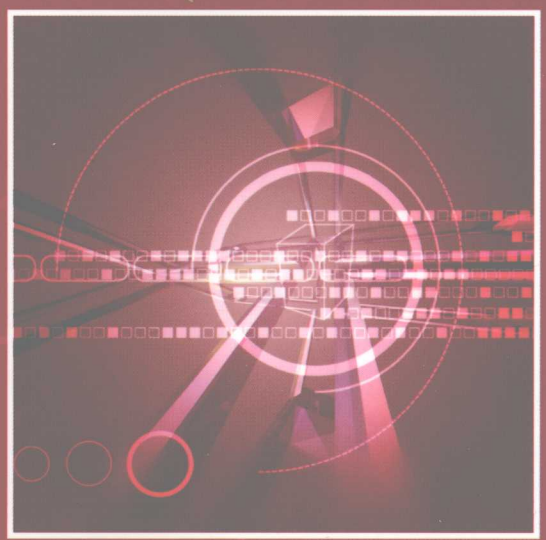
普通高等教育“十一五”计算机类规划教材

算法设计方法

吴哲辉　崔焕庆　马炳先　吴振寰　编著



免费电子课件

机械工业出版社
CHINA MACHINE PRESS

普通高等教育“十一五”计算机类规划教材

算法设计方法

吴哲辉　崔焕庆
马炳先　吴振寰　编著

机械工业出版社

全书共分为8章。第1章介绍了算法的基本概念以及算法描述和算法分析的基本知识。第2章至第7章分别论述了分治与递归算法、散列与凝聚算法、贪心算法、动态规划算法、回溯算法和分支限界算法。在每一章的开头，都先对相应的典型算法的基本思路进行详细、清晰的阐述，然后通过多种实际问题的求解，对该典型算法的设计方法作进一步的剖析。第8章对NP完全问题的基本理论进行讨论，并介绍了求解NP困难问题的近似算法和概率算法。

本书可作为计算机和相关专业的本科课程教材，也可作为科技人员的参考书。为方便教师教学，本书配有免费教学课件，欢迎选用本书作为教材的老师索取，索取邮箱：llm7785@sina.com。

图书在版编目（CIP）数据

算法设计方法/吴哲辉等编著. —北京：机械工业出版社，2008.7
普通高等教育“十一五”计算机类规划教材
ISBN 978-7-111-24707-4

Ⅰ.算… Ⅱ.吴… Ⅲ.电子计算机-算法设计-高等学校-教材 Ⅳ.TP301.6

中国版本图书馆CIP数据核字（2008）第108967号

机械工业出版社（北京市百万庄大街22号 邮政编码100037）
责任编辑：刘丽敏 版式设计：张世琴 责任校对：吴美英
封面设计：张 静 责任印制：邓 博
北京京丰印刷厂印刷
2008年10月第1版·第1次印刷
184mm×260mm·13印张·318千字
标准书号：ISBN 978-7-111-24707-4
定价：25.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
销售服务热线电话：（010）68326294
购书热线电话：（010）88379639 88379641 88379643
编辑热线电话：（010）88379726
封面无防伪标均为盗版

前　言

算法是计算机科学的核心内容之一，也是应用电子计算机求解实际问题的基础。对复杂的实际应用问题的求解，大多都归结为算法的设计，然后把求解算法转化为计算机程序。同算法设计密切相关的内容是算法分析，即算法的正确性证明和算法的时空复杂性分析。因此，国内外各高校都把算法设计与分析作为计算机专业的重要课程。

自20世纪七八十年代以来，国内一些高校先后编写和出版过多种算法设计与分析教材，同时也引进或翻译出版了一些国外的经典教材。就国内编写出版的教材而言，20世纪的教材大多从问题入手进行内容组织和处理，即在同一章中叙述一类问题的各种不同的算法设计方法。21世纪的教材则大多从方法入手来进行内容的组织和处理，即在一章中介绍一种典型的算法设计方法，而把适用于该种算法设计方法的各类问题作为实例在同一章中分节阐述。从问题入手转化为从方法入手编写教材，是学科发展（求解问题类大量增加）的必然，也是知识分类和内容组织的一种进步。不过，有些教材虽然从方法入手分章编写，但叙述重点还是放在对各类问题的求解算法的详细描述上，而对算法的设计思想的阐述仍不够透彻、突出。对各类问题的求解算法的详细描述虽然便于学生把书本上给出的算法转化为计算机程序，但如果不能理解和掌握各种典型算法的基本思想，就难以对适用于同类算法设计方法的（书本上未讨论过的）问题进行算法设计。

我们认为，算法设计与分析这门课程的主要教学目标是让学生领会和掌握各种典型算法的基本设计思路。因此一方面，我们采用从方法入手来对课程内容进行组织和处理；另一方面，在讲述每一种典型的算法设计方法时，首先对这种典型算法的基本思路进行充分的阐述，然后把适用于该典型算法的各类问题作为实例分节阐述，对每一类实际问题都要把典型算法设计思路在该类问题求解中的具体体现作为重点内容，并对所设计出的算法给出时间复杂性分析。本教材共分为8章。第1章是算法设计与分析概论，介绍了算法描述和算法分析的基本方法。第2~7章是本书的主体部分，依次讨论了分治与递归算法、散列与凝聚算法、贪心算法、动态规划算法、回溯算法和分支限界算法等典型的算法设计方法。第8章对NP完全问题进行了讨论，并介绍了求解NP困难问题常用的近似算法和概率算法。全书的讲授约需72学时。如果在各种典型算法中挑选一些问题类让学生自学，也可以用60左右学时完成本书的教学。

本书的第2章和第7章由崔焕庆撰写；第3、4章由吴振寰撰写；第5、6章由马炳先撰写；吴哲辉写了第1章和第8章，并对全部书稿进行校审。由于水平所限，书中难免有错误和不妥之处，敬请读者和专家批评指正。

作　者

2008年4月

目　录

第 1 章　算法设计与分析概论

算法是计算机科学的核心内容之一。借助电子计算机求解许多应用问题，最终都归结为算法设计。尽管类似于算法设计的一些求解问题的思想方法（例如，求两个数的最大公约数的欧几里德辗转相除法）早在古希腊时代就已被提出，但算法真正作为一个学科分支，还是在现代电子计算机问世以后才形成。因为对于许多复杂的计算问题来说，只有能够实现高速计算的电子计算机，才能将算法设计的思想方法转化为现实的求解结果。因此，“算法”又被称为“计算机算法”或“电子计算机算法”。

1.1　算法的定义和特征

通常的算法定义是：一个算法是求解某一类特定问题的一组有穷规则的集合。

首先，一个算法是一组规则，通常称之为算法步骤，这组规则是有穷的，即能用有限的形式表示出来。其次，一个算法是针对一类问题而设计的。如果给出了求解某类问题的一个算法，那么对这类问题的任意一组（一个）初始数据，这个算法都是有效的，也就是说，根据算法给出的求解步骤，都能求出这组初始数据所对应的计算结果。

算法通常具有以下 5 方面的特征。

（1）有限性

若给出了求解某类问题的一个算法，那么对于这类问题的任意一组初始数据，根据算法规则进行计算，运行总是能在有限时间内终止。因此，算法的有限性又称为算法的可终止性。

（2）确定性

算法规则中给出的每一个步骤都有确定的含义（动作），从而保证算法的每一步都能被确定地执行。

（3）可行性

算法规则中给出的每一步骤都是可执行的。这里包含两方面的含义：一种情况是给出的每一个计算步骤都是电子计算机可以直接执行的基本运算；另一种情况是如果求解的一类问题比较复杂，为了能够清晰表达算法思路，其中某些算法步骤并不详细写到基本运算，而是表述为“求解 × × 问题”或“对 × × 问题进行计算”，那么这里的“求解 × × 问题”的算法必须是已被进行过明确的描述，并为一般读者所熟知（至少是从有关文献中容易查找到）的。

（4）输入

一个算法在具体执行时都有一组（一个）输入。一个算法是针对一类问题而设计的。同一类问题，可以有不同的初始数据。用不同的初始数据作为输入，执行同一个算法可得到不同的计算结果。

（5）输出

对求解某类问题的一个算法，输入不同的初始数据可能得到不同的计算结果。对于计算结果，要进行输出。可以看出，算法的输出同输入有着密切的联系。

从以上关于算法的定义和特征的叙述，可以归结出关于算法的一个形式定义：(求解某类问题的)一个算法就是(求解该类问题的)对任意一组输入都能停机的一个确定图灵机。关于图灵机的基本概述要到第8章才给出，因此这里无法对这个定义给出详细的解释。不过可以指出，这是对算法的一个最严格的科学定义。

1.2 算法的描述

从学科的角度来说，算法主要包含两个方面的内容：算法设计和算法分析。算法设计主要研究怎样针对某一特定类型的问题设计出求解步骤，算法分析则要讨论所设计出来的算法步骤的正确性和复杂性。

对于设计出来的一类问题的求解步骤，需要一种表达方式，即算法描述。一方面，使他人可以通过这些算法描述来了解算法设计者的思路。另一方面，由于算法设计的最终目标是通过电子计算机来求解，因此这种表达方式最好应便于转化为电子计算机可执行的程序设计语言。然而，上面两方面的要求在许多情况下是矛盾的。要表述成计算机可执行的程序设计语言的方式，可能会因对细节描述过于详尽而掩盖了算法设计的基本思路。因此，对算法的描述往往采用一种折衷的方法，即用类似于某种程序设计语言(指高级语言)的格式，但并不要求完全按那种语言的规格来书写。对一些复杂问题的求解算法描述，还允许加入一些自然语言。

过去(20世纪八、九十年代)的算法书中，大多数采用类PASCAL语言来描述算法步骤，也有用类Algol语言、类Sparks语言来描述的。近年来的算法书则多用类C语言对算法步骤进行描述。本书中，我们对算法的描述不作统一约定。因为我们希望突出算法设计的基本方法。算法描述语言的选用以便于表述算法设计的基本思路为出发点。

接下来，通过一些实例来说明对算法的描述。

例1.1 冒泡排序算法。

排序是计算机处理工作中经常遇到的一项工作。据统计，计算机处理几乎有25%的时间花在排序上。冒泡排序是最常用的排序算法之一。这种排序算法的名称表达了算法的基本思想：轻者(小的元素)像气泡那样从水底往上浮。在排序过程中，从后面(理解为底部)开始，每次把相邻的两个元素作比较，当前面的元素大于后面的元素时，就交换它们的位置。这样，所有相邻的元素比较一遍以后最小的元素就被交换到了最前面(浮到上面)。下一轮比较时，就只需从第二个元素开始对元素序列重复上述工作。依此类推，直到把最后两个元素排好序。

根据这个思想，可以对冒泡排序算法进行描述。

算法1.1 冒泡排序

输入： 待排序数组A，其中有n个元素；

输出： 排好序的数组A。

```
bubblesort(floatA[ ],int n){
  int i,j;
```

```
    for(i=0;i<n;i++)
      for(j=n-1;j>i;j--)
        if(A[j]<A[j-1])
          swap(A+j,A+j-1);
}
```

这是用类 C 语言描述的一个算法。对于这个算法，我们作一些说明。首先，我们把冒泡排序算法写成一个过程(或函数)，是因为许多实际应用问题中，排序并不是整个问题的目标，而是解决问题过程中所需要做的一部分工作。把冒泡排序算法表述成一个函数，就便于编写解决整个问题的算法(主算法)在需要对某些元素进行排序时调用。事实上，在算法 1.1 中，我们也调用了一个函数 swap(*x, *y)。这是指把 $A[j]$ 和 $A[j-1]$ 两个元素进行交换。即把两个元素进行交换的算法也可以用一个过程表示如下。

算法 1.2 *元素交换*

输入： 待交换元素的位置 x 和 y；

输出： 交换后的结果仍存于 x 和 y 中。

```
swap(float *x,float *y){
  float u=*x;
  *x=*y;
  *y=u;
}
```

冒泡排序算法的执行过程，一般需要多次调用 swap 算法。有时候，我们也可以把调用 swap 算法的工作用自然语言来表示。即把算法 1.1 的主体表述为

```
for(i=0;i<n;i++)
  for(j=n-1;j>i;j--)
    if(A[j]<A[j-1])交换A[j]和A[j-1];
```

这里，用自然语言“交换 $A[j]$ 和 $A[j-1]$”替换了对 swap($A+j$,$A+j-1$)的调用。我们之所以承认这样写也是一种合理的算法描述方式，是建立在假定大家(大多数读者)都知道怎样实现把两个元素进行交换，即知道算法 1.2 描述的前提上的。

例 1.2 求最大公约数的辗转相除算法。

求两个数的最大公约数的辗转相除法是古希腊时代由欧几里德提出来的。今天来看，这也是一个非常高效的算法。它的基本思想是，用小的数作除数，大的数作为被除数，做除法求余，如果余数等于零，那么小的数就是两个数的最大公约数。否则用小的数做被除数，余数作为除数，再做除法求余，如此辗转相除下去，直到余数等于零。那时的除数就是要求的原本两个数的最大公约数。用算法来描述这个思想方法，可表述如下。

算法 1.3 *求最大公约数的辗转相除法*

输入： 两整数 m 和 n；

输出： m 和 n 的最大公约数。

```
int gcd(int m,int n)
{  int a=max{m,n};
   int b=min{m,n};
```

```
    int c;
    while(b!=0){
      c=a mod b;
      a=b;
      b=c;
    }
    return(a);
}
```

这个算法用 while 循环语句表述了辗转相除的思路。整个算法的核心是做除法求余的语句

$$c = a \bmod b;$$

与之配套的两个赋值语句

$$a = b;$$
$$b = c;$$

体现了辗转相除的思想。辗转相除的过程继续进行下去的条件是语句头

$$\text{while}(b!=0);$$

这个语句头就表述了“零不能做除数”的条件，又反映了这样的一个思想：只要 a、b 都不等于零，辗转相除的过程就一直进行下去，直到余数等于零为止。

另一方面，当 $b \neq 0$ 时，最大公约数满足

$$\gcd(a,b) = \gcd(b, a \bmod b)$$

根据这个性质，我们也可以设计一个递归调用自身的求最大公约数算法。

算法 1.4 *求最大公约数的递归算法*

输入： 两整数 m 和 n；

输出： m 和 n 的最大公约数。

```
int gcd(int m,int n)
{  int a=max{m,n};
   int b=min{m,n};
   int c;
   if(b==0)return(a);
   else{
     c=a mod b;
     return(gcd(b,c));
   }
}
```

在这个算法中，gcd(b,c)就是调用函数 gcd(m,n)自身。这种调用算法过程(或函数)自身的算法称为递归算法。许多问题的求解都可以归结为递归算法的设计。例如阶乘计算、梵塔问题等。

例 1.3 多项式求值的 Horner 算法。

设有一个 n 次多项式

$$P_n(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \cdots + a_{n-1}x + a_n$$

其中，系数 $a_0, a_1, a_2, \cdots, a_n$ 为已知值。给定 $x = x_0$ 计算多项式 $P_n(x_0)$ 的值。

对于这个问题，最直接的计算方法是先计算出 x_0 的各次幂 $x_0, x_0^2, \cdots x_0^n$，然后把 x_0 的各个幂分别同相应的系数相乘，最后把这些乘积累加，就得到所求的结果 $P_n(x_0)$。然而，这样做并不是最好的计算方法，这种做法需要进行比较多的运算次数。Horner 根据等式

$$P(x_0) = (\cdots((a_0x + a_1)x + a_2)x + \cdots + a_{n-1})x + a_n$$

设计出一个高效算法。为节省篇幅，下面只写出算法的主体部分。

算法 1.5 多项式求值的 Horner 算法

输入： 多项式的系数 $a_0, a_1, a_2, \cdots, a_n$ 和初值 x_0；

输出： 多项式 $P(x_0)$ 的值。

步骤：(仅主体部分)

```
P(x0) = a0;
for(i = 1; i <= n; i++)
    P(x0) = P(x0) * x0 + ai;
```

例 1.4 集合的并运算和交运算。

许多问题的求解过程中都涉及到集合运算，特别是并运算和交运算。在复杂问题的求解算法中，当需要进行集合的并运算或交运算时，往往用自然语言来表述，或简单地表述为“求 $A \cup B$”或“计算 $A \cap B$”等。作为那些复杂问题求解算法的基础，我们在这里讨论一下集合的并运算和交运算的算法描述。

首先，我们给出两个更为基本的集合运算算法。一个用 Member(b,A)表示，其功能是考查 b 是不是集合 A 的一个元素。另一个记为 Insert(b,A)，它的功能是在集合 A 中增加一个元素 b，这个运算以 b 原本不是 A 的元素为前提。

实现 Member(b,A)的实质性工作是查找。把 A 的元素逐个同 b 进行比较，若找到一个元素等于 b，就输出 1(表示 $b \in A$)并停止运算；当查找完集合 A 的全部元素都没有找出同 b 相等的元素时，就输出 0(即 $b \notin A$)。下面给出具体的算法描述。

算法 1.6 集合元素查找算法

输入： 集合 A 和待查找的元素 b，其中 $|A| = m$；

输出： b 在集合 A 中则返回 1，否则返回 0。

```
int Member(float b, float A[], int m){
  int i;
  for(i = 0; i < m; i++)
    if(A[i] == b) return(1);
  return(0);
}
```

在这个算法中，把集合定义为一个数组，是为了便于把算法转化为高级语言程序。因为在许多程序设计语言中，都没有定义集合这种数据类型，而数组则是各种程序设计语言都设置的数据类型。

Insert(b,A)的操作往往是在 Member(b,A)输出 0 后才使用的。假设用集合 S 作为 Insert(b,A)操作执行的结果，Insert(b,A)也可表示为 Insert(b,A,S)。

算法 1.7 向集合中插入元素算法

输入： 集合 A 和待插入的元素 b，其中 $|A|=m$；

输出： 集合 S，$|S|=m+1$。

```
Insert(float b,float A[ ],float S[ ],int m){
  int i;
  for(i=0;i<m;i++)
    S[i]=A[i];
  S[m]=b;
}
```

在算法 1.6 和算法 1.7 的基础上，容易写出集合的并运算和交运算的算法描述。集合 A 和 B 的并运算算法的设计思路是：以其中一个集合（设 A）为基础，先把的 A 的所有元素放入并集 S 中，对另一个集合 B 的每个元素 b_i，通过 Member(b_i,A)考查是否 $b_i \in A$，若 $b_i \notin A$，就通过 Insert(b_i,S,S)操作把 b_i 加入集合 S 中。

算法 1.8 集合的并运算算法

输入： 集合 A 和集合 B，其中 $|A|=m$，$|B|=n$；

输出： 集合 S，其中 $|S|=p$ 并且满足 $\max\{m,n\} \leqslant p \leqslant m+n$。

```
Union(float A[ ],int m,float B[ ],int n,float S[ ]){
  int i,j=m;
  S=A;
  for(i=0;i<n;i++)
    if(Member(B[i],A,j)==0)
      Insert(B[i],S,S,j++);
}
```

对集合 A 和 B 的交运算算法的设计是：先把交集 S 设置为空集，然后对集合 B 的每个元素 b_i 做 Member(b_i,A)操作，如果输出 1，则通过 Insert(b_i,S,S)操作把 b_i 加入到集合 S 中。

算法 1.9 集合的交运算算法

输入： 集合 A 和集合 B，其中 $|A|=m$，$|B|=n$；

输出： 集合 S，其中 $|S|=p$ 并且满足 $0 \leqslant p \leqslant \min\{m,n\}$。

```
Intersection(float A[ ],int m,float B[ ],int n,float S[ ]){
  int i,j=1;
  S=Φ;
  for(i=0;i<n;i++)
    if(Member(B[i],A,m)==1)
      Insert(B[i],S,S,j++);
}
```

这个算法中，我们首先把交集 S 定义为空集，然后用 Insert(b_i,S,S)表示把 b_i 加入到集合 S 中，所得到的新集合仍定义为 S，即用于下一次插入，这样就可以使 S 的元素根据 Member(b_i,A)的结果逐步增加。

1.3 算法分析

在文献[1]中，美国算法专家 Sara Baase 指出，算法分析的目的是为了尽可能地改进算法和在求解同一类问题的已提出的多个算法中选择一个好的算法。他指出算法选择的 5 个准则：

①正确性；

②工作量；

③占用空间；

④简单性；

⑤最优性。

我们认为，算法分析工作可归结为两部分：算法的正确性证明和算法的复杂性分析。算法的正确性证明主要包括算法的可终止性(即对任意输入，算法的执行都可以在有限步内终止)和算法的执行结果(输出)与问题(问题类)的求解要求相符两方面的证明。算法的复杂性分析是指算法的执行所需要的时间量和空间量的分析，其中对时间量的分析尤为重要。S. Basse 提出的算法选择准则中的第 4 条(简单性)是指算法设计思路应尽量直接反映问题的求解要求，这样便于对算法的正确性的理解和证明；第 5 条(最优性)则是在进行算法复杂性分析后才可能作出的判定和选择。

在许多关于算法的文献中，对求解一类问题的算法给出算法描述后，往往只对算法的复杂性(特别是时间复杂性)进行分析，而看不到同算法的正确性证明相关的内容。有的文献则只对算法的可终止性加以证明，而没有对算法执行的结果是否符合问题的求解要求加以证明。我们认为主要有以下原因：一些思路简单的算法(即符合 S. Basse 的第 4 条准则的算法)，由于算法的设计思路直接反映了问题的求解要求，其正确性十分明显，因此没有专门给出算法的正确性证明。另一方面，对于那些复杂问题的求解算法，在给出算法描述之前，往往都从理论上对算法设计思路进行阐述，甚至给出一些定理。这些阐述实际上就是算法正确性证明的一种表述方式。

算法的复杂性分析包括时间复杂性分析和空间复杂性分析。算法的时间复杂性亦即算法的效率，通常是指算法的执行所需要的各种基本运算的次数。算法的空间复杂性是指用电子计算机实现该算法所占用的内存单元数目。一般情况下，时间复杂性和空间复杂性都是同输入量的大小(求解问题的规模)密切相关的，都是输入量 n 的一个函数，分别用 $T(n)$ 和 $S(n)$ 来表示。由于 $T(n)$ 和 $S(n)$ 都是输入量 n 的函数，它们分别反映了该算法在时间量和空间量的要求程度。因此，许多文献称它们为时间复杂度和空间复杂度。在本书中，这两种名称都将被使用。“复杂性”或“复杂度”的名称将随着文字叙述的方便顺其自然地出现在相关叙述中。

通常情况下，人们对算法的时间复杂性比对算法的空间复杂性给予更多的关注。因为对大多数实际应用问题来说，求解算法的空间复杂度都是输入量的多项式函数，微电子技术的快速发展使得一般的电子计算机(即使是微型计算机)都能提供足够的内存空间以满足各种算法实现对空间量的要求。时间复杂性的情况则不一样。一方面，求解问题的算法对时间量的要求比对空间量的要求要高。理论上可以证明，对一个输入量为 n 的问题，如果求解该问

题所需的时间量的下确界和空间量的下确界分别为 $\inf(T(n))$ 和 $\inf(S(n))$，那么总有 $\inf(T(n)) \geqslant \inf(S(n))$。另一方面，许多实际应用问题（如导航、实时控制等）对相关算法的效率（时间复杂性）的要求是很高的。

有人认为，随着微电子技术的发展，近年来电子计算机的运算速度得到了若干个数量级的提高，因此除了少数领域外，算法的效率在许多场合下已不像过去那样显得十分重要，而且其重要性将随时间的推移越来越低。这种认识是不对的。当一个问题的输入量比较大时，时间复杂度的阶对算法实现所需时间量的影响决不是靠提高计算工具的计算速度所能弥补的。下面通过一个例子来说明这个道理。

假设甲和乙两人各自拥有一台电子计算机。甲的计算机运算速度比较慢，每秒只能进行 10 万（10^5）次基本运算；乙拥有的计算机要好得多，每秒能完成 1 亿（10^8）次基本运算。现有一个问题 P，由甲乙各自设计算法并在自己的计算机上求解。考虑到自己的计算机的运算速度低，甲在算法设计方面下了比较大的功夫，他花了五天时间设计出一个求解问题 P 的算法并编写出相应的计算机程序。他的算法的时间复杂度是输入量的 3 次函数，即当输入量为 n 时，需要 n^3 次基本运算来求解这个问题。乙认为自己的计算机性能好，不必在算法设计方面下太大功夫。他只用了两个小时就设计出算法并编写出相应的计算机程序。他设计出的算法的时间复杂度是输入量的（以 2 为底的）指数函数。即当输入量为 n 时，他的算法所需的基本运算次数为 2^n。现在我们计算一下，当输入量为 $n=50$ 时，他们各自用自己的算法和自己的计算机求解问题 P 所需要的时间。

甲用他自己设计的算法求解输入量 $n=50$ 的问题 P，共需要 $50^3=125000$ 次基本运算，在他自己的计算机上运行 1.25s 就能完成计算。计算时间同算法设计所用的时间（5 天）相比较，可以忽略不计。

乙用他自己设计的算法求解输入量 $n=50$ 的问题 P，所需的基本运算次数为 2^{50} 次。由于 $2^{50}>10^{15}$，而乙的计算机一年所能完成的基本运算次数为

$$365\times24\times3600\times10^8\approx2.7\times10^{14}$$

因此，用乙设计的算法，需要它的计算机连续运行 3 年多才能完成（输入量 $n=50$ 时）对问题 P 的计算。

从这个简单的例子可以看到，算法的时间复杂度比计算机性能对问题的求解效率的影响要大得多。

对于算法的复杂度（包括时间复杂度和空间复杂度），通常用界函数来表述。界函数又可分为上界、下界和精确界。

定义 1.1 设 $f(n)$ 和 $g(n)$ 为定义在自然数集上的两个函数。

（1）若存在正常数 c 和自然数 n_0，使得当 $n\geqslant n_0$ 时都有

$$f(n)\leqslant c\cdot g(n)$$

则称 $g(n)$ 为 $f(n)$ 的上界函数，记为 $f(n)=o(g(n))$。

（2）若存在正常数 c 和自然数 n_0，使得当 $n\geqslant n_0$ 时都有

$$f(n)\geqslant c\cdot g(n)$$

则称 $g(n)$ 为 $f(n)$ 的下界函数，记为 $f(n)=\Omega(g(n))$。

（3）若存在两个正常数 c_1、c_2 和自然数 n_0，使得当 $n\geqslant n_0$ 时都有

$$c_1\cdot g(n)\leqslant f(n)\leqslant c_2\cdot g(n)$$

则称 $g(n)$ 为 $f(n)$ 的精确界函数，记为 $f(n)=\Theta(g(n))$。

说一个算法的时间复杂度或空间复杂度时，一般是指出其上界、下界或精确界，大多数情况下是指上界或精确界。例如，当设计出一个算法后，往往要对这个算法的优劣进行评价。作为算法的设计者，为了强调算法的价值，要说明设计出来的算法计算量不是很大，就需要给出算法的时间复杂度和空间复杂度的上界或精确界，即给出函数 $f(n)$ 和 $g(n)$ 使得

$$T(n)=o(f(n)),S(n)=o(g(n))$$

或者

$$T(n)=\Theta(f(n)),S(n)=\Theta(g(n))$$

复杂度的下界函数也有使用的场合。例如，当对一类问题设计出两个不同的算法，要比较这两个算法的优劣时，就可能用到下界函数。以两个算法的时间复杂度 $T_1(n)$ 和 $T_2(n)$ 为例，如果找出一个界函数 $f(n)$，使得

$$T_1(n)=o(f(n)),T_2(n)=\Omega(f(n))$$

即第一个算法的时间复杂度以 $f(n)$ 为上界函数，而第二个算法的时间复杂度以 $f(n)$ 为下界函数，我们就可以下结论：第一个算法的效率不比第二个算法的效率低。

从定义 1.1 可以看出，算法复杂性的界函数加(减)或乘(除)一个常数，并不产生任何影响。即若 $a(a>0)$ 为一个常数，那么

$$o(f(n)+a)=o(f(n)),o(af(n))=o(f(n))$$

$$\Omega(f(n)+a)=\Omega(f(n)),\Omega(af(n))=\Omega(f(n))$$

$$\Theta(f(n)+a)=\Theta(f(n)),\Theta(af(n))=\Theta(f(n))$$

此外，由于对任意自然数 n 和 k，都有 $n^k \geqslant n^{k-1}$，当算法的时间(空间)复杂度的界函数为一个(n 的)多项式函数时，可以取这个多项式的最高次幂作为界函数，即(假设 $a_k \neq 0$)

$$o(a_k n^k+a_{k-1}n^{k-1}+\cdots+a_1 n+a_0)=o(n^k)$$

$$\Omega(a_k n^k+a_{k-1}n^{k-1}+\cdots+a_1 n+a_0)=\Omega(n^k)$$

$$\Theta(a_k n^k+a_{k-1}n^{k-1}+\cdots+a_1 n+a_0)=\Theta(n^k)$$

另一方面，对一个算法来说，实现对一个(一组)输入的计算，所需要的时间量和空间量不仅同输入量的大小(规模)有关，往往同输入的初始数据的分布情况(如排列形式等)也有密切的关系。因此，对算法的时间(空间)复杂度的分析，又提出了最坏情况下的复杂度和平均复杂度两个不同的概念。最坏情况下的复杂度是指(同样规模的输入量的)各种分布情况中，按计算量(或需要的存储空间)最大的情况计算出来的复杂度；平均复杂度则是把各种分布情况下的计算量(存储空间)根据各种分布出现的概率求出的加权平均值。算法理论研究中更注重最坏情况下的复杂度，而在实际应用中，对算法的平均复杂度也十分重视，有时会根据平均复杂度来决定对算法的选择，因此，平均复杂度也被称为期望复杂度。

在算法复杂性分析中，有时为了避免一些烦杂的计算，对某些操作(基本运算)的次数只作粗略的估算。作粗略估算时，对计算量往往是稍微放大，从而求出的上界函数也可能不是上确界。因此，算法的复杂性分析有时也称为算法复杂性估计(或估算)。

接下来，对 1.2 节给出的几个算法的时间复杂度作一下分析。

例 1.5 冒泡排序算法(算法 1.1)的时间复杂性分析。

算法 1.1 给出的冒泡排序算法主要包含两种基本运算：比较和交换。比较是指待排序数

组中两个相邻元素的比较；当通过比较运算发现两个相邻元素的排序顺序不符合要求时，就要把它们的位置交换。比较和交换运算都出现在双重嵌套循环语句的循环体中，比较运算是作为交换运算的条件而出现的。从这个双重嵌套的循环语句结构容易知道，比较运算共需进行

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

次。这是对任意分布的 n 个输入都需要做的运算次数。交换运算的情况则不同，在循环过程中每次执行到循环体时，只有当条件 $A[j]<A[j-1]$ 成立时才进行交换运算。因此在整个排序算法执行过程中，交换运算的次数同输入数据的分布(排列情况)有关。当待排序数组的元素之间满足关系

$$A[1]\leqslant A[2]\leqslant\cdots\leqslant A[n-1]\leqslant A[n] \tag{1.2}$$

时，一次交换运算都不需要执行。反之，如果待排序数组的元素满足关系

$$A[1]>A[2]>\cdots>A[n-1]>A[n] \tag{1.3}$$

那么每一次比较后都需要把相邻两个元素进行交换。在这种情况下，实现排序要进行的交换运算次数也等于$\frac{n(n-1)}{2}$。满足式(1.2)的分布情况是最好的情况，满足式(1.3)的分布情况就是最坏情况。也就是说，最坏情况下所需要的交换次数为$\frac{n(n-1)}{2}$。

下面讨论平均情况下执行算法 1.1 需要做交换运算的次数。为此先引入逆序的概念。在一个序列 $A[1],A[2],\cdots,A[n]$ 中，若存在 $i,j\in\{1,2,\cdots,n\}$，使得 $i<j$ 但 $A[i]>A[j]$，则说 $A[i]$ 和 $A[j]$ 构成一个逆序。显然，对 $A[1],A[2],\cdots,A[n]$ 的任一种排列情况，用算法 1.1 对这个序列进行排序，交换运算的次数就等于这个序列中逆序的个数。假设有 k 个逆序的排列个数为 $I_n(k)$，那么满足式(1.2)的排列和满足式(1.3)的排列都只有一种，即有

$$I_n(0)=I_n\left(\frac{n(n-1)}{2}\right)=1 \tag{1.4}$$

由于 n 个数的不同排列共有 $n!$ 种，而在各种排列中，逆序个数的最小值为 0，最大值为$\frac{n(n-1)}{2}$。因此各种分布的平均逆序个数为

$$\frac{1}{n!}\sum_{k=0}^{\frac{n(n-1)}{2}}k\cdot I_n(k) \tag{1.5}$$

这也就是算法 1.1 中需要进行交换运算次数的平均值。后面将证明，式(1.5)的值等于$\frac{n(n-1)}{4}$。可见平均交换运算次数和最坏情况下的交换运算次数都是输入量 n 的二次函数，即 $\Theta(n^2)$。当两个界函数的阶相等时(特别在阶比较低的情况下)，有时也会通过直接比较两个函数的大小来进行更具体的判定。在算法 1.1 中，平均情况下的交换运算次数只是最坏情况下交换运算次数的 1/2，这也从一定程度表示出两种不同情况下工作量之间的差别。另一方面，从算法 1.2 知，两个元素的交换运算可分解为 3 个赋值运算。可见，赋值运算是比交换运算更为基本的运算。在一般的算法时间复杂度分析中，给出交换运算的次数(的界函

数)也被大家所接受。

根据算法 1.1，比较运算的次数同输入的分布情况无关。不论输入的 n 元数组中，元素是怎样排列的，也需要做 $\frac{n(n-1)}{2}$ 次比较运算。这实际上是不必要的。譬如，当 n 个输入数的值和排列(或经过少量的交换运算后)满足式(1.2)时，排序的目标已经实现，这时就没有必要再进行比较(和交换)运算。算法 1.1 并没有考虑这些情况。对算法 1.1 稍作修改，可以(对许多输入分布情况)减少比较运算次数。具体做法是引入一些标志变量 f，f 的取值($f \in \{0,1\}$)由内层循环中是否进行过元素交换来确定；并根据 f 的取值来决定外层循环是否要继续进行。譬如，当执行内循环的某一轮并没有作交换运算时，就对 f 赋予数值 0，而当 $f=0$ 时，外层循环也不必再继续执行，整个算法终止。许多教科书中给出的冒泡排序算法都加入了标志变量 f，其作用就在此。

例 1.6 求最大公约数的辗转相除算法的时间复杂性分析。

用辗转相除法求两个正整数 a 和 b 的最大公约数，算法的时间复杂性表现为辗转相除的次数(轮数)，因为每轮只需做一次除法求余的运算。辗转相除的轮数(辗转次数)不仅同 a 和 b 两个数的大小有关，而且同两个数之间的关系也有关。譬如，即使 a、b 都是很大的数，当 $b=a-1$ 时，只需辗转两次(即做两轮除法)，算法就终止，并求出最大公约数(等于 1)。

下面分析一下用辗转相除算法求两个数的最大公约数(算法 1.3)在最坏情况下的时间复杂性。为此，引入斐波那契数列的概念。

斐波那契数列是由意大利数学家斐波那契(Fibonacci)提出来的。他在 1202 年出版的《算盘全书》中提出了兔子繁殖问题：假设一对兔子每月能生一对小兔(一雌一雄)，而每对小兔在出生后的第三个月开始又可以每月生一对小兔，问从一对刚出生的小兔开始，经过 50 个月拥有多少对兔子(假设在此过程中没有兔子死亡)？

这个问题可以通过这样的思路进行求解：假设一对兔子在出生后的一个月内称为小兔，第二个月称它们为大兔，那么大兔在下一个月就可以生一对小兔，那么序号为 0,1,2,3,…的各个月的小兔对数、大兔对数和兔子的总对数可以用表 1.1 表示。

表 1.1 兔子繁殖情况表

n(月份序号)	0	1	2	3	4	5	6	…
$F_n^{小}$(小兔对数)	1	0	1	1	2	3	5	…
$F_n^{大}$(大兔对数)	0	1	1	2	3	5	8	…
$F(n)$(兔子总对数)	1	1	2	3	5	8	13	…

表 1.1 中在第 n 个月中拥有小兔的对数用 $F_n^{小}$ 表示，在第 n 个月中拥有大兔的对数用 $F_n^{大}$ 表示，而 $F(n)$表示第 n 个月中拥有兔子的总对数。这样，第 50 个月拥有的兔子的对数就是 $F(50)$。观察表 1.1，可以得到各个月的兔子总对数满足关系 $F(n)=F(n-1)+F(n-2)$(当 $n \geqslant 2$ 时)。这样只要给出 $F(0)=1$ 和 $F(1)=1$，对于任意 $n \geqslant 2$，都可以通过 $F(n-1)$和 $F(n-2)$求出 $F(n)$。逐次进行下去，可以求出 $F(n)$的值。

斐波那契数列的概念就是在这个例子的基础上提出来的。如果去掉 $n \leqslant 50$ 的限制，把 n 的值任意非负整数，即设