第 1 部分　算法基础

本部分详细介绍算法设计与分析领域的经典技术、理论和方法，内容主要包括算法概述、递归与分治法、动态规划法、贪心算法、回溯法、分支限界法等。针对每种算法，均设计了应用实例，并按照问题提出、算法设计、算法实现（Java 语言）和算法复杂性分析的顺序进行了细致的讲解，讲解过程中涉及的算法推理和演算均配置了图解说明，以帮助读者清晰地掌握算法的设计思路与技巧。

第 1 章　算法概述

计算机操作系统、数据库系统及各种应用软件都是由一系列算法实现的。算法是解决某种问题的方法，是由若干指令组成的有穷序列。例如，设有一个问题，在三个数 a, b, c 中寻找最大的一个。可以设计如下算法：

第 1 步，设 $x = a$。

第 2 步，若 $b > x$，则 $x = b$。

第 3 步，若 $c > x$，则 $x = c$。

这个算法的思路就是顺序查看这三个数，把最大值复制到变量 x 中。算法结束时，x 中保存的是这三个数中的最大值。

上述例子只是一个求解很简单的问题的算法。然而，无论是求解简单问题的算法，还是求解复杂问题的算法，一般而言，都具有以下性质：

1）输入。算法可以有零个或多个外部量作为输入。大多数算法需要有外部量作为输入，例如上述算法就是如此；但有时也可以没有输入，如求 $1+2+\cdots+100$ 或求 100 以内的质数等。

2）输出。算法至少要产生一个量作为输出。例如，上述例子输出三个数中的最大值。算法是用来解决某种问题的方法，如果没有输出，那么这样的算法毫无意义。

3）确定性。确定性是指组成算法的每条指令需要清晰、无歧义。

4）有限性。有限性是指算法必须在执行有限步后能够停止，且执行每步指令的时间也要有限。例如，操作系统就不是算法，因为其在理论上可以无限时地运行。

为了让算法清晰易懂，需要选择一种好的描述方法。算法的描述方法有多种，如伪代码、自然语言、计算机语言等。

1）伪代码（Pseudocode）。伪代码是介于自然语言和计算机语言之间的文字和符号。它以编程语言的书写形式指明算法的职能，但描述时又不需要真正遵循程序设计语言的语法规则。因此，伪代码在算法描述方面具有较大的灵活性。

2）自然语言。自然语言算法描述就是用人们日常使用的语言描述求解问题的方法和步骤，是一种非形式化描述方法。用自然语言进行算法描述的优点是非常接近人类的思维习惯，即使不熟悉计算机语言的人也很容易理解算法的思想。然而，这种算法描述方法也有缺点，具体体现为自然语言在语法和语义上往往具有多义性，

描述烦琐，对程序流向的描述也不够直观、明了。

3）计算机语言（Computer Language）。无论采用何种描述方式，算法若要最终在计算机中执行，都需要转换为相应的计算机语言程序。因此，算法可以直接以计算机语言进行描述，如 C、C++、Java、Python 等计算机语言。

以上讲述了算法的定义、性质及描述方法。接下来介绍算法复杂性。算法复杂性是指算法运行所需的计算机资源的量。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。需要的资源越多，算法的复杂性越高；反之，算法的复杂性越低。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算法设计追求的主要目标。当给定的问题已有多种算法时，选择其中复杂性最低者，是在选用算法时需要遵循的重要准则。因此，算法的复杂性对算法设计及选用有重要的指导意义和实用价值。

需要注意的是，算法的复杂性并不能用具体的运算时间去衡量。因为算法的运行速度不仅取决于算法是怎样实现的，而且取决于算法的运算环境。不同性能的计算机，其运算环境有巨大的差别，同一个算法在高性能计算机上运行与在普通 PC 上运行，显然会有不同的执行时间；同一个算法用不同的编程语言编写，也会有不同的运算时间；即使是用同一种编程语言编写的同一个算法，用不同的编译系统实现时，也可能会有不同的运算速度。显然，并不能用具体的运算时间去衡量算法的复杂性。那么到底该如何衡量算法复杂性呢？既然不能用具体的量，那么衡量算法复杂性时显然必须用一个能从运行该算法的实际计算机中脱离出来的抽象的量。规定这个抽象的量只与三个因素有关：算法要求解的问题的规模、算法的输入和算法本身。

如果分别用 N, I 和 A 表示算法要求解问题的规模、算法的输入和算法本身，用 C 表示复杂性，那么有 $C=F(N,I,A)$，其中，$F(N,I,A)$ 是 N,I,A 的确定的三元函数。

一般把时间复杂性和空间复杂性分开，并分别用 T 和 S 来表示，于是有 $T=T(N,I,A)$ 和 $S=S(N,I,A)$。通常，让 A 隐含在复杂性函数名中，此时有 $T=T(N,I)$ 和 $S=S(N,I)$。

算法的时间复杂性越高，算法的执行时间越长；反之，执行时间越短。算法的空间复杂性越高，算法所需要的存储空间越多；反之，需要的存储空间越少。由于时间复杂性和空间复杂性概念雷同，计量方法相似，且空间复杂性分析相对简单，所以一般进行算法分析时主要讨论时间复杂性。

根据 $T=T(N,I)$ 的概念，它应该是该算法在一台抽象计算机上运行所需要的时间。设此抽象计算机提供的元运算有 k 种，分别记为 $O_1,O_2,\cdots,O_k$，设这些元运算每执行一次所需要的时间分别为 $t_1,t_2,\cdots,t_k$，设算法 A 中用到元运算 O_i 的次数为 e_i，$i=1,\cdots,k$, 则 $e_i=e_i(N,I)$，有

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

显然，不可能对规模 N 的每种合法的输入 I 都去统计 $e_i(N,I)$。因此，$T(N,I)$ 的表达式还需进一步简化；或者说，只能在规模 N 的某些或某类有代表性的合法输入中统计相应的 $e_i(N,I)$，以及评价时间复杂性。通常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 $T_{\text{avg}}(N)$：

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

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

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

其中，D_N 是规模为 N 的所有合法输入集合；I^* 是 D_N 中达到 $T_{\max}(N)$ 的一个输入；$\tilde{I}$ 是 D_N 中达到 $T_{\min}(N)$ 的一个输入；$P(I)$ 是出现输入为 I 的概率。以上三种情况下的时间复杂性各自从某个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，随着经济的发展、社会的进步、科学研究的深入，要求用计算机求解的问题越来越复杂，规模越来越大。人们关心的并不是较小的输入规模，而是在很大的输入实例下算法的性能。因此，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，即算法的运行时间随着输入规模的增长而增长，当规模达到最大时或最差输入状态发生时算法的性能。

一般来说，当 n 单调增加且趋于 ∞ 时，$T(n)$ 也单调增加且趋于 ∞。对于 $T(n)$，如果存在一个函数 $\tilde{T}(n)$，使得当 N 趋于 ∞ 时，有 $((T(n)-\tilde{T}(n))/T(n))\to 0$，那么称 $\tilde{T}(n)$ 是 $T(n)$ 当 n 趋于 ∞ 时的渐近性态或渐近复杂性。直观上，$\tilde{T}(n)$ 是 $T(n)$ 略去低阶项后剩余的主项，因此它确实比 $T(n)$ 简单。由于当 n 趋于 ∞ 时 $T(n)$ 渐近于 $\tilde{T}(n)$，因此有理由用 $\tilde{T}(n)$ 来替代 $T(n)$ 作为算法在 n 趋于 ∞ 时的复杂性度量。

下面以百钱买百鸡为例，说明渐近时间复杂性。公元前 5 世纪，中国古代数学家张丘建在其《算经》中提出了著名的“百钱买百鸡”问题：鸡翁一，值钱五，鸡母一，值钱三，鸡雏三，值钱一，百钱买百鸡，问翁、母、雏各几何？即一百个铜钱买了一百只鸡，其中公鸡一只 5 钱、母鸡一只 3 钱、雏鸡一钱 3 只，问一百只鸡中公鸡、母鸡、雏鸡各多少？

算法的伪代码如下：

```
for x = 0 to 100
   for y = 0 to 100
     for z = 0 to 100
        if  (x+y+z=100)  and  (5*x+3*y+z/3=100)  then
         System.out.println("  "+x+"  "+y+"  "+z)
        end if
      next z
```

```
        next y
    next x
```

百鸡问题算法的时间复杂性可以表示为

$$T(n) = (n+1)(n+1)(n+1) = n^3 + 3n^2 + 3n + 1$$

$T(n)$ 为百鸡问题算法复杂性函数。当 n 增大时，如当 n =100 万时，算法的执行时间主要取决于式子的第一项，而第二、三、四项对执行时间的影响只有第一项的几十万分之一，因此可以忽略不计。于是，可以用 $\tilde{T}(n) = n^3$ 来替代百鸡问题的时间复杂性 $T(n)$。由于 $\tilde{T}(n)$ 明显比 $T(n)$ 简单，因此这种替代明显地是对复杂性分析的一种简化，其大大降低了算法分析的难度，去除了精确计算产生的负担，使得算法分析任务变得可控。

下面引入表示算法渐近时间复杂性的记号：O, Ω, Θ。

1）大 O 表示法（算法运行时间的上界）

设 $f(n)$ 和 $g(n)$ 是定义在正整数集上的正函数。

若存在正常数 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 $f(n)$ 在 n 充分大时有上界，且 $g(n)$ 是它的一个上界，记为 $f(n) = O(g(n))$，也称 $f(n)$ 的阶不高于 $g(n)$ 的阶。大 O 表示法如图 1.1 所示。

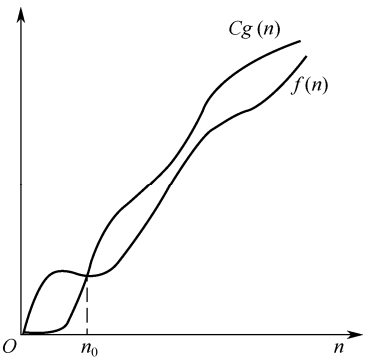

图 1.1　大 O 表示法

【例】 $f(n) = 8n^2 + 3n + 2$，求运行时间上界，用大 O 表示法表示。

令 $n_0 = 2$，当 $n \geqslant n_0$ 时，有 $f(n) \leqslant 8n^2 + 3n + n \leqslant 12n^2$。

令 $C = 12$，$g(n) = n^2$，所以有 $f(n) = O(g(n)) = O(n^2)$。

2）大 Ω 表示法（算法运行时间的下界）

设 $f(n)$ 和 $g(n)$ 是定义在正整数集上的正函数。

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 $f(n)$ 在 n 充分大时有下界，且 $g(n)$ 是它的一个下界，记为 $f(n) = \Omega(g(n))$，也称 $f(n)$ 的阶不低于

$g(n)$ 的阶。大 Ω 表示法如图 1.2 所示。

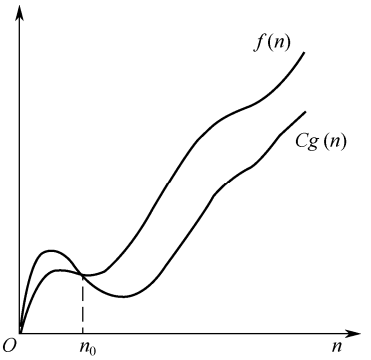

图 1.2　大 Ω 表示法

【例】 $f(n)=8n^2+3n+2$，求运行时间的下界，用大 Ω 表示法表示。

令 $n_0=8$，当 $n \geqslant n_0$ 时，有 $f(n) \geqslant 8n^2$。

令 $C=8$，$g(n)=n^2$，所以有 $f(n)=\Omega(g(n))=\Omega(n^2)$。

3）大 Θ 表示法（算法运行时间的准确界）

设 $f(n)$ 和 $g(n)$ 是定义在正整数集上的正函数。

若存在正常数 C_1, C_2 和自然数 n_0，使得当 $n \geqslant n_0$ 时有 $C_1 g(n) \leqslant f(n) \leqslant C_2 g(n)$，则称函数 $f(n)$ 在 n 充分大时有上界和下界，且 $C_1 g(n)$ 是它的一个下界，$C_2 g(n)$ 是它的一个上界，记为 $f(n)=\Theta(g(n))$，也称 $f(n)$ 的阶是 $\Theta(g(n))$。大 Θ 表示法如图 1.3 所示。

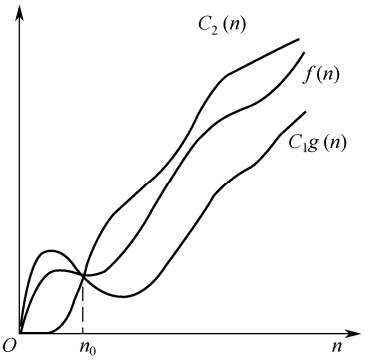

图 1.3　大 Θ 表示法

【例】 $f(n)=8n^2+3n+2$，求运行时间的准确界，用大 Θ 表示法表示。

令 $n_0=2$，当 $n \geqslant n_0$ 时，有 $f(n) \leqslant 12n^2$。

令 $n_0 = 0$，当 $n \geqslant n_0$ 时，有 $f(n) \geqslant 8n^2$。

令 $C_1 = 8$，$C_2 = 12$，$g(n) = n^2$，有 $8n^2 \leqslant f(n) \leqslant 12n^2$。

所以，$f(n) = \Theta(g(n)) = \Theta(n^2)$。

算法按运算时间可以分为两类，分别是多项式时间算法和指数时间算法。如果算法的时间复杂性与输入规模的一个确定的幂同阶，那么计算速度的提高可使解题规模以一个常数因子的倍数增加，习惯上把这类算法称为多项式时间算法。常见的多项式时间算法包括 $O(n), O(n\log n), O(n^2), O(n^3)$。而 $O(2^n), O(n!), O(n^n)$ 被称为指数时间算法。常见的算法时间复杂性如表 1.1 所示，常见的算法时间复杂性函数值对比如表 1.2 所示。

表 1.1　常见的算法时间复杂性

O	说　明
$O(1)$	称为常数时间（Constant time），表示不论输入的规模是多少，其执行时间总是一样
$O(\log n)$	称为次线性时间（Sub-linear time）或对数时间
$O(n)$	称为线性时间（Linear time），算法的运行时间随数据集的大小呈线性增长
$O(n\log n)$	称为线性乘对数时间
$O(n^2)$	称为平方时间（Quadratic time），算法的运行时间随数据集的大小呈二次方增长
$O(n^3)$	称为立方时间（Cubic time），算法的运行时间随数据集的大小呈三次方增长
$O(2^n)$	称为指数时间（Exponential time），算法的运行时间随数据集的大小呈 2 的 n 次方增长

表 1.2　常见的算法时间复杂性函数值对比

n	$O(\log n)$	$O(n)$	$O(n\log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$	$O(n!)$
10	3.32	10	3.3×10	1×10^2	1×10^3	1024	1×10^9
10^2	6.64	10^2	6.6×10^2	1×10^4	1×10^6	1.3×10^{30}	1.0×10^{198}
10^3	9.96	10^3	10×10^3	1×10^6	1×10^9	1.1×10^{301}	—
10^4	13.28	10^4	13×10^4	1×10^8	1×10^{12}	—	—
10^5	16.60	10^5	17×10^5	1×10^{10}	1×10^{15}	—	—