第 1 章　算法概述

学习要点

- 理解算法的概念
- 掌握算法在最坏情况、最好情况和平均情况下的计算复杂性概念
- 掌握算法复杂性的渐近性态的数学表述
- 了解 NP 类问题的基本概念

1.1　算法与程序

对于计算机科学来说，算法（Algorithm）的概念至关重要。例如，在一个大型软件系统的开发中，设计出有效的算法将起决定性的作用。通俗地讲，算法是指解决问题的一种方法或一个过程。更严格地讲，算法是由若干条指令组成的有穷序列，且满足下述 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，操作系统的各种任务可以看成一些单独的问题，每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。

描述算法可以有多种方式，如自然语言方式、表格方式等。本书采用 C++语言描述算法。C++语言的优点是类型丰富、语句精炼，具有面向过程和面向对象的双重特点。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。

1.2　算法复杂性分析

算法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，所需资源越多，该算法的复杂性越高；反之，所需资源越少，该算法的复杂性越低。对计算机资源，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。

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，选择复杂性最低者是选用算法时遵循的一个重要准则。因此，算法的复杂性分析对算法的设计或选用有着重要的指导意义和实用价值。

更确切地说，算法的复杂性是算法运行需要的计算机资源的量，需要时间资源的量称为时间复杂性，需要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，并从运

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要解的问题的规模、算法的输入和算法本身的函数。如果分别用 N、I 和 A 表示算法要解的问题的规模、算法的输入和算法本身，而且用 C 表示复杂性，那么应该有 $C=F(N, I, A)$，其中 $F(N, I, A)$是一个由 N、I 和 A 确定的三元函数。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，应该有 $T=T(N, I, A)$和 $S=S(N, I, A)$。通常，A 隐含在复杂性函数名当中，因而将 T 和 S 分别简写为 $T=T(N, I)$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，因此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、I 和 A，如何导出 $T(N, I)$和 $S(N, I)$的数学表达式，从而给出计算 $T(N, I)$和 $S(N, I)$的法则。下面以 $T(N, I)$为例，将复杂性函数具体化。

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

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

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 $T(N, I)$的表达式需要进一步简化，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（$i=1, 2, \cdots, k$）来评价其时间复杂性。

本书只考虑三种情况下的时间复杂性，即最坏情况、最好情况和平均情况下的时间复杂性，分别记为 $T_{\max}(N)$、$T_{\min}(N)$和 $T_{\text{avg}}(N)$。在数学上有

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

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

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

式中，D_N 是规模为 N 的合法输入的集合；I^* 是 D_N 中使 $T(N, I^*)$达到 $T_{\max}(N)$的合法输入；$\tilde{I}$ 是 D_{N} 中使 $T(N, \tilde{I})$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某个角度反映算法的效率，各有局限性，各有用处。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。

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。在此要引入复杂性渐近性态的概念。

设 $T(N)$是前面定义的关于算法 A 的复杂性函数。当 N 单调增大且趋于∞时，$T(N)$一般也将单调增大且趋于∞。对于 $T(N)$，如果存在 $\tilde{T}(N)$，当 $N\to\infty$时，使得 $(T(N)-\tilde{T}(N))/T(N)\to 0$，就说 $\tilde{T}(N)$是 $T(N)$当 $N\to\infty$时的渐近性态，或称 $\tilde{T}(N)$为算法 A 当 $N\to\infty$的渐近复杂性，而与 $T(N)$相区别。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 $T(N)$

中略去低阶项留下的主项，所以比 $T(N)$简单。比如，当 $T(N)=3N^2+4N\log N+7$ 时，$\tilde{T}(N)$的一个答案是 $3N^2$，因为这时有

$$(T(N)-\tilde{T}(N))/T(N)=\frac{4N\log N+7}{3N^2+4N\log N+7}\to 0\text{（当 }N\to\infty\text{时）}$$

显然，$3N^2$ 比 $3N^2+4N\log N+7$ 简单得多。

当 $N\to\infty$时，$T(N)$渐近于 $\tilde{T}(N)$，我们有理由用 $\tilde{T}(N)$替代 $T(N)$作为算法 A 在 $N\to\infty$时的复杂性的度量。由于 $\tilde{T}(N)$明显比 $T(N)$简单，这种替代则是对复杂性分析的一种简化。进一步考虑到，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，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 $\tilde{T}(N)$的阶就足够了，不必关心包含在 $\tilde{T}(N)$中的常数因子。所以，常常对 $\tilde{T}(N)$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。

上面给出了简化算法复杂性分析的方法和步骤，即只要考察当问题的规模充分大时，算法复杂性在渐近意义下的阶。为了与此简化的复杂性分析相匹配，需要引入以下渐近意义下的记号 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。如果存在正的常数 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 $f(N)=O(g(N))$。这时还说 $f(N)$的阶不高于 $g(N)$的阶。例如：

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 $2N^2+11N-10=O(N^2)$。

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

⑤ 作为一个反例，$N^3\neq O(N^2)$。因为若不然，则存在正的常数 C 和自然数 N_0，使得当 $N\geqslant N_0$ 时有 $N^3\leqslant CN^2$，即 $N\leqslant C$。显然，当取 $N=\max\{N_0,\lfloor\sqrt{C}\rfloor+1\}$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 $O(f)+O(g)=O(\max(f,g))$。

② $O(f)+O(g)=O(f+g)$。

③ $O(f)O(g)=O(fg)$。

④ 如果 $g(N)=O(f(N))$，则 $O(f)+O(g)=O(f)$。

⑤ $O(Cf(N))=O(f(N))$，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 $F(N)\leqslant C_1f(N)$。

类似地，设 $G(N)=O(g)$，则存在正的常数 C_2 和自然数 N_2，使得对所有的 $N\geqslant N_2$，有 $G(N)\leqslant C_2g(N)$。

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

$$F(N)\leqslant C_1f(N)\leqslant C_1h(N)\leqslant C_3h(N)$$

类似地，有

$$G(N)\leqslant C_2f(N)\leqslant C_2h(N)\leqslant C_3h(N)$$

因而 $$O(f)+O(g)=F(N)+G(N)\leqslant C_3h(N)+C_3h(N)$$
$$=2C_3h(N)=O(h)=O(\max(f,g))$$

其余规则的证明类似，留给读者作为练习。

应该指出，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，则评估越精确，结果就越有价值。

关于符号 Ω，文献里有两种定义。本书只采用其中的一种，定义如下：如果存在正的常数 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 $f(N)=\Omega(g(N))$。这时还说 $f(N)$的阶不低于 $g(N)$的阶。Ω 的这个定义的优点是与 O 的定义对称，缺点是当 $f(N)$对自然数的不同无穷子集有不同的表达式，且有不同的阶时，不能很好地刻画 $f(N)$的下界。比如，当

$$f(N)=\begin{cases}100 & N\text{为正偶数}\\ 6N^2 & N\text{为正奇数}\end{cases}$$

时，按上述定义，只能得到 $f(N)=\Omega(1)$，这是一个平凡的下界，对算法分析没有什么价值。然而，考虑到上述定义有与符号 O 定义的对称性，同时本书介绍的算法都没出现上例中的那种情况，所以本书还是选用它。

同样，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，则评估越精确，结果就越有价值。再则，这里的 Ω 只对问题的一个算法而言。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。

定义 $f(N)=\theta(g,(N))$当且仅当 $f(N)=O(g(N))$且 $f(N)=\Omega(g(N))$时，称为 $f(N)$与 $g(N)$同阶。

最后，如果对于任意给定的 $\varepsilon>0$，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 $g(N)$低，记为 $f(N)=o(g(N))$。例如：

$$4N\log N+7=o(3N^2+4N\log N+7)$$

1.3 NP 完全性理论

在计算机算法理论中，最深刻的问题之一是“从计算的观点来看，要解决的问题的内在复杂性如何？”它是“易”计算的还是“难”计算的？如果知道了一个问题的计算时间下界，就知道了对于该问题能设计出多有效的算法，从而可以较正确地评价对该问题提出的各种算法的效率，并进而确定对已有算法还有多少改进的余地。在许多情况下，要确定一个问题的内在计算复杂性是很困难的。已创造出的各种分析问题计算复杂性的方法和工具可以较准确地确定许多问题的计算复杂性。

问题的计算复杂性可以通过解决该问题所需计算量的多少来度量。如何区分一个问题是“易”还是“难”呢？人们通常将可在多项式时间内解决的问题看作“易”解问题，而将需要指数函数时间解决的问题看作“难”问题。这里所说的多项式时间和指数函数时间是针对问题的规模而言的，即解决问题所需的时间是问题规模的多项式函数或指数函数。对于实际遇到的许多问题，人们至今无法确切了解其内在的计算复杂性，因此只能用分类的方法将计

算复杂性大致相同的问题归类进行研究。而对于能够进行较彻底分析的问题则尽可能准确地确定其计算复杂性，从而获得对它的深刻理解。

本书中的许多算法都是多项式时间算法，即对规模为 n 的输入，算法在最坏情况下的计算时间为 $O(n^k)$，k 为一个常数。是否所有的问题都在多项式时间内可解呢？回答是否定的，如一些不可解问题（像著名的“图灵停机问题”）。任何计算机不论耗费多少时间也不能解该问题。有些问题虽然可以用计算机求解，但是对任意常数 k，它们都不能在 $O(n^k)$的时间内得到解答。

一般，将可由多项式时间算法求解的问题看作易解的问题，将需要超多项式时间才能求解的问题看作难解的问题。有许多问题从表面上看似乎并不比排序或图的搜索等问题更困难，然而至今人们还没有找到解决这些问题的多项式时间算法，也没有人能够证明这些问题需要超多项式时间下界。也就是说，这类问题的计算复杂性至今未知。为了研究这类问题的计算复杂性，人们提出了非确定性图灵机计算模型，使得许多问题可以在多项式时间内求解。

本书中讨论的许多问题是以最优化问题形式出现的，如旅行售货员问题、0-1 背包问题和最大团问题等。然而对每个最优化问题，都有一个与之对应的判定问题。第 5 章中要讨论的旅行售货员问题是一个典型的最优化问题。

最优化形式的旅行售货员问题可用图论语言形式描述如下。

设 $G=(V, E)$是一个带权图，图中各边的费用（权）为正数。图的一条周游路线是包括 V 中的每个顶点在内的一条回路。周游路线的费用是这条路线上所有边的费用之和。旅行售货员问题要在图 G 中找出费用最小的周游路线。

与之对应的**判定形式的旅行售货员问题**可描述如下。

对于给定的带权图 $G=(V, E)$和一个正数 d。判定形式的旅行售货员问题要求判定图 G 中是否存在总费用不超过 d 的周游路线。

在一般情况下，判定问题比相应的最优化问题多一个输入参数 d。从直观上看，判定问题要比相应的最优化问题容易求解。从一个最优化问题的多项式时间算法容易得到与之相应的判定问题的多项式时间算法。

所有可以在多项式时间内求解的判定问题构成 P 类问题（Polynomial Problem，多项式问题）。在通常情况下，解一个问题要比验证问题的一个解困难得多，特别在有时间限制的条件下更是如此。P 类问题是确定性计算模型下的易解问题类，而 NP 类问题是非确定性计算模型下的易验证问题类。

为了说明什么是 NP 类问题（Non-deterministic Polynomial Problem，非确定性多项式问题），需要引入非确定性算法的概念。非确定性算法将问题求解分为猜测和验证两个阶段。算法的猜测阶段是非确定性的，给出问题解的一个猜测。算法的验证阶段是确定性的，验证猜测阶段给出的解的正确性。设算法 A 是解一个判定问题 Q 的非确定性算法。如果算法 A 的验证阶段可以在多项式时间内完成，则称算法 A 是一个多项式时间非确定性算法，也称问题 Q 是非确定性多项式时间可解的。

所有非确定性多项式时间可解的判定问题构成 NP 类问题。例如，对于判定形式的旅行售货员问题，容易在多项式时间内验证其解的正确性，因此旅行售货员问题属于 NP 类。

从 P 类和 NP 类问题的定义容易看出，$P \subseteq NP$。反之，大多数的计算机科学家认为，NP 类中包含了不属于 P 类的问题，即 $P \neq NP$，但这个问题至今没有获得明确的解答。也许使大

多数计算机科学家相信 P≠NP 的最令人信服的理由是，存在一类 NP 完全问题，即 NPC 类问题。这类问题有一种令人惊奇的性质，即如果一个 NP 完全问题能在多项式时间内得到解决，那么 NP 中的每个问题都可以在多项式时间内求解，即 P=NP。尽管已进行多年研究，目前还没有一个 NP 完全问题有多项式时间算法。

获得“第一个 NP 完全问题”称号的是布尔表达式的可满足性问题，就是著名的 Cook 定理：**布尔表达式的可满足性问题 SAT 是 NP 完全的**。

Cook 定理的重要性是明显的，给出了第一个 NP 完全问题。这使得对于任何问题 Q，只要能证明 Q∈NP，而且可以在多项式时间内将 SAT 变换为问题 Q，便有 Q∈NPC。所以，人们很快证明了许多其他问题的 NP 完全性。这些 NP 完全问题都是直接或间接地以 SAT 的 NP 完全性为基础而得到证明的，由此逐渐生长出一棵以 SAT 为树根的 NP 完全问题树。其中，每个结点代表一个 NP 完全问题，该问题可在多项式时间内变换为它的任一儿子结点表示的问题。实际上，由树的连通性及多项式在复合变换下的封闭性可知，NP 完全问题树中任一结点表示的问题可以在多项式时间内变换为它的任一后裔结点表示的问题。目前，这棵 NP 完全问题树上已有几千个结点，还在继续生长。

下面介绍这棵 NP 完全树中的几个典型的 NP 完全问题。

1．合取范式的可满足性问题 CNF-SAT

给定一个合取范式 α，判定它是否可满足。如果一个布尔表达式是一些因子和之积，则称它为合取范式，简称 CNF（Conjunctive Normal Form）。这里的因子是变量 x 或 $\bar{x}$。例如，$(x_1+x_2)(x_2+x_3)(\bar{x}_1+\bar{x}_2+x_3)$是一个合取范式，而 $x_1x_2+x_3$ 不是合取范式。

2．三元合取范式的可满足性问题 3-SAT

给定一个三元合取范式 α，判定它是否可满足。

3．团问题 CLIQUE

给定一个无向图 $G=(V, E)$和一个正整数 k，判定图 G 是否包含一个 k 团，即是否存在 $V'\subseteq V$ 和$|V'|=k$，且对任意 $u, w\in V'$，有$(u, w)\in E$。

4．顶点覆盖问题 VERTEX-COVER

给定一个无向图 $G=(V, E)$和一个正整数 k，判定是否存在 $V'\subseteq V$ 和$|V'|=k$，使得对任意 $(u, v)\in E$ 有 $u\in V'$或 $v\in V'$，如果存在，就称 V'为图 G 的一个大小为 k 的顶点覆盖。

5．子集和问题 SUBSET-SUM

给定整数集合 S 和一个整数 t，判定是否存在 S 的一个子集 $S'\subseteq S$，使得 S'中整数的和为 t。例如，若 S={1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344}且 t=3754，则子集 S'={1, 16, 64, 256, 1040, 1093, 1284}是它的一个解。

6．哈密顿回路问题 HAM-CYCLE

给定无向图 $G=(V, E)$，判定其是否含有一条哈密顿回路。

7．旅行售货员问题 TSP

给定一个无向完全图 $G=(V, E)$及定义在 $V\times V$ 上的一个费用函数 c 和一个整数 k，判定 G

是否存在经过 V 中各顶点恰好一次的回路，使得该回路的费用不超过 k。

算法分析题 1

1-1 求下列函数的渐近表达式：

$3n^2+10n$；　　$n^2/10+2^n$；　　$21+1/n$；　　$\log n^3$；　　$10\log 3^n$

1-2 试论 $O(1)$和 $O(2)$的区别。

1-3 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？

1-4 （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，那么在新机器上用 t 秒时间能解输入规模为多大的问题？

1-5 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，若用 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 小时内分别能解输入规模为多大的问题？

1-6 对于下列各组函数 $f(n)$和 $g(n)$，确定 $f(n)=O(g(n))$或 $f(n)=\Omega(g(n))$或 $f(n)=\theta(g(n))$，并简述理由。

（1）$f(n)=\log n^2$；　$g(n)=\log n+5$　　（5）$f(n)=10$；　$g(n)=\log 10$

（2）$f(n)=\log n^2$；　$g(n)=\sqrt{n}$　　（6）$f(n)=\log^2 n$；　$g(n)=\log n$

（3）$f(n)=n$；　$g(n)=\log^2 n$　　（7）$f(n)=2^n$；　$g(n)=100n^2$

（4）$f(n)=n\log n+n$；　$g(n)=\log n$　　（8）$f(n)=2^n$；　$g(n)=3^n$

1-7 证明 $n!=o(n^n)$。

1-8 下面的算法段用于确定 n 的初始值。试分析该算法段所需计算时间的上界和下界。

```
while(n>1)
    if(odd(n))
        n = 3*n+1;
    else
        n = n/2;
```

1-9 证明：如果一个算法在平均情况下的计算时间复杂性为 $\theta(f(n))$，则该算法在最坏情况下所需的计算时间为 $\Omega(f(n))$。

算法实现题 1

1-1 统计数字问题。

问题描述：一本书的页码从自然数 1 开始顺序编码直到自然数 n。书的页码按照通常的习

惯编排，每个页码都不含多余的前导数字 0。例如，第 6 页用数字 6 表示而不是 06 或 006 等。数字计数问题要求对给定书的总页码 n，计算书的全部页码分别用到多少次数字 0, 1, 2, …, 9。

算法设计：给定表示书的总页码的十进制整数 n（$1 \leqslant n \leqslant 10^9$），计算书的全部页码中分别用到多少次数字 0, 1, 2, …, 9。

数据输入：输入数据由文件名为 input.txt 的文本文件提供。每个文件只有 1 行，给出表示书的总页码的整数 n。

结果输出：将计算结果输出到文件 output.txt。输出文件共 10 行，在第 k（k=1, 2, …, 10）行输出页码中用到数字 k–1 的次数。

输入文件示例	输出文件示例
input.txt	output.txt
11	1
	4
	1
	1
	1
	1
	1
	1
	1
	1

1-2　字典序问题。

问题描述：在数据加密和数据压缩中常需要对特殊的字符串进行编码。给定的字母表 A 由 26 个小写英文字母组成，即 A={a, b, …, z}。该字母表产生的升序字符串是指字符串中字母从左到右出现的次序与字母在字母表中出现的次序相同，且每个字符最多出现 1 次。例如，a、b、ab、bc、xyz 等字符串都是升序字符串。现在对字母表 A 产生的所有长度不超过 6 的升序字符串按照字典序排列并编码如下。

1	2	…	26	27	28	…
a	b	…	z	ab	ac	…

对于任意长度不超过 6 的升序字符串，迅速计算出它在上述字典中的编码。

算法设计：对于给定的长度不超过 6 的升序字符串，计算它在上述字典中的编码。

数据输入：输入数据由文件名为 input.txt 的文本文件提供。文件的第 1 行是一个正整数 k，表示接下来有 k 行。在接下来的 k 行中，每行给出一个字符串。

结果输出：将计算结果输出到文件 output.txt。文件有 k 行，每行对应一个字符串的编码。

输入文件示例	输出文件示例
input.txt	output.txt
2	1
a	2
b	

1-3　最多约数问题。

问题描述：正整数 x 的约数是能整除 x 的正整数。正整数 x 的约数个数记为 div(x)。例如，1、2、5、10 都是正整数 10 的约数，且 div(10)=4。设 a 和 b 是 2 个正整数，$a \leqslant b$，找出 a 和 b 之间约数个数最多的数 x。

算法设计：对于给定的 2 个正整数 $a \leqslant b$，计算 a 和 b 之间约数个数最多的数。

数据输入：输入数据由文件名为 input.txt 的文本文件提供。文件的第 1 行有 2 个正整数 a 和 b。

结果输出：若找到的 a 和 b 之间约数个数最多的数是 x，则将 div(x)输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
1 36	9

1-4　金币阵列问题。

问题描述：有 $m \times n$（$m \leqslant 100$，$n \leqslant 100$）枚金币在桌面上排成一个 m 行 n 列的金币阵列。每枚金币或正面朝上或背面朝上。用数字表示金币状态，0 表示金币正面朝上，1 表示金币背面朝上。

金币阵列游戏的规则是：① 每次可将任一行金币翻过来放在原来的位置上；② 每次可任选 2 列，交换这 2 列金币的位置。

算法设计：给定金币阵列的初始状态和目标状态，计算按金币游戏规则，将金币阵列从初始状态变换到目标状态所需的最少变换次数。

数据输入：由文件 input.txt 给出输入数据。文件中有多组数据。文件的第 1 行有 1 个正整数 k，表示有 k 组数据。每组数据的第 1 行有 2 个正整数 m 和 n。以下 m 行是金币阵列的初始状态，每行有 n 个数字表示该行金币的状态，0 表示正面朝上，1 表示背面朝上。接着的 m 行是金币阵列的目标状态。

结果输出：将计算出的最少变换次数按照输入数据的次序输出到文件 output.txt。相应数据无解时，输出−1。

输入文件示例	输出文件示例
input.txt	output.txt
2	2
4 3	−1
1 0 1	
0 0 0	
1 1 0	
1 0 1	
1 0 1	
1 1 1	
0 1 1	
1 0 1	
4 3	
1 0 1	
0 0 0	
1 0 0	
1 1 1	
1 1 0	
1 1 1	
0 1 1	
1 0 1	

1-5　最大间隙问题。

问题描述：最大间隙问题：给定 n 个实数 $x_1, x_2, \cdots, x_n$，求这 n 个数在实轴上相邻两个数之间的最大差值。假设对任何实数的下取整函数耗时 $O(1)$，设计解最大间隙问题的线性时间算法。

算法设计：对于给定的 n 个实数 $x_1, x_2, \cdots, x_n$，计算它们的最大间隙。

数据输入：输入数据由文件名为 input.txt 的文本文件提供。文件的第 1 行有 1 个正整数 n。接下来的 1 行中有 n 个实数 $x_1, x_2, \cdots, x_n$。

结果输出：将找到的最大间隙输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5	3.2
2.3 3.1 7.5 1.5 6.3	