第2章　算法分析

算法(algorithm)是为求解一个问题需要遵循的、被清楚指定的简单指令的集合。对于一个问题，一旦某种算法给定并且(以某种方式)被确定是正确的，那么重要的一步就是确定该算法将需要多少诸如时间或空间等资源量的问题。如果一个问题的求解算法竟然需要长达一年时间，那么这种算法就很难能有什么用处。同样，一个需要若干个 GB(gigabyte)的内存的算法在当前的大多数机器上也是无法使用的。

在这一章，我们将讨论：

- 如何估计一个程序所需要的时间。
- 如何将一个程序的运行时间从天或年降低到秒甚至更少。
- 粗心使用递归的后果。
- 将一个数自乘得到其幂，以及计算两个数的最大公因数的非常有效的算法。

2.1　数学基础

一般说来，估计算法资源消耗所需的分析是一个理论问题，因此需要一套正式的系统架构。我们先从某些数学定义开始。

本书将使用下列四个定义：

定义 2.1　如果存在正常数 c 和 n_0 使得当 $N \geqslant n_0$ 时 $T(N) \leqslant cf(N)$，则记为 $T(N)=O(f(N))$。

定义 2.2　如果存在正常数 c 和 n_0 使得当 $N \geqslant n_0$ 时 $T(N) \geqslant cg(N)$，则记为 $T(N)=\Omega(g(N))$。

定义 2.3　$T(N)=\Theta(h(N))$当且仅当 $T(N)=O(h(N))$和 $T(N)=\Omega(h(N))$。

定义 2.4　如果对每一正常数 c 都存在常数 n_0 使得当 $N>n_0$ 时 $T(N)<cp(N)$，则 $T(N)=o(p(N))$。有时也可以说，如果 $T(N)=O(p(N))$且 $T(N)\neq\Theta(p(N))$，则 $T(N)=o(p(N))$。

这些定义的目的是要在函数间建立一种相对的级别。给定两个函数，通常存在一些点，在这些点上一个函数的值小于另一个函数的值，因此，一般地宣称，比如说 $f(N)<g(N)$，是没有什么意义的。于是，我们比较它们的**相对增长率**(relative rate of growth)。当将相对增长率应用到算法分析时，我们将会明白为什么它是重要的度量。

虽然对于较小的 N 值 $1\,000N$ 要比 N^2 大，但 N^2 以更快的速度增长，因此 N^2 最终将是更大的函数。在这种情况下，$N=1\,000$ 是转折点。第一个定义是说，最后总会存在某个点 n_0 从它以后 $c \cdot f(N)$总是至少与 $T(N)$一样大，从而若忽略常数因子，则 $f(N)$至少与 $T(N)$一样大。在我们的例子中，$T(N)=1\,000N$，$f(N)=N^2$，$n_0=1\,000$ 而 $c=1$。我们也可以让 $n_0=10$ 而 $c=100$。因此，可以说 $1\,000N=O(N^2)$(N 平方级)。这种记法称为**大 *O* 标记法**。人们常常不说“……级的”，而是说“大 O……”。

如果用传统的不等式来计算增长率，那么第一个定义是说 $T(N)$的增长率小于或等于 $f(N)$的增长率。第二个定义 $T(N)=\Omega(g(N))$(念成“omega”)是说 $T(N)$的增长率大于或等于 $g(N)$