第1章 概 述

计算机算法的设计与分析，简称为计算机算法，是计算机科学领域中一个重要分支，它主要研究两方面问题：

1）如何分析一个给定算法的时间复杂度。

2）如何为给定的计算问题设计一个算法，使得它的复杂度最低。

通俗讲，这里的时间复杂度就是用多少时间，显然以上两个问题是紧密相连的。因实用价值不大，计算机算法课一般已不太讨论空间复杂度，即用多少存储单元的问题，本书也是这样。这一章，我们介绍算法的时间复杂度的基本概念以及分析时遵循的基本原则。

1.1 算法与数据结构及程序的关系

计算机算法课是数据结构课和程序设计课的后续课程。一个简单的问题是，学了数据结构课和程序设计课之后，为什么还要学算法？程序本身是算法吗？这一节逐步给出简单回答。在深入学习之后，读者自己会有更准确的理解和领悟。

1.1.1 什么是算法

简单地说，算法（algorithm）是为解决某一计算问题而设计的一个过程，其每一步必须能在计算机上实现并在有限时间内完成。广义地讲，一个用某一语言（比如 C++ 或机器语言）编写的程序也可称为算法。但是，为了理论分析的严格性和方便，我们对算法的定义加了某些限制。经过下面的讨论，我们会了解是哪些限制及为什么要加这些限制。

1.1.2 算法与数据结构的关系

算法与数据结构是密不可分的，除极少数算法外，几乎所有算法都需要数据结构的支持，而且数据结构的优劣往往决定算法的好坏。数据结构把输入的数据及运算过程中产生的中间数据以某种方式组织起来以便于动态地寻找、更改、插入、删除等。没有一种数据结构是万能的，我们应根据问题和算法的需要选用和设计数据结构，而在讨论数据结构时也必定会讨论其适用的算法。所以，数据结构课程与算法课程的内容往往有很大重叠。但是，数据结构课程需要解释其在计算机上的具体实现，而算法课程着重讨论在更为抽象的层次上解决问题的技巧及分析方法。打个比方，数据结构就好像汽车零件，例如发动机、车轮、车窗、车闸、座椅、灯光、方向盘等，而算法就好像是汽车总体设计。我们假定读者熟悉常用的一些数据结构，包括数组、队列、堆栈、二叉树等，而略去对它们的介绍。读者还应当具有基本的编写程序的知识。

1.1.3 算法与程序的关系

一段用某种计算机语言写成的源码，如果可以在计算机上运行并正确地解决一个问题，则称为一个程序。程序必须严格遵守该语言规定的语法（包括标点符号），并且编程时往往还必须考虑到计算机的物理限制，例如，最大允许的整数在 32 位机上和 16 位机上是不同的。而算法则不依赖于某种语言，更不依赖具体计算机的限制。只要步骤和逻辑正确，一个算法可以用任何一种语言表达。当然这种语言必须清楚无误地定义每一步骤且能够让稍懂程序的人看懂。这样，设计算法者可以着重考虑解题方法而免去不必要的琐碎的语法细节。所以，算法通常不是程序，但一定

可以用任一种语言的程序来实现。(我们假定机器有足够大的内存。)

1.1.4 选择排序的例子

在这一节，我们举一个例子——选择排序（selection sort）——来说明算法与程序的关系。排序问题就是要求把 n 个输入的数字从小到大（或从大到小）排好。我们假定这 n 个输入的数字是存放在数组 $A[1..n]$ 中。下面的算法称为选择排序。

【例 1-1】选择排序。

输入：$A[1], A[2], \cdots, A[n]$

输出：把输入的 n 个数重排使得 $A[1] \leqslant A[2] \leqslant \cdots \leqslant A[n]$

```
Selection-Sort (A[1..n])
1    for (i ← 1, i ⩽ n, i++)
2        key ← i
3        for (j ← i, j ⩽ n, j++)
4            if A[j] < A[key]
5                then key ← j
6            endif
7        endfor
8        A[i] ↔ A[key]
9    endfor
10   End
```

这个算法看上去像 C++ 程序，但不是。实际上，它不遵守目前为止任一个可在计算机上编译的语言规定的语法，但它把算法的步骤描述得很清楚。这个算法含有 n 步，对应于变量 i 从 1 变到 n。第一步，它把最小数选出并放在 $A[1]$中；第二步，它把余下的在 $A[2..n]$中的最小数选出并放在 $A[2]$中，……，第 i 步，它把余下的在 $A[i..n]$中最小的数选出并放在 $A[i]$中。当 $i = n$ 时，排序完成。算法中第 2 行到第 7 行表明该算法是用顺序比较的方法找到 $A[i..n]$中最小的数所在的位置 $A[key]$，然后交换 $A[i]$和 $A[key]$，该算法显然是正确的。

1.1.5 算法的伪码表示

从上一节例子中，我们看到算法的描述不依赖于某一语言，但又必须用某种语言去描述。我们把这个语言称为伪语言（pseudo language），而用该语言所表达的算法称为伪码（pseudo code）。不同的人可用不同的伪码写算法。本书允许任何含义清楚的伪码，例如上一节中的例 1-1。但是我们应当注意符号的一致性，例如我们用“←”表示赋值，则不要与“=”或“:=”混用。用伪码可以方便我们对算法的描述，有时还可以大大简化描述。例如，例 1-1 中的算法还可以描述如下。

【例 1-2】选择排序的另一种描述。

```
Selection-Sort (A[1..n])
1    for (i ← 1, i ⩽ n, i++)
2        find j such that A[j] = Min{A[i], A[i+1], ⋯, A[n]}
3                        //这里，符号 Min 表示找出集合中有最小值的元素
4        A[i] ↔ A[j]
5    endfor
6    End
```

显然，这样的伪码大大简化了描述，突显了思路和方法，且易于分析。本书的伪码以英文为主，适当加入中文注释，以简洁、准确为原则。

1.2　算法复杂度分析

简单地说，某个算法的时间复杂度（time complexity）是它对一组输入数据执行一次运算并产生输出所需要的时间。早期的计算机使用者往往只注重程序的正确性而忽略了其时间的复杂度，因为当时计算机的速度与人算、珠算、机械式计算机或电动计算机相比太快了，似乎不需要时间。但后来人们困惑地发现，调试好的程序在实际应用时却迟迟不能算出结果，因而往往导致上百万美元投资的失败。原来，程序是正确的，但是，随着所解问题的规模增大，每次执行的时间却成十倍、百倍、千倍地增长。所以，算法复杂度的分析成了一个十分重要和必须考虑的问题。

1.2.1　算法复杂度的度量

既然算法复杂度很重要，那我们该如何度量呢？直接在机器上测量执行一次算法的时间显然不是好办法，因为即使是同样的算法和同样的输入数据，在快速计算机上执行和在慢速计算机上执行的时间也是不同的。而且，不同的编译程序也会影响计算时间。这样的测量不能反映算法本身的优劣。

一个客观的度量方法是数一数在执行一次算法时共需要多少次基本运算。基本运算指一条指令或若干条指令可以完成的操作，例如加法、减法、乘法、除法、赋值、两个数的比较，等等。不同语言提供的基本运算集合也许不完全相同，但大部分相同。某一语言所允许的不同的基本运算的个数是一个常数，且很少超出 300。但是，这一办法说起来容易做起来难。就拿例 1-1 中这一简单算法来说，有变量 i 和 n 的比较，有 i 每次加 1 的运算，有数组 A 中数字的比较，等等。数出一个准确的数很困难。对于稍微复杂一点的算法，这一办法很难实行。所以我们必须简化对复杂度的度量。

解决的办法是，我们只统计算法中一个主要的基本运算被执行的次数并把它作为算法的复杂度。这里，主要的基本运算指的是在算法中被执行得最多的一个运算。这样，度量会大为简化。例如，在例 1-1 中，第 4 行的比较运算是一个主要运算。因此，选择排序的复杂度可以这样得出：

选出最小的数并放入 $A[1]$需要（$n-1$）次比较;

选出第二小的数并放入 $A[2]$需要（$n-2$）次比较;

…

选出第 i 小的数并放入 $A[i]$需要（$n-i$）次比较;

…

所以，总共需要的比较次数即复杂度是

$$T(n) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} \qquad (1.1)$$

那么，这种简化是否合理呢？我们在下面会继续讨论。

1.2.2　算法复杂度与输入数据规模的关系

算法的基本功能是将一组输入数据进行处理和运算，并产生和输出符合所解问题需要的结果。显然，一个算法的复杂度与输入数据的规模（通常用 n 表示）有关系。例如，（1.1）式表明，选择排序中 n 越大，则所需比较的次数就越多。我们当然希望算法中主要的基本运算的执行次数要少，但更重要的是，我们希望其复杂度随着 n 的增长而缓慢地增长。所以，一个算法的复杂度，就像（1.1）式那样，是一个以输入数据规模 n 为自变量的函数 $f(n)$。假定某一问题有两个算法，

其复杂度分别为 $f(n)$ 和 $g(n)$。如果在 n 趋向无穷大时，$f(n) < g(n)$，我们则认为第一个算法优于第二个算法。例如，$f(n) = 200n, g(n) = n^2$，$f(n)$ 比 $g(n)$ 好。注意，在 n 小于 200 时，$f(n)$ 并不优于 $g(n)$，所以在解决具体问题时，我们应当理论联系实际以确定用哪个算法。但是，在算法分析领域，我们只注重考虑在 n 趋向无穷大时的情形以便于理论的分析，这是因为函数增长快慢的本质在这种情形下才真正体现出来。所以，算法分析的结果对解决实际问题起着巨大的和决定性的指导作用，而理论联系实际的工作需要由解决实际问题的工作者去完成。本书将尽量把理论的讨论与实际问题相结合。

因为我们必须讨论 n 趋向无穷大时的情形，所以算法所解决的问题必须允许无穷多个不同大小的输入数据。比如排序问题，算法必须是能将任意多的数字排好序的一般方法。例 1-1 中选择排序是一个算法，因为 n 可以任意大。作为反例，如果一个程序周期性地计算每周 7 天测量的温度的平均值，那么这个程序不在我们讨论的算法范畴内。这是我们对算法定义的最主要的限制，这也是对我们要解决的问题的限制，即所解问题必须有无穷多个可能的不同规模的输入数据，而每一个具体的输入数据则称为该问题的一个实例（instance）。例如，5.1、3.2、8.6、9.7、4.0 这 5 个数可认为是排序问题的一个实例。解决一个问题的算法必须能够对任何一组输入数据进行运算并输出正确结果。

通常讨论的算法还有一些其他限制，例如，算法不可以无结果地永远不停地运算下去等。因本书所讨论的算法不会涉及这些问题，我们略去对它们的解释。上面的限制，即要求输入规模 n 可趋向无穷大，是最主要的。

1.2.3 输入数据规模的度量模型

因为算法的复杂度是输入数据规模大小（简称输入规模）的函数，那么如何来度量输入规模呢？一般所采用的模型是用输入数据中数字的个数或输入的集合中元素的个数 n 作为输入的大小。在这个模型下，每个数，不论其数值大小，均占有一个存储单元，而任何一个基本运算所需时间是常数并不受被操作数大小的影响。例如，做 5+7 和做 10 225 566+45 787 899 所需时间是一样的。这本书也采用这个模型。这个模型也有其缺点。比如，它把变量 n 也作为一般变量处理。这似乎不太合理。一方面允许 n 趋向无穷大，而另一方面又认为任何与 n 有关的基本运算像其他数字一样只需常数时间。这是理论不完美的地方。但是，这部分的影响一般小于算法中主要部分的复杂度，不影响算法优劣的比较。这一模型在实践中被广泛采用而且非常成功。

另一个模型是用输入数据的二进制表示中所用的比特数作为输入数据的规模。这个模型在输入数据只含一个数或几个数时很有用。比如，我们希望判断一个数 a 是否为质数时，输入数据只有一个数。这时，前面的模型失败，因为 n 始终为 1，而且数 a 往往是个非常大的数，用一个存储单元存储 a 是一个不合理的假设。这时，用其比特数表示输入规模是一个合理的模型。本书基本上用不到这个模型。

1.2.4 算法复杂度分析中的两个简化假设

因为算法复杂度的优劣取决于在 n 趋向无穷大时它增长的快慢，所以两个复杂度在 n 趋向无穷大时如相差在常数倍之内，则可以认为是同阶的函数。例如，$f(n) = 500n$ 和 $g(n) = 3n$ 是同阶无穷大。而 $f(n) = 5n$ 和 $g(n) = 5n^{1.01}$，尽管指数相差很小，却有本质不同。它们不同阶，因为在 n 趋向无穷大时 $g(n)$ 和 $f(n)$ 之比会大于任何一常数而无限增大。所以，在分析算法复杂度时，我们注重的是函数增长的阶的大小，而不计较常数因子的影响。正因为这个原因，我们在分析算法复杂度时只统计一个主要的基本运算被执行的次数就是很合理的。这是因为任一语言提供的不同的基

本运算的种类（如加、减、乘、除、比较等）是一个不大的常数 c，而一个算法中用到的不同的基本运算的种类更是它的一个小子集。假定一个算法中用到 k（$\leqslant c$）种不同运算，而算法中主要的基本运算被执行的次数是 $f(n)$，那么所有基本运算被执行的总数 $g(n)$ 满足关系 $f(n) \leqslant g(n) \leqslant kf(n)$，可见 $g(n)$ 和 $f(n)$ 是同阶的函数。为了使这两个同阶函数所对应的实际所需时间也是同阶函数，我们还需要两个假设：

1）任一基本运算所需时间是一个常数，不因被操作数大小而改变。这一点在 1.2.3 节中已提到，即运算时间与被操作数大小无关。这一假设是合理的，因为不论数字大小，计算机中二进制表示所用的比特数是一样的且硬件操作的时钟周期数也往往一样。即使有所不同，其差别也是在常数倍之内。

2）任何两个不同基本运算所需要的时间相同。例如，做一次减法与做一次乘法所需时间被认为相同。实际上会有不同，但也是在常数倍之内。

因为差别在常数因子之内的两个函数是同阶的，以上两个假设既合理又可大大简化我们的分析。它们的合理性是建立在输入规模 n 趋向无穷大的假设之上。这个假设是我们对算法的定义所加的最主要的限制或要求。我们对算法的理论分析和设计都是建立在这个假设之上的。

1.2.5　最好情况、最坏情况和平均情况的复杂度分析

同一个算法可能遇到各种不同的输入数据。即使两组输入数据有相同的规模，算法执行的时间也会大不相同，因而我们往往需要估计在遇到最有利的一组输入数据时算法所需要的时间是多少，在遇到最不利的一组输入数据时算法所需要的时间是多少，以及对各种输入情况下平均所需要的时间是多少。下面我们看一个简单例子。例 1-3 是一个线性搜索的算法。这个算法搜索数组 $A[1..n]$中的 n 个数，如发现某个数 $A[i]$（$1 \leqslant i \leqslant n$），等于要找的数 x，则报告序号 i，否则报告 0。

【例 1-3】线性搜索的算法。

Linear- Search (x, $A[1..n]$)
输入：x 和数组 $A[1..n]$
输出：如果 $A[i]=x$, $1 \leqslant i \leqslant n$，输出 i，否则输出 0。

```
1    i←1
2    while (i ⩽ n and x ≠ A[i])
3         i ← i + 1
4    endwhile
5    if i ⩽ n
6         then return (i)
7         else return (0)
8    endif
9    End
```

我们把数 x 和数组 $A[1..n]$ 之间比较的次数作为复杂度。最好的情况是 $A[1] = x$，这时算法只需要一次比较。当 x 不在数组中或者发现 $A[n]=x$，算法则需要比较 n 次，这是最坏情况。算法平均情况下的复杂度与各种情况的分布有关。假设 x 总是可以在数组中找到，并且 x 等于任一个数 $A[i]$的概率都是 $1/n$，$1 \leqslant i \leqslant n$，那么平均情况复杂度为：

$$\frac{1}{n}(1+2+\cdots+n)=\frac{1}{n}\frac{n(n+1)}{2}=\frac{n+1}{2}$$

显然，最坏情况的复杂度是最重要的和必须考虑到的，否则会因运算时间过长导致整个软件系统失败，而平均复杂度帮助我们了解长时间重复使用一个算法时可能会有的效率。最好情况的复杂度相对来讲不太重要而往往不作分析，但为了理论分析的需要，我们有时也会讨论。

1.3 函数增长渐近性态的比较

由上面讨论知，算法复杂度的优劣取决于在输入规模 n 趋向无穷大时其增长速度的快慢，即所谓的阶。这一节我们介绍评估和比较两个函数随 n 增长时渐近性态优劣的常用方法和记号，以及常见的一些复杂度函数。

1.3.1 三种比较关系及 O、Ω、Θ 记号

在比较两个复杂度函数时一般有三种情况，即等阶、高阶、低阶。然而，我们用得更多的三种关系是“不高于”、“不低于”和“等阶于”，分别用记号 O（读作大 oh）、Ω（读作大 omega）和Θ（读作大 theta）表示。下面逐一讨论。

定义 1.1 设 $f(n)$和 $g(n)$是两个定义域为自然数的正函数（即值域为正实数的函数）。如果存在一个常数 $c>0$ 和某个自然数 n_0 使得对任一 $n \geqslant n_0$，都有关系 $f(n) \leqslant cg(n)$，我们则说 $f(n)$的阶不高于 $g(n)$的阶，并记作 $f(n)=O(g(n))$。

【例 1-4】证明 $n^3+2n+5=O(n^3)$。

证明：因为当 $n \geqslant 1$ 时，我们有 $n^3+2n+5 \leqslant n^3+2n^3+5n^3=8n^3$，所以 $n^3+2n+5=O(n^3)$。（取 $c=8, n_0=1$。）■

定义 1.2 设 $f(n)$和 $g(n)$是两个定义域为自然数的正函数。如果存在一个常数 $c>0$ 和某个自然数 n_0 使得对任一 $n \geqslant n_0$，都有关系 $f(n) \geqslant cg(n)$，我们则说 $f(n)$的阶不低于 $g(n)$的阶，并记作 $f(n)=\Omega(g(n))$。

【例 1-5】证明 $n^2=\Omega(n\lg n)$。（注意，在计算机科学领域，不明示的对数底规定为 2。）

证明：因为当 $n \geqslant 1$ 时，我们有 $n>\lg n$，所以 $n^2=\Omega(n\lg n)$。（取 $c=1, n_0=1$。）■

定义 1.3 设 $f(n)$和 $g(n)$是两个定义域为自然数的正函数。如果关系 $f(n)=O(g(n))$和 $f(n)=\Omega(g(n))$同时成立，我们则说 $f(n)$与 $g(n)$同阶，并记作 $f(n)=\Theta(g(n))$。

【例 1-6】证明 $n^3+2n+5=\Theta(n^3)$。

证明：因为例 1-4 已经证明了 $n^3+2n+5=O(n^3)$，只需证明 $n^3+2n+5=\Omega(n^3)$。我们注意到当 $n \geqslant 1$ 时，$n^3+2n+5>n^3$，所以 $n^3+2n+5=\Omega(n^3)$，这样也就证明了 $n^3+2n+5=\Theta(n^3)$。■

1.3.2 表示算法复杂度的常用函数

多项式函数是用于表示算法复杂度的最常见的函数之一。容易证明任一个多项式函数与其首项同阶。我们把它总结在定理 1.1 中。

定理 1.1 假设 $p(n)=a_kn^k+a_{k-1}n^{k-1}+a_{k-2}n^{k-2}+\cdots+a_1n^1+a_0$ 是一个变量为 n 的多项式，其中系数 $a_k>0$，那么 $p(n)=\Theta(n^k)$。

证明：我们先证明 $p(n)=O(n^k)$。有以下演算：

$$\begin{aligned} p(n) &= a_kn^k+a_{k-1}n^{k-1}+a_{k-2}n^{k-2}+\cdots+a_1n^1+a_0 \\ &\leqslant a_kn^k+|a_{k-1}|n^{k-1}+|a_{k-2}|n^{k-2}+\cdots+|a_1|n^1+|a_0| \\ &\leqslant a_kn^k+|a_{k-1}|n^k+|a_{k-2}|n^k+\cdots+|a_1|n^k+|a_0|n^k \\ &\leqslant (a_k+|a_{k-1}|+|a_{k-2}|+\cdots+|a_1|+|a_0|)n^k \\ &\leqslant Cn^k \end{aligned}$$

这里，$C=(a_k+|a_{k-1}|+|a_{k-2}|+\cdots+|a_1|+|a_0|)$是一个大于零的常数，所以 $p(n)=O(n^k)$。

现在证明 $p(n)=\Omega(n^k)$。

$$\begin{aligned}p(n)&=a_kn^k+a_{k-1}n^{k-1}+a_{k-2}n^{k-2}+\cdots+a_1n^1+a_0\\&\geqslant a_kn^k-|a_{k-1}|n^{k-1}-|a_{k-2}|n^{k-2}-\cdots-|a_1|n^1-|a_0|\\&\geqslant a_kn^k-|a_{k-1}|n^{k-1}-|a_{k-2}|n^{k-1}-\cdots-|a_1|n^{k-1}-|a_0|n^{k-1}\\&\geqslant a_kn^k-(|a_{k-1}|+|a_{k-2}|+\cdots+|a_1|+|a_0|)n^{k-1}\\&=a_kn^k-Dn^{k-1}\end{aligned}$$

这里，$D=(|a_{k-1}|+|a_{k-2}|+\cdots+|a_1|+|a_0|)$是一个非负常数。因此，我们有

$$p(n)\geqslant a_kn^k-Dn^{k-1}=a_kn^k\left(1-\frac{D}{a_k}\frac{1}{n}\right)$$

当 $n\geqslant\lceil 2D/a_k\rceil$时，

$$p(n)\geqslant a_kn^k\left(1-\frac{D}{a_k}\frac{1}{n}\right)\geqslant\frac{1}{2}\ a_kn^k$$

所以，$p(n)=\Omega(n^k)$。（取 $c=\frac{1}{2}a_k$, $n_0=\lceil 2D/a_k\rceil$。）

这也就证明了 $p(n)=\Theta(n^k)$。 ■

如果一个算法的复杂度是$\Theta(n^k)$，k 是一个正整数，我们称该复杂度为 k 阶多项式。例如，选择排序的复杂度是 2 阶多项式 $\Theta(n^2)$。当 $k=1$ 时，则称之为线性复杂度。另一常见函数是以 lg$n$ 为变量的多项式，或以 lglg$n$ 为变量的多项式，例如 $(\lg n)^2$、$(\lg\lg n)^3$ 等。而很多复杂度是以这两种函数为因子的函数，例如 $n^2(\lg n)^3(\lg\lg n)$。

如果一个函数在变量 n 趋向于无穷时比任何一个多项式都高阶，则称之为超多项式（super polynomial），例如指数函数 2^n、阶乘函数 $n!$，以及 n^n 等。这些函数增长极快以至于任何多项式与之相比都以零为极限。

【例 1-7】证明对任意正整数 k，$\lim\limits_{n\to\infty}\dfrac{n^k}{2^n}=0$。

证明：由微积分中洛必达法则，

$$\lim_{n\to\infty}\frac{n^k}{2^n}=\lim_{n\to\infty}\frac{kn^{k-1}}{2^n\ln 2}=\lim_{n\to\infty}\frac{k(k-1)n^{k-2}}{2^n(\ln 2)(\ln 2)}=\lim_{n\to\infty}\frac{k!}{2^n(\ln 2)^k}=0$$ ■

由上可见，如果一个算法的复杂度是指数函数，则该算法基本上是无法应用的，因为它增长得太快了。为了对其增长有一个感性认识，我们举下面的一个例子。据说俄国的一个沙皇与当时的一个象棋大师下棋输了，他问这个大师希望得到什么奖赏。这位大师说，棋盘有 $8\times 8=64$ 个格子，希望从第一个格子得到一粒米开始，每数下一个格子，得到的米粒数加倍。这就是说，他要得的米粒总数为 $1+2+4+\cdots+2^{63}=2^{64}-1$。这个数字有多大呢？$2^{64}-1=16\times(2^{10})^6-1>16\times 1000^6=16\times 10^{18}$。作者曾数过某种大米，一两大约有 2500 粒，也就是说，一吨米有 5 千万粒。所以，这位象棋大师要的米大约是 $16\times 10^{18}/(5\times 10^7)>3\times 10^{11}$ 吨，也就是大于 3 千亿吨。以每人每年 1000 斤米计算，这些米可供全世界 60 亿人吃 100 年。这大大超过沙俄一年的粮食收成。如果某算法需要 2^n 次运算，则当 $n=64$ 时，一台超级计算机（以每秒 1 万亿次运算计）也需要 16×10^6 秒或 185 天之多。

【例 1-8】证明对任意小正整数$\varepsilon>0$，$\lim\limits_{n\to\infty}\dfrac{\lg n}{n^\varepsilon}=0$。

证明：由微积分中洛必达法则，

$$\lim_{n\to\infty}\frac{\lg n}{n^{\varepsilon}}=\lim_{n\to\infty}\frac{\ln n}{(\ln 2)n^{\varepsilon}}=\frac{1}{\ln 2}\lim_{n\to\infty}\frac{1/n}{\varepsilon n^{\varepsilon-1}}=\frac{1}{\ln 2}\lim_{n\to\infty}\frac{1}{\varepsilon n^{\varepsilon}}=0$$ ■

例 1-8 说明对数函数的阶比任一多项式函数或幂函数阶要小。实际上，对任何正整数 k 和任意小正数 $\varepsilon>0$，都有 $(\lg n)^k=O(n^{\varepsilon})$。我们当然希望算法复杂度 $T(n)$越小越好，最好是常数（constant），记为 $T(n)=\Theta(1)$。在常数和对数之间还有其他一些函数，例如，lg*n。该函数定义为对 n 连续进行对数运算的次数使其值小于或等于 1。例如 lg*16 = 3，因为 lglglg16 = 1。在本书中，我们不常用这些函数，等用到时再作介绍。

1.4 算法复杂度与问题复杂度的关系

当为某一问题设计算法时，我们总是追求最好的复杂度。但是，怎样才能知道已达到最佳呢？我们必须考虑问题的复杂度。这一节简单介绍问题复杂度和算法复杂度的关系。

1.4.1 问题复杂度是算法复杂度的下界

问题的复杂度就是任一个解决该问题的算法所必需的运算次数。例如，任何一个用比较大小的办法将 n 个数排序的算法需要至少 lg$n$! 次比较才行。这个结果将在第 4 章讨论。那么，lg$n$! 就是（基于比较的）排序问题的复杂度。因为没有一个算法可以用少于 lg$n$! 次比较解决排序问题，lg$n$! 就成了算法复杂度的下界。通常我们使用的是在大Ω意义下的下界，即任一比较排序算法的复杂度必定为$\Omega(\lg n!)=\Omega(n\lg n)$。所以，如果可以证明某个问题至少需要$\Omega(g(n))$运算次数，那么 $\Omega(g(n))$ 就是所有解决该问题的算法的复杂度的下界。

反之，如果某一算法的复杂度是 $O(f(n))$，那么它所解决的问题的复杂度不会超过 $O(f(n))$。因此任一算法的复杂度也是其所解决的问题的复杂度的上界。通常在已知的某问题的复杂度下界和该问题最好算法的复杂度之间存在距离，算法工作者的任务就是努力寻找更好的下界或更优的上界。找出问题的复杂度，即找出其算法的下界，是一重要的工作，因为它可以告诉我们是否还有改进当前算法的余地而省去许多徒劳无功的努力。

1.4.2 问题复杂度与最佳算法

如果一个排序算法的复杂度是 $O(\lg n!)$，那么这个算法称为是（渐近）最佳的。当然，如果一个排序算法正好需要$\lceil \lg n! \rceil$次比较，它就是一个绝对最佳的算法。显然，找到一个绝对最佳的算法通常非常困难，因而最佳算法一般是指渐近最佳。当某算法的复杂度与所解问题的下界（渐近）吻合时，该算法是一个（渐近）最优的算法。

1.4.3 易处理问题和难处理问题

如果某一算法的复杂度是超多项式的，那么这个算法基本上是没有用处的，因为它需要的时间太长。如果一个问题的复杂度本身就是超多项式的，则称为难处理问题（intractable）。反之，称为易处理问题（tractable）。如果一个问题是难处理问题，那么我们只能依赖近似算法或启发式算法来解决。判断一个问题是易处理还是难处理似乎比找到该问题的复杂度容易，因为找到了复杂度就可以判断该问题是易处理还是难处理。可是，有相当多的问题看似简单，判断却十分困难。这样一类问题称为 NP 完全问题。其准确的定义会在第 14 章讨论。现在人们知道的是，如果任何一个 NP 完全问题被证明有超多项式下界，那么所有这类问题都有超多项式下界。反之，如果任何一个 NP 完全问题可以在多项式时间内解决，则所有 NP 完全问题都可以有多项式算法去解。但是，到目前为止，没有人能对任何一个 NP 完全问题给出多项式算法或证明它只能有超多项式算法，这就是著名的 P = NP 或 P≠ NP 的猜想问题。

习题

1．假设$f(n)$和$g(n)$是两个定义域为自然数的正函数。证明$f(n)+g(n)=\Theta(\max(f(n),g(n)))$。

2．假设$f(n)=\Theta(p(n))$，$g(n)=\Theta(q(n))$，并且都是正函数，证明以下结论。

（a）$f(n)+g(n)=\Theta(p(n)+q(n))$

（b）$f(n)*g(n)=\Theta(p(n)*q(n))$

（c）$f(n)/g(n)=\Theta(p(n)/q(n))$

3．对以下每个函数，用Θ记号表示与其同阶的只含一项的函数。例如，$f(n)=(n+1)^3$可表示为$f(n)=\Theta(n^3)$。

（a）$f(n)=n^2+n\lg n$

（b）$f(n)=\dfrac{\sqrt{n}(n\lg n+2n)}{\lg^2 n+n}$

（c）$f(n)=\dfrac{(n^2+\lg n)(n+1)}{n+n^2}$

4．用Θ记号表示对下面一段程序中语句$x\leftarrow x+1$被执行的次数的估计。

```
for i ← 1 to n
    for j ← i to 3i
        x ← x + 1;
    endfor
endfor
```

5．对以下每个级数和$T(n)$，用Θ记号表示与其同阶的只含一项的函数。

（a）$T(n)=\sum_{k=1}^{n}\dfrac{k^3\lg k+1}{k^2+k+1}$

（b）$T(n)=\sum_{k=1}^{n}\dfrac{\sqrt{k}+k\lg k+8}{3k^2\lg k+5k+1}$