Algorithm Analysis: The Big-O Notation

Just as a problem is analyzed before writing the algorithm and the computer program, after an algorithm is designed it should also be analyzed. Usually, there are various ways to design a particular algorithm. Certain algorithms take very little computer time to execute, whereas others take a considerable amount of time.

Let us consider the following problem. The holiday season is approaching and a gift shop is expecting sales to be double or even triple the regular amount. They have hired extra delivery people to deliver the packages on time. The company calculates the shortest distance from the shop to a particular destination and hands the route to the driver. Suppose that 50 packages are to be delivered to 50 different houses. The shop, while making the route, finds that the 50 houses are one mile apart and are in the same area. (See Figure 1-1, in which each dot represents a house and the distance between houses is 1 mile.)

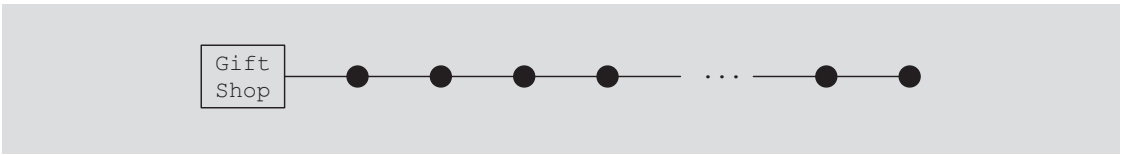

FIGURE 1-1 Gift shop and each dot representing a house

To deliver 50 packages to their destinations, one of the drivers picks up all 50 packages, drives one mile to the first house and delivers the first package. Then he drives another mile and delivers the second package, drives another mile and delivers the third package, and so on. Figure 1-2 illustrates this delivery scheme.

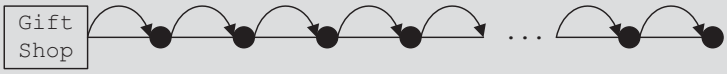

FIGURE 1-2 Package delivering scheme

It now follows that using this scheme, the distance driven by the driver to deliver the packages is:

$$1 + 1 + 1 + ... + 1 = 50 \text{ miles}$$

Therefore, the total distance traveled by the driver to deliver the packages and then getting back to the shop is:

$$50 + 50 = 100 \text{ miles}$$

Another driver has a similar route to deliver another set of 50 packages. The driver looks at the route and delivers the packages as follows: The driver picks up the first package, drives one mile to the first house, delivers the package, and then comes back to the shop. Next, the driver picks up the second package, drives 2 miles, delivers the second package, and then returns to the shop. The driver then picks up the third package, drives 3 miles, delivers the package, and comes back to the shop. Figure 1-3 illustrates this delivery scheme.

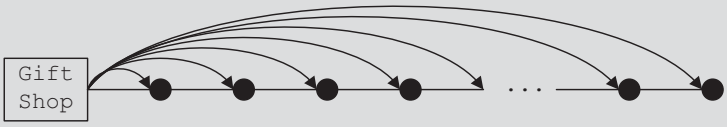

FIGURE 1-3 Another package delivery scheme

The driver delivers only one package at a time. After delivering a package, the driver comes back to the shop to pick up and deliver the second package. Using this scheme, the total distance traveled by this driver to deliver the packages and then getting back to the store is:

$$2 \cdot (1 + 2 + 3 + \ldots + 50) = 2550 \text{ miles}$$

Now suppose that there are n packages to be delivered to n houses, and each house is one mile apart from each other, as shown in Figure 1-1. If the packages are delivered using the first scheme, the following equation gives the total distance traveled:

$$1 + 1 + \ldots + 1 + n = 2n \tag{1–1}$$

If the packages are delivered using the second method, the distance traveled is:

$$2 \cdot (1 + 2 + 3 + \ldots + n) = 2 \cdot (n(n + 1)/2) = n^2 + n \tag{1–2}$$

In Equation (1-1), we say that the distance traveled is a function of n. Let us consider Equation (1-2). In this equation, for large values of n, we will find that the term consisting of n^2 will become the dominant term and the term containing n will be negligible. In this case, we say that the distance traveled is a function of n^2. Table 1-1 evaluates Equations (1-1) and (1-2) for certain values of n. (The table also shows the value of n^2.)

TABLE 1-1 Various values of n, $2n$, n^2, and $n^2 + n$

n	$2n$	n^2	$n^2 + n$
1	2	1	2
10	20	100	110
100	200	10,000	10,100
1000	2000	1,000,000	1,001,000
10,000	20,000	100,000,000	100,010,000

While analyzing a particular algorithm, we usually count the number of operations performed by the algorithm. We focus on the number of operations, not on the actual computer time to execute the algorithm. This is because a particular algorithm can be implemented on a variety of computers and the speed of the computer can affect the execution time. However, the number of operations performed by the algorithm would be the same on each computer. Let us consider the following examples.

EXAMPLE 1-1

Consider the following algorithm. (Assume that all variables are properly declared.)

```
cout << "Enter two numbers";                          //Line 1

cin >> num1 >> num2;                                  //Line 2

if (num1 >= num2)                                     //Line 3
    max = num1;                                       //Line 4
else                                                  //Line 5
    max = num2;                                       //Line 6

cout << "The maximum number is: " << max << endl;     //Line 7
```

Line 1 has one operation, <<; Line 2 has two operations; Line 3 has one operation, >=; Line 4 has one operation, =; Line 6 has one operation; and Line 7 has three operations. Either Line 4 or Line 6 executes. Therefore, the total number of operations executed in the preceding code is $1 + 2 + 1 + 1 + 3 = 8$. In this algorithm, the number of operations executed is fixed.

EXAMPLE 1-2

Consider the following algorithm:

```
cout << "Enter positive integers ending with -1" << endl;  //Line 1

count = 0;                                                  //Line 2
sum = 0;                                                    //Line 3

cin >> num;                                                 //Line 4

while (num != -1)                                           //Line 5
{
    sum = sum + num;                                        //Line 6
    count++;                                                //Line 7
    cin >> num;                                             //Line 8
}

cout << "The sum of the numbers is: " << sum << endl;       //Line 9

if (count != 0)                                             //Line 10
    average = sum / count;                                  //Line 11
else                                                        //Line 12
    average = 0;                                            //Line 13

cout << "The average is: " << average << endl;              //Line 14
```

This algorithm has five operations (Lines 1 through 4) before the `while` loop. Similarly, there are nine or eight operations after the `while` loop, depending on whether Line 11 or Line 13 executes.

Line 5 has one operation, and four operations within the `while` loop (Lines 6 through 8). Thus, Lines 5 through 8 have five operations. If the `while` loop executes 10 times, these five operations execute 10 times. One extra operation is also executed at Line 5 to terminate the loop. Therefore, the number of operations executed is 51 from Lines 5 through 8.

If the `while` loop executes 10 times, the total number of operations executed is:

$10 \cdot 5 + 1 + 5 + 9$ or $10 \cdot 5 + 1 + 5 + 8$

that is,

$10 \cdot 5 + 15$ or $10 \cdot 5 + 14$

We can generalize it to the case when the `while` loop executes n times. If the `while` loop executes n times, the number of operations executed is:

$5n + 15$ or $5n + 14$

In these expressions, for very large values of n, the term $5n$ becomes the dominating term and the terms 15 and 14 become negligible.

Usually, in an algorithm, certain operations are dominant. For example, in the preceding algorithm, to add numbers, the dominant operation is in Line 6. Similarly, in a search algorithm, because the search item is compared with the items in the list, the dominant operations would be comparison, that is, the relational operation. Therefore, in the case of a search algorithm, we count the number of comparisons. For another example, suppose that we write a program to multiply matrices. The multiplication of matrices involves addition and multiplication. Because multiplication takes more computer time to execute, to analyze a matrix multiplication algorithm, we count the number of multiplications.

In addition to developing algorithms, we also provide a reasonable analysis of each algorithm. If there are various algorithms to accomplish a particular task, the algorithm analysis allows the programmer to choose between various options.

Suppose that an algorithm performs $f(n)$ basic operations to accomplish a task, where n is the size of the problem. Suppose that you want to determine whether an item is in a list. Moreover, suppose that the size of the list is n. To determine whether the item is in the list, there are various algorithms, as you will see in Chapter 9. However, the basic method is to compare the item with the items in the list. Therefore, the performance of the algorithm depends on the number of comparisons.

Thus, in the case of a search, n is the size of the list and $f(n)$ becomes the count function, that is, $f(n)$ gives the number of comparisons done by the search algorithm. Suppose that, on a particular computer, it takes c units of computer time to execute one operation. Thus, the computer time it would take to execute $f(n)$ operations is $cf(n)$. Clearly, the constant c depends on the speed of the computer and, therefore, varies from computer to computer. However, $f(n)$, the number of basic operations, is the same on each computer. If we know how the function $f(n)$ grows as the size of the problem grows, we can determine the efficiency of the algorithm. Consider Table 1-2.

TABLE 1-2 Growth rates of various functions

n	$\log_2 n$	$n \log_2 n$	n^2	2^n
1	0	0	1	2
2	1	2	2	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296

Table 1-2 shows how certain functions grow as the parameter n, that is, the problem size, grows. Suppose that the problem size is doubled. From Table 1-2, it follows that if the number of basic operations is a function of $f(n) = n^2$, the number of basic operations is quadrupled. If the number of basic operations is a function of $f(n) = 2^n$, the number of basic operations is squared. However, if the number of operations is a function of $f(n) = \log_2 n$, the change in the number of basic operations is insignificant.

Suppose that a computer can execute 1 billion basic operations per second. Table 1-3 shows the time that the computer takes to execute $f(n)$ basic operations.

TABLE 1-3 Time for $f(n)$ instructions on a computer that executes 1 billion instructions per second

n	$f(n) = n$	$f(n) = \log_2 n$	$f(n) = n\log_2 n$	$f(n) = n^2$	$f(n) = 2^n$
10	0.01μs	0.003μs	0.033μs	0.1μs	1μs
20	0.02μs	0.004μs	0.086μs	0.4μs	1ms
30	0.03μs	0.005μs	0.147μs	0.9μs	1s
40	0.04μs	0.005μs	0.213μs	1.6μs	18.3min
50	0.05μs	0.006μs	0.282μs	2.5μs	13 days
100	0.10μs	0.007μs	0.664μs	10μs	4×10^{13} years
1000	1.00μs	0.010μs	9.966μs	1ms	
10,000	10μs	0.013μs	130μs	100ms	
100,000	0.10ms	0.017μs	1.67ms	10s	
1,000,000	1 ms	0.020μs	19.93ms	16.7m	
10,000,000	0.01s	0.023μs	0.23s	1.16 days	
100,000,000	0.10s	0.027μs	2.66s	115.7 days	

In Table 1-3, $1\mu s = 10^{-6}$ seconds and $1ms = 10^{-3}$ seconds.

Figure 1-4 shows the growth rate of functions in Table 1-3.

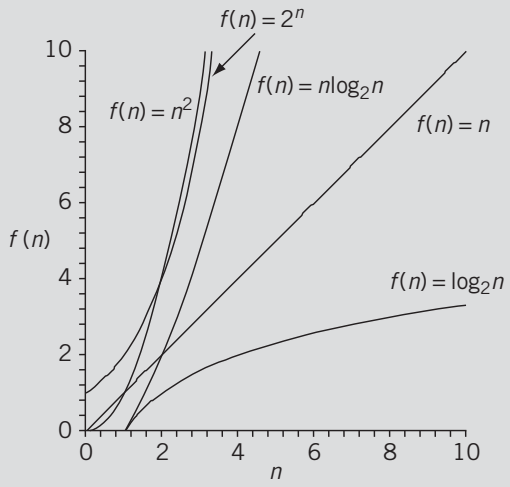

FIGURE 1-4 Growth rate of various functions

The remainder of this section develops a notation that shows how a function $f(n)$ grows as n increases without bound. That is, we develop a notation that is useful in describing the behavior of the algorithm, which gives us the most useful information about the algorithm. First, we define the term *asymptotic*.

Let f be a function of n. By the term **asymptotic**, we mean the study of the function f as n becomes larger and larger without bound.

Consider the functions $g(n) = n^2$ and $f(n) = n^2 + 4n + 20$. Clearly, the function g does not contain any linear term, that is, the coefficient of n in g is zero. Consider Table 1-4.

TABLE 1-4 Growth rate of n^2 and $n^2 + 4n + 20$

n	$g(n) = n^2$	$f(n) = n^2 + 4n + 20$
10	100	160
50	2500	2720
100	10,000	10,420
1000	1,000,000	1,004,020
10,000	100,000,000	100,040,020

Clearly, as n becomes larger and larger the term $4n + 20$ in $f(n)$ becomes insignificant, and the term n^2 becomes the dominant term. For large values of n, we can predict the behavior of $f(n)$ by looking at the behavior of $g(n)$. In algorithm analysis, if the complexity of a function can be described by the complexity of a quadratic function without the linear term, we say that the function is of $O(n^2)$, called Big-O of n^2.

Let f and g be real-valued functions. Assume that f and g are nonnegative, that is, for all real numbers n, $f(n) \geq 0$ and $g(n) \geq 0$.

Definition: We say that $f(n)$ is **Big-O** of $g(n)$, written $f(n) = O(g(n))$, if there exists positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

EXAMPLE 1-3

Let $f(n) = a$, where a is a nonnegative real number and $n \geq 0$. Note that f is a constant function. Now

$f(n) = a \leq a \cdot 1$ for all $n \geq a$.

Let $c = a$, $n_0 = a$, and $g(n) = 1$. Then $f(n) \leq cg(n)$ for all $n \geq n_0$. It now follows that $f(n) = O(g(n)) = O(1)$.

From Example 1-3, it follows that if f is a nonnegative constant function, then f is $O(1)$.

EXAMPLE 1-4

Let $f(n) = 2n + 5$, $n \geq 0$. Note that

$f(n) = 2n + 5 \leq 2n + n = 3n$ for all $n \geq 5$.

Let $c = 3$, $n_0 = 5$, and $g(n) = n$. Then $f(n) \leq cg(n)$ for all $n \geq 5$. It now follows that $f(n) = O(g(n)) = O(n)$.

EXAMPLE 1-5

Let $f(n) = n^2 + 3n + 2$, $g(n) = n^2$, $n \geq 0$. Note that $3n + 2 \leq n^2$ for all $n \geq 4$. This implies that

$f(n) = n^2 + 3n + 2 \leq n^2 + n^2 \leq 2n^2 = 2g(n)$ for all $n \geq 4$.

Let $c = 2$ and $n_0 = 4$. Then $f(n) \leq cg(n)$ for all $n \geq 4$. It now follows that $f(n) = O(g(n)) = O(n^2)$.

In general, we can prove the following theorem. Here we state the theorem without proof.

Theorem: Let $f(n)$ be a nonnegative real-valued function such that

$f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$,

where a_i's are real numbers, $a_m \neq 0$, $n \geq 0$, and m is a nonnegative integer. Then $f(n) = O(n^m)$.

In Example 1-6, we use the preceding theorem to establish the Big-O of certain functions.

EXAMPLE 1-6

In the following, $f(n)$ is a nonnegative real-valued function.

Function	Big-O
$f(n) = an + b$, where a and b are real numbers and a is nonzero.	$f(n) = O(n)$
$f(n) = n^2 + 5n + 1$	$f(n) = O(n^2)$
$f(n) = 4n^6 + 3n^3 + 1$	$f(n) = O(n^6)$
$f(n) = 10n^7 + 23$	$f(n) = O(n^7)$
$f(n) = 6n^{15}$	$f(n) = O(n^{15})$

EXAMPLE 1-7

Suppose that $f(n) = 2\log_2 n + a$, where a is a real number. It can be shown that $f(n) = O(\log_2 n)$.

EXAMPLE 1-8

Consider the following code, where `m` and `n` are `int` variables and their values are nonnegative:

```
for (int i = 0; i < m; i++)          //Line 1
    for (int j = 0; j < n; j++)      //Line 2
        cout << i * j << endl;       //Line 3
```

This code contains nested `for` loops. The outer `for` loop, at Line 1, executes m times. For each iteration of the outer loop, the inner loop, at Line 2, executes n times. For each iteration of the inner loop, the output statement in Line 3 executes. It follows that the total number of iterations of the nested `for` loop is mn. So the number of times the statement in Line 3 executes is mn. Therefore, this algorithm is $O(mn)$. Note that if $m = n$, then this algorithm is $O(n^2)$.

Table 1-5 shows some common Big-O functions that appear in the algorithm analysis. Let $f(n) = O(g(n))$ where n is the problem size.

TABLE 1-5 Some Big-O functions that appear in algorithm analysis

Function $g(n)$	Growth rate of $f(n)$
$g(n) = 1$	The growth rate is constant and so does not depend on n, the size of the problem.
$g(n) = \log_2 n$	The growth rate is a function of $\log_2 n$. Because a logarithm function grows slowly, the growth rate of the function f is also slow.
$g(n) = n$	The growth rate is linear. The growth rate of f is directly proportional to the size of the problem.
$g(n) = n\log_2 n$	The growth rate is faster than the linear algorithm.
$g(n) = n^2$	The growth rate of such functions increases rapidly with the size of the problem. The growth rate is quadrupled when the problem size is doubled.
$g(n) = 2^n$	The growth rate is exponential. The growth rate is squared when the problem size is doubled.

NOTE

It can be shown that

$O(1) \leq O(\log_2 n) \leq O(n) \leq O(n\log_2 n) \leq O(n^2) \leq O(2^n)$.