CHAPTER 1

BASIC CONCEPTS

1.1 OVERVIEW: SYSTEM LIFE CYCLE

We assume that our readers have a strong background in structured programming, typically attained through the completion of an elementary programming course. Such an initial course usually emphasizes mastering a programming language's syntax (its grammar rules) and applying this language to the solution of several relatively small problems. These problems are frequently chosen so that they use a particular language construct. For example, the programming problem might require the use of arrays or **while** loops.

In this text we want to move you beyond these rudiments by providing you with the tools and techniques necessary to design and implement large-scale computer systems. We believe that a solid foundation in data abstraction, algorithm specification, and performance analysis and measurement provides the necessary methodology. In this chapter, we will discuss each of these areas in detail. We also will briefly discuss recursive programming because many of you probably have only a fleeting acquaintance with this important technique. However, before we begin we want to place these tools in a context that views programming as more than writing code. Good programmers regard large-scale computer programs as systems that contain many complex interacting parts. As systems, these programs undergo a development process called the system life cycle. We consider this cycle as consisting of requirements, analysis, design, coding, and verification phases. Although we will consider them separately, these phases are highly interrelated and follow only a very crude sequential time frame. The Selected Readings and References section lists several sources on the system life cycle and its various phases that will provide you with additional information.

(1) **Requirements.** All large programming projects begin with a set of specifications that define the purpose of the project. These requirements describe the information that we, the programmers, are given (input) and the results that we must produce (output). Frequently the initial specifications are defined vaguely, and we must develop rigorous input and output descriptions that include all cases.

(2) **Analysis.** After we have delineated carefully the system's requirements, the analysis phase begins in earnest. In this phase, we begin to break the problem down into manageable pieces. There are two approaches to analysis: bottom-up and top-down. The bottom-up approach is an older, unstructured strategy that places an early emphasis on the coding fine points. Since the programmer does not have a master plan for the project, the resulting program frequently has many loosely connected, error-ridden segments. Bottom-up analysis is akin to constructing a building from a generic blueprint. That is, we view all buildings identically; they must have walls, a roof, plumbing, and heating. The specific purpose to which the building will be put is irrelevant from this perspective. Although few of us would want to live in a home constructed using this technique, many programmers, particularly beginning ones, believe that they can create good, error-free programs without prior planning.

In contrast, the top-down approach begins with the purpose that the program will serve and uses this end product to divide the program into manageable segments. This technique generates diagrams that are used to design the system. Frequently, several alternate solutions to the programming problem are developed and compared during this phase.

(3) **Design**. This phase continues the work done in the analysis phase. The designer approaches the system from the perspectives of both the data objects that the program needs and the operations performed on them. The first perspective leads to the creation of abstract data types, while the second requires the specification of algorithms and a consideration of algorithm design strategies. For example, suppose that we are designing a scheduling system for a university. Typical data objects might include students, courses, and professors. Typical operations might include inserting, removing, and searching within each object or between them. That is, we might want to add a course to the list of university courses, or search for the courses taught by some professor.

Since the abstract data types and the algorithm specifications are language-independent, we postpone implementation decisions. Although we must specify the information required for each data object, we ignore coding details. For example, we might decide that the student data object should include name, social security number, major, and phone number. However, we would not yet pick a specific implementation for the list of students. As we will see in later chapters, there are several possibilities including arrays, linked lists, or trees. By deferring implementation issues as long as possible, we not only create a system that could be written in several programming languages, but we also have time to pick the most efficient implementations within our chosen language.

(4) Refinement and coding. In this phase, we choose representations for our data objects and write algorithms for each operation on them. The order in which we do this is crucial because a data object's representation can determine the efficiency of the algorithms related to it. Typically this means that we should write those algorithms that are independent of the data objects first.

Frequently at this point we realize that we could have created a much better system. Perhaps we have spoken with a friend who has worked on a similar project, or we realize that one of our alternate designs is superior. If our original design is good, it can absorb changes easily. In fact, this is a reason for avoiding an early commitment to coding details. If we must scrap our work entirely, we can take comfort in the fact that we will be able to write the new system more quickly and with fewer errors.

(5) Verification. This phase consists of developing correctness proofs for the program, testing the program with a variety of input data, and removing errors. Each of these areas has been researched extensively, and a complete discussion is beyond the scope of this text. However, we want to summarize briefly the important aspects of each area.

Correctness proofs: Programs can be proven correct using the same techniques that abound in mathematics. Unfortunately, these proofs are very time-consuming, and difficult to develop for large projects. Frequently scheduling constraints prevent the development of a complete set of proofs for a large system. However, selecting algorithms that have been proven correct can reduce the number of errors. In this text, we will provide you with an arsenal of algorithms, some of which have been proven correct using formal techniques, that you may apply to many programming problems.

Testing: We can construct our correctness proofs before and during the coding phase since our algorithms need not be written in a specific programming language. Testing, however, requires the working code and sets of test data. This data should be developed carefully so that it includes all possible scenarios. Frequently beginning programmers assume that if their program ran without producing a syntax error, it must be correct. Little thought is given to the input data, and usually only one set of data is used. Good test data should verify that every piece of code runs correctly. For example, if our program contains a **switch** statement, our test data should be chosen so that we can check each **case** within the **switch** statement.

Initial system tests focus on verifying that a program runs correctly. While this is a crucial concern, a program's running time is also important. An error-free program that runs slowly is of little value. Theoretical estimates of running time exist for many algorithms and we will derive these estimates as we introduce new algorithms. In addition, we may want to gather performance estimates for portions of our code. Constructing these timing tests is also a topic that we pursue later in this chapter.

Error removal. If done properly, the correctness proofs and system tests will indicate erroneous code. The ease with which we can remove these errors depends on the design and coding decisions made earlier. A large undocumented program written in "spaghetti" code is a programmer's nightmare. When debugging such programs, each corrected error possibly generates several new errors. On the other hand, debugging a well-documented program that is divided into autonomous units that interact through parameters is far easier. This is especially true if each unit is tested separately and then integrated into the system.

1.2 POINTERS AND DYNAMIC MEMORY ALLOCATION

1.2.1 Pointers

Pointers are fundamental to C and C provides extensive support for them. Actually, for any type *T* in C there is a corresponding type pointer-to-*T*. The actual value of a pointer type is an address of memory. The two most important operators used with the pointer type are:

- & the address operator
- * the dereferencing (or indirection) operator

If we have the declaration:

```
int i, *pi;
```

then *i* is an integer variable and *pi* is a pointer to an integer. If we say:

```
pi = &i;
```

then *&i* returns the address of *i* and assigns it as the value of *pi*. To assign a value to *i* we can say:

```
i = 10;
```

or

```
*pi = 10;
```

In both cases the integer 10 is stored as the value of *i*. In the second case, the * in front of the pointer *pi* causes it to be dereferenced, by which we mean that instead of storing 10 into the pointer, 10 is stored into the location pointed at by the pointer *pi*.

There are other operations we can do on pointers. We may assign a pointer to a variable of type pointer. Since a pointer is just a nonnegative integer number, C allows us to perform arithmetic operations such as addition, subtraction, multiplication, and division, on pointers. We also can determine if one pointer is greater than, less than, or equal to another, and we can convert pointers explicitly to integers.

The size of a pointer can be different on different computers. In some cases the size of a pointer on a computer can vary. For example, the size of a pointer to a **char** can be longer than a pointer to a **float**. C has a special value that it treats as a null pointer. The null pointer points to no object or function. Typically the null pointer is represented by the integer 0. The C macro *NULL* is defined to be this constant. The null pointer can be used in relational expressions, where it is interpreted as false. Therefore, to test for the null pointer in C we can say:

```
if (pi == NULL)
```

or more simply:

```
if (!pi)
```

1.2.2 Dynamic Memory Allocation

In your program you may wish to acquire space in which you will store information. When you write your program you may not know how much space you will need (for example, the size of an array may depend on an input to the program), nor do you wish to allocate some very large area that may never be required. To solve this problem C provides a mechanism, called a *heap*, for allocating storage at run-time. Whenever you need a new area of memory, you may call a function, *malloc*, and request the amount you need. If the memory is available, a pointer to the start of an area of memory of the required size is returned. When the requested memory is not available, the pointer *NULL* is returned. At a later time when you no longer need an area of memory, you may free it by calling another function, *free*, and return the area of memory to the system. Once an area of memory is freed, it is improper to use it. Program 1.1 shows how we might allocate and deallocate storage to pointer variables.

The call to *malloc* includes a parameter that determines the size of storage required to hold the **int** or the **float**. The result is a pointer to the first byte of a storage area of the proper size. The type of the result can vary. On some systems the result of *malloc* is a **char** *, a pointer to a **char**. However, those who use ANSI C will find that

```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
printf("an integer = %d, a float = %f\n", *pi, *pf);
free(pi);
free(pf);
```

Program 1.1: Allocation and deallocation of memory

the result is **void ***. The notation (*int* *) and (*float* *) are *type cast* expressions, which may be omitted in Program 1.1. They transform the resulting pointer into a pointer to the correct type. The pointer is then assigned to the proper pointer variable. The *free* function deallocates an area of memory previously allocated by *malloc*. In some versions of C, *free* expects an argument that is a **char ***, while ANSI C expects **void ***. However, the casting of the argument is generally omitted in the call to *free*.

Since there is the possibilty that a call to *malloc* may fail for lack of sufficient memory, we can write a more robust version of Program Program 1.1 by replacing the lines of code that invoke *malloc* by the code

```
if ((pi = (int *) malloc(sizeof(int))) == NULL ||
    (pf = (float *) malloc(sizeof(float))) == NULL)
{fprintf(stderr, "Insufficient memory");
 exit(EXIT_FAILURE);
}
```

or by the equivalent code

```
if (!(pi = malloc(sizeof(int))) ||
    !(pf = malloc(sizeof(float))))
{fprintf(stderr, "Insufficient memory");
 exit(EXIT_FAILURE);
}
```

Since *malloc* may be invoked from several places in your program, it is often convenient to define a macro that invokes *malloc* and exits when *malloc* fails. A possible macro definition is:

```
#define MALLOC(p,s) \
   if (!((p) = malloc(s))) {\
      fprintf(stderr, "Insufficient memory"); \
      exit(EXIT_FAILURE);\
   }
```

Now, the two lines of Program 1.1 that invoke *malloc* may be replaced by the code

```
MALLOC(pi, sizeof(int));
MALLOC(pf, sizeof(float));
```

In Program 1.1 if we insert the line:

```
          pf = (float *) malloc(sizeof(float));
```

immediately after the *printf* statement, then the pointer to the storage used to hold the value 3.14 has disappeared. Now there is no way to retrieve this storage. This is an example of a *dangling reference*. Whenever all pointers to a dynamically allocated area of storage are lost, the storage is lost to the program. As we examine programs that make use of pointers and dynamic storage, we will make it a point to always return storage after we no longer need it.

1.2.3 Pointers Can Be Dangerous

When programming in C, it is a wise practice to set all pointers to *NULL* when they are not actually pointing to an object. This makes it less likely that you will attempt to access an area of memory that is either out of range of your program or that does not contain a pointer reference to a legitimate object. On some computers, it is possible to dereference the null pointer and the result is *NULL*, permitting execution to continue. On other computers, the result is whatever the bits are in location zero, often producing a serious error.

Another wise programming tactic is to use explicit **type casts** when converting between pointer types. For example:

```
pi = malloc(sizeof(int));
     /* assign to pi a pointer to int */
pf = (float *) pi;
     /* casts an int pointer to a float pointer */
```

Another area of concern is that in many systems, pointers have the same size as type **int**. Since **int** is the default type specifier, some programmers omit the return type when defining a function. The return type defaults to **int** which can later be interpreted as a pointer. This has proven to be a dangerous practice on some computers and the programmer is urged to define explicit return types for functions.

1.3 ALGORITHM SPECIFICATION

1.3.1 Introduction

The concept of an algorithm is fundamental to computer science. Algorithms exist for many common problems, and designing efficient algorithms plays a crucial role in developing large-scale computer systems. Therefore, before we proceed further we need to discuss this concept more fully. We begin with a definition.

Definition: 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.** There are zero or more quantities that 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 basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible. □

In computational theory, one distinguishes between an algorithm and a program, the latter of which does not have to satisfy the fourth 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 will always terminate, we will 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 will present most of our algorithms in C, occasionally resorting to a combination of English and C for our specifications. Two examples should help to illustrate the process of translating a problem into an algorithm.

Example 1.1 [*Selection sort*]: Suppose we must devise a program that sorts a set of $n \geq 1$ integers. A simple solution is given by the following:

From those integers 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 since it leaves several unanswered questions. For example, it does not tell us where and how the integers are initially stored, or where we should place the result. We assume that the integers are stored in an array, *list*, such that the *i*th integer is stored in the *i*th position, *list*[i], $0 \leq i < n$. Program 1.2 is our first attempt at deriving a solution. Notice that it is written partially in C and partially in English.

```
for (i = 0; i < n; i++) {
  Examine list[i] to list[n-1] and suppose that the
  smallest integer is  at list[min];

  Interchange list[i] and list[min];
}
```

Program 1.2: Selection sort algorithm

To turn Program 1.2 into a real C program, two clearly defined subtasks remain: finding the smallest integer and interchanging it with *list*[i]. We can solve the latter problem using either a function (Program 1.3) or a macro.

```
void swap(int *x, int *y)
{/* both parameters are pointers to ints */
   int temp = *x;   /* declares temp as an int and assigns
                 to it the contents of what x points to */
   *x = *y; /* stores what y points to into the location
               where x points */
   *y = temp; /* places the contents of temp in location
                 pointed to by y */
}
```

Program 1.3: Swap function

Using the function, suppose a and b are declared as **ints**. To swap their values one would say:

```
swap(&a, &b);
```

passing to *swap* the addresses of *a* and *b*. The macro version of swap is:

```
#define SWAP(x,y,t) ((t) = (x), (x) = (y), (y) = (t))
```

The function's code is easier to read than that of the macro but the macro works with any data type.

We can solve the first subtask by assuming that the minimum is *list* [*i*], checking *list* [*i*] with *list* [*i* +1], *list* [*i* +2], $\cdots$, *list* [*n* −1]. Whenever we find a smaller number we make it the new minimum. When we reach *list* [*n* −1] we are finished. Putting all these observations together gives us *sort* (Program 1.4). Program 1.4 contains a complete program which you may run on your computer. The program uses the *rand* function defined in *math.h* to randomly generate a list of numbers which are then passed into *sort*. At this point, we should ask if this function works correctly.

Theorem 1.1: Function *sort* (*list*,*n*) correctly sorts a set of $n \geq 1$ integers. The result remains in *list* [0], $\cdots$, *list* [*n* −1] such that *list* [0] ≤ *list* [1] ≤ $\cdots$ ≤ *list* [*n* −1].

Proof: When the outer **for** loop completes its iteration for $i = q$, we have *list* [*q*] ≤ *list* [*r*], $q < r < n$. Further, on subsequent iterations, $i > q$ and *list* [0] through *list* [*q*] are unchanged. Hence following the last iteration of the outer **for** loop (i.e., $i = n - 2$), we have *list* [0] ≤ *list* [1] ≤ $\cdots$ ≤ *list* [*n* −1]. □

Example 1.2 [***Binary search***]**:** Assume that we have $n \geq 1$ distinct integers that are already sorted and stored in the array *list*. That is, *list* [0] ≤ *list* [1] ≤ $\cdots$ ≤ *list* [*n* −1]. We must figure out if an integer *searchnum* is in this list. If it is we should return an index, *i*, such that *list*[*i*] = *searchnum*. If *searchnum* is not present, we should return −1. Since the list is sorted we may use the following method to search for the value.

Let *left* and *right*, respectively, denote the left and right ends of the list to be searched. Initially, *left* = 0 and *right* = *n*−1. Let *middle* = (*left*+*right*)/2 be the middle position in the list. If we compare *list* [*middle*] with *searchnum*, we obtain one of three results:

(1) **searchnum < list[middle]**. In this case, if *searchnum* is present, it must be in the positions between 0 and *middle* − 1. Therefore, we set *right* to *middle* − 1.

(2) **searchnum = list[middle].** In this case, we return *middle*.

(3) **searchnum > list[middle].** In this case, if *searchnum* is present, it must be in the positions between *middle* + 1 and *n* − 1. So, we set *left* to *middle* + 1.

```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ((t) = (x), (x)= (y), (y) = (t))
void sort(int [],int); /*selection sort */
void main(void)
{
   int i,n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d",&n);
   if( n < 1 || n > MAX_SIZE) {
     fprintf(stderr, "Improper value of n\n");
     exit(EXIT_FAILURE);
   }
   for (i = 0; i < n; i++) {/*randomly generate numbers*/
      list[i] = rand() % 1000;
      printf("%d  ",list[i]);
   }
   sort(list,n);
   printf("\n Sorted array:\n ");
   for (i = 0; i < n; i++) /* print out sorted numbers */
      printf("%d  ",list[i]);
   printf("\n");
}
void sort(int list[],int n)
{
   int i, j, min, temp;
   for (i = 0; i < n-1; i++)  {
      min = i;
      for (j = i+1; j < n; j++)
         if (list[j] < list[min])
            min = j;
      SWAP(list[i],list[min],temp);
   }
}
```

Program 1.4: Selection sort

If *searchnum* has not been found and there are still integers to check, we recalculate *middle* and continue the search. Program 1.5 implements this searching strategy. The algorithm contains two subtasks: (1) determining if there are any integers left to check, and (2) comparing *searchnum* to *list*[*middle*].

```
while (there are more integers to check ) {
   middle = (left + right) / 2;
   if (searchnum < list[middle])
      right = middle - 1;
   else if (searchnum == list[middle])
           return middle;
        else left = middle + 1;
}
```

Program 1.5: Searching a sorted list

We can handle the comparisons through either a function or a macro. In either case, we must specify values to signify less than, equal, or greater than. We will use the strategy followed in C's library functions:

- We return a negative number (−1) if the first number is less than the second.
- We return a 0 if the two numbers are equal.
- We return a positive number (1) if the first number is greater than the second.

Although we present both a function (Program 1.6) and a macro, we will use the macro throughout the text since it works with any data type.

```
int compare(int x, int y)
{/* compare x and y, return -1 for less than, 0 for equal,
    1 for greater */
   if (x < y) return -1;
   else if (x == y) return 0;
        else return 1;
}
```

Program 1.6: Comparison of two integers

The macro version is:

```
#define COMPARE(x,y) (((x) < (y)) ? -1: ((x) == (y))? 0: 1)
```

We are now ready to tackle the first subtask: determining if there are any elements left to check. You will recall that our initial algorithm indicated that a comparison could cause us to move either our left or right index. Assuming we keep moving these indices, we will eventually find the element, or the indices will cross, that is, the left index will have a higher value than the right index. Since these indices delineate the search boundaries, once they cross, we have nothing left to check. Putting all this information together gives us *binsearch* (Program 1.7).

```
int binsearch(int list[], int searchnum, int left,
                                          int right)
{/* search list[0] <= list[1] <=  · · ·  <= list[n-1] for
 searchnum. Return its position if found. Otherwise
 return -1 */
   int  middle;
   while (left <= right)  {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: left = middle + 1;
                  break;
         case 0 : return middle;
         case 1 : right = middle - 1;
      }
   }
   return -1;
}
```

Program 1.7: Searching an ordered list

The search strategy just outlined is called *binary search.* □

The previous examples have shown that algorithms are implemented as functions in C. Indeed functions are the primary vehicle used to divide a large program into manageable pieces. They make the program easier to read, and, because the functions can be tested separately, increase the probability that it will run correctly. Often we will declare a function first and provide its definition later. In this way the compiler is made aware that a name refers to a legal function that will be defined later. In C, groups of functions can be compiled separately, thereby establishing libraries containing groups of logically related algorithms.

1.3.2 Recursive Algorithms

Typically, beginning programmers view a function as something that is invoked (called) by another function. It executes its code and then returns control to the calling function. This perspective ignores the fact that functions can call themselves (*direct recursion*) or they may call other functions that invoke the calling function again (*indirect recursion*). These recursive mechanisms are not only extremely powerful, but they also frequently allow us to express an otherwise complex process in very clear terms. It is for these reasons that we introduce recursion here.

Frequently computer science students regard recursion as a mystical technique that is useful for only a few special problems such as computing factorials or Ackermann's function. This is unfortunate because any function that we can write using assignment, **if-else**, and **while** statements can be written recursively. Often this recursive function is easier to understand than its iterative counterpart.

How do we determine when we should express an algorithm recursively? One instance is when the problem itself is defined recursively. Factorials and Fibonacci numbers fit into this category as do binomial coefficients where:

$$\begin{bmatrix} n \\ m \end{bmatrix} = \frac{n!}{m!(n-m)!}$$

can be recursively computed by the formula:

$$\begin{bmatrix} n \\ m \end{bmatrix} = \begin{bmatrix} n-1 \\ m \end{bmatrix} + \begin{bmatrix} n-1 \\ m-1 \end{bmatrix}$$

We would like to use two examples to show you how to develop a recursive algorithm. In the first example, we take the binary search function that we created in Example 1.2 and transform it into a recursive function. In the second example, we recursively generate all possible permutations of a list of characters.

Example 1.3 [*Binary search*]: Program 1.7 gave the iterative version of a binary search. To transform this function into a recursive one, we must (1) establish boundary conditions that terminate the recursive calls, and (2) implement the recursive calls so that each call brings us one step closer to a solution. If we examine Program 1.7 carefully we can see that there are two ways to terminate the search: one signaling a success (*list*[*middle*] = *searchnum*), the other signaling a failure (the left and right indices cross). We do not need to change the code when the function terminates successfully. However, the **while** statement that is used to trigger the unsuccessful search needs to be replaced with an equivalent **if** statement whose **then** clause invokes the function recursively.

Creating recursive calls that move us closer to a solution is also simple since it requires only passing the new *left* or *right* index as a parameter in the next recursive call. Program 1.8 implements the recursive binary search. Notice that although the code has changed, the recursive function call is identical to that of the iterative function. □

```
int binsearch(int list[], int searchnum, int left,
                                         int right)
{/* search list[0] <= list[1] <= ... <= list[n-1] for
    searchnum. Return its position if found. Otherwise
    return -1 */
   int middle;
   if (left <= right) {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: return
            binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return
            binsearch(list, searchnum, left, middle - 1);
      }
   }
   return -1;
}
```

Program 1.8: Recursive implementation of binary search

Example 1.4 [***Permutations***]: Given a set of $n \geq 1$ elements, print out all possible permutations of this set. For example, if the set is $\{a, b, c\}$, then the set of permutations is $\{(a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a)\}$. It is easy to see that, given n elements, there are $n!$ permutations. We can obtain a simple algorithm for generating the permutations if we look at the set $\{a, b, c, d\}$. We can construct the set of permutations by printing:

(1) a followed by all permutations of (b, c, d)

(2) b followed by all permutations of (a, c, d)

(3) c followed by all permutations of (a, b, d)

(4) d followed by all permutations of (a, b, c)

The clue to the recursive solution is the phrase "followed by all permutations." 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 the development of Program 1.9. We assume that *list* is a character array. Notice that it recursively generates permutations until $i = n$. The initial function call is *perm* (*list*, 0, $n - 1$).

Try to simulate Program 1.9 on the three-element set $\{a, b, c\}$. Each recursive call

```
void perm(char *list, int i, int n)
{/* generate all the permutations of list[i] to list[n] */
   int j, temp;
   if (i == n) {
      for (j = 0; j <= n; j++)
         printf("%c", list[j]);
      printf("    ");
   }
   else {
   /* list[i] to list[n] has more than one permutation,
      generate these recursively */
      for (j = i; j <= n; j++) {
         SWAP(list[i],list[j],temp);
         perm(list,i+1,n);
         SWAP(list[i],list[j],temp);
      }
   }
}
```

Program 1.9: Recursive permutation generator

of *perm* produces new local copies of the parameters *list*, *i*, and *n*. The value of *i* will differ from invocation to invocation, but *n* will not. The parameter *list* is an array pointer and its value also will not vary from call to call. □

We will encounter recursion several more times since many of the algorithms that appear in subsequent chapters are recursively defined. This is particularly true of algorithms that operate on lists (Chapter 4) and binary trees (Chapter 5).

EXERCISES

In the last several examples, we showed you how to translate a problem into a program. We have avoided the issues of data abstraction and algorithm design strategies, choosing to focus on developing a function from an English description, or transforming an iterative algorithm into a recursive one. In the exercises that follow, we want you to use the same approach. For each programming problem, try to develop an algorithm, translate it into a function, and show that it works correctly. Your correctness "proof" can employ an analysis of the algorithm or a suitable set of test runs.

1. Consider the two statements:

   (a) Is $n = 2$ the largest value of n for which there exist positive integers x, y, and z such that $x^n + y^n = z^n$ has a solution?

(b) Store 5 divided by zero into x and go to statement 10.

Both fail to satisfy one of the five criteria of an algorithm. Which criterion do they violate?

2. Horner's rule is a strategy for evaluating a polynomial $A(x) =$

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 + a_0$$

at point x_0 using a minimum number of multiplications. This rule is:

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

Write a C program to evaluate a polynomial using Horner's rule.

3. Given n Boolean variables $x_1, \cdots, x_n$, we wish to print all possible combinations of truth values they can assume. For instance, if $n = 2$, there are four possibilities: <*true*, *true*>, <*false*, *true*>, <*true*, *false*>, and <*false*, *false*>. Write a C program to do this.
4. Write a C program that prints out the integer values of x, y, z in ascending order.
5. The pigeon hole principle states that if a function f has n distinct inputs but less than n distinct outputs then there are two inputs a and b such that $a \neq b$ and $f(a) = f(b)$. Write a C program to find the values a and b for which the range values are equal.
6. Given n, a positive integer, determine if n is the sum its divisors, that is, if n is the sum of all t such that $1 \leq t < n$ and t divides n.
7. 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 C function to compute $n!$.
8. 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 C function to compute f_i.
9. Write an iterative function to compute a binomial coefficient, then transform it into an equivalent recursive function.
10. Ackerman's function $A(m, n)$ is defined as:

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

This function is studied because it grows very quickly for small values of m and n. Write recursive and iterative versions of this function.

11. [***Towers of Hanoi***] There are three towers and 64 disks of different diameters placed on the first tower. The disks are in order of decreasing diameter as one scans up the tower. Monks were reputedly supposed to move the disk from tower

1 to tower 3 obeying the rules:

(a) Only one disk can be moved at any time.

(b) No disk can be placed on top of a disk with a smaller diameter.

Write a recursive function that prints out the sequence of moves needed to accomplish this task.

12. If S is a set of n elements the power set of S is the set of all possible subsets of S. For example, if $S = \{a, b, c\}$, then *powerset* $(S) = \{ \{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Write a recursive function to compute *powerset(S)*.

1.4 DATA ABSTRACTION

The reader is no doubt familiar with the basic data types of C. These include **char**, **int**, **float**, and **double**. Some of these data types may be modified by the keywords **short**, **long**, and **unsigned**. Ultimately, the real world abstractions we wish to deal with must be represented in terms of these data types. In addition to these basic types, C helps us by providing two mechanisms for grouping data together. These are the array and the structure. *Arrays* are collections of elements of the same basic data type. They are declared implicitly, for example, *int list*[5] defines a five-element array of integers whose legitimate subscripts are in the range 0 $\cdots$ 4. *Structs* are collections of elements whose data types need not be the same. They are explicitly defined. For example,

```
struct {
       char lastName;
       int studentId;
       char grade;
       } student;
```

defines a structure with three fields, two of type character and one of type integer. The structure name is *student*. Details of C structures are provided in Chapter 2.

All programming languages provide at least a minimal set of predefined data types, plus the ability to construct new, or *user-defined types*. It is appropriate to ask the question, "What is a data type?"

Definition: A *data type* is a collection of *objects* and a set of *operations* that act on those objects. □

Whether your program is dealing with predefined data types or user-defined data types, these two aspects must be considered: objects and operations. For example, the data type **int** consists of the objects $\{0, +1, -1, +2, -2, \cdots$. INT_MAX, INT_MIN$\}$, where INT_MAX and INT_MIN are the largest and smallest integers that can be represented on your machine. (They are defined in *limits.h*.) The operations on integers are many, and would certainly include the arithmetic operators +, –, *, /, and %. There is also testing for equality/inequality and the operation that assigns an integer to a variable. In all of these cases, there is the name of the operation, which may be a prefix operator, such as *atoi*, or an infix operator, such as +. Whether an operation is defined in the language or in a library, its name, possible arguments and results must be specified.

In addition to knowing all of the facts about the operations on a data type, we might also want to know about how the objects of the data type are represented. For example on most computers a **char** is represented as a bit string occupying 1 byte of memory, whereas an **int** might occupy 2 or possibly 4 bytes of memory. If 2 eight-bit bytes are used, then *INT–MAX* is $2^{15} - 1 = 32{,}767$.

Knowing the representation of the objects of a data type can be useful and dangerous. By knowing the representation we can often write algorithms that make use of it. However, if we ever want to change the representation of these objects, we also must change the routines that make use of it. It has been observed by many software designers that hiding the representation of objects of a data type from its users is a good design strategy. In that case, the user is constrained to manipulate the objects solely through the functions that are provided. The designer may still alter the representation as long as the new implementations of the operations do not change the user interface. This means that users will not have to recode their algorithms.

Definition: An *abstract data type (ADT)* is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operations. □

Some programming languages provide explicit mechanisms to support the distinction between specification and implementation. For example, Ada has a concept called a *package*, and C++ has a concept called a *class*. Both of these assist the programmer in implementing abstract data types. Although C does not have an explicit mechanism for implementing ADTs, it is still possible and desirable to design your data types using the same notion.

How does the specification of the operations of an ADT differ from the implementation of the operations? The specification consists of the names of every function, the type of its arguments, and the type of its result. There should also be a description of what the function does, but without appealing to internal representation or implementation details. This requirement is quite important, and it implies that an abstract data type is *implementation-independent*. Furthermore, it is possible to classify the functions of a data type into several categories:

(1) **Creator/constructor**: These functions create a new instance of the designated type.

(2) **Transformers**: These functions also create an instance of the designated type, generally by using one or more other instances. The difference between

constructors and transformers will become more clear with some examples.

(3) **Observers/reporters**: These functions provide information about an instance of the type, but they do not change the instance.

Typically, an ADT definition will include at least one function from each of these three categories.

Throughout this text, we will emphasize the distinction between specification and implementation. In order to help us do this, we will typically begin with an ADT definition of the object that we intend to study. This will permit the reader to grasp the essential elements of the object, without having the discussion complicated by the representation of the objects or by the actual implementation of the operations. Once the ADT definition is fully explained we will move on to discussions of representation and implementation. These are quite important in the study of data structures. In order to help us accomplish this goal, we introduce a notation for expressing an ADT.

Example 1.5 [***Abstract data type*** *NaturalNumber*]**:** As this is the first example of an ADT, we will spend some time explaining the notation. ADT 1.1 contains the ADT definition of *NaturalNumber*.

```
ADT NaturalNumber is
  objects: an ordered subrange of the integers starting at zero and ending at the
  maximum integer (INT_MAX) on the computer
  functions:
    for all x, y ∈ NaturalNumber; TRUE, FALSE ∈ Boolean
    and where +, −, <, and == are the usual integer operations

    NaturalNumber Zero( )            ::=  0
    Boolean IsZero(x)                ::=  if (x) return FALSE
                                          else return TRUE
    Boolean Equal(x, y)              ::=  if (x == y) return TRUE
                                          else return FALSE
    NaturalNumber Successor(x)       ::=  if (x == INT_MAX) return x
                                          else return x + 1
    NaturalNumber Add(x, y)          ::=  if ((x + y) <= INT_MAX) return x + y
                                          else return INT_MAX
    NaturalNumber Subtract(x, y)     ::=  if (x < y) return 0
                                          else return x − y

end NaturalNumber
```

ADT 1.1: Abstract data type *NaturalNumber*

The ADT definition begins with the name of the ADT. There are two main sections in the definition: the objects and the functions. The objects are defined in terms of the integers, but we make no explicit reference to their representation. The function definitions are a bit more complicated. First, the definitions use the symbols x and y to denote two elements of the data type *NaturalNumber*, while *TRUE* and *FALSE* are elements of the data type *Boolean*. In addition, the definition makes use of functions that are defined on the set of integers, namely, plus, minus, equals, and less than. This is an indication that in order to define one data type, we may need to use operations from another data type. For each function, we place the result type to the left of the function name and a definition of the function to the right. The symbols "::=" should be read as "is defined as."

The first function, *Zero*, has no arguments and returns the natural number zero. This is a constructor function. The function *Successor(x)* returns the next natural number in sequence. This is an example of a transformer function. Notice that if there is no next number in sequence, that is, if the value of x is already *INT_MAX*, then we define the action of *Successor* to return *INT_MAX*. Some programmers might prefer that in such a case *Successor* return an error flag. This is also perfectly permissible. Other transformer functions are *Add* and *Subtract*. They might also return an error condition, although here we decided to return an element of the set *NaturalNumber*. □

ADT 1.1 shows you the general form that all ADT definitions will follow. However, in most of our further examples, the function definitions will not be so close to C functions. In fact, the nature of an ADT argues that we avoid implementation details. Therefore, we will usually use a form of structured English to explain the meaning of the functions. Often, there will be a discrepency even between the number of parameters used in the ADT definition of a function and its C implementation. To avoid confusion between the ADT definition of a function and its C implementation, ADT names begin with an upper case letter while C names begin with a lower case letter.

EXERCISES

For each of these exercises, provide a definition of the abstract data type using the form illustrated in ADT 1.1.

1. Add the following operations to the *NaturalNumber* ADT: *Predecessor*, *IsGreater*, *Multiply*, *Divide*.
2. Create an ADT, *Set*. Use the standard mathematics definition and include the following operations: *Create*, *Insert*, *Remove*, *IsIn*, *Union*, *Intersection*, *Difference*.
3. Create an ADT, *Bag*. In mathematics a *bag* is similar to a *set* except that a *bag* may contain duplicate elements. The minimal operations should include: *Create*, *Insert*, *Remove*, and *IsIn*.
4. Create an ADT, *Boolean*. The minimal operations are *And*, *Or*, *Not*, *Xor* (Exclusive or), *Equivalent*, and *Implies*.

1.5 PERFORMANCE ANALYSIS

One of the goals of this book is to develop your skills for making evaluative judgments about programs. There are many criteria upon which we can judge a program, including:

(1) Does the program meet the original specifications of the task?

(2) Does it work correctly?

(3) Does the program contain documentation that shows how to use it and how it works?

(4) Does the program effectively use functions to create logical units?

(5) Is the program's code readable?

Although the above criteria are vitally important, particularly in the development of large systems, it is difficult to explain how to achieve them. The criteria are associated with the development of a good programming style and this takes experience and practice. We hope that the examples used throughout this text will help you improve your programming style. However, we also can judge a program on more concrete criteria, and so we add two more criteria to our list.

(6) Does the program efficiently use primary and secondary storage?

(7) Is the program's running time acceptable for the task?

These criteria focus on performance evaluation, which we can loosely divide into two distinct fields. The first field focuses on obtaining estimates of time and space that are machine independent. We call this field *performance analysis*, but its subject matter is the heart of an important branch of computer science known as *complexity theory*. The second field, which we call *performance measurement*, obtains machine-dependent running times. These times are used to identify inefficient code segments. In this section we discuss performance analysis, and in the next we discuss performance measurement. We begin our discussion with definitions of the space and time complexity of a program.

Definition: The *space complexity* of a program is the amount of memory that it needs to run to completion. The *time complexity* of a program is the amount of computer time that it needs to run to completion. □

1.5.1 Space Complexity

The space needed by a program is the sum of the following components:

(1) **Fixed space requirements:** This component refers to space requirements that do not depend on the number and size of the program's inputs and outputs. The fixed requirements include the instruction space (space needed to store the code), space for simple variables, fixed-size structured variables (such as **structs**), and constants.

(2) **Variable space requirements:** This component consists of the space needed by structured variables whose size depends on the particular instance, I, of the problem being solved. It also includes the additional space required when a function uses recursion. The variable space requirement of a program P working on an instance I is denoted $S_P(I)$. $S_P(I)$ is usually given as a function of some *characteristics* of the instance I. Commonly used characteristics include the number, size, and values of the inputs and outputs associated with I. For example, if our input is an array containing n numbers then n is an instance characteristic. If n is the only instance charcteristic we wish to use when computing $S_P(I)$, we will use $S_P(n)$ to represent $S_P(I)$.

We can express the total space requirement $S(P)$ of any program as:

$$S(P) = c + S_P(I)$$

where c is a constant representing the fixed space requirements. When analyzing the space complexity of a program we are usually concerned with only the variable space requirements. This is particularly true when we want to compare the space complexity of several programs. Let us look at a few examples.

Example 1.6: We have a function, *abc* (Program 1.10), which accepts three simple variables as input and returns a simple value as output. According to the classification given, this function has only fixed space requirements. Therefore, $S_{abc}(I) = 0$. □

```
float abc(float a, float b, float c)
{
   return a+b+b*c+(a+b-c)/(a+b)+4.00;
}
```

Program 1.10: Simple arithmetic function

Example 1.7: We want to add a list of numbers (Program 1.11). Although the output is a simple value, the input includes an array. Therefore, the variable space requirement depends on how the array is passed into the function. Programming languages like Pascal may pass arrays by value. This means that the entire array is copied into temporary storage before the function is executed. In these languages the variable space requirement for this program is $S_{sum}(I) = S_{sum}(n) = n$, where n is the size of the array. C passes all parameters by value. When an array is passed as an argument to a function, C interprets it as passing the address of the first element of the array. C does not copy the array. Therefore, $S_{sum}(n) = 0$. □

```
float sum(float list[], int n)
{
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
      tempsum += list[i];
   return tempsum;
}
```

Program 1.11: Iterative function for summing a list of numbers

Example 1.8: Program 1.12 also adds a list of numbers, but this time the summation is handled recursively. This means that the compiler must save the parameters, the local variables, and the return address for each recursive call.

```
float rsum(float list[], int n)
{
   if (n) return rsum(list,n-1) + list[n-1];
   return 0;
}
```

Program 1.12: Recursive function for summing a list of numbers

In this example, the space needed for one recursive call is the number of bytes required for the two parameters and the return address. We can use the *sizeof* function to find the number of bytes required by each type. Figure 1.1 shows the number of bytes required for one recursive call under the assumption that an integer and a pointer each require 4 bytes.

If the array has $n = MAX\_SIZE$ numbers, the total variable space needed for the recursive version is $S_{rsum}(MAX\_SIZE) = 12*MAX\_SIZE$. If $MAX\_SIZE = 1000$, the variable space needed by the recursive version is $12*1000 = 12{,}000$ bytes. The iterative version has no variable space requirement. As you can see, the recursive version has a far greater overhead than its iterative counterpart. □

Type	Name	Number of bytes
parameter: array pointer	*list*[]	4
parameter: integer	*n*	4
return address: (used internally)		4
TOTAL per recursive call		12

Figure 1.1: Space needed for one recursive call of Program 1.12

EXERCISES

1. Determine the space complexity of the iterative and recursive factorial functions created in Exercise 7, Section 1.3.
2. Determine the space complexity of the iterative and recursive Fibonacci number functions created in Exercise 8, Section 1.3.
3. Determine the space complexity of the iterative and recursive binomial coefficient functions created in Exercise 9, Section 1.3.
4. Determine the space complexity of the function created in Exercise 5, Section 1.3 (pigeon hole principle).
5. Determine the space complexity of the function created in Exercise 12, Section 1.3 (powerset problem).

1.5.2 Time Complexity

The time, $T(P)$, taken by a program, P, is the sum of its *compile time* and its *run* (or *execution*) *time*. The compile time is similar to the fixed space component since it does not depend on the instance characteristics. In addition, once we have verified that the program runs correctly, we may run it many times without recompilation. Consequently, we are really concerned only with the program's execution time, T_P.

Determining T_P is not an easy task because it requires a detailed knowledge of the compiler's attributes. That is, we must know how the compiler translates our source program into object code. For example, suppose we have a simple program that adds and subtracts numbers. Letting n denote the instance characteristic, we might express $T_P(n)$ as:

$$T_P(n) = c_a ADD(n) + c_s SUB(n) + c_l LDA(n) + c_{st} STA(n)$$

where c_a, c_s, c_l, c_{st} are constants that refer to the time needed to perform each operation, and *ADD*, *SUB*, *LDA*, *STA* are the number of additions, subtractions, loads, and stores that are performed when the program is run with instance characteristic n.

Obtaining such a detailed estimate of running time is rarely worth the effort. If we must know the running time, the best approach is to use the system clock to time the program. We will do this later in the chapter. Alternately, we could count the number of operations the program performs. This gives us a machine-independent estimate, but we must know how to divide the program into distinct steps.

Definition: A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics. □

Note that the amount of computing represented by one program step may be different from that represented by another step. So, for example, we may count a simple assignment statement of the form $a = 2$ as one step and also count a more complex statement such as $a = 2*b+3*c/d-e+f/g/a/b/c$ as one step. The only requirement is that the time required to execute each statement that is counted as one step be independent of the instance characteristics.

We can determine the number of steps that a program or a function needs to solve a particular problem instance by creating a global variable, *count*, which has an initial value of 0 and then inserting statements that increment count by the number of program steps required by each executable statement.

Example 1.9 [*Iterative summing of a list of numbers*]: We want to obtain the step count for the sum function discussed earlier (Program 1.11). Program 1.13 shows where to place the *count* statements. Notice that we only need to worry about the executable statements, which automatically eliminates the function header, and the second variable declaration from consideration.

```
float sum(float list[], int n)
{
   float tempsum = 0;  count++; /* for assignment */
   int i;
   for (i = 0; i < n; i++)  {
      count++;                  /* for the for loop */
      tempsum += list[i]; count++;   /* for assignment */
   }
   count++; /* last execution of for */
   count++; /* for return */  return tempsum;
}
```

Program 1.13: Program 1.11 with count statements

Since our chief concern is determining the final count, we can eliminate most of the program statements from Program 1.13 to obtain a simpler program Program 1.14 that computes the same value for *count*. This simplification makes it easier to express the count arithmetically. Examining Program 1.14, we can see that if *count*'s initial value is 0, its final value will be $2n + 3$. Thus, each invocation of *sum* executes a total of $2n + 3$ steps. □

```
float sum(float list[], int n)
{
  float tempsum = 0;
  int i;
  for (i = 0; i < n; i++)
    count += 2;
  count +=3;
  return 0;
}
```

Program 1.14: Simplified version of Program 1.13

Example 1.10 [***Recursive summing of a list of numbers***]: We want to obtain the step count for the recursive version of the summing function. Program 1.15 contains the original function (Program 1.12) with the step counts added.

```
float rsum(float list[], int n)
{
  count++;       /* for if conditional */
  if (n) {
    count++;  /* for return and rsum invocation */
    return rsum(list,n-1) + list[n-1];
  }
  count++;
  return list[0];
}
```

Program 1.15: Program 1.12 with count statements added

To determine the step count for this function, we first need to figure out the step count for the boundary condition of $n = 0$. Looking at Program 1.15, we can see that when $n = 0$ only the **if** conditional and the second **return** statement are executed. So, the total step count for $n = 0$ is 2. For $n > 0$, the **if** conditional and the first **return** statement are executed. So each recursive call with $n > 0$ adds two to the step count. Since there are n such function calls and these are followed by one with $n = 0$, the step count for the function is $2n + 2$.

Surprisingly, the recursive function actually has a lower step count than its iterative counterpart. However, we must remember that the step count only tells us how many steps are executed, it does not tell us how much time each step takes. Thus, although the recursive function has fewer steps, it typically runs more slowly than the iterative version as its steps, on average, take more time than those of the iterative version. □

Example 1.11 [*Matrix addition*]: We want to determine the step count for a function that adds two-dimensional arrays (Program 1.16). The arrays a and b are added and the result is returned in array c. All of the arrays are of size $rows \times cols$. Program 1.17 shows the *add* function with the step counts introduced. As in the previous examples, we want to express the total count in terms of the size of the inputs, in this case *rows* and *cols*. To make the count easier to decipher, we can combine counts that appear within a single loop. This operation gives us Program 1.18.

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
                 int c[][MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < rows; i++)
      for (j = 0; j < cols; j++)
         c[i][j] = a[i][j] + b[i][j];
}
```

Program 1.16: Matrix addition

For Program 1.18, we can see that if *count* is initially 0, it will be $2rows \cdot cols + 2rows + 1$ on termination. This analysis suggests that we should interchange the matrices if the number of rows is significantly larger than the number of columns. □

By physically placing count statements within our functions we can run the functions and obtain precise counts for various instance characteristics. Another way to obtain step counts is to use a tabular method. To construct a step count table we first determine the step count for each statement. We call this the *steps/execution*, or *s/e* for short. Next we figure out the number of times that each statement is executed. We call this the *frequency*. The frequency of a nonexecutable statement is zero. Multiplying s/e

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
                  int c[][MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < rows; i++) {
      count++;  /* for i for loop */
      for (j = 0; j < cols; j++) {
         count++; /* for j for loop */
         c[i][j] = a[i][j] + b[i][j];
         count++; /*  for assignment statement */
      }
      count++; /* last time of j for loop */
   }
   count++; /* last time of i for loop */
}
```

Program 1.17: Matrix addition with count statements

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
                  int c[][MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++)
         count += 2;
      count += 2;
   }
   count++;
}
```

Program 1.18: Simplification of Program 1.17

by the frequency, gives us the *total steps* for each statement. Summing these totals, gives us the step count for the entire function. Although this seems like a very complicated process, in fact, it is quite easy. Let us redo our three previous examples using the tabular approach.

Example 1.12 [***Iterative function to sum a list of numbers***]**:** Figure 1.2 contains the step count table for Program 1.11. To construct the table, we first entered the steps/execution for each statement. Next, we figured out the frequency column. The **for** loop at line 5 complicated matters slightly. However, since the loop starts at 0 and terminates when i is equal to n, its frequency is $n + 1$. The body of the loop (line 6) only executes n times since it is not executed when $i = n$. We then obtained the total steps for each statement and the final step count. □

Statement	s/e	Frequency	Total steps
float sum(float list[], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for (i = 0; i < n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Figure 1.2: Step count table for Program 1.11

Example 1.13 [***Recursive function to sum a list of numbers***]**:** Figure 1.3 shows the step count table for Program 1.13. □

Example 1.14 [***Matrix addition***]**:** Figure 1.4 contains the step count table for the matrix addition function. □

Summary

The time complexity of a program is given by the number of steps taken by the program to compute the function it was written for. The number of steps is itself a function of the instance characteristics. While any specific instance may have several characteristics (e.g., the number of inputs, the number of outputs, the magnitudes of the inputs and outputs, etc.), the number 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 program, we might be interested in

Statement	s/e	Frequency	Total steps
float rsum(float list[], int n)	0	0	0
{	0	0	0
if (n)	1	$n+1$	$n+1$
return rsum(list,n−1) + list[n−1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			$2n+2$

Figure 1.3: Step count table for recursive summing function

Statement	s/e	Frequency	Total Steps
void add(int a[][MAX_SIZE] $\cdots$)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i=0; i<rows; i++)	1	$rows+1$	$rows+1$
for (j = 0; j < cols; j++)	1	$rows \cdot (cols+1)$	$rows \cdot cols + rows$
c[i][j] = a[i][j] + b[i][j];	1	$rows \cdot cols$	$rows \cdot cols$
}	0	0	0
Total			$2rows \cdot cols + 2rows+1$

Figure 1.4: Step count table for matrix addition

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 a program 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 function *add* the choice of characteristics was the number of rows and the number of columns in the matrices being added.

Once the relevant characteristics (n, m, p, q, r, $\ldots$) 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, . . .). 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, etc., be counted as one step.

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 elements, and the number of rows and columns. For many programs, the time complexity is not dependent solely on the number of inputs or outputs or some other easily specified characteristic. Consider the function *binsearch* (Program 1.7). This function searches an ordered list. A natural parameter with respect to which you might wish to determine the step count is the number, n, of elements in the list. That is, we would like to know how the computing time changes as we change the number of elements n. The parameter n is inadequate. For the same n, the step count varies with the position of the element *searchnum* that is being searched for. 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 steps 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 paramenters. The *worst-case step count* is the maximum number of steps that can be executed for the given paramenters. The *average step count* is the average number of steps executed on instances with the given parameters.

EXERCISES

1. Redo Exercise 2, Section 1.3 (Horner's rule for evaluating polynomials), so that step counts are introduced into the function. Express the total count as an equation.
2. Redo Exercise 3, Section 1.3 (truth tables), so that steps counts are introduced into the function. Express the total count as an equation.
3. Redo Exercise 4, Section 1.3 so that step counts are introduced into the function. Express the total count as an equation.
4. (a) Rewrite Program 1.19 so that step counts are introduced into the function.
   (b) Simplify the resulting function by eliminating statements.
   (c) Determine the value of *count* when the function ends.
   (d) Write the step count table for the function.
5. Repeat Exercise 4 with Program 1.20.
6. Repeat Exercise 4 with Program 1.21
7. Repeat Exercise 4 with Program 1.22

```
void printMatrix(int matrix[][MAX_SIZE], int rows,
                                         int cols)
{
   int i, j;
   for (i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++)
         printf("%d",matrix[i][j]);
      printf("\n");
   }
}
```

Program 1.19: Printing out a matrix

```
void mult(int a[][MAX_SIZE], int b[][MAX_SIZE],
                             int c[][MAX_SIZE])
{
   int i, j, k;
   for (i = 0; i < MAX_SIZE; i++)
      for (j = 0; j < MAX_SIZE; j++) {
         c[i][j] = 0;
         for (k = 0; k < MAX_SIZE; k++)
            c[i][j] += a[i][k] * b[k][j];
      }
}
```

Program 1.20: Matrix multiplication function

1.5.3 Asymptotic Notation (O, Ω, Θ)

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

Determining the exact step count (either worst case or average) of a program can prove to be an exceedingly difficult task. Expending immense effort to determine the step count exactly isn't a very worthwhile endeavor as 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 isn't very

```
void prod(int a[][MAX_SIZE], int b[][MAX_SIZE],
     int c[][MAX_SIZE], int rowsa, int colsb, int colsa)
{
   int i, j, k;
   for (i = 0; i < rowsa; i++)
      for (j = 0; j < colsb; j++) {
         c[i][j] = 0;
         for (k = 0; k < colsa; k++)
            c[i][j] += a[i][k] * b[k][j];
      }
}
```

Program 1.21: Matrix product function

```
void transpose(int a[][MAX_SIZE])
{
   int i, j, temp;
   for (i = 0; i < MAX_SIZE-1; i++)
      for (j = i+1; j < MAX_SIZE; j++)
         SWAP(a[i][j], a[j][i], temp);
}
```

Program 1.22: Matrix transposition function

useful for comparative purposes. An exception to this is when the difference in the step counts of two programs is very large as in $3n+3$ versus $100n+10$. We might feel quite safe in predicting that the program with step count $3n+3$ will run in less time than the one with step count $100n+10$. But even in this case, it isn't 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 $c_1 n^2 \le T_P(n) \le c_2 n^2$ or $T_Q(n,m) = c_1 n + c_2 m$ where c_1 and c_2 are nonnegative constants. This is so because if we have two programs with a complexity of $c_1 n^2 + c_2 n$ and $c_3 n$, respectively, then we know that the one with complexity $c_3 n$ will be faster than the one with complexity $c_1 n^2 + c_2 n$ for sufficiently large values of n. For small values of n, either program could be faster (depending on c_1, c_2, and c_3). If $c_1 = 1$, $c_2 = 2$, and $c_3 = 100$ then $c_1 n^2 + c_2 n \le c_3 n$ for $n \le 98$ and $c_1 n^2 + c_2 n > c_3 n$ for $n > 98$. If $c_1 = 1$, $c_2 = 2$, and $c_3 = 1000$, then $c_1 n^2 + c_2 n \le c_3 n$ for $n \le 998$.

No matter what the values of c_1, c_2, and c_3, there will be an n beyond which the program with complexity $c_3 n$ will be faster than the one with complexity $c_1 n^2 + c_2 n$. This value of n will be called the ***break even point***. If the break even point is 0 then the program with complexity $c_3 n$ is always faster (or at least as fast). The exact break even point cannot be determined analytically. The programs 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 adequate to know that one program has complexity $c_1 n^2 + c_2 n$ and the other $c_3 n$ 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.

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

Definition: [Big "oh"] $f(n) = O(g(n))$ (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 $f(n) \le cg(n)$ for all n, $n \ge n_0$. □

Example 1.15: $3n + 2 = O(n)$ as $3n + 2 \le 4n$ for all $n \ge 2$. $3n + 3 = O(n)$ as $3n + 3 \le 4n$ for all $n \ge 3$. $100n + 6 = O(n)$ as $100n + 6 \le 101n$ for $n \ge 10$. $10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \le 11n^2$ for $n \ge 5$. $1000n^2 + 100n - 6 = O(n^2)$ as $1000n^2 + 100n - 6 \le 1001n^2$ for $n \ge 100$. $6*2^n + n^2 = O(2^n)$ as $6*2^n + n^2 \le 7*2^n$ for $n \ge 4$. $3n + 3 = O(n^2)$ as $3n + 3 \le 3n^2$ for $n \ge 2$. $10n^2 + 4n + 2 = O(n^4)$ as $10n^2 + 4n + 2 \le 10n^4$ for $n \ge 2$. $3n + 2 \ne O(1)$ as $3n + 2$ is not less than or equal to c for any constant c and all n, $n \ge n_0$. $10n^2 + 4n + 2 \ne O(n)$. □

We write $O(1)$ to mean a computing time which 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 will see most often in this book.

As illustrated by the previous example, the statement $f(n) = O(g(n))$ only states that $g(n)$ is an upper bound on the value of $f(n)$ for all n, $n \ge n_0$. It doesn't say anything about how good this bound is. Notice that $n = O(n^2)$, $n = O(n^{2.5})$, $n = O(n^3)$, $n = O(2^n)$, etc. In order 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 shall often say $3n + 3 = O(n)$, we shall almost never say $3n + 3 = O(n^2)$ 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 as 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 $f(n)$ (i.e., the $g(n)$ in $f(n) = O(g(n))$) when $f(n)$ is a polynomial in n.

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

Proof: $f(n) \le \sum_{i=0}^{m} |a_i| n^i$

$$\le n^m \sum_{0}^{m} |a_i| n^{i-m}$$

$$\le n^m \sum_{0}^{m} |a_i|, \text{ for } n \ge 1$$

So, $f(n) = O(n^m)$. □

Definition: [Omega] $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist positive constants c and n_0 such that $f(n) \ge cg(n)$ for all n, $n \ge n_0$. □

Example 1.16: $3n + 2 = \Omega(n)$ as $3n + 2 \ge 3n$ for $n \ge 1$ (actually the inequality holds for $n \ge 0$ but the definition of Ω requires an $n_0 > 0$). $3n + 3 = \Omega(n)$ as $3n + 3 \ge 3n$ for $n \ge 1$. $100n + 6 = \Omega(n)$ as $100n + 6 \ge 100n$ for $n \ge 1$. $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \ge n^2$ for $n \ge 1$. $6*2^n + n^2 = \Omega(2^n)$ as $6*2^n + n^2 \ge 2^n$ for $n \ge 1$. 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 $6*2^n + n^2 = \Omega(1)$. □

As in the case of the "big oh" notation, there are several functions $g(n)$ for which $f(n) = \Omega(g(n))$. $g(n)$ 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 shall say that $3n + 3 = \Omega(n)$ and that $6*2^n + n^2 = \Omega(2^n)$, we shall almost never say that $3n + 3 = \Omega(1)$ or that $6*2^n + n^2 = \Omega(1)$ even though both these statements are correct.

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

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

Proof: Left as an exercise. □

Definition: [Theta] $f(n) = \Theta(g(n))$ (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 $c_1 g(n) \le f(n) \le c_2 g(n)$ for all n, $n \ge n_0$. □

Example 1.17: $3n + 2 = \Theta(n)$ 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 "big oh" and omega notations. $f(n) = \Theta(g(n))$ 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 has been 1. This is in accordance with practice. We shall almost never find ourselves saying that $3n + 3 = O(3n)$, or that $10 = O(100)$, or that $10n^2 + 4n + 2 = \Omega(4n^2)$, or that $6*2^n + n^2 = \Omega(6*2^n)$, or that $6*2^n + n^2 = \Theta(4*2^n)$, even though each of these statements is true.

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

Proof: Left as an exercise. □

Let us reexamine the time complexity analyses of the previous section. For function *sum* (Program 1.12) we had determined that $T_{sum}(n) = 2n + 3$. So, $T_{sum}(n) = \Theta(n)$. $T_{rsum}(n) = 2n + 2 = \Theta(n)$ and $T_{add}(rows, cols) = 2rows.cols + 2rows + 1 = \Theta(rows.cols)$.

While 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 one has 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 program and then adding up these complexities.

Example 1.18 [*Complexity of matrix addition*]: Using a tabular approach, we construct the table of Figure 1.5. This is quite similar to Figure 1.4. However, instead of putting in exact step counts, we put in asymptotic ones. For nonexecutable statements, we enter a step count of 0. Constructing a table such as the one in Figure 1.5 is actually easier than constructing the one is Figure 1.4. For example, it is harder to obtain the exact step count of $rows.(cols+1)$ for line 5 than it is to see that line 5 has an asymptotic complexity that is $\Theta(rows.cols)$. To obtain the asymptotic complexity of the function, we can add the asymptotic complexities of the individual program lines. Alternately, since the number of lines is a constant (i.e., is independent of the instance characteristics), we may simply take the maximum of the line complexities. Using either approach, we obtain $\Theta(rows.cols)$ as the asymptotic complexity. □

Example 1.19 [*Binary search*]: Let us obtain the time complexity of the binary search function *binsearch* (Program 1.7). The instance characteristic we shall use is the number

Statement	Asymptotic complexity
`void add(int a[][MAX_SIZE] ··· )`	0
`{`	0
`    int i, j;`	0
`    for (i=0; i<rows; i++)`	$\Theta(rows)$
`        for (j = 0; j < cols; j++)`	$\Theta(rows.cols)$
`            c[i][j] = a[i][j] + b[i][j];`	$\Theta(rows.cols)$
`}`	0
Total	$\Theta(rows.cols)$

Figure 1.5: Time complexity of matrix addition

n of elements in the list. Each iteration of the **while** loop takes $\Theta(1)$ time. We can show that the **while** loop is iterated at most $\lceil \log_2(n+1) \rceil$ times. Since an asymptotic analysis is being performed, we don't need such an accurate count of the worst-case number of iterations. Each iteration except for the last results in a decrease in the size of the segment of *list* that has to be searched by a factor of about 2. That is, the value of *right* − *left* + 1 reduces by a factor of about 2 on each iteration. So, this loop is iterated $\Theta(\log n)$ times in the worst case. As each iteration takes $\Theta(1)$ time, the overall worst-case complexity of *binsearch* is $\Theta(\log n)$. Notice that the best case complexity is $\Theta(1)$ as in the best case *searchnum* is found in the first iteration of the **while** loop. □

Example 1.20 [*Permutations*]: Consider function *perm* (Program 1.9). When $i = n$, the time taken is $\Theta(n)$. When $i < n$, the **else** clause is entered. The **for** loop of this clause is entered $n - i + 1$ times. Each iteration of this loop takes $\Theta(n + T_{perm}(i + 1, n))$ time. So, $T_{perm}(i, n) = \Theta((n - i + 1)(n + T_{perm}(i + 1, n)))$ when $i < n$. Since, $T_{perm}(i + 1, n)$, is at least n when $i + 1 \leq n$, we get $T_{perm}(i, n) = \Theta((n - i + 1)T_{perm}(i + 1, n))$ for $i < n$. Solving this recurrence, we obtain $T_{perm}(1,n) = \Theta(n(n!))$, $n \geq 1$. □

Example 1.21 [*Magic square*]: As our last example of complexity analysis, we use a problem from recreational mathematics, the creation of a magic square. A *magic square* is an $n \times n$ matrix of the integers from 1 to n^2 such that the sum of each row and column and the two major diagonals is the same. Figure 1.6 shows a magic square for the case $n = 5$. In this example, the common sum is 65.

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

Put a one in the middle box of the top row. Go up and left assigning numbers in increasing order to empty boxes. If your move causes you to jump off the square (that is,

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 1.6: Magic square for $n = 5$

you go beyond the square's boundaries), figure out where you would be if you landed on a box on the opposite side of the square. Continue with this box. If a box is occupied, go down instead of up and continue.

We created Figure 1.6 using Coxeter's rule. Program 1.23 contains the coded algorithm. Let n denote the size of the magic square (i.e., the value of the variable *size* in Program 1.23. The **if** statements that check for errors in the value of n take $\Theta(1)$ time. The two nested **for** loops have a complexity $\Theta(n^2)$. Each iteration of the next **for** loop takes $\Theta(1)$ time. This loop is iterated $\Theta(n^2)$ time. So, its complexity is $\Theta(n^2)$. The nested **for** loops that output the magic square also take $\Theta(n^2)$ time. So, the asymptotic complexity of Program 1.23 is $\Theta(n^2)$. □

```
#include <stdio.h>
#define MAX_SIZE  15 /* maximum size of square */
void main(void)
{/* construct a magic square, iteratively */
   int square[MAX_SIZE][MAX_SIZE];
   int i, j, row, column;     /* indexes */
   int count;                 /* counter */
   int size;                  /* square size */

   printf("Enter the size of the square: ");
   scanf("%d", &size);
   /* check for input errors */
   if (size < 1 || size > MAX_SIZE + 1) {
      fprintf(stderr, "Error!  Size is out of range\n");
      exit(EXIT FAILURE);
   }
   if (!(size % 2)) {
      fprintf(stderr, "Error!  Size is even\n");
      exit(EXIT_FAILURE);
   }
   for (i = 0; i < size; i++)
      for (j = 0; j < size; j++)
         square[i][j] = 0;
   square[0][(size-1) / 2] = 1; /* middle of first row */
   /* i and j are current position */
   i = 0;
   j = (size - 1) / 2;
   for (count = 2; count <= size * size; count++) {
      row = (i-1 < 0) ? (size - 1) : (i - 1); /*up*/
      column = (j-1 < 0) ? (size - 1) : (j - 1); /*left*/
      if (square[row][column])  /*down*/
         i = (++i) % size;
      else {                    /* square is unoccupied */
         i = row;
         j = (j-1 < 0) ? (size - 1) : --j;
      }
      square[i][j] = count;
   }
   /* output the magic square */
   printf(" Magic Square of size %d : \n\n",size);
   for (i = 0; i < size; i++) {
      for (j = 0; j < size; j++)
         printf("%5d", square[i][j]);
      printf("\n");
   }
   printf("\n\n");
}
```

Program 1.23: Magic square program

When we analyze programs in the following chapters, we will normally confine ourselves to providing an upper bound on the complexity of the program. That is, we will normally use only the big oh notation. We do this because this is the current trend in practice. In many of our analyses the theta notation could have been used in place of the big oh notation as the complexity bound obtained is both an upper and a lower bound for the program.

EXERCISES

1. Show that the following statements are correct:
   (a) $5n^2 - 6n = \Theta(n^2)$
   (b) $n! = O(n^n)$
   (c) $2n^2 + n \log n = \Theta(n^2)$
   (d) $\sum_{i=0}^{n} i^2 = \Theta(n^3)$
   (e) $\sum_{i=0}^{n} i^3 = \Theta(n^4)$
   (f) $n^{2^n} + 6 \cdot 2^n = \Theta(n^{2^n})$
   (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) $n^k + n + n^k \log n = \Theta(n^k \log n)$ for all $k \geq 1$.
   (k) $10n^3 + 15n^4 + 100n^2 2^n = O(n^2 2^n)$
2. Show that the following statements 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) $n^3 2^n + 6n^2 3^n = O(n^2 2^n)$
   (e) $3^n = O(2^n)$
3. Prove Theorem 1.3.
4. Prove Theorem 1.4.
5. Determine the worst-case complexity of Program 1.19.
6. Determine the worst-case complexity of Program 1.22.
7. Compare the two functions n^2 and $20n + 4$ for various values of n. Determine when the second function becomes smaller than the first.
8. Write an equivalent recursive version of the magic square program (Program 1.23).

1.5.4 Practical Complexities

We have seen that the time complexity of a program 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 may also be used to compare two programs P and Q that perform the same task. Assume that program P has complexity $\Theta(n)$ and program Q is of complexity $\Theta(n^2)$. We can assert that program P is faster than program Q for "sufficiently large" n. To see the validity of this assertion, observe that the actual computing time of P is bounded from above by cn for some constant c and for all n, $n \geq n_1$, while 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 $n \geq c/d$, program P is faster than program 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 the assertion of the preceding discussion. When deciding which of the two programs to use, we must know whether the n we are dealing with is, in fact, "sufficiently large." If program P actually runs in $10^6 n$ milliseconds while program Q runs in n^2 milliseconds and if we always have $n \leq 10^6$, then, other factors being equal, program Q is the one to use, other factors being equal.

To get a feel for how the various functions grow with n, you are advised to study Figures 1.7 and 1.8 very closely. As you can see, the function 2^n grows very rapidly with n. In fact, if a program 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 1 billion steps per second, this would require about 18.3 minutes. If $n = 50$, the same program would run for about 13 days on this computer. When $n = 60$, about 310.56 years will be required to execute the program and when $n = 100$, about $4*10^{13}$ years will be needed. So, we may conclude that the utility of programs 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	4096	65,536
5	32	160	1024	32,768	4,294,967,296

Figure 1.7: Function values

Programs that have a complexity that is a polynomial of high degree are also of limited utility. For example, if a program needs n^{10} steps, then using our 1 billion steps per second computer we will need 10 seconds when $n = 10$; 3,171 years when $n = 100$; and $3.17*10^{13}$ years when $n = 1000$. If the program's complexity had been n^3 steps instead, then we would need 1 second when $n = 1000$; 110.67 minutes when $n = 10{,}000$; and 11.57 days when $n = 100{,}000$.

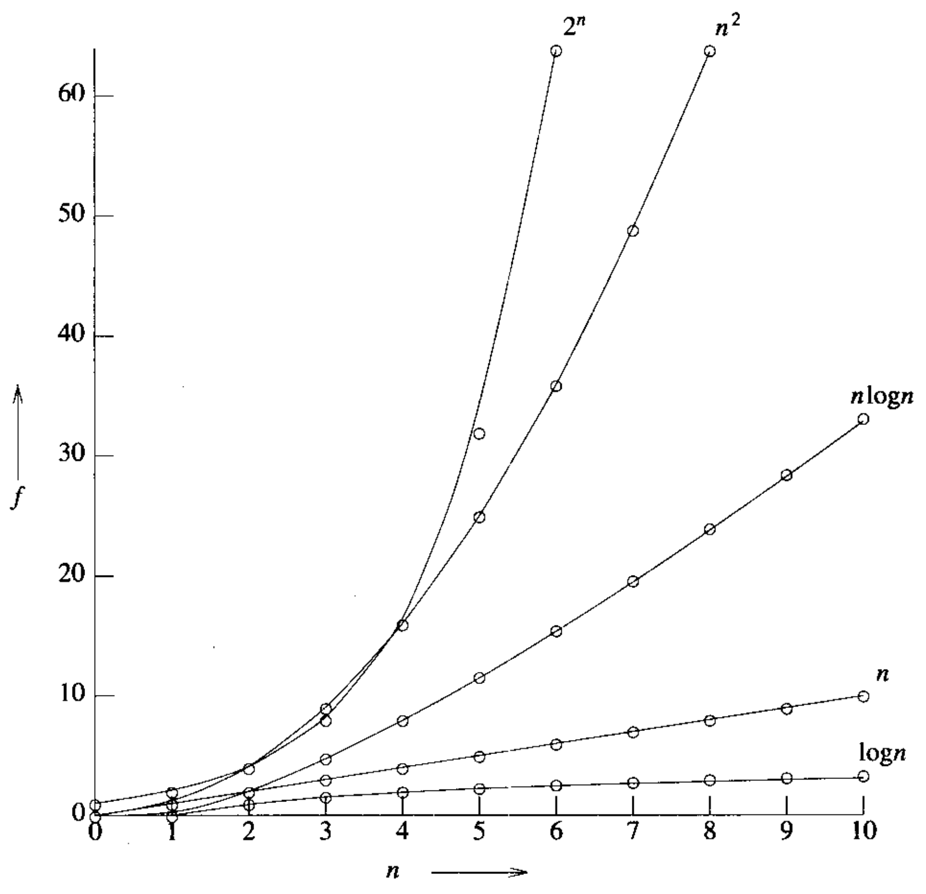

Figure 1.8 Plot of function values

Figure 1.9 gives the time needed by a 1 billion instructions per second computer to execute a program 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 programs of small complexity (such as n, $n\log n$, n^2, n^3) are feasible. Further, this is the case even if one could build a computer capable of executing 10^{12} instructions per second. In this case, the computing times of Figure 1.9 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.

	$f(n)$						
n	n	$n\log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μs	.03 μs	.1 μs	1 μs	10 μs	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 h	1 ms
30	.03 μ	.15 μ	.9 μ	27 μ	810 μ	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121 d	18 m
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 y	13 d
100	.10 μs	.66 μs	10 μs	1 ms	100 ms	3171 y	$4*10^{13}$ y
10^3	1 μs	9.96 μs	1 ms	1 s	16.67 m	$3.17*10^{13}$ y	$32*10^{283}$ y
10^4	10 μs	130 μs	100 ms	16.67 m	115.7 d	$3.17*10^{23}$ y	
10^5	100 μs	1.66 ms	10 s	11.57 d	3171 y	$3.17*10^{33}$ y	
10^6	1 ms	19.92 ms	16.67 m	31.71 y	$3.17*10^7$ y	$3.17*10^{43}$ y	

μs = microsecond = 10^{-6} seconds; ms = milliseconds = 10^{-3} seconds
s = seconds; m = minutes; h = hours; d = days; y = years

Figure 1.9: Times on a 1-billion-steps-per-second computer

1.6 PERFORMANCE MEASUREMENT

1.6.1 Clocking

Although performance analysis gives us a powerful tool for assessing an algorithm's space and time complexity, at some point we also must consider how the algorithm executes on our machine. This consideration moves us from the realm of analysis to that of measurement. We will concentrate our discussion on measuring time.

The functions we need to time events are part of C's standard library, and are accessed through the statement: *#include <time.h>*. There are actually two different methods for timing events in C. Figure 1.10 shows the major differences between these two methods.

Method 1 uses *clock* to time events. This function gives the amount of processor time that has elapsed since the program began running. To time an event we use *clock* twice, once at the start of the event and once at the end. The time is returned as a built-in type, *clock_t*. The total time required by an event is its start time subtracted from its stop time. Since this result could be any legitimate numeric type, we **type cast** it to **double**. In addition, since this result is measured as internal processor time, we must divide it by the number of clock ticks per second to obtain the result in seconds. In ANSI C, the ticks per second is held in the built-in constant, *CLOCKS_PER_SEC*. We found that

	Method 1	Method 2
Start timing	start = clock();	start = time(NULL);
Stop timing	stop = clock();	stop = time(NULL);
Type returned	clock_t	time_t
Result in seconds	duration = ((double) (stop−start)) / CLOCKS_PER_SEC;	duration = (double) difftime(stop,start);

Figure 1.10: Event timing in C

this method was far more accurate on our machine. However, the second method does not require a knowledge of the ticks per second, which is why we also present it here.

Method 2 uses *time*. This function returns the time, measured in seconds, as the built-in type *time_t*. Unlike *clock*, *time* has one parameter, which specifies a location to hold the time. Since we do not want to keep the time, we pass in a *NULL* value for this parameter. As was true of Method 1, we use *time* at the start and the end of the event we want to time. We then pass these two times into *difftime*, which returns the difference between two times measured in seconds. Since the type of this result is *time_t*, we **type cast** it to **double** before printing it out.

Example 1.22 [***Worst-case performance of selection sort***]: The worst case for selection sort occurs when the elements are in reverse order. That is, we want to sort into ascending order an array that is currently in descending order. To conduct our timing tests, we varied the size of the array from 0, 10, 20 , $\cdots$, 90, 100, 200 , $\cdots$, 1000. Program 1.24 contains the code we used to conduct the timing tests. (The code for the sort function is given in Program 1.4 and for the purposes of Program 1.24 is assumed to be in a file named *selectionSort.h*).

To conduct the timing tests, we used a **for** loop to control the size of the array. At each iteration, a new reverse ordered array of n numbers was created. We called *clock* immediately before we invoked *sort* and immediately after it returned. Surprisingly, the duration output for each n was 0! What went wrong? Although our timing program (Program 1.24) is logically correct, it fails to measure run times accurately because the events we are trying to time are too short! Since there is a measurement error of ± 1 tick, Program 1.24 returns accurate results only when the sort time is much more than 1 tick. Program 1.25 is a more accurate timing program for selection sort. In this program, for each n, we do the sort as many times as needed to bring the total time up to 1 second (1000 ticks). This program has some inaccuracies of its own. For example, the reported time includes the time to initialize the array that is to be sorted. However, this

```
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
   int i, n, step = 10;
   int a[MAX_SIZE];
   double duration;
   clock_t start;

   /* times for n = 0, 10, ..., 100, 200, ..., 1000 */
   printf("    n     time\n");
   for (n = 0; n <= 1000; n += step)
   {/* get time for size n */

      /* initialize with worst-case data */
      for (i = 0; i < n; i++)
         a[i] = n - i;

      start = clock( );
      sort(a, n);
      duration = ((double) (clock() - start))
                              / CLOCKS_PER_SEC;
      printf("%6d   %f\n", n, duration);
      if (n == 100) step = 100;
   }
}
```

Program 1.24: First timing program for selection sort

initialization time is small compared to the actual sort time (O(n) vs O(n^2)). In case, the initialization time is a concern, we may measure the initialization time using a separate experiment and subtract this from the time reported by Program 1.24.

The results from Program 1.24 are displayed in Figures 1.11 and 1.12. The curve of Figure 1.12 resembles the n^2 curve displayed in Figure 1.8. This agrees with our analysis of selection sort. □

```
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
   int i, n, step = 10;
   int a[MAX_SIZE];
   double duration;

   /* times for n = 0, 10, ..., 100, 200, ..., 1000 */
   printf("    n    repetitions     time\n");
   for (n = 0; n <= 1000; n += step)
   {
      /* get time for size n */
      long repetitions = 0;
      clock_t start = clock( );
      do
      {
         repetitions++;

         /* initialize with worst-case data */
         for (i = 0; i < n; i++)
            a[i] = n - i;

         sort(a, n);
      } while (clock( ) - start < 1000);
           /* repeat until enough time has elapsed */

      duration = ((double) (clock() - start))
                             / CLOCKS_PER_SEC;
      duration /= repetitions;
      printf("%6d  %9d   %f\n", n, repetitions, duration);
      if (n == 100) step = 100;
   }
}
```

Program 1.25: More accurate timing program for selection sort

n	repetitions	time
0	8690714	0.000000
10	2370915	0.000000
20	604948	0.000002
30	329505	0.000003
40	205605	0.000005
50	145353	0.000007
60	110206	0.000009
70	85037	0.000012
80	65751	0.000015
90	54012	0.000019
100	44058	0.000023
200	12582	0.000079
300	5780	0.000173
400	3344	0.000299
500	2096	0.000477
600	1516	0.000660
700	1106	0.000904
800	852	0.001174
900	681	0.001468
1000	550	0.001818

Figure 1.11: Worst-case performance of selection sort (seconds)

1.6.2 Generating Test Data

Generating a data set that results in the worst-case performance of a program isn't 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

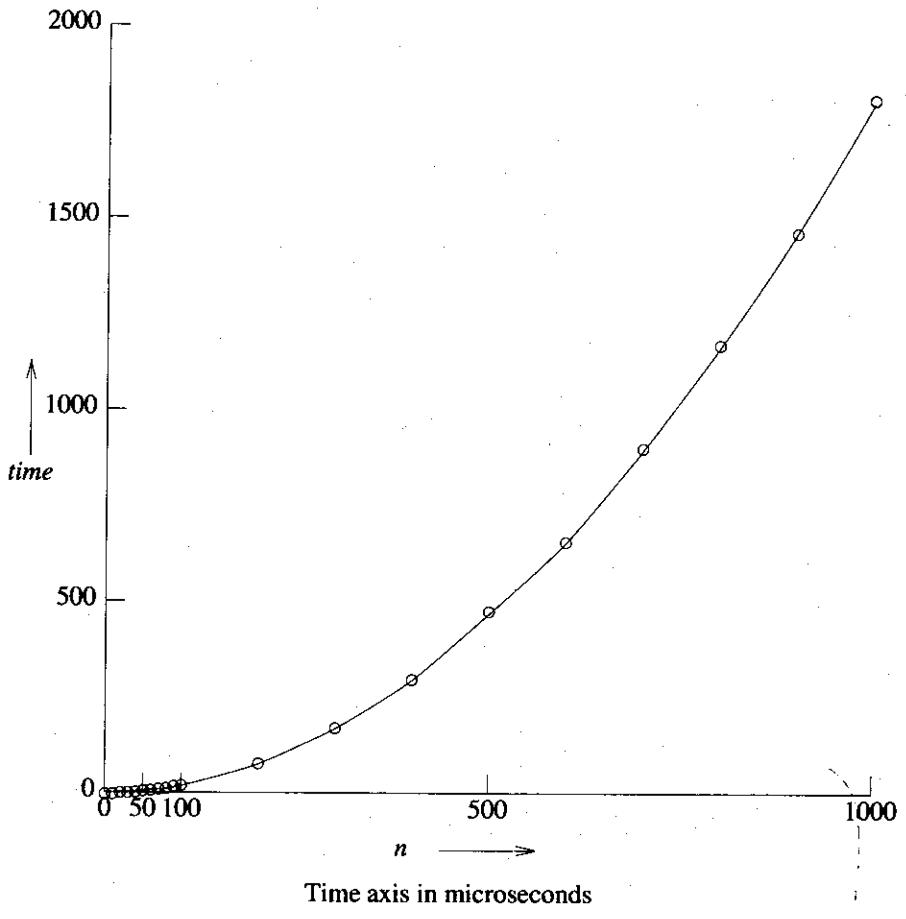

Figure 1.12: Graph of worst-case performance of selection sort

possible instances of a given characteristic. While it is possible to do this for sequential and binary search, it is not possible for a sort program. 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.

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 shall not go into it here.

EXERCISES

Each of the following exercises requires you to create a timing program. You must pick arrays of appropriate sizes and use the proper timing construct. Present you results in table and graph form, and summarize your findings.

1. Repeat the experiment of Example 1.22. This time make sure that all measured times have an accuracy of at least 10%. Times are to be obtained for the same values of *n* as in the example. Plot the measured times as a function of *n*.
2. Compare the worst-case performance of the iterative (Program 1.11) and recursive (Program 1.12) list summing functions.
3. Compare the worst-case performance of the iterative (Program 1.7) and recursive (Program 1.8) binary search functions.
4. (a) Translate the iterative version of sequential search (Program 1.26) into an equivalent recursive function.
   (b) Analyze the worst-case complexity of your function.
   (c) Measure the worst-case performance of the recursive sequential search function and compare with the results we provided for the iterative version.
5. Measure the worst-case performance of *add* (Program 1.16).
6. Measure the worst-case performance of *mult* (Program 1.20).

1.7 SELECTED READINGS AND REFERENCES

A good introduction to programming in C can be found in the text *C: An advanced introduction* by Narain Gehani, Silicon Press, NJ, 1995. *Testing computer software*, 2nd Edition, by C. Kaner, J. Falk, and H. Nguyen, John Wiley, New York, NY, 1999 has a more thorough treatment of software testing and debugging techniques.

The following books provide asymptotic analyses for several programs: Handbook of data structures and applications edited by D. Mehta and S. Sahni, Chapman & Hall/CRC, Boca Raton, 2005, *Fundamentals of Computer Algorithms* by E. Horowitz, S. Sahni, and S. Rajasekaran, W. H. Freeman and Co., New York, NY, 1998; *Introduction to Algorithms*, Second Edition, by T. Cormen, C. Leiserson, and R. Rivest, McGraw-Hill, New York, NY, 2002; and *Compared to What: An Introduction to the Analysis of Algorithms* by G. Rawlins, W. H. Freeman and Co., NY, 1992.