Exercise Show that $n(n-1)/2 \in \Theta(n^2)$.

Sometimes we wish to indicate that one function has strictly smaller asymptotic growth than another. We can use the following definition.

Definition $o(f)$ is the set of functions $g:\mathbf{N}\rightarrow\mathbf{R}^*$ such that $\lim_{n\rightarrow\infty} g(n)/f(n) = 0$

1.4.2 How Important Is Order?

Table 1.1[4] shows the running times for several actual algorithms for the same problem. (The last column does not correspond to an algorithm for the problem; it is included to demonstrate how fast exponential functions grow, and hence how bad exponential algorithms are.) Look over the entries in the table to see how fast the running time increases with input size for the algorithms of higher complexities. One of the important lessons in the table is that the high constant factors on the $\Theta(n)$ and $\Theta(n\lg n)$ algorithms do not make them slower than the other algorithms except for very small inputs.

The second part of the table looks at the effect of asymptotic growth on the increase in the size of the input that can be handled with more computer time (or by using a faster computer). It is *not* true in general that if we multiply the time (or speed) by 60 we can handle an input 60 times as large; that is true only for algorithms whose complexity is in $O(n)$. The $\Theta(n^2)$ algorithm, for example, can handle an input only $\sqrt{60}$ times as large.

To further drive home the point that the order of the running time of an algorithm is more important than a constant factor (for large inputs), look at Table 1.2.[5]

Table 1.1
How functions grow.

	Algorithm	1	2	3	4	
Time (in μsecs.)		$33n$	$46n\lg n$	$13n^2$	$3.4n^3$	2^n
Time to solve for input size	n=10	.00033 sec.	.0015 sec.	.0013 sec.	.0034 sec.	.001 sec.
	n=100	.003 sec.	.03 sec.	.13 sec.	3.4 sec.	4×10^{14}
	n=1000	.033 sec.	.45 sec.	13 sec.	.94 hrs.	centuries
	n=10,000	.33 sec.	6.1 sec.	22 min.	39 days	
	n=100,000	3.3sec.	1.3 min.	1.5 days	108 years	
Approx. maximum input size in	1 sec.	30,000	2000	280	67	20
	1 min.	1,800,000	82,000	2200	260	26

[4] This table (except the last column) is adapted from *Programming Pearls* by Jon Bentley (Addison-Wesley, Reading, Mass., 1986) and is reproduced here with permission.

[5] This table is also from *Programming Pearls* by Jon Bentley and is reproduced here with permission.

Table 1.2
Order wins out.

n	Cray-1 Fortran $3n^3$ nanosec.	TRS-80 Basic $19{,}500{,}000n$ nanosec.
10	3 microsec.	200 millisec.
100	3 millisec.	2 sec.
1000	3 sec.	20 sec.
2500	50 sec.	50 sec.
10,000	49 min.	3.2 min.
1,000,000	95 years	5.4 hours

Cray-1 is a trademark of Cray Research, Inc.
TRS-80 is a trademark of Tandy Corporation.

A program for the cubic algorithm from Table 1.1 was written for the Cray-1 supercomputer; it ran in $3n^3$ nanoseconds for input of size n. The linear algorithm was programmed on a TRS-80; it ran in $19.5n$ milliseconds (which is $19{,}500{,}000n$ nanoseconds). Even though the constant on the linear algorithm is 6.5 million times as big as the constant on the cubic algorithm, the linear algorithm is faster for input sizes $n \geq 2500$. (Whether one considers this a large or small input size would depend on the context of the problem.)

If we focus on the order of functions (thus including, say, n and $1{,}000{,}000n$ in the same class), then when we can show that two functions are *not* of the same order, we are making a strong statement about the difference between the algorithms described by those functions. If two functions *are* of the same order, they may differ by a large constant factor. The constant, though, is irrelevant to the effects of improved computer speed on the maximum input size an algorithm can handle in a given amount of time. That is, the constant is irrelevant to the increase between the last two rows of Table 1.1. Let us look a little more closely at the meaning of those numbers.

Suppose we fix on a certain amount of time (one second, one minute — the specific choice is unimportant). Let s be the maximum input size a particular algorithm can handle within that amount of time. Now suppose we allow t times as much time (or our computer speed increases by a factor of t, either because technology has improved, or simply because we went out and bought a more expensive machine). Table 1.3 shows the effect of the speedup for several complexities.

The values in the third column are computed by observing that

$$f(s_{new}) = \text{number of steps after speedup}$$

$$= t \text{ times the number of steps before the speedup } = tf(s)$$

and solving

$$f(s_{new}) = tf(s)$$

for s_{new}.