第1章 引 论

学习要点

- 理解算法的概念
- 理解什么是程序，程序与算法的区别和内在联系
- 能够列举求解问题的基本步骤
- 掌握算法在最坏情况、最好情况和平均情况下的时间复杂性
- 掌握算法复杂性的渐近性态的数学表述
- 了解表达算法的抽象机制
- 熟悉数据类型和数据结构的概念
- 熟悉抽象数据类型的基本概念
- 理解数据结构、数据类型和抽象数据类型三者的区别和联系
- 掌握用 C 语言描述数据结构与算法的方法
- 理解递归的概念

1.1 算法及其复杂性的概念

1.1.1 算法与程序

对于计算机科学来说，算法（Algorithm）的概念是至关重要的。例如，在一个大型软件系统的开发中，设计出有效的算法起决定性作用。通俗地讲，算法是指解决问题的一种方法或一个过程。严格地讲，算法是由若干条指令组成的有穷序列，且具有下述 4 条性质。

① 输入：有零个或多个由外部提供的量作为算法的输入。

② 输出：算法产生至少一个量作为输出。

③ 确定性：组成算法的每条指令是清晰的，无歧义的。

④ 有限性：算法中每条指令的执行次数是有限的，执行每条指令的时间也是有限的。

程序（Program）与算法不同。程序是算法用某种程序设计语言的具体实现。程序可以不满足算法的性质④。例如，操作系统是一个在无限循环中执行的程序，因而不是一个算法。然而可把操作系统的各种任务看成一些单独的问题，每一个问题由操作系统中的一个子程序通过特定的算法来实现，该子程序得到输出结果后便终止。

1.1.2 算法复杂性的概念

一个算法的复杂性的高低体现在运行该算法所需要的计算机资源的多少上，所需要的资源越多，该算法的复杂性越高；反之，所需要的资源越少，该算法的复杂性越低。计算机资源中最重要的是时间和空间（即存储器）资源。因此，算法的复杂性有时间复杂性和空间复

杂性之分。

不言而喻，对于任意给定的问题，设计出复杂性尽可能低的算法是设计算法时追求的一个重要目标。另一方面，当给定的问题已有多种算法时，选择其中复杂性最低者，是选用算法应遵循的一个重要准则。因此，算法复杂性分析对算法的设计或选用有重要的指导意义和实用价值。

确切地说，算法的复杂性是指运行算法所需要的计算机资源的量。需要的时间资源的量称为时间复杂性，需要的空间资源的量称为空间复杂性。这个量应该集中反映算法的效率，而从运行该算法的实际计算机中抽象出来。换句话说，这个量应该是只依赖于算法要解的问题的规模和算法的输入的函数。

如果分别用 n 和 I 表示算法要解的问题的规模和算法的输入，用 C 表示复杂性，那么算法复杂性可表示为 $C(n,I)$。如果把时间复杂性和空间复杂性分开，并分别用 T 和 S 来表示，那么应该有 $T=T(n,I)$ 和 $S=S(n,I)$。由于时间复杂性与空间复杂性概念类同，计量方法相似，且空间复杂性分析相对简单些，所以本书将主要讨论时间复杂性。现在的问题是如何将复杂性函数具体化，即对于给定的 n 和 I，如何导出 $T(n,I)$ 和 $S(n,I)$ 的数学表达式，给出计算 $T(n,I)$ 和 $S(n,I)$的法则。下面以 $T(n,I)$ 为例，将复杂性函数具体化。

根据 $T(n,I)$ 的概念，它应该是算法在一台抽象的计算机上运行所需要的时间。设此抽象的计算机所提供的元运算有 k 种，分别记为 $O_1,O_2,\cdots,O_k$。又设每执行一次这些元运算所需要的时间分别为 $t_1,t_2,\cdots,t_k$。对于给定的算法 A，设经统计，用到元运算 O_i 的次数为 e_i，$i=1,2,\cdots,k$。显然，对于每一个 i，$1\leqslant i\leqslant k$，e_i 是 n 和 I 的函数，即 $e_i=e_i(n,I)$。那么有

$$T(n,I)=\sum_{i=1}^{k}t_i e_i(n,I)$$

式中，t_i ($i=1,2,\cdots,k$)是与 n 和 I 无关的常数。

显然，不可能对规模为 n 的每一种合法的输入 I 都统计 $e_i(n,I)$，$i=1,2,\cdots,k$。因此 $T(n,I)$ 的表达式还要进一步简化。或者说，只能在规模为 n 的某些或某类有代表性的合法输入中统计相应的 e_i，$i=1,2,\cdots,k$，评价时间复杂性。

通常考虑最坏、最好和平均三种情况下的时间复杂性，并分别记为 $T_{\max}(n)$，$T_{\min}(n)$ 和 $T_{\text{avg}}(n)$。在数学上有

$$T_{\max}(n)=\max_{I\in D_n}T(n,I)=\max_{I\in D_n}\sum_{i=1}^{k}t_i e_i(n,I)=\sum_{i=1}^{k}t_i e_i(n,I^*)=T(n,I^*)$$

$$T_{\min}(n)=\min_{I\in D_n}T(n,I)=\min_{I\in D_n}\sum_{i=1}^{k}t_i e_i(n,I)=\sum_{i=1}^{k}t_i e_i(n,\tilde{I})=T(n,\tilde{I})$$

$$T_{\text{avg}}(n)=\sum_{I\in D_n}P(I)T(n,I)=\sum_{I\in D_n}P(I)\sum_{i=1}^{k}t_i e_i(n,I)$$

式中，D_n 是规模为 n 的合法输入的集合；I^* 是 D_n 中一个使 $T(n,I^*)$ 达到 $T_{\max}(n)$ 的合法输入；$\tilde{I}$ 是 D_n 中一个使 $T(n,\tilde{I})$ 达到 $T_{\min}(n)$ 的合法输入；而 $P(I)$ 是在算法的应用中出现输入 I 的概率。

以上三种情况下的时间复杂性从不同角度反映算法的效率，各有各的局限性，也各有各的用处。实践表明，可操作性最好且最有实际价值的是最坏情况下的时间复杂性。本书对算法时

间复杂性分析的重点将放在这种情况下。

1.1.3 算法复杂性的渐近性态

随着经济的发展、社会的进步和科学研究的深入，要求用计算机解决的问题越来越复杂，规模越来越大。对求解这类问题的算法进行复杂性分析具有特别重要的意义，因而要特别关注。在此引入复杂性渐近性态的概念。

设$T(n)$是前面所定义的关于算法A的复杂性函数。一般来说，当n单调增加且趋于∞时，$T(n)$也将单调增加趋于∞。对于$T(n)$，若存在$\tilde{T}(n)$，使得当$n\to\infty$时，有$\frac{T(n)-\tilde{T}(n)}{T(n)}\to 0$，则$\tilde{T}(n)$是$T(n)$当$n\to\infty$时的渐近性态，或称$\tilde{T}(n)$为算法$A$当$n\to\infty$时的渐近复杂性，以示与$T(n)$区别。因为在数学上，$\tilde{T}(n)$是$T(n)$当$n\to\infty$时的渐近表达式；直观上，$\tilde{T}(n)$是$T(n)$中略去低阶项所留下的主项，所以它比$T(n)$更简单。

例如，当$T(n)=3n^2+4n\log n$[注]$+7$时，$\tilde{T}(n)$的一个答案是$3n^2$，因为这时有

$$\lim_{n\to\infty}\frac{T(n)-\tilde{T}(n)}{T(n)}=\lim_{n\to\infty}\frac{4n\log n+7}{3n^2+4n\log n+7}=0$$

显然，$3n^2$比$3n^2+4n\log n+7$简单得多。

由于当$n\to\infty$时$T(n)$渐近于$\tilde{T}(n)$，所以可以用$\tilde{T}(n)$来替代$T(n)$，作为算法A在$n\to\infty$时的复杂性的度量。而且$\tilde{T}(n)$明显地比$T(n)$简单，这种替代是对复杂性分析的一种简化。还要进一步考虑分析算法的复杂性的目的在于比较求解同一问题的两个不同算法的效率。而当要比较的两个算法的渐近复杂性的阶不相同时，只要能确定出各自的阶，就可以判定哪一个算法的效率高。换句话说，这时的渐近复杂性分析只要关心$\tilde{T}(n)$的阶就够了，不必关心包含在$\tilde{T}(n)$中的常数因子。因此又可对$\tilde{T}(n)$的分析进一步简化，即假设算法中用到的所有不同的元运算各执行一次所需要的时间都是一个单位时间。

上面已经给出了简化算法复杂性分析的方法，即只需要考查当问题的规模充分大时，算法复杂性在渐近意义下的阶。本书的算法分析都将这么进行。为此引入渐近意义下的记号O，Ω，θ和o。

以下设$f(n)$和$g(n)$是定义在正数集上的正函数。

若存在正的常数C和自然数n_0，使得当$n\geqslant n_0$时有$f(n)\leqslant Cg(n)$，则称函数$f(n)$当n充分大时上有界，且$g(n)$是它的一个上界，记为$f(n)=O(g(n))$。这时还称$f(n)$的阶不高于$g(n)$的阶。

举几个例子如下。

（1）因为对所有的$n\geqslant 1$有$3n\leqslant 4n$，所以$3n=O(n)$。

（2）因为当$n\geqslant 1$时有$n+1024\leqslant 1025n$，所以$n+1024=O(n)$。

（3）因为当$n\geqslant 10$时有$2n^2+11n-10\leqslant 3n^2$，所以$2n^2+11n-10=O(n^2)$。

注：没有标注底的 log 可默认为以 2 为底。在算法分析中，可不关心对数的底，这是因为不同的底只相差一个常数，对算法复杂性没有任何影响。

（4）因为对所有 $n \geqslant 1$ 有 $n^2 \leqslant n^3$，所以 $n^2 = O(n^3)$。

（5）一个反例 $n^3 \neq O(n^2)$。因为若不然，则存在正的常数 C 和自然数 n_0，使得当 $n \geqslant n_0$ 有 $n^3 \leqslant Cn^2$，即 $n \leqslant C$。显然，当取 $n = \max\{n_0, \lfloor C \rfloor + 1\}$ 时这个不等式不成立，所以 $n^3 \neq O(n^2)$。

按照符号 O 的定义，容易证明它有如下运算规则：

（1）$O(f) + O(g) = O\big(\max(f, g)\big)$；

（2）$O(f) + O(g) = O(f + g)$；

（3）$O(f)O(g) = O(fg)$；

（4）若 $g(n) = O\big(f(n)\big)$，则 $O(f) + O(g) = O(f)$；

（5）$O\big(Cf(n)\big) = O\big(f(n)\big)$，其中 C 是一个正的常数；

（6）$f = O(f)$。

规则（1）的**证明**：设 $F(n) = O(f)$。根据符号 O 的定义，存在正常数 C_1 和自然数 n_1，使得对所有的 $n \geqslant n_1$，有 $F(n) \leqslant C_1 f(n)$。类似地，设 $G(n) = O(g)$，则存在正常数 C_2 和自然数 n_2，使得对所有的 $n \geqslant n_2$ 有 $G(n) \leqslant C_2 g(n)$。

令 $C_3 = \max\{C_1, C_2\}$，$n_3 = \max\{n_1, n_2\}$，$h(n) = \max\{f, g\}$，则对所有的 $n \geqslant n_3$，有

$$F(n) \leqslant C_1 f(n) \leqslant C_1 h(n) \leqslant C_3 h(n)$$

类似地有

$$G(n) \leqslant C_2 f(n) \leqslant C_2 h(n) \leqslant C_3 h(n)$$

因而

$$\begin{aligned} O(f) + O(g) &= F(n) + G(n) \\ &\leqslant C_3 h(n) + C_3 h(n) \\ &= 2C_3 h(n) \\ &= O(h) \\ &= O\big(\max(f, g)\big) \end{aligned}$$

其余规则的证明类似，留给读者作为练习。

根据符号 O 的定义，用它评估算法的复杂性，得到的只是当规模充分大时的一个上界。这个上界的阶越低，评估就越精确，结果就越有价值。

与渐近复杂性有关的另一记号是 Ω，其定义如下：若存在正常数 C 和自然数 n_0，使得当 $n \geqslant n_0$ 时有 $f(n) \geqslant Cg(n)$，则称函数 $f(n)$ 当 n 充分大时下有界，且 $g(n)$ 是它的一个下界，记为 $f(n) = \Omega\big(g(n)\big)$。这时我们还说 $f(n)$ 的阶不低于 $g(n)$ 的阶。

用 Ω 评估算法的复杂性，得到的只是该复杂性的一个下界。这个下界的阶越高，评估就越精确，结果就越有价值。这里的 Ω 只是对问题的一个算法而言的。如果它是对一个问题的所有算法或某类算法而言的，即对于一个问题和任意给定的充分大的规模 n，下界在该问题的所有算法或某类算法的复杂性中取，那么它将更有意义。这时得到的相应下界，称为问题的下界或某类算法的下界。它常常与符号 O 配合，以证明某问题的一个特定算法是该问题的最优算法，或该问题的某算法类中的最优算法。

现在来看符号 θ 的定义。定义 $f(n) = \theta\big(g(n)\big)$ 当且仅当 $f(n) = O\big(g(n)\big)$ 且 $f(n) = \Omega\big(g(n)\big)$。这时认为 $f(n)$ 与 $g(n)$ 同阶。

最后，若对于任意给定的$\varepsilon>0$，都存在正整数n_0，使得当 $n\geqslant n_0$ 时有 $f(n)/g(n)<\varepsilon$，则称函数 $f(n)$ 当 n 充分大时的阶比 $g(n)$ 的低，记为 $f(n)=O(g(n))$。

例如，$4n\log n+7=O(3n^2+4n\log n+7)$。

1.2 算法的表达与数据表示

1.2.1 问题求解

用计算机解决一个稍复杂的实际问题，一般都要进行如下步骤。

（1）将实际问题数学化，即把实际问题抽象为一个带有一般性的数学问题。这一步要引入一些数学概念，精确地阐述数学问题，弄清问题的已知条件和所要求的结果，以及在已知条件和所要求的结果之间存在着的隐式或显式的联系。

（2）对于确定的数学问题，设计求解的方法，即算法设计。这一步要建立问题的求解模型，即确定问题的数据模型并在此模型上定义一组运算，然后借助于对这组运算的执行和控制，从已知数据出发导出所要求的结果，形成算法并用自然语言来表述。这种语言不是程序设计语言，不能被计算机接受。

（3）用计算机上的一种程序设计语言来表达已设计好的算法。即将非形式化的自然语言表达的算法转变为用一种程序设计语言表达的算法。这一步称为程序设计或程序编制。

（4）在计算机上编辑、调试和测试编制好的程序，直到输出所要求的结果。

在上述问题求解的过程中，求解问题的算法及其实现是核心内容。本章着重考虑第（2）步，而且把注意力集中在算法表达的抽象机制上，目的是引入抽象数据类型的重要概念，同时为大型程序设计提供一种自顶向下逐步求精的模块化方法，即运用抽象数据类型来描述程序的方法。

1.2.2 表达算法的抽象机制

算法是一个运算序列。它的所有运算定义在一类特定的数据模型上，并以解决一类特定问题为目标。算法的程序表达归根结底是算法要素的程序表达，因为一旦算法的每一项要素都已经用程序清楚地表达，整个算法的程序表达也就不成问题了。

算法实现有如下三要素：

（1）作为运算序列中各种运算的对象和结果的数据；

（2）运算序列中的各种运算；

（3）运算序列中的控制转移。

这三要素依序分别简称为数据、运算和控制。

由于算法层出不穷，千变万化，其运算的对象数据和得到的结果数据名目繁多。最简单最基本的有布尔值数据、字符数据、整数和实数数据等；稍复杂的有向量、矩阵、记录等数据；更复杂的有集合、树和图，还有声音、图形、图像等数据。

同样，运算种类也五花八门。最基本最初等的有赋值运算、算术运算、逻辑运算和关系运算等；稍复杂的有算术表达式、逻辑表达式等；更复杂的有函数值计算、向量运算、矩阵运算、集合运算，以及表、栈、队列、树和图上的运算等；此外，还有以上列举的运算的复合和嵌套。

控制转移相对单纯。在串行计算中，它只有顺序、分支、循环、递归和无条件转移等几种。

最早的程序设计语言是机器语言，即具体的计算机上的一个指令集。当时，在计算机上运行的所有算法都必须直接用机器语言来表达，计算机才能接受。算法的运算序列（包括运算对象和运算结果）都必须转换为指令序列，其中的每一条指令都以编码（指令码和地址码）的形式出现。这与用高级程序设计语言表达的算法相差甚远。对于没受过程序设计专门训练的人来说，程序可读性极差。

用机器语言表达算法的数据、运算和控制十分繁杂，因为机器语言所提供的指令太初等、太原始。机器语言只能表达算术运算、按位逻辑运算和数的大小比较运算等。稍复杂的运算都必须分解为最初等的运算，才能用相应的指令替代它。机器语言能直接表达的数据只有最原始的位、字节和字三种。算法中即使是最简单的数据，如布尔值、字符、整数、实数，也必须映射到位、字节和字中，还要给它们分配存储单元。对于算法中有结构的数据的表达，则要麻烦得多。机器语言所提供的控制转移指令也只有无条件转移、条件转移、进入子程序和从子程序返回等最基本的几种。用它们构造循环、形成分支、调用函数都要事先做许多准备，还需要许多经验和技巧。

直接用机器语言表达算法有许多缺点，如下所述。

（1）大量繁杂的细节牵制着程序员，使他们不可能有更多的时间和精力去从事创造性的劳动，执行对他们来说更为重要的任务，如确保程序的正确性、高效性。

（2）程序员既要把握程序设计的全局又要深入每一个局部直到实现的细节，即使智力超群的程序员也常常会顾此失彼，屡出差错，因而所编制的程序可靠性差，且开发周期长。

（3）由于用机器语言进行程序设计的思维和表达方式与人们的习惯大相径庭，只有经过较长时间职业训练的程序员才能胜任，使得程序设计曲高和寡。

（4）机器语言的书面形式全是“密码”，可读性差，不便于交流与合作。

（5）机器语言高度依赖于具体的计算机，可移植性和可重用性差。

克服上述缺点的办法是对程序设计语言进行抽象，让它尽可能接近算法语言。为此，人们首先注意到的是可读性和可移植性，因为它们较容易通过抽象得到改善。

汇编语言实现了对机器语言的抽象，它将机器语言的每一条指令符号化：以记忆符号替代指令码，以符号地址替代地址码，使含义显现在符号上而不再隐藏在编码中。另一方面，汇编语言摆脱了具体计算机的限制，可在具有不同指令集的计算机上运行，只要该计算机配备了汇编语言的一个汇编程序。这无疑是机器语言朝算法语言靠拢迈出的重要一步。但它离算法语言还太远，程序员还不能从分解算法的数据、运算和控制，直至细化到汇编可直接表达的指令等繁杂的事务中解脱出来。

高级程序设计语言的出现使算法的程序表达产生了一次飞跃。算法最终要表达为具体计算机上的机器语言才能在该计算机上运行，得到所需要的结果。但汇编语言的实践启发人们，表达成机器语言不必一步到位，可以分两步走，即先表达成一种中间语言，然后转换成机器语言。汇编语言作为一种中间语言，并没有获得很大成功，原因是它离算法语言还太远。这便促使人们去设计一种尽量接受算法语言的规范语言，即高级程序设计语言，让程序员可以方便地表达算法，然后借助于规范的高级语言到规范的机器语言的“翻译”，最终将算法表达为机器语言。而且，由于高级语言和机器语言都具有规范性，这里的“翻译”完全可以机械化地由计算

机来完成，就像汇编语言翻译成机器语言一样，只要计算机配备一个编译程序即可。上述两步，前一步由程序员去完成，后一步由编译程序去完成。在规定好它们各自该做什么之后，这两步是完全独立的。前一步要做的只是用高级语言正确地表达给定的算法，产生一个高级语言程序；后一步要做的只是将第一步得到的高级语言程序翻译成机器语言程序。至于程序员如何用高级语言表达算法，编译程序如何将高级语言表达的算法翻译成机器语言表达的算法，二者毫不相干。

处理从算法语言最终表达成机器语言这一复杂过程的上述思想方法就是一种抽象。汇编语言和高级语言的出现都是这种抽象的范例。与汇编语言相比，高级语言的巨大成功在于它在数据、运算和控制三方面的表达中引入了许多使之十分接近算法语言的概念和工具，大大提高了抽象表达算法的能力。

在运算方面，高级语言除允许原封不动地运用算法语言的算术运算、逻辑运算、关系运算、算术表达式和逻辑表达式外，还引入了强有力的函数等工具，并让用户自定义。这一工具的重要性不仅在于它精简了重复的程序文本段，还在于它反映出程序的二级抽象。在函数调用级，人们只关心它能做什么，不必关心它如何做。只是到定义函数时，人们才给出如何做的细节。用过高级语言的读者都知道，一旦函数的名称、参数和功能被规定清楚，在程序中调用它们便与在程序的头部说明它们完全分开。可以修改一个函数甚至更换函数体而不影响调用该函数。如果把函数名看成运算名，把参数看成运算的对象或运算的结果，那么，函数调用和初等运算的引用就完全一样。利用函数及函数的复合或嵌套可以很自然地表达算法语言中任何复杂的运算。

在数据表示方面，高级语言引入了数据类型的概念，即把所有的数据加以分类。每一个数据（包括表达式）或每一个数据变量都属于其中确定的一类。这一类数据称为一个数据类型。数据类型是数据或数据变量类属的说明，它指示该数据或数据变量可能取的值的全体。对于无结构的数据，高级语言除提供标准的基本数据类型外，还提供用户可自定义的枚举类型、子界类型和指针类型等。这些类型的使用方式都符合人们在算法语言中的使用习惯。对于有结构的数据，高级语言提供了数组、记录、集合和文件等标准的结构数据类型。其中，数组是科学计算中的向量、矩阵的抽象；记录是商业和管理中的记录的抽象；集合是数学中小集合的势集的抽象；文件是外存储数据（如磁盘中的数据等）的抽象。人们可以利用所提供的基本数据类型，按数组、记录、集合和文件的构造规则构造有结构的数据。此外，还允许用户利用标准的结构数据类型，通过复合或嵌套构造更复杂、更高层的结构数据。这使得高级语言中的数据类型呈明显的分层。高级语言中数据类型的分层是没有穷尽的，因而用它们可以表达算法语言中任何复杂层次的数据。

在控制方面，高级语言通常提供表达算法控制转移的如下方式：

（1）默认的顺序控制；

（2）条件（分支）控制；

（3）选择（情况）控制；

（4）循环控制；

（5）函数调用，包括递归函数调用；

（6）无条件转移。

以上算法控制转移表达方式不仅满足了算法语言中所有控制表达的要求，而且不再像机器

语言或汇编语言那样原始、烦琐、隐晦，而是如上面所看到的，与自然语言的表达相差无几。

高级程序设计语言是对机器语言的进一步抽象，它带来的好处主要是：

（1）高级语言更接近算法语言，易学易用，一般工程技术人员只需要几周时间的培训就可以从事简单的程序开发工作；

（2）高级语言为程序员提供了结构化程序设计的环境和工具，使得设计出来的程序可读性好，可维护性强，可靠性高；

（3）高级语言不依赖于机器语言，与具体的计算机硬件关系不大，因而写出来的程序可移植性好，重用率高；

（4）由于把繁杂的事务交给了编译程序去做，所以自动化程度高，开发周期短，且程序员得以解脱，可以集中时间和精力去从事更为重要的创造性劳动，提高程序的质量。

1.3 抽象数据类型

1.3.1 抽象数据类型的基本概念

与机器语言和汇编语言相比，高级语言的出现大大简化了程序设计。但算法从非形式的自然语言表达转换为形式化的高级语言表达，仍然是一个复杂的过程，要做很多繁杂的事情，因而仍然需要进一步抽象。

设计一个明确的数学问题的算法，总是先选用该问题的一个数据模型；接着，弄清所选用的数据模型在已知条件下的初始状态和要求的结果状态，以及这两个状态之间的隐含关系；然后探索从已知初始状态到结果状态所必需的运算步骤；最后把这些步骤记录下来，就是求解该问题的算法。

按照自顶向下逐步求精的原则，在探索运算步骤时，首先应该考虑算法顶层的运算步骤，再考虑底层的运算步骤。

顶层的运算步骤是指定义在数据模型级上的运算步骤，或称宏观步骤。它们组成算法的主干部分。这部分算法通常用非形式的自然语言表达。其中涉及的数据是数据模型中的一个变量，暂时不关心它的数据结构；涉及的运算以数据模型中的数据变量作为运算对象，或作为运算结果，或二者兼而有之，简称为定义在数据模型上的运算。由于暂时不关心变量的数据结构，所以这些运算都带有抽象性质，不含运算的细节。

底层的运算步骤是指顶层抽象运算的具体实现。它们依赖于数据模型的结构及其具体表示。因此，底层的运算步骤包括两部分：一是数据模型的具体表示；二是定义在该数据模型上的运算的具体实现。可以把它们理解为微观运算。于是，底层运算是顶层运算的细化，底层运算为顶层运算服务。为了将顶层算法与底层算法隔开，使二者在设计时不会互相牵制、互相影响，必须对二者的接口进行一次抽象。让底层只通过这个接口为顶层服务，顶层也只通过这个接口调用底层的运算。这个接口就是抽象数据类型，其英文术语是 Abstract Data Types，简记为 ADT。

抽象数据类型是算法设计和程序设计中的重要概念。严格地说，它是算法的一个数据模型连同定义在该模型上并作为该算法构件的一组运算。这个概念明确地把数据模型与该模型上的运算紧密地联系起来。数据模型上的运算依赖于数据模型的具体表示，因为数据模型上的运算

以数据模型中的数据变量作为运算对象，或运算结果，或二者兼而有之。另一方面，有了数据模型的具体表示，以及数据模型上运算的具体实现，运算的效率随之确定。如何选择数据模型的具体表示，使该模型上的各种运算的效率都尽可能高？对于不同的运算组，为使该运算组中所有运算的效率都尽可能高，其相应的数据模型的具体表示是不同的。在这个意义下，数据模型的具体表示又反过来依赖于数据模型上定义的那些运算。特别是当不同运算的效率互相制约时，还必须事先将所有的运算按使用频度排序，让所选择的数据模型的具体表示先保证使用频度较高的运算有较高的效率。数据模型与定义在该模型上的运算之间存在着的这种密不可分的联系是抽象数据类型的概念产生的背景和依据。

抽象数据类型的概念并不是全新的概念。它实际上是基本数据类型概念的引申和发展。用过高级语言进行算法设计和程序设计的人都知道，基本数据类型已隐含着数据模型和定义在该模型上的运算的统一。事实上基本数据类型中的逻辑类型就是逻辑值数据模型与三种逻辑运算（或、与、非）的统一体；整数类型就是整数值数据模型与四种算术运算（加、减、乘、除）的统一体；实型和字符型等也类同。每一种基本数据类型都连带着一组基本运算。只是由于这些基本数据类型中的数据模型的具体表示、基本运算和具体实现都很规范，都可以通过系统内置而隐蔽起来，使人们看不到它们的封装，而在算法与程序设计中直接使用基本数据类型名和相关的运算名，不究其内部细节，所以没有意识到抽象数据类型的概念已经孕育在基本数据类型的概念之中。

回到定义算法的顶层和底层的接口，即定义抽象数据类型。根据抽象数据类型的概念，对抽象数据类型进行定义就是约定抽象数据类型的名字，同时约定在该类型上定义的一组运算的各个运算的名字，明确各个运算分别有多少个参数，这些参数的含义和顺序，以及运算的功能。一旦定义清楚，在算法的顶层就可以像引用基本数据类型那样，十分简便地引用抽象数据类型；同时算法的底层就有了设计的依据和目标。顶层和底层都与抽象数据类型的定义打交道。顶层运算与底层运算没有直接的联系。因此只要严格按照定义办，顶层算法的设计和底层算法的设计就可以互相独立，互不影响，实现对它们的隔离，达到抽象的目的。

在定义了抽象数据类型之后，算法底层的设计任务如下：

（1）对于每一个抽象数据类型，赋予其具体的构造数据类型，或者说，对于每一个抽象数据类型名，赋予其具体的数据结构；

（2）对于每一个抽象类型上所定义的每一个运算名，赋予其具体的运算内容，或者说，赋予其具体的函数。

因此底层算法的设计就是数据结构的设计和函数的设计。用高级语言表达，就是构造数据类型的定义和函数的说明。

由于实际问题千奇百怪，数据模型千姿百态，问题求解的算法千变万化，抽象数据类型的设计和实现不可能像基本数据类型那样规范。它要求算法设计和程序设计人员因时因地自行筹划，目标是使抽象数据类型对外的整体效率尽可能高。本书在介绍各种抽象数据类型时会给出一些范例，供设计和实现时参考选用。

1.3.2 使用抽象数据类型的好处

使用抽象数据类型将给算法和程序设计带来很多好处，如下所述。

（1）算法顶层的设计与底层的实现分离，在进行顶层设计时不必考虑所用的数据和运算如

何表示和实现；反之，在进行数据表示和底层运算实现时，只要定义清楚抽象数据类型，而不必考虑在什么场合引用。这样做，算法和程序设计的复杂性降低了，条理性增强了，既能提高开发程序原型的速度，又能减少开发过程中的差错，保证编出来的程序有较高的可靠性。

（2）算法设计与数据结构设计隔开，允许数据结构自由选择、从容比较、优化算法和提高程序运行的效率。

（3）数据模型和该模型上的运算统一在抽象数据类型中，反映了它们之间内在的互相依赖和互相制约的关系，便于空间和时间耗费的折中，灵活地满足用户的要求。

（4）由于顶层设计和底层实现的局部化，在设计中出现的差错也是局部的，因而容易查找，容易纠正。在设计中常常要做的增、删、改也都是局部的，因而也都很容易进行。因此，用抽象数据类型表述的程序具有较好的可维护性。

（5）编出来的程序自然地呈现模块化，而且抽象数据类型的表示和实现都可以封装起来，便于移植和重用。

（6）为自顶向下逐步求精和模块化提供一种有效的途径和工具。

（7）编出来的程序结构清晰，层次分明，便于程序正确性的证明和复杂性的分析。

1.4 数据结构、数据类型和抽象数据类型

数据结构、数据类型和抽象数据类型，这三个术语在字面上既不同又相近，反映出它们在含义上既有区别又有联系。

数据结构是在整个计算机科学与技术领域中广泛使用的术语。它用来反映数据的内部构成，即数据由哪些成分数据构成，以什么方式构成，呈什么结构。数据结构有逻辑上的数据结构和物理上的数据结构之分。逻辑上的数据结构反映成分数据之间的逻辑关系；物理上的数据结构反映成分数据在计算机内的存储安排。数据结构是数据存在的形式。

数据是按照数据结构分类的，具有相同数据结构的数据属于同一类。同一类数据的全体称为一个数据类型。在高级程序设计语言中，数据类型用来说明数据在数据分类中的归属。它是数据的一种属性。这个属性限定了该数据的变化范围。为了解题的需要，根据数据结构的种类，高级语言定义了一系列的数据类型。不同的高级语言所定义的数据类型不尽相同。简单数据类型对应于简单的数据结构；构造数据类型对应于复杂的数据结构；在复杂的数据结构里，允许成分数据本身具有复杂的数据结构，因此，构造数据类型允许复合嵌套；指针类型对应于数据结构中成分数据之间的关系，表面上属于简单数据类型，实际上都指向复杂的成分数据（即构造数据类型中的数据），因此单独分出一类。

由于数据类型是按照数据结构划分的，所以一类数据结构对应着一种数据类型。一个数据变量在高级语言中的类型说明，必须是该变量所具有的数据结构所对应的数据类型。

最常用的数据结构是数组结构和记录结构。

数组结构的特点如下。

① 成分数据的个数固定，它们之间的逻辑关系由成分数据的序号（数组的下标）来体现。这些成分数据按照序号的先后顺序排列起来。

② 每一个成分数据具有相同的结构（可以是简单结构，也可以是复杂结构），因而属于同一数据类型（相应地是简单数据类型或构造数据类型）。这种同一的数据类型称为基类型。

③ 所有成分数据被依序安排在一片连续的存储单元中。概括起来，数组结构是一个线性的、均匀的、可随机访问其成分数据的结构。由于这种结构有这些良好的特性，所以最常被人们采用。

记录结构是另一种常用的数据结构，它的特点如下。

① 与数组结构一样，成分数据的个数固定。但成分数据之间没有自然序，它们处于平等地位。每一个成分数据被称为一个域并赋予域名。不同的域有不同的域名。

② 不同的域允许有不同的结构，因而允许属于不同的数据类型。

③ 与数组结构一样，可以随机访问其成分数据，但访问的途径是靠域名。在高级语言中记录结构对应的数据类型是记录结构类型。

抽象数据类型的含义在前面已有专门叙述，它可理解为数据类型的进一步抽象，即把数据类型和数据类型上的运算绑定并封装。引入抽象数据类型的目的是把数据类型的表示和数据类型上运算的实现，与这些数据类型和运算在程序中的引用隔开，使它们相互独立。对于抽象数据类型的描述，除必须描述它的数据结构外，还必须描述定义在它上面的运算。抽象数据类型上定义的运算以该抽象数据类型的数据所应具有的数据结构为基础。

1.5 用 C 语言描述数据结构与算法

描述数据结构与算法可以有多种方式，如自然语言方式、表格方式等。在本书中，采用 C 语言来描述数据结构与算法。C 语言的优点是类型丰富，语句精练，使用灵活。用 C 语言来描述算法可使整个算法结构紧凑，可读性强。在本书中，有时为了更好地阐明算法的思路，还采用 C 语言与自然语言相结合的方式来描述算法。本节对 C 语言的若干重要特性进行简要概述和回顾。

1.5.1 变量和指针

1. 变量

变量是程序设计语言对存储单元的抽象，它具有以下属性。

变量名（name）：变量名是用于标识变量的符号。

地址（address）：变量的地址是变量所占据的存储单元的地址。变量的地址属性也称为左值。

大小（size）：变量的大小指该变量所占据的存储空间的数量（以字节数来衡量）。

类型（type）：变量的类型指变量所取的值域及对变量所能执行的运算集。

值（value）：变量的值是指变量所占据的存储单元中的内容。这些内容的意义由变量的类型决定。变量的值属性也称为右值。

生命期（lifetime）：变量的生命期是指在执行程序期间变量存在的时段。

作用域（scope）：变量的作用域是指在程序中变量被引用的语句范围。

2. 指针变量

C 语言中的指针变量是一个 type*类型的变量，其中 type 为任一已定义的数据类型。

指针变量用于存放对象的存储地址。例如：

```
int k, n,*p;
n=8;
```

```
p=&n;
k=*p;
```

其中，*p* 是一个指向 int 的指针。通过间接引用指针来存取指针所指向的变量。

1.5.2 函数与参数传递

1. 函数

C 语言中函数定义包括 4 个部分：函数名、形参表、返回类型和函数体。函数的使用者通过函数名来调用该函数。调用函数时，将实参传递给形参作为函数的输入，函数体中的处理程序实现该函数的功能，最后将得到的结果作为返回值输出。下面的函数 max 是一个简单函数的例子。

```
1   int max(int x,int y)
2   {
3     return x>y?x:y;
4   }
```

其中，max 是函数名；函数后圆括号中的 int x 和 int y 是形参；函数前面的 int 是返回类型；花括号内是函数体，它实现函数的具体功能。

C 语言中函数一般都有一个返回值。函数的返回值表示函数的计算结果或函数执行状态。如果所定义的函数不需要返回值，可使用 void 来表示它的返回类型。函数的返回值通过函数体中的 return 语句返回。return 语句的作用是返回一个与返回类型相同类型的值，并中止函数的执行。

2. 参数传递

在 C 语言中调用函数时传递给形参表的实参必须与形参在类型、个数和顺序上保持一致。参数传递有两种方式。一种是按值传递。在这种方式下，把实参的值传递到函数局部工作区相应的副本中。函数使用副本执行必要的计算，因此函数实际修改的是副本的值，实参的值不变。

另一种是按地址传递。在这种方式下，需将形参声明为指针类型，即在参数名前加上符号“*”。当一个实参与一个指针类型结合时，被传递的不是实参的值，而是实参的地址。函数通过地址存取被引用的实参。执行函数调用后，实参的值将发生改变。例如：

```
1   void swap(int *x, int *y)
2   {
3      int temp = *x;
4      *x = *y;
5      *y = temp;
6   }
```

其中，函数调用 swap(*&x*,*&y*)交换变量 *x* 和 *y* 的值。

在 C 语言中数组参数的传递属特殊情形。数组作为形参可按值传递方式声明，但实际传递的是数组第一个元素的地址，因此在函数体内对于形参数组所进行的任何改变都会在实参数组中反映出来。

在函数语句 max(int x,int y)和 swap(int *x, int *y)的形参中都明确指定参数 *x* 和 *y* 的类型是

int。要对其他类型的数据，如 long 或 float，完成同样的运算就需要重写相应的函数。例如，对于数据类型 float 可以将函数 max 重写为

```
float max(float x,float y)
{
   return x>y?x:y;
}
```

如何将函数表示成与数据类型无关的形式？一般来说有多种不同途径可以达到此目的。最简单的一种方式是用 typedef 来定义一个一般的数据类型。用这个一般的数据类型来定义函数。在对具体数据类型调用函数时，只要在 typedef 中指明数据类型即可。例如，用 typedef 来定义一个一般的数据类型 num 如下。

```
typedef int num;
num max(num x,num y)
{
   return x>y?x:y;
}
```

如果要对数据类型 double 调用函数 max，只要在 typedef 中将 int 改成 double 形如：typedef double num。无须改变函数 max。

1.5.3 结构

1. 定义结构

C 语言的结构（Structure）为自定义数据类型提供了灵活方便的方法，可用于实现抽象数据类型的思想，将说明与实现分离。

结构由结构名和结构的数据成员组成。说明结构的标准形式如下。

```
struct 结构名
{
   数据成员列表;
};
```

2. 指向结构的指针

指向结构的指针值是相应的结构变量所占据的内存空间的首地址。

例如，若已经定义了一个结构 st，则语句 struct st *p; 定义一个指向结构 st 的指针。

3. 用 typedef 定义新数据类型

关键字 typedef 常与结构一起用于定义新数据类型。

下面是用 typedef 和结构定义矩形数据类型 Rectangle 的例子。

```
typedef struct recnode
{
  int x,y,h,w;/* (x,y)是矩形左下角点的坐标; h 是矩形的高; w 是矩形的宽。*/
}Recnode;
```

4．访问结构变量的数据成员

对于结构类型的变量，用圆点运算符（.）访问结构变量的数据成员。定义为指向结构的指针类型的变量用箭头运算符（->）访问结构变量的数据成员。例如：

```
Recnode rr;
Rectangle R;
R=&rr;
rr.x=1; rr.y=1; rr.h=12; rr.w=13;
printf("x=%d  y=%d \n",R->x,R->y);
printf("h=%d  w=%d \n",rr.h,rr.w);
```

其中，rr 是一个结构类型的变量，*R* 是一个指向结构的指针类型的变量。

5．新数据类型变量初始化

使用自定义数据类型变量前通常需要初始化操作。下面的函数用于说明一个 Rectangle 型变量并对其初始化。

```
Rectangle RecInit()
{
  Rectangle R=(Rectangle)malloc(sizeof *R);
  R->x=0; R->y=0; R->h=0; R->w=0;
  return R;
}
```

1.5.4　动态存储分配

1．动态存储分配函数 malloc()和 free()

C 语言的标准函数 malloc()和 free()可用于动态存储分配。例如：

```
  char *str;
  /* 为字符串分配内存 */
  if ((str=(char *)malloc(10))==0){
     printf("内存不足 \n");
     exit(1); /* 退出 */
  }
  strcpy(str, "Hello");
  /* 显示字符串 */
  printf("字符串 %s\n", str);
  /* 释放内存 */
  free(str);
```

2．动态数组

为了在运行时创建一个大小可动态变化的一维浮点数组 *x*，可先将 *x* 声明为一个 float 类型的指针。然后用函数 malloc()为数组动态地分配存储空间。例如，语句 float *x = malloc(n*sizeof (float)) 创建一个大小为 *n* 的一维浮点数组，然后可用 $x[0],x[1],\cdots,x[n-1]$访问每个数组元素。

3．二维数组

C 语言提供了多种声明二维数组的机制。在许多情况下，当形参是一个二维数组时，必须指定第二维的大小。例如，*a*[ ][10]是一个合法的形参，*a*[ ][ ]则不是。为了克服这种限制，可以使用动态分配的二维数组。例如，下面的函数创建一个 int 类型的动态工作数组，这个数组有 *r* 行和 *c* 列。

```
int **malloc2d(int r,int c)
{
    int **t=(int **)malloc(r*sizeof(int*));
    for(int i=0;i<r;i++)t[i]=(int *)malloc(c*sizeof(int));
    return t;
}
```

其他类型的二维动态数组可用类似方法创建。在程序中动态分配的数组必须在退出程序之前用 free 来释放动态分配的空间，否则容易造成内存泄漏。

1.6 递归

1.6.1 递归的基本概念

直接或间接地调用自身的算法称为递归算法。用函数自身给出定义的函数称为递归函数。在数据结构与算法设计中，递归技术是十分有用的。使用递归技术往往使函数的定义和算法的描述简捷，易于理解。有些数据结构，如二叉树等，由于其本身固有的递归特性，特别适合用递归的形式来描述。另外，还有一些问题，虽然其本身并没有明显的递归结构，但用递归技术来求解可使算法简捷易懂，易于分析。

下面用实例说明递归的概念及其应用范围。

1．阶乘函数

阶乘函数可递归地定义为

$$n!=\begin{cases}1, & n=0\\ n(n-1)!, & n>0\end{cases}$$

阶乘函数的自变量 n 的定义域是非负整数。递归式的第一式给出了这个函数的初始值，是非递归定义的。每个递归函数都必须有非递归定义的初始值，否则，递归函数就无法计算。递归式的第二式是用较小自变量的函数值来表达较大自变量的函数值的方式来定义 $n!$。定义式的左右两边都引用了阶乘记号，是递归定义式，可递归地计算如下。

```
int factorial(int n)
{
   if (n==0) return 1;
   return n*factorial(n-1);
}
```

2．Fibonacci 数列

无穷数列 1,1,2,3,5,8,13,21,34,55…称为 Fibonacci 数列。它可以递归地定义为

$$F(n)=\begin{cases}1, & n=0\\ 1, & n=1\\ F(n-1)+F(n-2), & n>1\end{cases}$$

这是一个递归关系式，它说明当$n>1$时，这个数列的第 n 项的值是它前面两项之和。它用两个较小的自变量的函数值来定义较大自变量的函数值，所以需要两个初始值 $F(0)$和 $F(1)$。

第 n 个 Fibonacci 数可递归地计算如下。

```
1  int fibonacci(int n)
2  {
3     if (n<=1) return 1;
4     return fibonacci(n-1)+fibonacci(n-2);
5  }
```

上述两例中的函数也可用如下非递归方式定义：

$$n!=1\times 2\times 3\times\cdots\times(n-1)\times n$$

$$F(n)=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$$

在实现递归算法时，用递归调用函数通常比较耗时，且占用较多空间。例如，用上面描述的递归算法 fibonacci 来计算 Fibonacci 数列就需要耗费指数级 $O(F(n))$ 时间和系统堆栈空间。这是由于在递归调用时重复计算了许多子问题，浪费了计算资源。解决这一问题的有效方法是采用动态规划算法。其基本思想是将计算过的子问题的解保存在一个数组中。在需要子问题的解时，只要从数组中取出，而不必重新计算。这样每个子问题只要解 1 次就可以了，从而节省了计算时间和空间。以计算 Fibonacci 数列为例，用数组 f 来存储子问题的解，相应的动态规划算法如下。

```
1  int fibonacci(int n)
2  {
3     f[0]=f[1]=1;
4     for(int i=2;i<=n;i++)f[i]=f[i-1]+f[i-2];
5     return f[n];
6  }
```

这个算法只需要 $O(n)$ 计算时间和空间。注意，算法实际上只要保留前 2 个子问题的解就足够了。用数组滚动技术可以将算法的空间需求进一步减少到 $O(1)$ 。

```
1  int fibonacci(int n)
2  {
3     f[0]=f[1]=1;
4     while(--n)f[n&1]=f[0]+f[1];
5     return f[1];
6  }
```

3. 排列问题

设 $R=\{r_1,r_2,\cdots,r_n\}$ 是要进行排列的 n 个元素， $R_i=R-\{r_i\}$ 。R 中元素的全排列记为

perm(R)。(r_i)perm(R)表示在全排列 perm(R)的每一个排列前加上前缀 r_i 得到的排列。R 的全排列可归纳定义如下：

当 $n=1$ 时，perm(R)=(r)，其中 r 是集合 R 中唯一的元素；

当 $n>1$ 时，perm(R)由(r_1)perm(R_1),(r_2)perm(R_2),…,(r_n)perm(R_n)构成。

依此递归定义，可设计产生全排列的递归算法如下。

```
1   void perm(int list[], int k, int m)
2   {/* 产生list[k:m]的所有排列 */
3     if(k==m){/* 已排定 */
4       for (int i=0;i<=m;i++) printf ("%d ",list[i]);
5       printf ("\n");
6     }
7     else /* 递归产生排列 */
8       for(int i=k;i<=m;i++){
9         swap(&list[k],&list[i]);
10        perm(list,k+1,m);
11        swap(&list[k],&list[i]);
12      }
13  }
```

算法 perm(list,k,m)递归地产生所有前缀是 list[0:k−1]，后缀是 list[k:m]的全排列的所有排列，调用算法 perm(list,0,n−1)则产生 list[0:n−1]的全排列。

在一般情况下，$k<m$。算法将 list[k:m]中每一个元素分别与 list[k]中元素交换。然后递归地计算 list[k+1:m]的全排列，并将计算结果作为 list[0:k]的后缀。算法中的函数 swap 用于交换两个表元素值。

1.6.2 间接递归

前面介绍的递归函数都是直接调用其自身，这类递归函数称为直接递归函数。间接递归函数通过调用别的函数间接地调用其自身。

例如，计算正弦和余弦函数的递归式为

$$\sin 2\theta = 2\sin\theta\cos\theta$$

$$\cos 2\theta = 1-2\sin^2\theta$$

当 x 充分小时，可以用泰勒展开式计算

$$\sin x = x - \frac{1}{6}x^3$$

$$\cos x = 1 - \frac{1}{2}x^2$$

由此可以设计计算 sin$x$ 和 cos$x$ 的间接递归函数如下。

```
1   double s(double x)
2   {
3     if (-0.005<x && x<0.005) return x-x*x*x/6;
4     return 2*s(x/2)*c(x/2);
5   }
6
```

```
7   double c(double x)
8   {
9     if (-0.005<x && x<0.005) return 1.0-x*x/2;
10    return 1.0-2*s(x/2)*s(x/2);
11  }
```

本 章 小 结

本章介绍了算法的基本概念、表达算法的抽象机制，以及算法的计算复杂性概念和分析方法。简要阐述了数据类型、数据结构和抽象数据类型的基本概念，以及这 3 个重要概念的区别和内在联系。概述了 C 语言的若干重要特性和采用 C 语言与自然语言相结合的方式描述算法的方法。最后介绍了递归的概念，以及递归在数据结构和算法设计中的应用。本章内容是后续各章叙述算法和描述数据结构的基础和准备。

习 题 1

1.1 试列举在 C 语言编程环境中下列基本数据类型变量能表示的最大数和最小数。

（1）int；（2）long int；（3）short int；（4）float；（5）double。

1.2 什么是抽象数据类型？试述抽象数据类型与数据结构的区别和联系。

1.3 试用 C 语言的结构类型定义表示复数的抽象数据类型。

（1）在复数内部用浮点数定义其实部和虚部。

（2）设计实现复数的 +，−，*，/ 等运算的函数。

1.4 求下列函数尽可能简单的渐近表达式：

$3n^2+10n$； $\frac{1}{10}n^2+2^n$； $21+\frac{1}{n}$； $\log n^3$； $10\log 3^n$。

1.5 试述 $O(1)$ 和 $O(2)$ 的区别。

1.6 说明下列各表达式当 n 在什么范围内取值时效率最高。

$4n^2$， $\log n$， 3^n， $20n$， 2， $n^{2/3}$。

1.7 按照渐近阶从低到高的顺序排列以下表达式：$4n^2$， $\log n$， 3^n， $20n$， 2， $n^{2/3}$。又 $n!$ 应该排在哪一位?

1.8 （1）假设某算法在输入规模为 n 时的计算时间为 $T(n)=3\times 2^n$。在某台计算机上实现并完成该算法的时间为 t 秒。现有另一台计算机，其运行速度为第一台的 64 倍，那么在这台新机器上用同一算法在 t 秒内能解输入规模为多大的问题?

（2）若上述算法的计算时间改进为 $T(n)=n^2$，其余条件不变，则在新机器上用 t 秒时间能解输入规模为多大的问题?

（3）若上述算法的计算时间进一步改进为 $T(n)=8$，其余条件不变，则在新机器上用 t 秒时间能解输入规模为多大的问题?

1.9 硬件厂商 XYZ 公司宣称他们最新研制的微处理器运行速度为其竞争对手 ABC 公司同类产品的 100 倍。对于计算复杂性分别为 n, n^2, n^3 和 $n!$ 的各算法，如果用 ABC 公司的计算机在 1 小时内能解输入规模为 n 的问题，那么用 XYZ 公司的计算机在 1 小时内能解输入规模为多大

的问题？

1.10　对于下列各组函数 $f(n)$ 和 $g(n)$，确定 $f(n)=O(g(n))$，$f(n)=\Omega(g(n))$ 还是 $f(n)=\theta(g(n))$？并简述理由。

（1）$f(n)=\log n^2$；$g(n)=\log n+5$

（2）$f(n)=\log n^2$；$g(n)=\sqrt{n}$

（3）$f(n)=n$；$g(n)=\log^2 n$

（4）$f(n)=n\log n+n$；$g(n)=\log n$

（5）$f(n)=10$；$g(n)=\log 10$

（6）$f(n)=\log^2 n$；$g(n)=\log n$

（7）$f(n)=2^n$；$g(n)=100n^2$

（8）$f(n)=2^n$；$g(n)=3^n$

1.11　证明：若一个算法在平均情况下的时间复杂性为 $\theta(f(n))$，则该算法在最坏情况下所需的计算时间为 $\Omega(f(n))$。

1.12　证明：可以在 $O(\log n)$ 时间内计算 $n!$ 和 $F(n)$ 的值。

算法实验题 1

算法实验题 1.1　哥德巴赫猜想问题。

★ 问题描述：哥德巴赫猜想：任何大偶数均可表示为 2 个素数之和。

★ 实验任务：验证哥德巴赫猜想。计算给定的大偶数可以表示为多少对素数之和。例如，大偶数 10 可以表示为 2 对素数 3，7 或 5，5 之和。

★ 数据输入：由文件 input.txt 给出输入数据。每行有 1 个大偶数，文件以数字 0 结尾。

★ 结果输出：将计算出的相应的素数分解数输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
4	1
6	1
8	1
10	2

算法实验题 1.2　连续整数和问题。

★ 问题描述：大部分的正整数可以表示为 2 个以上连续整数之和。如 6=1+2+3，9=5+4=2+3+4。

★ 实验任务：连续整数和问题要求计算给定的正整数可以表示为多少个 2 个以上连续整数之和。

★ 数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数。

★ 结果输出：将计算出的相应的连续整数分解数输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
9	2

算法实验题 1.3　随机决策森林问题。

★ 问题描述：在机器学习算法中，随机决策森林是一个包含多个互相独立决策树的分类器，并且其输出的类别是由个别决策树输出的类别的众数而定的。假定有 n 棵互相独立的决策树，它们都能预测抛一枚硬币的结果是正面朝上还是反面朝上，并且预测准确率都是 81%。考虑如下的随机森林集成算法：如果这 n 棵独立的决策树中有超过半数是预测正面，则算法预测正面，否则算法预测反面。如此可以计算出当 $n=3$ 时，随机森林集成算法预测正确的概率是90.54%。相比于单一决策树模型，随机森林集成算法显著地提高了预测准确率。事实上，只要$n\geqslant 3$，随机森林集成算法的预测准确率都高于 90%。

在随机决策森林问题中，给定单棵独立的决策树预测的准确率 p 和随机森林集成算法希望达到的预测准确率 q，计算至少需要多少棵互相独立的决策树组成随机决策森林才能使随机森林集成算法的预测准确率高于 q。

★ 实验任务：对于给定单棵独立的决策树预测的准确率 p 和随机森林集成算法希望达到的预测准确率 q，设计一个算法来计算至少需要多少棵互相独立的决策树组成随机决策森林才能使随机森林集成算法的预测准确率高于 q。

★ 数据输入：由文件 input.txt 给出输入数据。有 k 组测试数据（$1\leqslant k\leqslant 10$），每行给出一组测试数据。每组测试数据由 2 个实数 p 和 q 组成。其中，p 是单棵独立的决策树预测的准确率，q 是随机森林集成算法希望达到的预测准确率。

★ 结果输出：将计算结果输出到文件 output.txt 中。依次输出各组测试数据至少需要的决策树棵数。每行输出一个数。若不存在满足要求的随机决策森林则输出−1。

输入文件示例	输出文件示例
input.txt	output.txt
0.81 0.90	3
0.60 0.90	41
0.40 0.40	1
0.90 0.80	1
0.49 0.50	−1

算法实验题 1.4　与 1 共舞数字问题。

★ 问题描述：若将正整数用二进制数来表示，则可以将它看成由 0 和 1 组成的字符串。在一个长度为 n 的 0-1 字符串中，若每个 0 的左侧必有 1 个 1 与其相邻，则称它所表示的数字为与 1 共舞数字。

例如，当 n=1 时，长度为 1 的 0-1 字符串有：0 和 1。只有 1 是与 1 共舞数字。当 n=3 时，长度为 3 的 0-1 字符串有：000，001，010，011，100，101，110 和 111。其中，101，110 和 111 是与 1 共舞数字。

★ 实验任务：对于给定正整数 n，计算长度为 n 的 0-1 字符串中有多少个与 1 共舞数字。

★ 数据输入：由文件 input.txt 给出输入数据。有 k 组测试数据（$1\leqslant k\leqslant 10\,000$），每行给出一个正整数 n。

★ 结果输出：将计算结果输出到文件 output.txt 中。依次输出各组给定正整数 n 的与 1 共舞数字个数 mod 10^9。

输入文件示例	输出文件示例
input.txt	output.txt
1	1
3	3