CHAPTER TWO

Analysis of Algorithms

> In computer science there is no history of critical experiments that decide between the validity of various theories, as there are in physical sciences.
>
> Juris Hartmanis [421]

Algorithms are precise procedures that can be turned into computer programs. A classical example is Euclid's algorithm, which specifies the exact steps toward computing the greatest common divisor. Problems such as the greatest common divisor are therefore said to be **computable**, whereas those that do not admit algorithms are **uncomputable**. A computable problem may have complexity so high that no efficient algorithms exist. In this case, it is said to be **intractable**. The difficulty of pricing certain financial instruments may be linked to their intrinsic complexity [169].

The hardest part of software implementation is developing the algorithm [264]. Algorithms in this book are expressed in an informal style called a **pseudocode**. A pseudocode conveys the algorithmic ideas without getting tied up in syntax. Pseudocode programs are specified in sufficient detail as to make their coding in a programming language straightforward. This chapter outlines the conventions used in pseudocode programs.

2.1 Complexity

Precisely predicting the performance of a program is difficult. It depends on such diverse factors as the machine it runs on, the programming language it is written in, the compiler used to generate the binary code, the workload of the computer, and so on. Although the actual running time is the only valid criterion for performance [717], we need measures of complexity that are machine independent in order to have a grip on the expected performance.

We start with a set of basic operations that are assumed to take one unit of time. Logical comparisons ($\leq$, $=$, $\geq$, and so on) and arithmetic operations of finite precision ($+$, $-$, $\times$, $/$, exponentiation, logarithm, and so on) are among them. The total number of these operations is then used as the total work done by an algorithm, called its **computational complexity**. Similarly, the **space complexity** is the amount of memory space used by an algorithm. The purpose here is to concentrate on the abstract complexity of an algorithm instead of its implementation, which involves so many details that we can never fully take them into account. Complexity serves

Algorithm for searching an element:

input: $x, n, A_i\ (1 \le i \le n)$;
integer k;
for ($k = 1$ to n)
    if [$x = A_k$] return k;
return not-found;

Figure 2.1: Sequential search algorithm.

as a good guide to an algorithm's actual running time. Because space complexity is seldom an issue in this book, the term complexity is used to refer exclusively to computational complexity.

The complexity of an algorithm is expressed as a function of the size of its input. Consider the search algorithm in Fig. 2.1. It looks for a given element by comparing it sequentially with every element in an array of length n. Apparently the worst-case complexity is n comparisons, which occurs when the matching element is the last element of the array or when there is no match. There are other operations to be sure. The for loop, for example, uses a loop variable k that has to be incremented for each execution of the loop and compared against the loop bound n. We do not need to count them because we care about the **asymptotic** growth rate, not the exact number of operations; the derivation of the latter can be quite involved, and its effects on real-world performance cannot be pinpointed anyway [37, 227]. The complexity from maintaining the loop is therefore subsumed by the complexity of the body of the loop.

2.2 Analysis of Algorithms

We are interested in worst-case measures. It is true that worst cases may not occur in practice. But an average-case analysis must assume a distribution on the input, whose validity is hard to certify. To further suppress unnecessary details, we are concerned with the rate of growth of the complexity only as the input gets larger, ignoring constant factors and small inputs. The focus is on the asymptotic growth rate, as mentioned in Section 2.1.

Let $\boldsymbol{R}$ denote the set of real numbers, $\boldsymbol{R}^+$ the set of positive real numbers, and $\boldsymbol{N} = \{0, 1, 2, \ldots, \}$. The following definition lays out the notation needed to formulate complexity.

DEFINITION 2.2.1 We say that $g = O(f)$ if $g(n) \le cf(n)$ for some nonnegative c and sufficiently large n, where $f, g : \boldsymbol{N} \to \boldsymbol{R}^+$.

EXAMPLE 2.2.2 The base of a logarithm is not important for asymptotic analysis because

$$\log_a x = \frac{\log_e x}{\log_e a} = O(\log_e x),$$

where $e = 2.71828\ldots$. We abbreviate $\log_e x$ as $\ln x$.

EXAMPLE 2.2.3 Let $f(n) = n^3$ and $g(n) = 3.5 \times n^2 + \ln n + \sin n$. Clearly, $g = O(f)$ because $g(n)$ is less than n^3 for sufficiently large n. On the other hand, $f \ne O(g)$.

Denote the input size by N. An algorithm runs in **logarithmic time** if its complexity is $O(\log N)$. An algorithm runs in **linear time** if its complexity is $O(N)$. The sequential search algorithm in Fig. 2.1, for example, has a complexity of $O(N)$

because it has $N = n+2$ inputs and carries out $O(n)$ operations. A complexity of $O(N \log N)$ typifies sorting and various divide-and-conquer types of algorithms. An algorithm runs in **quadratic time** if its complexity is $O(N^2)$. Many elementary matrix computations such as matrix–vector multiplication have this complexity. An algorithm runs in **cubic time** if its complexity is $O(N^3)$. Typical examples are matrix–matrix multiplication and solving simultaneous linear equations. An algorithm runs in **exponential time** if its complexity is $O(2^N)$. Problems that *require* exponential time are clearly intractable. It is possible for an exponential-time algorithm to perform well on "typical" inputs, however. The foundations for computational complexity were laid in the 1960s [710].

➤ **Exercise 2.2.1** Show that $f + g = O(f)$ if $g = O(f)$.

➤ **Exercise 2.2.2** Prove the following relations: (1) $\sum_{i=1}^{n} i = O(n^2)$, (2) $\sum_{i=1}^{n} i^2 = O(n^3)$, (3) $\sum_{i=0}^{\log_2 n} 2^i = O(n)$, (4) $\sum_{i=0}^{\alpha \log_2 n} 2^i = O(n^\alpha)$, (5) $n \sum_{i=0}^{n} i^{-1} = O(n \ln n)$.

2.3 Description of Algorithms

Universally accepted mathematical symbols are respected. Therefore $+$, $-$, $\times$, $/$, $<$, $>$, $\leq$, $\geq$, and $=$ mean addition, subtraction, and so on. The symbol $:=$ denotes assignment. For example, $a := b$ assigns the value of b to the variable a. The statement `return` a says that a is returned by the algorithm.

The construct

`for` $(i = a$ `to` $b)$ $\{ \cdots \}$

means that the statements enclosed in braces ({ and }) are executed $b - a + 1$ times, with i equal to $a, a+1, \ldots, b$, in that order. The construct

`for` $(i = a$ `down to` $b)$ $\{ \cdots \}$

means the statements enclosed in braces are executed $a - b + 1$ times, with i equal to $a, a-1, \ldots, b$, in that order. The construct

`while` $[\, S \,]$ $\{ \cdots \}$

executes the statements enclosed in braces until the condition S is violated. For example, `while` $[a = b]$ $\{ \cdots \}$ runs until a is not equal to b. The construct

`if` $[\, S \,]$ $\{\, T_1 \,\}$ `else` $\{\, T_2 \,\}$

executes T_1 if the expression S is true and T_2 if the expression S is false. The statement `break` causes the current `for` loop to exit. The enclosing brackets can be dropped if there is only a single statement within.

The construct $a[\, n \,]$ allocates an array of n elements $a[\, 0 \,], \ldots, a[\, n-1 \,]$. The construct $a[\, n \,][\, m \,]$ allocates the following $n \times m$ array (note that the indices start from *zero*, not *one*):

$$\begin{array}{ccc} a[\,0\,][\,0\,] & \cdots & a[\,0\,][\,m-1\,] \\ \vdots & \ddots & \vdots \\ a[\,n-1\,][\,0\,] & \cdots & a[\,n-1\,][\,m-1\,] \end{array} .$$

Although the zero-based indexing scheme is more convenient in many cases, the one-based indexing scheme may be preferred in others. So we use $a[\, 1..n \,][\, 1..m \,]$

to denote an array with the following $n \times m$ elements,

$$a[\,1\,][\,1\,], a[\,1\,][\,2\,], \ldots, a[\,n\,][\,m-1\,], a[\,n\,][\,m\,].$$

Symbols such as $a[\ ]$ and $a[\ ][\ ]$ are used to reference the entire array. Anything following // is treated as comment.

2.4 Software Implementation

Implementation turns an algorithm into a computer program on specific computer platforms. Design, coding, debugging, and module testing are all integral parts of implementation.[1] A key to a productive software project is the reuse of code, either from previous projects or commercial products [650]. The current trend toward object-oriented programming and standardization promises to promote software reuse.

The choice of algorithms in software projects has to be viewed within the context of a larger system. The overall system design might limit the choices to only a few alternatives [791]. This constraint usually arises from the requirements of other parts of the system and very often reflects the fact that most pieces of code are written for an existing system [714].

I now correct a common misconception about the importance of performance. People tend to think that a reduction of the running time from, say, 10 s to 5 s is not as significant as that from 10 min to 5 min. This view rests on the observation that a 5-s difference is not as critical as a 5-min difference. This is wrong. A 5-s difference can be easily turned into a 5-min difference if there are 60 such tasks to perform. A significant reduction in the running time for an important problem is always desirable.

Finally, a word of caution on the term **recursion**. Computer science usually reserves the word for the way of attacking a problem by solving smaller instances of the same problem. Take sorting a list of numbers as an example. One recursive strategy is to sort the first half of the list and the second half of the list separately before merging them. Note that the two sorting subproblems are indeed smaller in size than the original problem. Consistent with most books in finance, however, in this book the term "recursion" is used loosely to mean "iteration." Adhering to the strict computer science usage will usually result in problem formulations that lead to highly inefficient pricing algorithms.

NOTE

1. Software errors can be costly. For example, they were responsible for the crash of the maiden flight of the Ariane 5 that was launched on June 4, 1996, at a cost of half a billion U.S. dollars [606].

> Probably only a person with some mathematical knowledge would think of beginning with 0 instead of with 1.
>
> Bertrand Russell (1872–1970), *Introduction to Mathematical Philosophy*