

Figure 2.8 shows that an $O(\log n)$ algorithm is inherently better than an $O(n)$ algorithm. Regardless of constant factors and slower-growing terms, the $O(\log n)$ algorithm will eventually overtake the $O(n)$ algorithm as n grows.

We have now introduced the ***O*-notation** for algorithm complexity. The notation $O(X)$ stands for ‘of order X ’, and means that the algorithm’s time (or space) requirement grows proportionately to X . X characterizes the algorithm’s growth rate, neglecting slower-growing terms and constant factors. In general, X depends on the algorithm’s input data.

Table 2.9 summarizes the most common time complexities. These are common enough to have acquired verbal descriptions: for example, we say that the simple power algorithm is a *linear-time* algorithm, while the smart power algorithm is a *log-time* algorithm. The complexities are arranged in order of growth rate: $O(1)$ is the slowest-growing, and $O(2^n)$ is the fastest-growing.

It is important to develop our intuitions about what these complexities tell us, in practical terms, about the time efficiency of algorithms. Table 2.10 shows some numerical information, and Figure 2.11 shows the same information graphically. As we have already noted, $\log n$ grows more gradually than n , so an $O(\log n)$ algorithm is better than an $O(n)$ algorithm that solves the same problem. Of course, the constant 1 does not grow at all, so an $O(1)$ or constant-time algorithm is best of all – if we can find one!

Now study the growth rate of $n \log n$. This grows more steeply than n , so an $O(n)$ algorithm is better than an $O(n \log n)$ algorithm that solves the same problem.

Now study the growth rates of n^2 and n^3 . These grow ever more steeply, so an $O(n \log n)$ algorithm is better than an $O(n^2)$ algorithm, which in turn is better than an $O(n^3)$ algorithm, that solves the same problem. One way to look at these algorithms is this: every time n is doubled, $O(n^2)$ is multiplied by four and $O(n^3)$ is multiplied by eight. And every time n is multiplied by 10, $O(n^2)$ is multiplied by 100 and $O(n^3)$ is multiplied by 1000! These numbers are discouraging. If the best algorithm we can find is $O(n^2)$ or $O(n^3)$, we have to accept that the algorithm will rapidly slow down as n increases. Such an algorithm is often too slow to be of practical use.

While n^2 and n^3 grow steeply, 2^n grows at a stupendous rate. Every time n is incremented by 10, $O(2^n)$ is multiplied by over 1000! As n is increased from 10 to 20 to 30 to 40, 2^n grows from a thousand to a million to a billion to a trillion. If the algorithm performs 2^n operations at the rate of a million per second, its time requirement will grow from a millisecond to a second to over 16 minutes to over 11 days! Such an

Table 2.9 Summary of common time complexities.

Complexity	Verbal description	Feasibility
$O(1)$	constant time	feasible
$O(\log n)$	log time	feasible
$O(n)$	linear time	feasible
$O(n \log n)$	log linear time	feasible
$O(n^2)$	quadratic time	sometimes feasible
$O(n^3)$	cubic time	less often feasible
$O(2^n)$	exponential time	rarely feasible