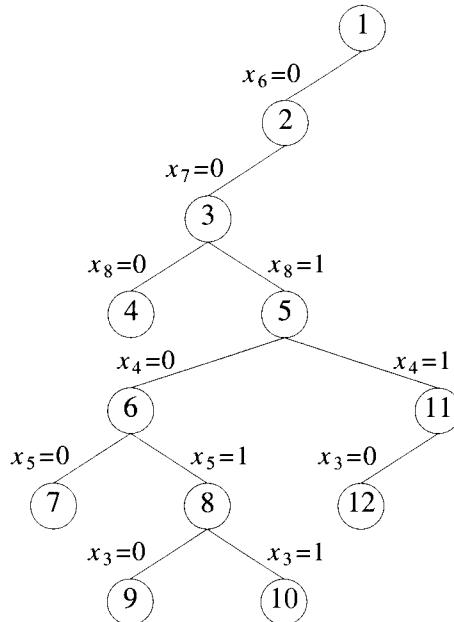


Figure 7.16 Part of the dynamic state space tree generated in Example 7.7

EXERCISES

1. (a) Present a backtracking algorithm for solving the knapsack optimization problem using the variable tuple size formulation.
(b) Draw the portion of the state space tree your algorithm will generate when solving the knapsack instance of Example 7.7.
2. Complete the state space tree of Figure 7.16.
3. Give a backtracking algorithm for the knapsack problem using the dynamic state space tree discussed in this section.
4. [Programming project] (a) Program the algorithms of Exercises 1 and 3. Run these two programs and **BKnap** using the following data: $p =$

$\{11, 21, 31, 33, 43, 53, 55, 65\}$, $w = \{1, 11, 21, 23, 33, 43, 45, 55\}$, $m = 110$, and $n = 8$. Which algorithm do you expect to perform best?

- (b) Now program the dynamic programming algorithm of Section 5.7 for the knapsack problem. Use the heuristics suggested at the end of Section 5.7. Obtain computing times and compare this program with the backtracking programs.
- 5. (a) Obtain a knapsack instance for which more nodes are generated by the backtracking algorithm using a dynamic tree than using a static tree.
- (b) Obtain a knapsack instance for which more nodes are generated by the backtracking algorithm using a static tree than using a dynamic tree.
- (c) Strengthen the backtracking algorithms with the following heuristic: Build an array $minw[\cdot]$ with the property that $minw[i]$ is the index of the object that has least weight among objects $i, i+1, \dots, n$. Now any E -node at which decisions for x_1, \dots, x_{i-1} have been made and at which the unutilized knapsack capacity is less than $w[minw[i]]$ can be terminated provided the profit earned up to this node is no more than the maximum determined so far. Incorporate this into your programs of Exercise 4(a). Rerun the new programs on the same data sets and see what (if any) improvements result.

7.7 REFERENCES AND READINGS

An early modern account of backtracking was given by R. J. Walker. The technique for estimating the efficiency of a backtrack program was first proposed by M. Hall and D. E. Knuth and the dynamic partitioning scheme for the 0/1 knapsack problem was proposed by H. Greenberg and R. Hegerich. Experimental results showing static trees to be superior for this problem can be found in “Computing partitions with applications to the knapsack problem,” by E. Horowitz and S. Sahni, *Journal of the ACM* 21, no. 2 (1974): 277–292.

Data presented in the above paper shows that the divide-and-conquer dynamic programming algorithm for the knapsack problem is superior to BKnap.

For a proof of the four-color theorem see *Every Planar Map is Four Colorable*, by K. I. Appel, American Mathematical Society, Providence, RI, 1989.