The analysis of algorithms is a major task in computer science. In order to compare algorithms, we must have some criteria to measure the efficiency of our algorithms. The differences between the algorithms may be immaterial for processing a small number of data items, but these differences grow with the amount of data.

The ***computational complexity*** of an algorithm is the function $f(n)$ which gives the running time and/or storage space requirement of the algorithm in terms of the size n of the input data. The factor of time is usually more important than that of space, so efficiency considerations usually focus on the amount of time elapsed when processing data. For this course and our discussion, the term "complexity" shall refer to the running time of the algorithm.

To evaluate an algorithm's efficiency, real-time units such as microseconds and nanoseconds should not be used. Rather, logical units that express a relationship between the size n of a file or an array and the amount of time t required to process the data should be used. A function expressing the relationship between n and t is usually very complex, and calculating such a function is important only for large bodies of data. Any terms that do not substantially change the function's magnitude should be eliminated from the function. The resulting function gives only an approximate measure of efficiency of the original function. However, this approximation is sufficiently close to the original, especially for a function that processes large quantities of data. This measure of efficiency is called ***asymptotic complexity*** and is used when disregarding certain terms of a function to express the efficiency of an algorithm, or when calculating a function is difficult or impossible, and only approximations can be found. Consider the following example:

$$f(n) = n^2 + 100n + \log_{10}n + 1000$$

For small values of n, the last term, 1000, is the largest. When n equals 10, the second ($100n$) and last (1000) terms are on equal footing with the other terms, making a small contribution to the function value. When n reaches the value of 100, the first and the second terms make the same contribution to the result. But when n becomes larger than 100, the contribution of the second term becomes less significant. Hence, for large values of n, the value of the function f depends mainly on the value of the first term, as Figure 2.1 demonstrates. Other terms can be disregarded for large n.

FIGURE 2.1 The growth rate of all terms of function $f(n) = n^2 + 100n + \log_{10}n + 1{,}000$.

n	f(n)	n^2		100n		$\log_{10}n$		1,000	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

The above discussion leads us to the question of finding the complexity function $f(n)$ for certain cases. The two cases usually investigated in complexity theory are:

- **Worst case:** the maximum value of $f(n)$ for any possible input
- **Average case:** the expected value of $f(n)$

Sometimes we also consider the minimum value of $f(n)$, called the **best case**.

The analysis of average case assumes that all possible permutations of an input data set are equally likely. Also suppose the numbers n_1, n_2, …, n_k occur with respective probabilities p_1, p_2, …, p_k. Then the expectation or average value E is given by

$$E = n_1p_1 + n_2p_2 + \ldots + n_kp_k$$

The complexity of the average case of an algorithm is usually much more complicated to analyze than that of the worst case. Also, the assumption for the average case may not actually apply to real situations. So for our discussion, the complexity of an algorithm shall mean the function which gives the running time of the worst case in terms of the input size. This is a fair assumption, since the complexity of the average case for many algorithms is proportional to the worst case.

Rate of Growth

Big-O Notation

The most commonly used notation for specifying asymptotic complexity—that is, for estimating the rate of function growth—is the big-O notation introduced in 1894 by Paul Bachmann.

$f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that $f(n) \leq cg(n)$ for all $n \geq N$.

This definition reads: f is big-O of g if there is a positive number c such that f is not larger than cg for sufficiently large ns; that is, for all ns larger than some number N. The relationship between f and g can be expressed by stating either that $g(n)$ is an upper bound on the value of $f(n)$ or that, in the long run, f grows at most as fast as g.

The problem with this definition is that, first, it states only that there must exist certain c and N, but it does not give any hint of how to calculate these constants. Second, it does not put any restrictions on these values and gives little guidance in situations when there are many candidates. In fact, there are usually infinitely many pairs of cs and Ns that can be given for the same pair of functions f and g. For example, for $f(n) = 2n^2 + 3n + 1 = O(n^2)$, where $g(n) = n^2$, candidate values for c and N are shown in figure.

Different values of c and N for function $f(n) = 2n^2 + 3n + 1 = O(n^2)$ calculated according to the definition of big-O.

c	≥ 6	$\geq 3\frac{3}{4}$	$\geq 3\frac{1}{9}$	$\geq 2\frac{13}{16}$	$\geq 2\frac{16}{25}$	…	→	2
N	1	2	3	4	5	…	→	∞

We obtain these values by solving the inequality: $2n^2 + 3n + 1 \leq cn^2$ or equivalently

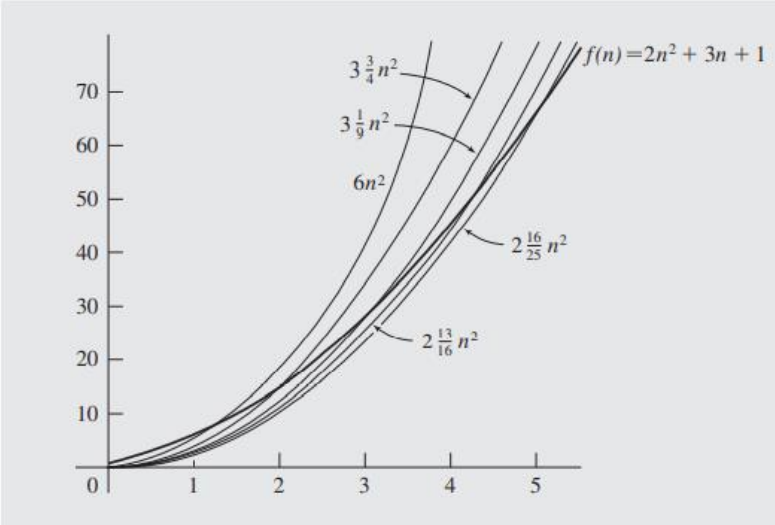
$$2 + \frac{3}{n} + \frac{1}{n^2} \leq c$$

for different *n*s. The first inequality results in substituting the quadratic function from the equation for $f(n)$ in the definition of the big-O notation and n^2 for $g(n)$. Because it is one inequality with two unknowns, different pairs of constants c and N for the same function $g(= n^2)$ can be determined. To choose the best c and N, it should be determined for which N a certain term in f becomes the largest and stays the largest. In the equation, the only candidates for the largest term are $2n^2$ and $3n$; these terms can be compared using the inequality $2n^2 > 3n$ that holds for $n > 1.5$. Thus, $N = 2$ and $c \geq 3\frac{3}{4}$, as figure indicates.

What is the practical significance of the pairs of constants just listed? All of them are related to the same function $g(n) = n^2$ and to the same $f(n)$. For a fixed g, an infinite number of pairs of cs and Ns can be identified. The point is that f and g grow at the same rate. The definition states, however, that g is almost always greater than or equal to f if it is multiplied by a constant c. "Almost always" means for all ns not less than a constant N.

The crux of the matter is that the value of c depends on which N is chosen, and vice versa. For example, if 1 is chosen as the value of N—that is, if g is multiplied by c so that $cg(n)$ will not be less than f right away—then c has to be equal to 6 or greater. If $cg(n)$is greater than or equal to $f(n)$ starting from $n = 2$, then it is enough that c is equal to 3.75. The constant c has to be at least $3\frac{1}{9}$ if $cg(n)$ is not less than $f(n)$ starting from $n = 3$. Figure below shows the graphs of the functions f and g. The function g is plotted with different coefficients c. Also, N is always a point where the functions $cg(n)$ and f intersect each other.

Comparison of functions for different values of c and N



Properties of Big-O Notation

Big-O notation has some helpful properties that can be used when estimating the efficiency of algorithms.

Fact 1. (transitivity) If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

(This can be rephrased as $O(O(g(n)))$ is $O(g(n))$.)

Fact 2. If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n) + g(n)$ is $O(h(n))$.

Fact 3. The function an^k is $O(n^k)$.

Fact 4. The function n^k is $O(n^{k+j})$ for any positive j.

Fact 5. If $f(n) = cg(n)$, then $f(n)$ is $O(g(n))$.

Fact 6. The function $\log_a n$ is $O(\log_b n)$ for any positive numbers a and $b \neq 1$.

Fact 7. $\log_a n$ is $O(\lg n)$ for any positive $a \neq 1$, where $\lg n = \log_2 n$.

Ω Notation

Big-O notation refers to the upper bounds of functions. There is a symmetrical definition for a lower bound in the definition of big-Ω:

The function f(n) is $\Omega(g(n))$ if there exist positive numbers c and N such that $f(n) \geq cg(n)$ for all $n \geq N$.

This definition reads: f is Ω (big-omega) of g if there is a positive number c such that f is at least equal to cg for almost all ns. In other words, $cg(n)$ is a lower bound on the size of $f(n)$, or, in the long run, f grows at least at the rate of g.

The only difference between this definition and the definition of big-O notation is the direction of the inequality; one definition can be turned into the other by replacing "≥" with "≤." There is an interconnection between these two notations expressed by the equivalence

$f(n)$ is $\Omega(g(n))$ iff $g(n)$ is $O(f(n))$

Ω notation suffers from the same profusion problem as big-O notation: There is an unlimited number of choices for the constants c and N.

Θ Notation

Note that there is a common ground for big-O and Ω notations indicated by the equalities in the definitions of these notations: Big-O is defined in terms of "≤" and Ω in terms of "≥"; "=" is included in both inequalities. This suggests a way of restricting the sets of possible lower and upper bounds. This restriction can be accomplished by the following definition of Θ (theta) notation:

$f(n)$ is $\Theta(g(n))$ if there exist positive numbers c_1 , c_2 , and N such that $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq N$.

This definition reads: f has an order of magnitude g, f is on the order of g, or both functions grow at the same rate in the long run. We see that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

The only function just listed that is both big-O and Ω of the function in previous example is n^2. However, it is not the only choice, and there are still an infinite number of choices, because the functions $2n^2$, $3n^2$, $4n^2$,. . . are also Θ of that function. But it is rather obvious that the simplest, n^2, will be chosen. When applying any of these notations (big-O, Ω, and Θ), do not forget that they are approximations that hide some detail that in many cases may be considered important.