3.3 Asymptotic Analysis

In algorithm analysis, we focus on the growth rate of the running time as a function of the input size n, taking a "big-picture" approach. For example, it is often enough just to know that the running time of an algorithm ***grows proportionally to*** n.

We analyze algorithms using a mathematical notation for functions that disregards constant factors. Namely, we characterize the running times of algorithms by using functions that map the size of the input, n, to values that correspond to the main factor that determines the growth rate in terms of n. This approach reflects that each basic step in a pseudo-code description or a high-level language implementation may correspond to a small number of primitive operations. Thus, we can perform an analysis of an algorithm by estimating the number of primitive operations executed up to a constant factor, rather than getting bogged down in language-specific or hardware-specific analysis of the exact number of operations that execute on the computer.

As a tangible example, we revisit the goal of finding the largest element of a Python list; we first used this example when introducing for loops on page 21 of Section 1.4.2. Code Fragment 3.1 presents a function named find_max for this task.

```
def find_max(data):
  """Return the maximum element from a nonempty Python list."""
  biggest = data[0]                     # The initial value to beat
  for val in data:                      # For each value:
    if val > biggest                    # if it is greater than the best so far,
      biggest = val                     # we have found a new best (so far)
  return biggest                        # When loop ends, biggest is the max
```

Code Fragment 3.1: A function that returns the maximum value of a Python list.

This is a classic example of an algorithm with a running time that grows proportional to n, as the loop executes once for each data element, with some fixed number of primitive operations executing for each pass. In the remainder of this section, we provide a framework to formalize this claim.

3.3.1 The "Big-Oh" Notation

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq cg(n), \quad \text{for} \quad n \geq n_0.$$

This definition is often referred to as the "big-Oh" notation, for it is sometimes pronounced as "$f(n)$ is ***big-Oh*** of $g(n)$." Figure 3.5 illustrates the general definition.

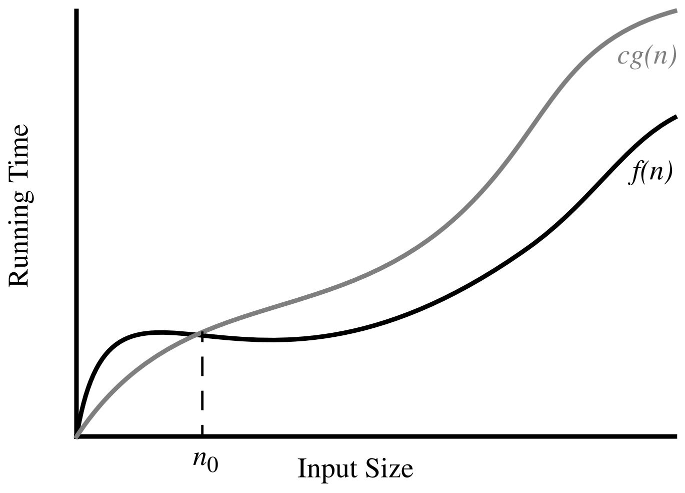

Figure 3.5: Illustrating the "big-Oh" notation. The function $f(n)$ is $O(g(n))$, since $f(n) \le c \cdot g(n)$ when $n \ge n_0$.

Example 3.6: *The function* $8n+5$ *is* $O(n)$.

Justification: By the big-Oh definition, we need to find a real constant $c > 0$ and an integer constant $n_0 \ge 1$ such that $8n+5 \le cn$ for every integer $n \ge n_0$. It is easy to see that a possible choice is $c = 9$ and $n_0 = 5$. Indeed, this is one of infinitely many choices available because there is a trade-off between c and n_0. For example, we could rely on constants $c = 13$ and $n_0 = 1$. ■

The big-Oh notation allows us to say that a function $f(n)$ is "less than or equal to" another function $g(n)$ up to a constant factor and in the ***asymptotic*** sense as n grows toward infinity. This ability comes from the fact that the definition uses "$\le$" to compare $f(n)$ to a $g(n)$ times a constant, c, for the asymptotic cases when $n \ge n_0$. However, it is considered poor taste to say "$f(n) \le O(g(n))$," since the big-Oh already denotes the "less-than-or-equal-to" concept. Likewise, although common, it is not fully correct to say "$f(n) = O(g(n))$," with the usual understanding of the "$=$" relation, because there is no way to make sense of the symmetric statement, "$O(g(n)) = f(n)$." It is best to say,

"$f(n)$ ***is*** $O(g(n))$."

Alternatively, we can say "$f(n)$ is ***order of*** $g(n)$." For the more mathematically inclined, it is also correct to say, "$f(n) \in O(g(n))$," for the big-Oh notation, technically speaking, denotes a whole collection of functions. In this book, we will stick to presenting big-Oh statements as "$f(n)$ ***is*** $O(g(n))$." Even with this interpretation, there is considerable freedom in how we can use arithmetic operations with the big-Oh notation, and with this freedom comes a certain amount of responsibility.

Characterizing Running Times Using the Big-Oh Notation

The big-Oh notation is used widely to characterize running times and space bounds in terms of some parameter n, which varies from problem to problem, but is always defined as a chosen measure of the "size" of the problem. For example, if we are interested in finding the largest element in a sequence, as with the find_max algorithm, we should let n denote the number of elements in that collection. Using the big-Oh notation, we can write the following mathematically precise statement on the running time of algorithm find_max (Code Fragment 3.1) for ***any*** computer.

Proposition 3.7: *The algorithm, find_max, for computing the maximum element of a list of n numbers, runs in $O(n)$ time.*

Justification: The initialization before the loop begins requires only a constant number of primitive operations. Each iteration of the loop also requires only a constant number of primitive operations, and the loop executes n times. Therefore, we account for the number of primitive operations being $c' + c'' \cdot n$ for appropriate constants c' and c'' that reflect, respectively, the work performed during initialization and the loop body. Because each primitive operation runs in constant time, we have that the running time of algorithm find_max on an input of size n is at most a constant times n; that is, we conclude that the running time of algorithm find_max is $O(n)$. ■

Some Properties of the Big-Oh Notation

The big-Oh notation allows us to ignore constant factors and lower-order terms and focus on the main components of a function that affect its growth.

Example 3.8: $5n^4 + 3n^3 + 2n^2 + 4n + 1$ *is* $O(n^4)$.

Justification: *Note that* $5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5+3+2+4+1)n^4 = cn^4$, *for* $c = 15$, *when* $n \geq n_0 = 1$. ■

In fact, we can characterize the growth rate of any polynomial function.

Proposition 3.9: *If $f(n)$ is a polynomial of degree d, that is,*

$$f(n) = a_0 + a_1 n + \cdots + a_d n^d,$$

and $a_d > 0$, then $f(n)$ is $O(n^d)$.

Justification: Note that, for $n \geq 1$, we have $1 \leq n \leq n^2 \leq \cdots \leq n^d$; hence,

$$a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d \leq (|a_0| + |a_1| + |a_2| + \cdots + |a_d|)\, n^d.$$

We show that $f(n)$ is $O(n^d)$ by defining $c = |a_0| + |a_1| + \cdots + |a_d|$ and $n_0 = 1$. ■

Thus, the highest-degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial. We consider some additional properties of the big-Oh notation in the exercises. Let us consider some further examples here, focusing on combinations of the seven fundamental functions used in algorithm design. We rely on the mathematical fact that $\log n \le n$ for $n \ge 1$.

Example 3.10: $5n^2 + 3n\log n + 2n + 5$ *is* $O(n^2)$.

Justification: $5n^2+3n\log n+2n+5 \le (5+3+2+5)n^2 = cn^2$, for $c = 15$, when $n \ge n_0 = 1$. ■

Example 3.11: $20n^3 + 10n\log n + 5$ *is* $O(n^3)$.

Justification: $20n^3 + 10n\log n + 5 \le 35n^3$, for $n \ge 1$. ■

Example 3.12: $3\log n + 2$ *is* $O(\log n)$.

Justification: $3\log n + 2 \le 5\log n$, for $n \ge 2$. Note that $\log n$ is zero for $n = 1$. That is why we use $n \ge n_0 = 2$ in this case. ■

Example 3.13: 2^{n+2} *is* $O(2^n)$.

Justification: $2^{n+2} = 2^n \cdot 2^2 = 4 \cdot 2^n$; hence, we can take $c = 4$ and $n_0 = 1$ in this case. ■

Example 3.14: $2n + 100\log n$ *is* $O(n)$.

Justification: $2n + 100\log n \le 102n$, for $n \ge n_0 = 1$; hence, we can take $c = 102$ in this case. ■

Characterizing Functions in Simplest Terms

In general, we should use the big-Oh notation to characterize a function as closely as possible. While it is true that the function $f(n) = 4n^3 + 3n^2$ is $O(n^5)$ or even $O(n^4)$, it is more accurate to say that $f(n)$ is $O(n^3)$. Consider, by way of analogy, a scenario where a hungry traveler driving along a long country road happens upon a local farmer walking home from a market. If the traveler asks the farmer how much longer he must drive before he can find some food, it may be truthful for the farmer to say, "certainly no longer than 12 hours," but it is much more accurate (and helpful) for him to say, "you can find a market just a few minutes drive up this road." Thus, even with the big-Oh notation, we should strive as much as possible to tell the whole truth.

It is also considered poor taste to include constant factors and lower-order terms in the big-Oh notation. For example, it is not fashionable to say that the function $2n^2$ is $O(4n^2 + 6n\log n)$, although this is completely correct. We should strive instead to describe the function in the big-Oh in ***simplest terms***.

The seven functions listed in Section 3.2 are the most common functions used in conjunction with the big-Oh notation to characterize the running times and space usage of algorithms. Indeed, we typically use the names of these functions to refer to the running times of the algorithms they characterize. So, for example, we would say that an algorithm that runs in worst-case time $4n^2 + n\log n$ is a ***quadratic-time*** algorithm, since it runs in $O(n^2)$ time. Likewise, an algorithm running in time at most $5n + 20\log n + 4$ would be called a ***linear-time*** algorithm.

Big-Omega

Just as the big-Oh notation provides an asymptotic way of saying that a function is "less than or equal to" another function, the following notations provide an asymptotic way of saying that a function grows at a rate that is "greater than or equal to" that of another.

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $\Omega(g(n))$, pronounced "$f(n)$ is big-Omega of $g(n)$," if $g(n)$ is $O(f(n))$, that is, there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \geq cg(n), \quad \text{for} \quad n \geq n_0.$$

This definition allows us to say asymptotically that one function is greater than or equal to another, up to a constant factor.

Example 3.15: $3n\log n - 2n$ *is* $\Omega(n\log n)$.

Justification: $3n\log n - 2n = n\log n + 2n(\log n - 1) \geq n\log n$ for $n \geq 2$; hence, we can take $c = 1$ and $n_0 = 2$ in this case. ■

Big-Theta

In addition, there is a notation that allows us to say that two functions grow at the same rate, up to constant factors. We say that $f(n)$ is $\Theta(g(n))$, pronounced "$f(n)$ is big-Theta of $g(n)$," if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$, that is, there are real constants $c' > 0$ and $c'' > 0$, and an integer constant $n_0 \geq 1$ such that

$$c'g(n) \leq f(n) \leq c''g(n), \quad \text{for} \quad n \geq n_0.$$

Example 3.16: $3n\log n + 4n + 5\log n$ *is* $\Theta(n\log n)$.

Justification: $3n\log n \leq 3n\log n + 4n + 5\log n \leq (3+4+5)n\log n$ for $n \geq 2$. ■

3.3.2 Comparative Analysis

Suppose two algorithms solving the same problem are available: an algorithm A, which has a running time of $O(n)$, and an algorithm B, which has a running time of $O(n^2)$. Which algorithm is better? We know that n is $O(n^2)$, which implies that algorithm A is ***asymptotically better*** than algorithm B, although for a small value of n, B may have a lower running time than A.

We can use the big-Oh notation to order classes of functions by asymptotic growth rate. Our seven functions are ordered by increasing growth rate in the following sequence, that is, if a function $f(n)$ precedes a function $g(n)$ in the sequence, then $f(n)$ is $O(g(n))$:

$$1, \quad \log n, \quad n, \quad n\log n, \quad n^2, \quad n^3, \quad 2^n.$$

We illustrate the growth rates of the seven functions in Table 3.2. (See also Figure 3.4 from Section 3.2.1.)

n	$\log n$	n	$n\log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Table 3.2: Selected values of fundamental functions in algorithm analysis.

We further illustrate the importance of the asymptotic viewpoint in Table 3.3. This table explores the maximum size allowed for an input instance that is processed by an algorithm in 1 second, 1 minute, and 1 hour. It shows the importance of good algorithm design, because an asymptotically slow algorithm is beaten in the long run by an asymptotically faster algorithm, even if the constant factor for the asymptotically faster algorithm is worse.

Running	Maximum Problem Size (n)		
Time (μs)	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

Table 3.3: Maximum size of a problem that can be solved in 1 second, 1 minute, and 1 hour, for various running times measured in microseconds.

The importance of good algorithm design goes beyond just what can be solved effectively on a given computer, however. As shown in Table 3.4, even if we achieve a dramatic speedup in hardware, we still cannot overcome the handicap of an asymptotically slow algorithm. This table shows the new maximum problem size achievable for any fixed amount of time, assuming algorithms with the given running times are now run on a computer 256 times faster than the previous one.

Running Time	New Maximum Problem Size
$400n$	$256m$
$2n^2$	$16m$
2^n	$m+8$

Table 3.4: Increase in the maximum size of a problem that can be solved in a fixed amount of time, by using a computer that is 256 times faster than the previous one. Each entry is a function of m, the previous maximum problem size.

Some Words of Caution

A few words of caution about asymptotic notation are in order at this point. First, note that the use of the big-Oh and related notations can be somewhat misleading should the constant factors they "hide" be very large. For example, while it is true that the function $10^{100}n$ is $O(n)$, if this is the running time of an algorithm being compared to one whose running time is $10n\log n$, we should prefer the $O(n\log n)$-time algorithm, even though the linear-time algorithm is asymptotically faster. This preference is because the constant factor, 10^{100}, which is called "one googol," is believed by many astronomers to be an upper bound on the number of atoms in the observable universe. So we are unlikely to ever have a real-world problem that has this number as its input size. Thus, even when using the big-Oh notation, we should at least be somewhat mindful of the constant factors and lower-order terms we are "hiding."

The observation above raises the issue of what constitutes a "fast" algorithm. Generally speaking, any algorithm running in $O(n\log n)$ time (with a reasonable constant factor) should be considered efficient. Even an $O(n^2)$-time function may be fast enough in some contexts, that is, when n is small. But an algorithm running in $O(2^n)$ time should almost never be considered efficient.

Exponential Running Times

There is a famous story about the inventor of the game of chess. He asked only that his king pay him 1 grain of rice for the first square on the board, 2 grains for the second, 4 grains for the third, 8 for the fourth, and so on. It is an interesting test of programming skills to write a program to compute exactly the number of grains of rice the king would have to pay.

If we must draw a line between efficient and inefficient algorithms, therefore, it is natural to make this distinction be that between those algorithms running in polynomial time and those running in exponential time. That is, make the distinction between algorithms with a running time that is $O(n^c)$, for some constant $c > 1$, and those with a running time that is $O(b^n)$, for some constant $b > 1$. Like so many notions we have discussed in this section, this too should be taken with a "grain of salt," for an algorithm running in $O(n^{100})$ time should probably not be considered "efficient." Even so, the distinction between polynomial-time and exponential-time algorithms is considered a robust measure of tractability.