number of words—our measure of the size of an instance is off *only by a multiplicative constant*. So an algorithm that performs a computation using integers stored using 64 bits may take twice as long as a similar algorithm coded using integers stored in 32 bits.

Algorithmic researchers accept that they are unable to compute with pinpoint accuracy the costs involved in using a particular encoding in an implementation. Therefore, they assert that performance costs that differ by a multiplicative constant are *asymptotically equivalent*, or in other words, will not matter as the problem size continues to grow. As an example, we can expect 64-bit integers to require more processing time than 32-bit integers, but we should be able to ignore that and assume that a good algorithm for a million 32-bit integers will also be good for a million 64-bit integers. Although such a definition would be impractical for real-world situations (who would be satisfied to learn they must pay a bill that is 1,000 times greater than expected?), it serves as the universal means by which algorithms are compared.

For all algorithms in this book, the constants are small for virtually all platforms. However, when implementing an algorithm in production code, you must pay attention to the details reflected by the constants. This asymptotic approach is useful since it can predict the performance of an algorithm on a large problem instance based on the performance on small problem instances. It helps determine the largest problem instance that can be handled by a particular algorithm implementation (Bentley, 1999).

To store collections of information, most programming languages support *arrays*, contiguous regions of memory indexed by an integer i to enable rapid access to the i^{th} element. An array is one-dimensional when each element fits into a word in the platform (e.g., an array of integers or Boolean values). Some arrays extend into multiple dimensions, enabling more complex data representations.

Rate of Growth of Functions

We describe the behavior of an algorithm by representing the *rate of growth of its execution time* as a function of the size of the input problem instance. Characterizing an algorithm's performance in this way is a common abstraction that ignores numerous details. To use this measure properly requires an awareness of the details hidden by the abstraction. Every program is run on a computing platform, which is a general term meant to encompass:

- The computer on which the program is run, its CPU, data cache, floating-point unit (FPU), and other on-chip features
- The programming language in which the program is written, along with the compiler/interpreter and optimization settings for generated code
- The operating system
- Other processes being run in the background

We assume that changing the platform will change the execution time of the program by a constant factor, and that we can therefore ignore platform differences in conformance with the asymptotically equivalent principle described earlier.

To place this discussion in context, we briefly discuss the **Sequential Search** algorithm, presented later in Chapter 5. **Sequential Search** examines a list of $n \geq 1$ distinct elements, one at a time, until a desired value, v, is found. For now, assume that:

- There are n distinct elements in the list
- The list contains the desired value v
- Each element in the list is equally likely to be the desired value v

To understand the performance of **Sequential Search**, we must know how many elements it examines "on average." Since v is known to be in the list and each element is equally likely to be v, the average number of examined elements, $E(n)$, is the sum of the number of elements examined for each of the n values divided by n. Mathematically:

$$E(n) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n(n+1)}{2n} = \frac{1}{2}n + \frac{1}{2}$$

Thus, **Sequential Search** examines about half of the elements in a list of n distinct elements subject to these assumptions. If the number of elements in the list doubles, then **Sequential Search** should examine about twice as many elements; the expected number of probes is a *linear* function of n. That is, the expected number of probes is "about" $c*n$ for some constant c; here, $c = 0.5$. A fundamental fact of performance analysis is that the constant c is unimportant in the long run, because the most important cost factor is the size of the problem instance, n. As n gets larger and larger, the error in claiming that:

$$\frac{1}{2}n \approx \frac{1}{2}n + \frac{1}{2}$$

becomes less significant. In fact, the ratio between the two sides of this approximation approaches 1. That is:

$$\lim_{n \to \infty} \frac{\left(\frac{1}{2}n\right)}{\left(\frac{1}{2}n + \frac{1}{2}\right)} = 1$$

although the error in the estimation is significant for small values of n. In this context, we say the rate of growth of the expected number of elements that **Sequential Search** examines is linear. That is, we ignore the constant multiplier and are concerned only when the size of a problem instance is large.