第 1 章

进入算法的世界

计算机（Computer）是一种具备了数据计算与信息处理功能的电子设备。它可以接受人类所设计的指令或程序设计语言，经过运算处理后输出期待的结果。

对于有志于从事信息技术专业领域的人员来说，数据结构（Data Structure）是一门与计算机硬件和软件息息相关的学科，称得上是从计算机问世以来经久不衰的热门学科。这门学科研究的重点在计算机程序设计领域，即研究如何将计算机中相关数据或信息的组合以某种方式组织起来进行有效的加工和处理，其中包含算法（Algorithm）、数据存储的结构、排序、查找、树、图及哈希函数等。

随着信息与网络科技的高速发展，在目前这个物联网（Internet of Things，IOT）与云运算（Cloud Computing）的时代，程序设计能力已经被看成是国力的象征，有条件的中小学校都将程序设计（或称为“编程”）列入学生信息课的学习内容，在大专院校里，程序设计已不再只是信息技术相关科系的“专利”了。程序设计已经是接受全民义务制教育的学生们应该具备的基本能力，只有将“创意”通过“设计过程”与计算机相结合，才能让新一代人才轻松应对这个快速变迁的云计算时代（见图 1-1）。

图 1-1

没有最好的程序设计语言，只有是否适合的程序设计语言。程序设计语言本来就只是工具，从来都不是算法的重点。我们知道，一个程序能否快速而高效地完成预定的任务，算法才是其中的关键因素。本章将介绍算法的基本概念和算法性能的分析，并介绍一些基本的数据结构，以作为后续章节讨论的基础，让读者逐步认识算法。

提　示

“云”其实泛指“网络”，因为工程师在网络结构示意图中通常习惯用“云朵”图来代表不同的网络。云计算是指将网络中的运算能力提供出来作为一种服务，只要用户可以通过网络登录远程服务器进行操作，就能使用这种运算资源。

物联网（Internet of Things，IOT）是近年来信息产业中一个非常热门的话题，各种配备了传感器的物品，如 RFID、环境传感器、全球定位系统（GPS）等与因特网结合起来，并通过网络技术让各种实体对象、自动化设备彼此沟通与交换信息，也就是通过网络把所有东西都连接在一起。

1.1　生活中处处都存在算法

算法（Algorithm）是计算机科学中程序设计领域的核心理论之一，每个人每天都会用到一些算法。算法也是人类使用计算机解决问题的技巧之一，不但可用于计算机领域，而且在数学、物理甚至是每天的生活中都应用广泛。在日常生活中有许多工作可以使用算法来描述，例如员工的工作报告、宠物的饲养过程、厨师准备美食的食谱、学生的课程表等。我们几乎每天都在使用的各种搜索引擎也必须借助不断更新的算法来运行，如图 1-2 所示。

图 1-2

特别是在算法与大数据的结合下，这门学科演化出“千奇百怪”的应用，例如当我们拨打某个银行信用卡客户服务中心的电话时，很可能就先经过后台算法的过滤，帮我们找出一位最“合我

们胃口”的客服人员来与我们交谈。在互联网时代，通过大数据的分析，网店还可以进一步了解产品购买和需求的人群，甚至一些知名 IT 企业在面试过程中也会测验面试人员对算法的了解程度（见图 1-3）。

图 1-3

提　示

大数据（又称为海量数据，Big Data）由 IBM 公司于 2010 年提出，是指在一定时效（Velocity）内进行大量（Volume）、多样性（Variety）、低价值密度（Value）、真实性（Veracity）数据的获得、分析、处理、保存等操作，主要特性包含 Volume（大量）、Velocity（时效性）、Variety（多样性）、Value（低价值密度）和 Veracity（真实性）。大数据解决了商业智能无法处理的非结构化与半结构化数据。

1.1.1　算法的定义

在韦氏辞典中算法定义为：A procedure for solving a mathematical problem in a finite number of steps，即“在有限步骤内解决数学问题的过程。”如果运用在计算机领域中，我们也可以把算法定义成：“为了解决某项工作或某个问题，所需要有限数量的机械性或重复性指令与计算步骤。”

我们知道可整除两个整数的最大整数被称为这两个整数的最大公约数，而辗转相除法可以用来求出两个整数的最大公约数，即可以使用这个辗转相除法的算法来求解。下面我们使用 while 循环来设计一个 C 语言程序，根据输入的两个整数求解最大公约数（Greatest Common Divisor，GCD）。辗转相除法用 C 语言来描述的算法过程如下：

```
if (Num1 < Num2)
{
    TmpNum = Num1;
    Num1 = Num2;
    Num2 = TmpNum;       /* 找出两个数中的较大值 */
}
while (Num2 != 0)
{
    TmpNum = Num1 % Num2;    /* 求两个数的余数 */
```

```
        Num1 = Num2;
        Num2 = TmpNum;          /* 辗转相除法 */
    }
    printf("最大公约数(GCD) = %d\n", Num1);
```

1.1.2 算法的条件

在计算机系统中算法更是不可或缺的一环，有一个著名的公式“计算机程序 = 算法 + 数据结构”，它从另一个角度阐述算法的概念与定义，也表述了算法、数据结构和计算机程序之间的关系。在了解了认识算法的定义之后，说明一下算法所必须符合的 5 个条件，如图 1-4 和表 1-1 所示。

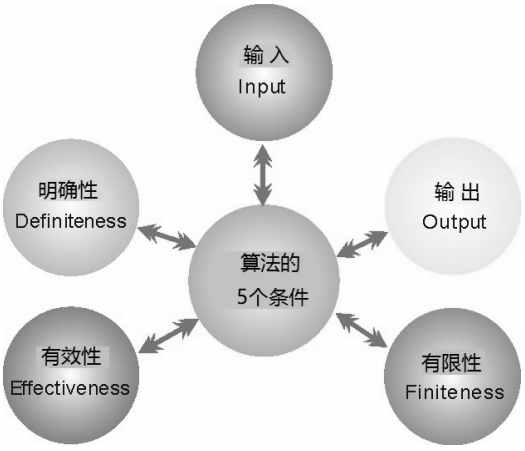

图 1-4

表 1-1 算法必须符合的 5 个条件

算法的特性	内容与说明
输入（Input）	0 个或多个输入数据，这些输入必须有清楚的描述或定义
输出（Output）	至少会有一个输出结果，不能没有输出结果
明确性（Definiteness）	每一个指令或步骤必须是简洁明确的
有限性（Finiteness）	在有限步骤后一定会结束，不会产生无限循环
有效性（Effectiveness）	步骤清晰且可行，能让用户用纸笔计算而求出答案

我们认识了算法的定义与条件后，接着要思考一下用什么方法来表达算法比较合适。其实算法的主要目的在于让人们了解所执行工作的流程与步骤，只要清楚地体现出算法的 5 个条件即可。

常用的算法一般可以用中文、英文、数字等文字方式来描述，也就是用自然语言来描述算法的具体步骤。例如，图 1-5 所示就是小华早上去上学并买早餐的简单文字算法。

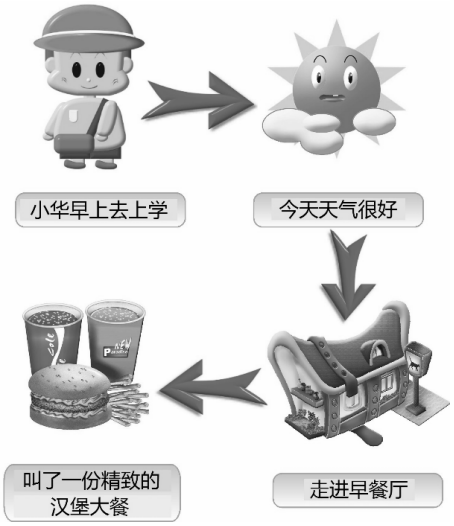

图 1-5

常用的算法也可以用可读性高的高级程序设计语言或伪语言（Pseudo-Language）来描述或者表达。以下算法是用 C 语言描述的，给 Pow()函数传入两个数 x、y，求 x 的 y 次方的值，即求 x^y 的值：

```
float Pow( float x, int y )
{
    float p = 1;
    int i;
    for( i = 1; i <= y; i++ )
        p *= x;

    return p;
}

int main(void)
{
    float x;
    int y;

    printf( "请输入次方运算（ex.2^3）: " );
    scanf( "%f^%d", &x, &y );
    printf( "次方运算结果: %.4f\n", Pow(x, y) );
    /* 调用 Pow()函数，并输出计算结果 */
}
```

提　示

伪语言（Pseudo-Language）是接近高级程序设计的语言，也是一种不能直接放进计算机中执行的语言。一般需要一种特定的预处理器（Preprocessor），或者用人工编写转换成真正的计算机语言，经常使用的有 SPARKS、PASCAL-LIKE 等。

流程图（Flow Diagram）是一种以图形符号来表示算法的通用方法。例如，输入一个数值，并判断是奇数还是偶数，如图 1-6 所示。

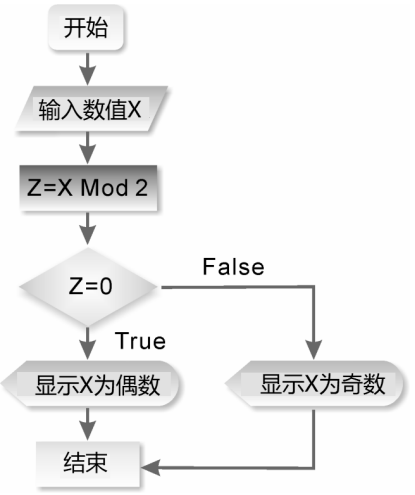

图 1-6

提　示

算法和过程（Procedure）有何不同？与流程图又有什么关系？

算法和过程是有所区别的，因为过程不一定要满足有限性的要求，如操作系统或机器上运行的过程。除非宕机，否则永远在等待循环中（Waiting Loop），这也违反了算法 5 个条件中的“有限性”。另外，只要是算法，就都能够使用流程图来表示，但是由于过程流程图可包含无限循环，因此无法使用算法来表达。

1.1.3 时间复杂度 $O(f(n))$

大家可能会想，那么应该怎么评估一个算法的好坏呢？例如，可以把某个算法执行步骤的计数来作为衡量运行时间的标准，例如同样是程序语句：

```
a = a + 1
```

与

```
a = a + 0.3 / 0.7 * 10005
```

由于涉及变量存储类型与表达式的复杂度，因此真正绝对精确的运行时间一定不相同。不过话说回来，如此大费周章地去考虑程序的运行时间往往寸步难行，而且毫无意义，此时可以利用一种“概量”的概念来衡量运行时间，我们称之为“时间复杂度”（Time Complexity）。其详细定义如下：

在一个完全理想状态下的计算机中，我们定义 $T(n)$来表示程序执行所要花费的时间，其中 n 代表数据输入量。当然程序的运行时间（Worse Case Executing Time）或最大运行时间是时间复杂度的衡量标准，一般以 Big-Oh 表示。

在分析算法的时间复杂度时，往往用函数来表示它的成长率（Rate of Growth），其实时间复杂度是一种“渐近表示法”（Asymptotic Notation）。

$O(f(n))$可视为某算法在计算机中所需运行时间不会超过某一常数倍的 $f(n)$。也就是说，当某算法的运行时间 $T(n)$的时间复杂度（Time Complexity）为 $O(f(n))$（读成 big-oh of f(n)或 order is f(n)）时，意思是存在两个常数 c 与 n_0，若 $n \geq n_0$，则 $T(n) \leq cf(n)$。$f(n)$又称为运行时间的成长率（Rate of Growth）。由于在估算算法复杂度时采取“宁可高估不要低估”的原则，因此估计出来的复杂度是算法真正所需运行时间的上限。请大家看以下范例，以了解时间复杂度的意义。

范例▶ 假如运行时间 $T(n)=3n^3+2n^2+5n$，求时间复杂度。

解答▶ 首先找出常数 c 与 n_0。当 $n_0=0$、$c=10$ 时，若 $n \geq n_0$，则 $3n^3+2n^2+5n \leq 10n^3$，因此得知时间复杂度为 $O(n^3)$。

事实上，时间复杂度只是执行次数的一个概略的量度，并非真实的执行次数。而 Big-Oh 则是一种用来表示最坏运行时间的表现方式，也是最常用于在描述时间复杂度的渐近式表示法。常见的 Big-Oh 可参考表 1-2 和图 1-7。

表 1-2　常见的 Big-Oh

Big-Oh	特色与说明
$O(1)$	称为常数时间（Constant Time），表示算法的运行时间是一个常数倍
$O(n)$	称为线性时间（Linear Time），表示执行的时间会随着数据集合的大小而线性增长
$O(\log_2 n)$	称为次线性时间（Sub-Linear Time），成长速度比线性时间还慢，而比常数时间还快
$O(n^2)$	称为平方时间（Quadratic Time），算法的运行时间会成二次方的增长
$O(n^3)$	称为立方时间（Cubic Time），算法的运行时间会成三次方的增长
(2^n)	称为指数时间（Exponential Time），算法的运行时间会成 2 的 n 次方增长。例如，解决 Nonpolynomial Problem 问题算法的时间复杂度为 $O(2^n)$
$O(n\log_2 n)$	称为线性乘对数时间，介于线性和二次方增长的中间模式

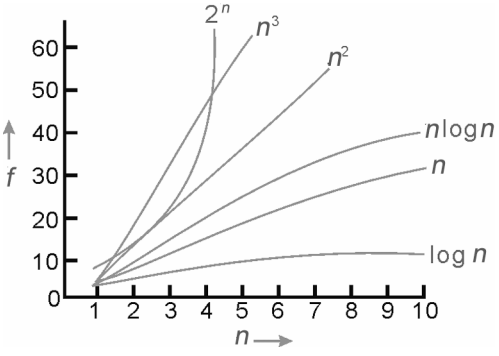

图 1-7

$n \geq 16$ 时，时间复杂度的优劣比较关系如下：

$$O(1)<O(\log_2 n)<O(n)<O(n\log_2 n)<O(n^2)<O(n^3)<O(2^n)$$

1.2 常见算法介绍

善用算法是培养程序设计逻辑很重要的步骤。许多实际的问题都可用多个可行的算法来解决，但是要从中找出最佳的解决算法是一项挑战。本节将为大家介绍一些近年来相当知名的算法，帮助大家更加了解不同算法的概念与技巧，以便日后更有能力分析各种算法的优劣。

1.2.1 分治法

分治法（Divide and Conquer，也称为“分而治之法”）是一种很重要的算法，我们可以应用分治法来逐一拆解复杂的问题，核心思想就是将一个难以直接解决的大问题依照相同的概念分割成两个或更多的子问题，以便各个击破。其实，任何一个可以用程序求解的问题所需的计算时间都与其规模有关，问题的规模越小，越容易直接求解。分割问题也是遇到大问题的解决方式，可以使子问题规模不断缩小，直到这些子问题简单到足以解决，最后将各子问题的解合并得到原问题的最终解答。这个算法应用相当广泛，如快速排序法（Quick Sort）、递归算法（Recursion）、大整数乘法。

下面我们就以一个实际的例子来说明。如果有 8 幅很难画的图，就可以分成两组各 4 幅画来完成；如果还是觉得复杂，就分成 4 组，每组各两幅画来完成。采用相同模式反复分割问题，这就是最简单的分治法的核心思想，如图 1-8 所示。

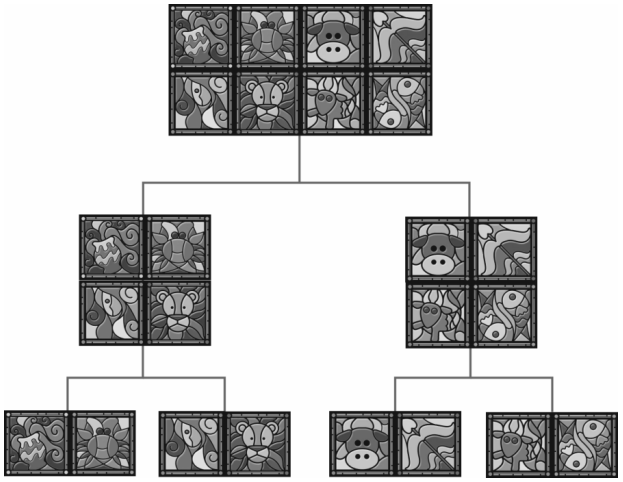

图 1-8

再举个例子，如果你被委派做一个项目的规划，规划这个项目有 8 个章节的主题，如果只靠一个人独立完成，不但时间比较长，而且有些规划的内容可能不是自己的专长，这时就可以按照这 8 个章节的特性分给 2 个项目负责人去完成。不过，为了让这个规划更快完成，又能找到适合的分类，再分别将其分割成 2 章，并分派给更多不同的项目成员，如此一来，每个成员只需负责其中 2 个章节，经过这样的分配，就可以将原先的大项目简化成 4 个小项目，并委派给 4 个成员去完成。

以此类推，根据分治法的核心思想，又可以将其切割成 8 个小主题，委派给 8 个成员去分别完成，因为参与人员较多，所以所需时间缩减到原先一个人独立完成的时间。这个例子的分治法解决方案的示意图如图 1-9 所示。

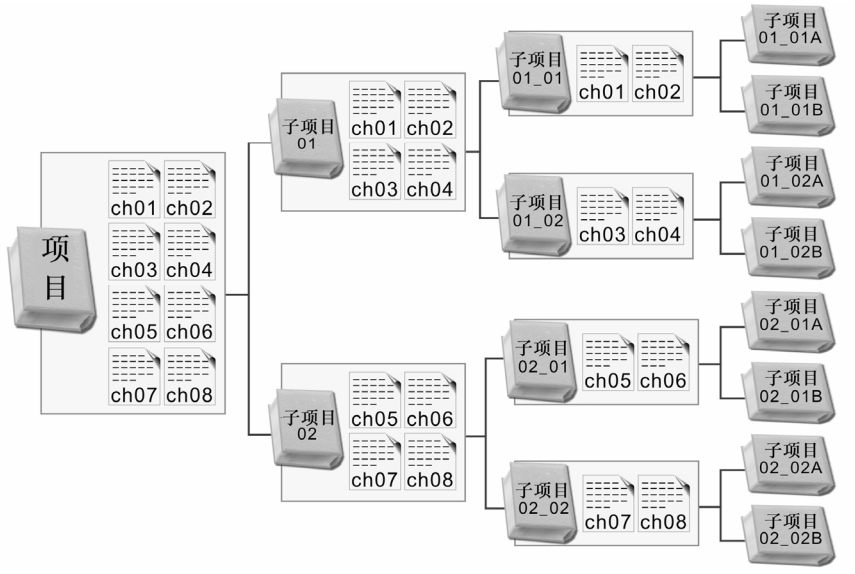

图 1-9

分治法也可以应用在数字的分类与排序上，如果要以人工的方式将散落在地上的打印稿从第 1 页整理并排序到第 100 页，可以有两种做法。一种方法是逐一捡起打印稿，并逐一按页码顺序插入到正确的位置。但是这样有一个缺点，就是排序和整理的过程较为繁杂，而且比较浪费时间。另一种方法是应用分治法的原理，先行将页码 1 到页码 10 放在一起，页码 11 到页码 20 放在一起，以此类推，将页码 91 到页码 100 放在一起，也就是说，将原先的 100 页分类为 10 个页码区间，然后分别对 10 堆页码进行整理，再从页码小到大的分组合并起来，轻易恢复到原先的稿件顺序。通过分治法可以让原先复杂的问题，变成规则更简单、数量更少、速度更快且更容易轻易解决的小问题。

1.2.2　递归法

递归是一种很特殊的算法，分治法和递归法很像一对孪生兄弟，都是将一个复杂的算法问题进行分解，让规模越来越小，最终使子问题容易求解。递归在早期人工智能所用的语言（如 Lisp、Prolog）中几乎是整个语言运行的核心，现在许多程序设计语言（包括 C、C++、Java、Python 等）都具备递归功能。简单来说，对程序设计人员的实现而言，“函数”（或称为子程序）不单纯只是能够被其他函数调用（或引用）的程序单元，在某些程序设计语言中还提供了自己调用自己的功能，这两种调用的功能就是所谓的“递归”。

从程序设计语言的角度来说，谈到递归的定义，可以这样来描述：假如一个函数或子程序是由自身所定义或调用的，就称为递归（Recursion）。它至少要定义两个条件，包括一个可以反复执行的递归过程与一个跳出执行过程的出口。

提　示

“尾递归”（Tail Recursion）就是函数或子程序的最后一条语句为递归调用，因为每次调用后，再回到前一次调用的第一条语句就是 return 语句，所以不需要再进行任何运算工作了。

阶乘函数是数学上很有名的函数，对递归法而言，也可以看成是很典型的范例，一般以符号“！”来代表阶乘。如 4 的阶乘可写为 4!，*n*!则表示为：

$$n! = n \times (n-1) * (n-2) * \ldots * 1$$

下面逐步分解它的运算过程，以观察出其规律性。

```
5!  = (5 * 4!)
    = 5 * (4 * 3!)
    = 5 * 4 * (3 * 2!)
    = 5 * 4 * 3 * (2 * 1)
    = 5 * 4 * (3 * 2)
    = 5 * (4 * 6)
    = (5 * 24)
    = 120
```

用 C 语言编写的 *n*!递归函数算法如下，请注意其中所应用的递归基本条件：一个反复的过程；一个递归终止的条件，确保有跳出递归过程的出口。

```
int factorial(int i)
{
    int sum;
    if(i == 0)  /* 递归终止的条件，跳出递归过程的出口 */
        return(1);
    else
        sum = i * factorial(i-1);  /* sum=n*(n-1)!，反复执行的递归过程 */
    return sum;
}
```

以上是用阶乘函数的范例来说明递归的运行方式，在系统中具体实现递归时，则要用到堆栈的数据结构。所谓堆栈（Stack），就是一组相同数据类型的集合，所有的操作均在这个结构的顶端进行，具有“后进先出”（Last In First Out，LIFO）的特性。有关堆栈的详细功能说明与实现，请参考第 2 章及第 6 章。

我们再来看著名的斐波那契数列（Fibonacci Polynomial）的求解。斐波那契数列的基本定义为：

$$F_n = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ F_{n-1}+F_{n-2} & n=2,3,4,5,6\ldots（n\text{ 为正整数}） \end{cases}$$

简单来说，这个数列的第 0 项是 0，第 1 项是 1，之后各项的值是由其前面两项值相加的结果（后面的每项值都是其前两项值的和）。根据斐波那契数列的定义，可以尝试把它设计成递归形式。

```
int fib(int n)
{
```

```
    if(n==0) return 0;
    if(n==1)
       return 1;
    else
       return fib(n-1) + fib(n-2);/*递归引用本身 2 次*/
}
```

【范例程序：CH01_01.c】

下面设计一个计算第 *n* 项斐波拉契数列的递归程序。

```
#include <stdio.h>
#include <stdlib.h>

int fib(int);                              /* fib()函数的原型声明 */

int main(void)
{
   int i,n;
   printf("请输入要计算到第几项斐波拉契数列：");
   scanf("%d",&n);

   for(i=0;i<=n;i++)                              /* 计算前 n 项斐波拉契数列 */
      printf("fib(%d)=%d\n",i,fib(i));


   return 0;
}

int fib(int n)                /* 定义函数 fib()*/
{

   if (n==0)
      return 0;                 /* 如果 n=0，则返回 0*/
   else if(n==1 || n==2)      /* 如果 n=1 或 n=2，则返回 1 */
      return 1;
   else                       /* 否则返回 fib(n-1) + fib(n-2) */
      return (fib(n-1) + fib(n-2));
}
```

【执行结果】参考图 1-10。

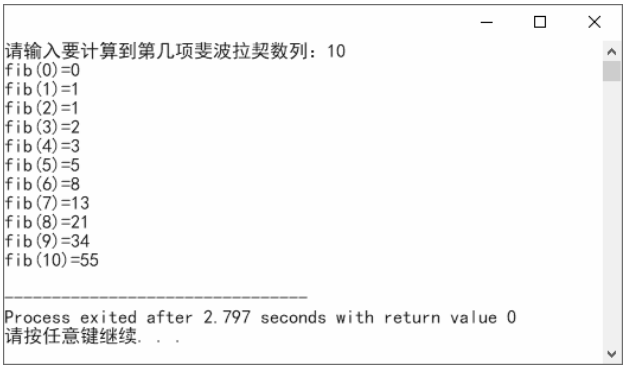

图 1-10

1.2.3 贪心法

贪心法（Greed Method）又称为贪婪算法，从某一起点开始，就是在每一个解决问题步骤中使用贪心原则，即采取在当前状态下最有利或最优化的选择，不断地改进该解答，持续在每一步骤中选择最佳的方法，并且逐步逼近给定的目标，当达到某一步骤不能再继续前进时算法停止，以尽可能快地求得更好的解。

贪心法的精神虽然是把求解的问题分成若干个子问题，不过不能保证求得的最后解是最佳的，贪心法容易过早做决定，只能求满足某些约束条件可行解的范围，不过在有些问题中却可以得到最佳解，经常用于求图的最小生成树（MST）、最短路径与霍哈夫曼编码等。

我们来看一个简单的例子（后面的货币系统不是现实的情况，只为了举例），如图 1-11 所示。假设我们今天去便利商店买了几听可乐，总价是 24 元，我们付给售货员 100 元，并且我们希望不要找太多硬币，即硬币的总数量最少，该如何找钱呢？假如目前的硬币有 50 元、10 元、5 元、1 元 4 种，从贪心法的策略来说，应找的钱总数是 76 元，所以一开始选择 50 元的硬币一枚，接下来就是 10 元的硬币两枚，最后是 5 元的硬币和 1 元的硬币各一枚，总共 5 枚硬币，这个结果也确实是最佳解答。

图 1-11

贪心法也适合用于旅游某些景点的判断，假如我们要从图 1-12 中的顶点 5 走到顶点 3，最短的路径该怎么走才好呢？以贪心法来说，当然是先走到顶点 1 最近，接着选择走到顶点 2，最后从顶点 2 走到顶点 5，这样的距离是 28。可是从图 1-12 中我们发现直接从顶点 5 走到顶点 3 才是最短的距离。也就是说，在这种情况下，是没有办法以贪心法规则来找到最佳解答的。

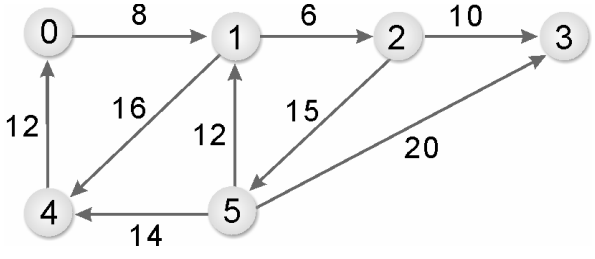

图 1-12

1.2.4　动态规划法

动态规划法（Dynamic Programming Algorithm，DPA）类似于分治法，在 20 世纪 50 年代初由美国数学家 R. E. Bellman 发明，用于研究多阶段决策过程的优化过程与求得一个问题的最佳解。动态规划法主要的做法是：如果一个问题答案与子问题相关，就能将大问题拆解成各个小问题，其中与分治法最大不同的地方是可以让每一个子问题的答案被存储起来，以供下次求解时直接取用。这样的做法不但能减少再次计算的时间，并可将这些解组合成大问题的解答，故而使用动态规划可以解决重复计算的问题。

例如，前面斐波拉契数列采用的是类似分治法的递归法，如果改用动态规划法，那么已计算过的数据就不必重复计算了，也不会再往下递归，因而实现了提高性能的目的。如果我们想求斐波拉契数列的第 4 项数 Fib(4)，那么它的递归过程可以用图 1-13 表示出来。

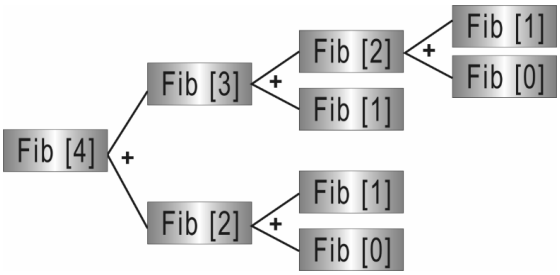

图 1-13

从上面的执行路径图中我们可以得知递归调用了 9 次，而执行加法运算 4 次，Fib(1)与 Fib(0)共执行了 3 次，重复计算影响了执行性能。我们根据动态规划法的思想，可以将算法修改如下（以 C 语言为例）：

```
int output[1000]={0}; //fibonacci 的暂存区

int fib(int n)
{
    int result;
    result=output[n];
    if (result==0)
    {
        if(n==0)
            return 0;
        if(n==1)
            return 1;
        else
           return (fib(n-1)+fib(n-2));
       output[n]=result;
    }
    return result;
}
```

1.2.5 迭代法

迭代法（Iterative Method）无法使用公式一次求解，而需要使用迭代。

【范例程序：CH01_02.c】

下面以 C 语言用 for 循环设计一个计算 1!~*n*!的阶乘程序。

```
/* 以 for 循环计算 n! */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int i,j,n,sum = 1;
    printf("请输入 n=");
    scanf("%d",&n);

    for(i=0;i<=n;i++)          /* 0~n 的阶乘 */
    {
        for(j=i;j>0;j--) /* n!=n*(n-1)*(n-2)*...*1 */
            sum *= j; /* sum=sum*j */
        printf("%d!=%3d\n",i,sum);
        sum = 1;
    }
    return 0;
}
```

【执行结果】参考图 1-14。

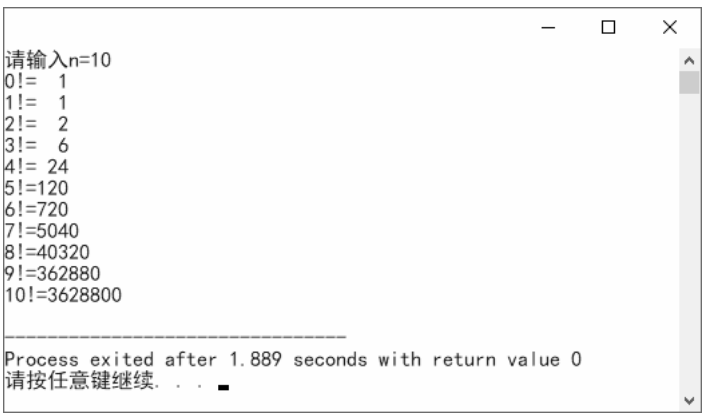

图 1-14

上述例子采用的是一种固定执行次数的迭代法，当遇到一个问题时，如果无法一次以公式求解，又不能确定要执行多少次，就可以使用 while 循环。

while 循环必须加入控制变量的起始值及递增或递减表达式，并且在编写循环过程时必须检查离开循环体的条件是否存在，如果条件不存在，就会让循环体一直执行而无法停止，导致“无限循环”。循环结构通常需要具备以下 3 个要件：

（1）变量初始值。

（2）循环条件判断表达式。

（3）调整变量增减值。

例如：

```
int i=0,sum=0;
while(i<10)
{
    i++;              /* 执行循环一次则加一，控制循环的条件变量 */
    sum=i+sum;
}
printf("%d!=%d",i,sum);
```

当 i 小于 10 时会执行 while 循环体内的语句，所以 i 会加 1，直到 i 等于 10。当条件判断表达式为 false 时，就会跳离循环了。

1.2.6　枚举法

枚举法（又称穷举法）是一种常见的数学方法，是我们在日常工作中使用比较多的一种算法，其核心思想就是列举所有的可能。根据问题要求，逐一列举问题的解答，或者为了便于解决问题，把问题分为不重复、不遗漏的有限种情况，逐一列举各种情况并加以解决，最终达到解决整个问题的目的。像枚举法这种分析问题、解决问题的方法，得到的结果总是正确的，缺点是速度太慢。

例如，我们想将 A 与 B 两个字符串连接起来，就是将 B 字符串中的每一个字符从第一个字符开始逐步连接到 A 字符串的最后一个字符，如图 1-15 所示。

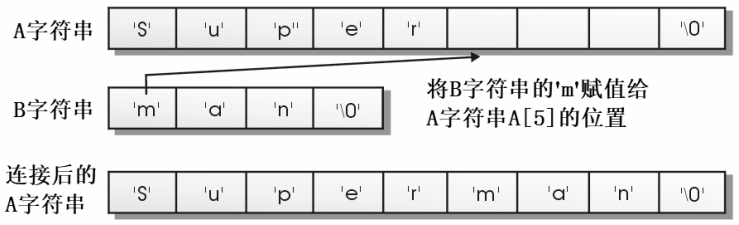

图 1-15

再来看一个例子：1000 依次减去 1，2，3……直到哪一个数时，相减的结果开始为负数？这是很纯粹的枚举法应用，只要按序减去 1，2，3，4，5，6，7，8……?

1000-1-2-3-4-5-6….-? < 0

用 C 语言写成的算法如下：

```
int x;
int num;
x=1;
num=1000;
while (num>=0) /* while 循环 */
{
```

```
        num-=x;
        x=x+1;
    }
    printf("%d",x-1);
```

简单来说，枚举法的核心概念就是将要分析的项目在不遗漏的情况下逐一列举出来，再从所列举的项目中找到自己所需要的目标对象。我们再举一个例子来加深大家的印象，如果我们希望列出 1~500 之间所有 5 的倍数（整数），用枚举法就是从 1 开始到 500 逐一列出所有的整数并枚举，同时检查该枚举的数字是否为 5 的倍数，如果不是，则不加以理会，如果是，则加以输出。

用 C 语言编写的算法如下：

```
    for(num=1; num<=500; num++)
        if (num%5 ==0)
            printf("%d 是 5 的倍数\n",num);
```

1.2.7 回溯法

回溯法（Backtracking）也算是枚举法中的一种。对于某些问题而言，回溯法是一种可以找出所有（或一部分）解的一般性算法，同时避免枚举不正确的数值。一旦发现不正确的数值，就不再递归到下一层，而是回溯到上一层，以节省时间，是一种走不通就退回再走的方式。它的特点主要是在搜索过程中寻找问题的解，当发现不满足求解条件时，就回溯（返回），尝试别的路径，避免无效搜索。

例如，老鼠走迷宫就是一种“回溯法”（Backtracking）的应用。老鼠走迷宫问题的陈述是：假设把一只大老鼠放在一个没有盖子的大迷宫盒的入口处，盒中有许多墙使得大部分的路径都被挡住而无法前进。老鼠可以按照尝试错误的方法找到出口。不过，这只老鼠必须具备走错路时就会退回来并把走过的路记下来，避免下次走重复的路，就这样直到找到出口为止。简单来说，老鼠行进时，必须遵守以下 3 个原则。

（1）一次只能走一格。
（2）遇到墙无法往前走时，则退回一步找找看是否有其他的路可以走。
（3）走过的路不会再走第二次。

在编写走迷宫程序之前，我们先来了解如何在计算机中表现一个仿真迷宫的方式。这时可以使用二维数组 MAZE[row][col]，并符合以下规则。

MAZE[*i*][*j*] = 1 表示[*i*][*j*]处有墙，无法通过；
= 0表示[*i*][*j*]处无墙，可通行；
MAZE[1][1]是入口，MAZE[*m*][*n*]是出口。

图 1-16 就是一个使用 10×12 二维数组的仿真迷宫地图。

假设老鼠从左上角的 MAZE[1][1]进入，从右下角的 MAZE[8][10]出来，老鼠当前位置以 MAZE[*x*][*y*]表示，那么我们可以将老鼠可能移动的方向表示成如图 1-17 所示。

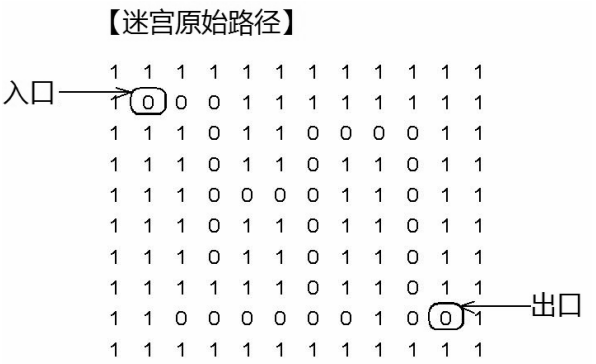

图 1-16

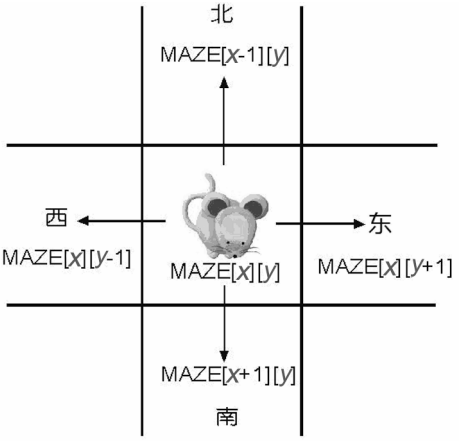

图 1-17

如图 1-17 所示，老鼠可以选择的方向共有 4 个，分别为东、西、南、北，但并非每个位置都有 4 个方向可以选择，必须看情况来决定，如 T 字形的路口，就只有东、西、南 3 个方向可以选择。

我们可以使用链表来记录走过的位置，并且将走过的位置对应的数组元素内容标记为 2，然后将这个位置放入堆栈再进行下一次的选择。如果走到死胡同并且还没有抵达终点，就退出上一个位置，并退回去直至回到上一个岔路后再选择其他的路。由于每次新加入的位置必定会在堆栈的顶端，因此堆栈顶端指针所指的方格编号便是当前搜索迷宫出口的老鼠所在的位置。如此重复这些动作直至走到出口为止。在图 1-18 和图 1-19 中，以小球来代表迷宫中的老鼠。

图 1-18

图 1-19

上面这样一个迷宫搜索的过程可以用下面的算法来加以描述。

```
01 if(上一格可走)
02 {
03     加入方格编号到堆栈;
04     往上走;
05     判断是否为出口;
```

```
06 }
07 else if(下一格可走)
08 {
09     加入方格编号到堆栈;
10     往下走;
11     判断是否为出口;
12 }
13 else if(左一格可走)
14 {
15     加入方格编号到堆栈;
16     往左走;
17     判断是否为出口;
18 }
19 else if(右一格可走)
20 {
21     加入方格编号到堆栈;
22     往右走;
23     判断是否为出口;
24 }
25 else
26 {
27     从堆栈删除一方格编号;
28     从堆栈中取出一方格编号;
29     往回走;
30 }
```

上面的算法是每次进行移动时所执行的操作，其主要是判断当前所在位置的上、下、左、右是否有可以前进的方格，若找到可前进的方格，则将该方格的编号压入到记录移动路径的堆栈中并向该方格移动；若四周没有可走的方格（第 25 行），也就是当前所在的方格无法走出迷宫，则必须退回到前一格重新检查是否有其他可走的路径。所以在上面算法中的第 27 行会将当前所在位置的方格编号从堆栈中删除，之后第 28 行再弹出的就是前一次所走过的方格编号。

以下是迷宫问题 C 程序的具体实现。

【范例程序：CH01_03.c】

使用堆栈结构来帮助找出老鼠走迷宫的路线，其中 0 表示墙、2 表示入口、3 表示出口、6 表示老鼠走过的路线。

```
01   #include <stdio.h>
02   #include <stdlib.h>
03   #define EAST  MAZE[x][y+1]  /*定义东方的相对位置*/
04   #define WEST  MAZE[x][y-1]  /*定义西方的相对位置*/
05   #define SOUTH MAZE[x+1][y] /*定义南方的相对位置*/
06   #define NORTH MAZE[x-1][y] /*定义北方的相对位置*/
07   #define ExitX 8            /*定义出口的 X 坐标在第 8 列*/
08   #define ExitY 10           /*定义出口的 Y 坐标在第 10 行*/
09   struct list
```

```
{
   int x,y;
   struct list* next;
};
typedef struct list node;
typedef node* link;
int MAZE[10][12] = {2,1,1,1,1,0,0,0,1,1,1,1,  /*声明迷宫数组*/
                    1,0,0,0,1,1,1,1,1,1,1,1,
                    1,1,1,0,1,1,0,0,0,0,1,1,
                    1,1,1,0,1,1,0,1,1,0,1,1,
                    1,1,1,0,0,0,0,1,1,0,1,1,
                    1,1,1,0,1,1,0,1,1,0,1,1,
                    1,1,1,0,1,1,0,1,1,0,1,1,
                    1,1,1,0,1,1,0,0,1,0,1,1,
                    1,1,0,0,0,0,0,0,1,0,0,1,
                    1,1,1,1,1,1,1,1,1,1,1,3};
link push(link stack,int x,int y)
{
   link newnode;
   newnode = (link)malloc(sizeof(node));
   if(!newnode)
   {
      printf("Error!内存分配失败!\n");
      return NULL;
   }
   newnode->x=x;
   newnode->y=y;
   newnode->next=stack;
   stack=newnode;
   return stack;
}
link pop(link stack,int* x,int* y)
{
   link top;
   if(stack!=NULL)
   {
      top=stack;
      stack=stack->next;
      *x=top->x;
      *y=top->y;
      free(top);
      return stack;
   }
   else
      *x=-1;
   return stack;
}
int chkExit(int x,int y,int ex,int ey)
{
   if(x==ex&&y==ey)
   {
      if(NORTH==1||SOUTH==1||WEST==1||EAST==2)
           return 1;
      if(NORTH==1||SOUTH==1||WEST==2||EAST==1)
           return 1;
      if(NORTH==1||SOUTH==2||WEST==1||EAST==1)
           return 1;
      if(NORTH==2||SOUTH==1||WEST==1||EAST==1)
           return 1;
```

```
69        }
70        return 0;
71    }
72
73    int main()
74    {
75        int i,j,x,y;
76        link path = NULL;
77        x=1;    /*入口的 X 坐标*/
78        y=1;    /*入口的 Y 坐标*/
79        printf("[迷宫模拟图(0 表示墙,2 表示入口,3 表示出口]\n"); /*打印出迷宫的路径图*/
80        for(i=0;i<10;i++)
81        {
82            for(j=0;j<12;j++)
83                 printf("%2d",MAZE[i][j]);
84            printf("\n");
85        }
86        while(x<=ExitX&&y<=ExitY)
87        {
88            MAZE[x][y]=6;
89            if(NORTH==0)
90            {
91                 x -= 1;
92                 path=push(path,x,y);
93            }
94            else if(SOUTH==0)
95            {
96                 x+=1;
97                 path=push(path,x,y);
98            }
99            else if(WEST==0)
100           {
101                y-=1;
102                path=push(path,x,y);
103           }
104           else if(EAST==0)
105           {
106                y+=1;
107                path=push(path,x,y);
108           }
109           else if(chkExit(x,y,ExitX,ExitY)==1) /*检查是否走到出口了*/
110                break;
111           else
112           {
113                MAZE[x][y]=2;
114                path=pop(path,&x,&y);
115           }
116       }
117       printf("----------------------------\n");
118       printf("[6 表示老鼠走过的路线]\n"); /*打印出老鼠走完迷宫后的路径图*/
119       printf("----------------------------\n");
120       for(i=0;i<10;i++)
121       {
122           for(j=0;j<12;j++)
123                printf("%2d",MAZE[i][j]);
124           printf("\n");
125       }
126
```

```
127      return 0;
128  }
```

【执行结果】参考图 1-20。

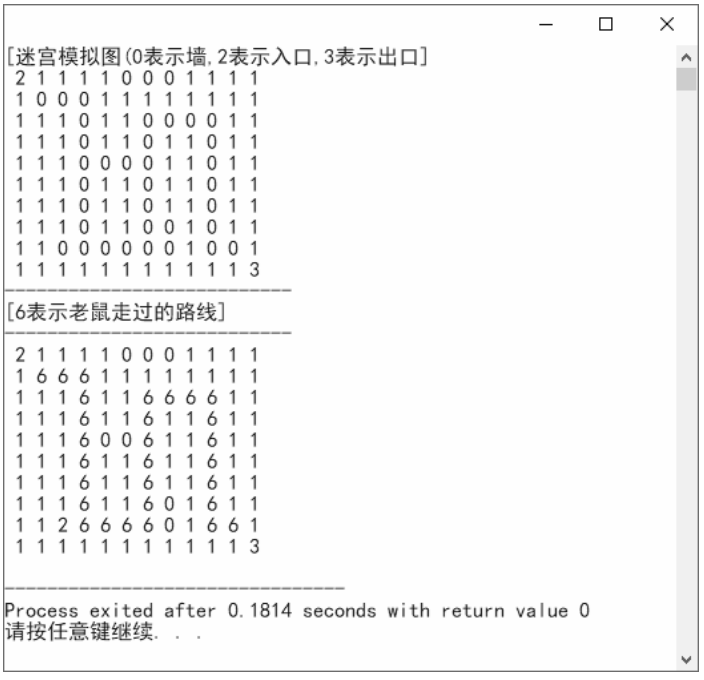

图 1-20

课后习题

1. 以下 C 程序片段是否相当严谨地表达出算法的含义？

```
count＝0;
while(count < > 3)
```

2. 在下列程序的循环部分中，实际执行的次数与时间复杂度是什么？

```
for i=1 to n
    for j=i to n
        for k =j to n
            { end of k Loop }
    { end of j Loop }
{ end of i Loop }
```

3. 试证明 $f(n) = a_m n^m + ... + a_1 n + a_0$，则 $f(n) = O(nm)$。

4. 以下程序的 Big-Oh 是什么？

```
Total=0;
for(i=1; i<=n ; i++)
    total=total+i*i;
```

5. 算法必须符合哪 5 个条件？
6. 试简述分治法的核心思想。
7. 递归至少要定义哪两个条件？
8. 试简述贪心法的主要核心概念。
9. 简述动态规划法与分治法的差异。
10. 什么是迭代法？试简述之。
11. 枚举法的核心概念是什么？试简述之。
12. 回溯法的核心概念是什么？试简述之。
13. 编写一个算法来求取函数 $f(n)$。$f(n)$的定义如下：

$$f(n)\begin{cases} n^n & n\geqslant 1 \\ 0 & \text{其他} \end{cases}$$

第 2 章

常用数据结构

人们设计和制造计算机的主要原因之一就是用它们来存储和管理一些数字化的数据和信息。当我们要求计算机解决问题时，必须以计算机了解的模式来描述问题。数据结构是数据的表示方法，也就是指计算机中存储数据的方法。我们可以将数据结构看成是在数据处理过程中一种分析、存储、组织数据的方法与逻辑，它考虑到了数据之间的特性与相互关系。简单来说，数据结构的定义就是一种程序设计优化的方法论，不仅讨论到存储的数据，同时也考虑到彼此之间的关系与运算，目的是加快程序的执行速度与减少内存占用的空间。例如，图书馆的书籍管理就是一种数据结构的应用，如图 2-1 所示。

图 2-1

2.1 认识数据结构

在信息技术发达的今日，我们日常的生活已经和计算机密不可分了。计算机与数据是息息相关的，并且计算机具有处理速度快与存储容量大两大特点（见图 2-2），因而在数据处理的角色上更为举足轻重。数据结构和相关的算法就是数据进入计算机进行处理的一套完整逻辑。在进行程序设计时，对于要存储和处理的一类数据，程序员必须选择一种数据结构来进行这类数据的添加、修改、删除、存储等操作，如果在选择数据结构时做了错误的决定，那么程序执行起来将可能变得非

常低效，如果选错了数据类型，后果就更加不堪设想了。

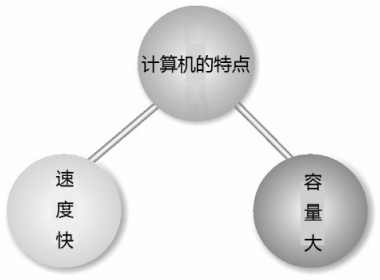

图 2-2

例如，医院会将事先设计好的个人病历表格准备好，当有新的病人上门时，就请他们自行填写，随后管理人员可能按照某种次序（例如姓氏或年龄）将病历表加以分类，然后用文件夹或档案柜加以收藏。日后当病人回诊时，只要询问病人的姓名或年龄，管理人员就可以快速地从文件夹或档案柜中找出病人的病历表。这个档案柜中所存放的病历表就是一种数据结构概念的应用，如图 2-3 所示。

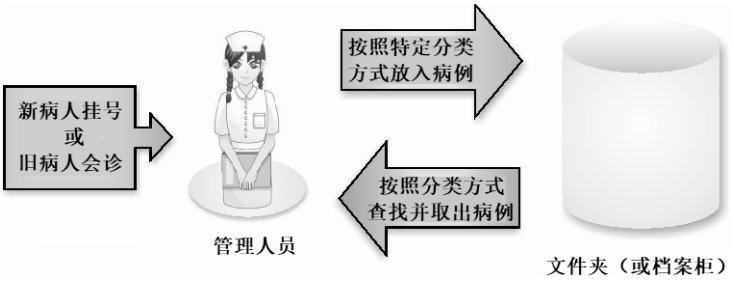

图 2-3

“数据表”（见图 2-4）中的数据结构就是一种二维矩阵，纵的方向称为“列”（Column，或者“栏”），横的方向称为“行”（Row），每一张数据表的最上面一行用来存放数据项的名称，称为“字段名”（Field Name），而除了字段名这一行之外，其他都用来存放一项项数据，称之为“值”（Value）。

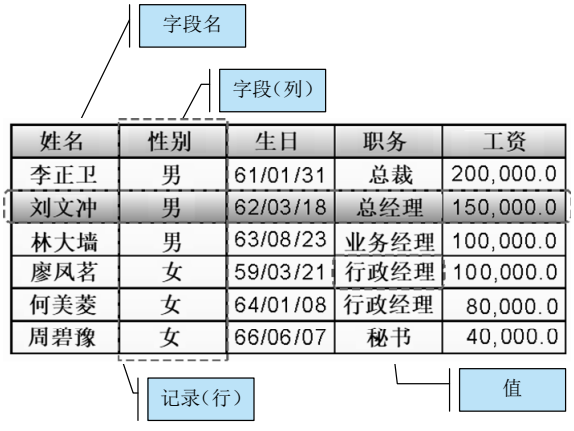

图 2-4

数据与信息

谈到数据结构，首先必须了解什么是数据（Data）与信息（Information）。从字义上来看，数据（Data）指的是一种未经处理的原始文字（Word）、数字（Number）、符号（Symbol）或图形（Graph）等。我们可将数据分为两大类：一类为数值数据（Numeric Data），例如 0, 1, 2, 3, …, 9 等所组成的可用运算符（Operator）来进行运算的数据；另一类为字符数据（Alphanumeric Data），像 A, B, C, …, +,*等非数值数据（Non-Numeric Data）。例如，姓名或我们常看到的课表、通讯录等都可泛称是一种“数据”（Data）。

信息（Information）就是利用大量的数据，经过有系统的整理、分析、筛选处理而提炼出来的，且具有参考价格以及提供决策依据的文字、数字、符号或图表。在近代的“信息革命”浪潮中，如何掌握信息、利用信息可以说是个人或事业团体发展成功的重要原因。充分发挥计算机的优势，更能让信息的价值发挥到淋漓尽致的境界。

不过，大家可能会有疑问：“那么数据和信息的角色是否绝对一成不变呢？”这倒也不一定，同一份文件可能在某种情况下为数据，而在另一种情况下为信息。例如，“广州市每周的平均气温是 25℃”，这段文字只是陈述事实的一种数据，我们并无法判定广州市是一个炎热或者凉爽的城市。

例如，一个学生的语文成绩是 90 分，我们可以说这是一项成绩的数据，不过无法判断它具备什么含义。如果经过排序（Sorting）等处理，就可以知道这个学生语文成绩在班上同学中的名次，也就清楚了在这班学生中成绩相对的优良程度，这时它就成为一种信息，而排序是数据结构的一种应用。

从严谨的角度来形容“数据处理”，就是用人力或机器设备对数据进行系统的整理，如记录、排序、合并、计算、统计等，以使原始的数据符合需求，成为有用的信息。图 2-5 所示即为使用计算机进行数据处理的过程。

图 2-5

数据结构主要是表示数据在计算机内存中所存储的位置及其模式，通常可以分为以下 3 种类型。

（1）基本数据类型（Primitive Data Type）

不能以其他类型来定义的数据类型，或称为标量数据类型（Scalar Data Type），几乎所有的程序设计语言都会为标量数据类型提供一组基本数据类型，例如 C 语言中的基本数据类型就包括了 int、float、double、char、void 等。

（2）结构化数据类型（Structured Data Type）

结构化数据类型也称为虚拟数据类型（Virtual Data Type），是一种比基本数据类型更高一级的数据类型，例如字符串（String）、数组（Array）、指针（Pointer）、列表（List）、文件（File）等。

（3）抽象数据类型（Abstract Data Type，ADT）

我们可以将一种数据类型看成是一种值的集合，以及在这些值上所进行的运算及其所代表的属性所成的集合。“抽象数据类型”（Abstract Data Type，ADT）比结构数据类型更高级，是指一个数学模型以及定义在此数学模型上的一组数学运算或操作。也就是说，ADT 在计算机中表示的是一种“信息隐藏”（Information Hiding）的程序设计思想以及信息之间某一种特定的关系模式。例如，堆栈（Stack）就是一种典型数据抽象类型，具有后进先出（Last In First Out）的数据操作方式。

2.2 数据结构的种类

数据结构可通过程序设计语言所提供的数据类型、引用及其他操作加以实现，我们知道一个程序能否快速而高效地完成预定的任务取决于是否选对了数据结构，而程序是否能清楚而正确地把问题解决则取决于算法，所以我们可以认为“数据结构加上算法等于高效的可执行程序”，如图 2-6 所示。

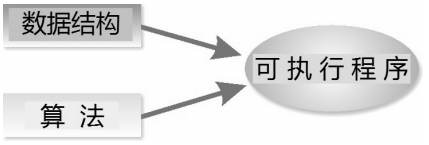

图 2-6

不同种类的数据结构适合于不同种类的应用，选择适当的数据结构是让算法发挥最大效能的主要考虑因素，精心选择的数据结构可以带来最优效率的算法。然而，不管是哪种情况，数据结构的选择都是至关重要的。下面我们将为大家介绍一些常见的数据结构。

2.2.1 数组

“数组”（Array）结构其实就是一排紧密相邻的可数内存，并提供了一个能够直接访问单一数据内容的计算方法。我们其实可以想象一下自家的信箱，每个信箱都有住址，其中路名就是名称，而信箱号码就是索引（注：在数组中也称为“下标”），如图 2-7 所示。邮递员可以按照信件上的住址把信件直接投递到指定的信箱中，这就好比程序设计语言中数组的名称表示一块紧密相邻内存的起始位置，而数组的索引（或下标）功能则用来表示从此内存起始位置的第几个区块。

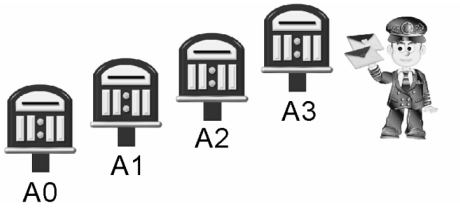

图 2-7

通常数组的使用可以分为一维数组、二维数组与多维数组等，其基本的工作原理都相同。例如，下面的 C 语言语句声明了一个名称为 Score、长度为 5 的数组（Array，示意图如图 2-8 所示）：

```
int Score[5];
```

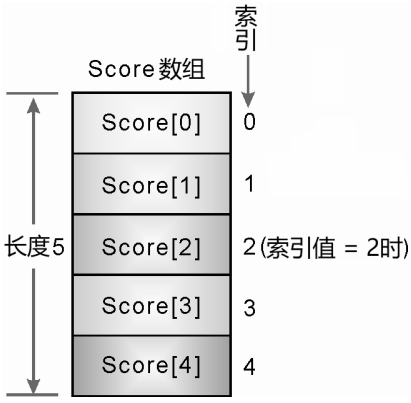

图 2-8

1．二维数组

二维数组（Two-dimension Array）可视为一维数组的扩展，都是用于处理数据类型相同的数据，差别只在于维数的声明。例如，一个含有 *m***n* 个元素的二维数组 *A* (1:*m*, 1:*n*)，*m* 代表行数，*n* 代表列数。例如，*A*[4][4]数组中各个元素在直观平面上的排列方式如图 2-9 所示。

图 2-9

在 C 语言中，二维数组的声明格式如下：

```
数据类型 二维数组名[行大小][列大小];
```

以数组 number [2][3]来说明，number 为一个 2 行 3 列的二维数组，也可以视为 2*3 的矩阵。在存取二维数组中的元素时，使用的索引值仍然是从 0 开始计算。在二维数组设置初始值时，为了方便区分行与列，除了最外层的{}外，最好以{}括住每一行元素的初始值，并以“,”隔开每个数

组元素，语法如下：

```
数据类型 数组名[n][列大小]={ {第 0 行初值},{第 1 行初值},…,{第 n-1 行初值} }
```

例如：

```
int number [2][3]={{1,2,3},{2,3,4}};
```

上面的 number[0]或称为第一行的索引，存放着另一个数组；number[1]或称为第二行的索引，存放着另一个数组，以此类推。第一行索引有 3 列，分别存放着 3 个元素，其位置 number[0][0]存储着数值 1，number[0][1]存储着数值 2，以此类推。所以 number 是 2*3 的二维数组，其行和列的索引示意如表 2-1 所示。

表 2-1　2*3 的二维数组示意

	列索引[0]	列索引[1]	列索引[2]
行索引[0]	1	2	3
行索引[1]	2	3	4

2．三维数组

现在让我们来看看三维数组（Three-dimension Array）。基本上三维数组的表示法和二维数组一样，都可视为一维数组的延伸，如果数组为三维数组，就可以看作是一个立方体。

将 arr[2][3][4]三维数组想象成空间上的立方体，如图 2-10 所示。

例如，在 C 语言中三维数组声明的方式如下：

```
int num[2][3][3]={{{33,45,67},
                   {23,71,56},
                   {55,38,66}},
                  {{21,9,15 },
                   {38,69,18},
                   {90,101,89}}};//声明三维数组
```

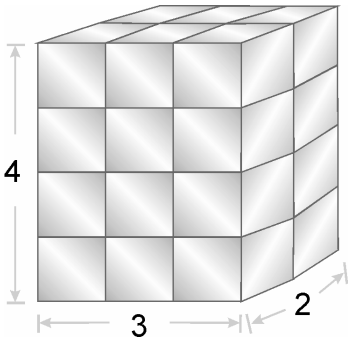

图 2-10

2.2.2　链表

链表（Linked List）是由许多相同数据类型的数据项按特定顺序排列而成的线性表。链表的特性是各个数据项在计算机内存中的位置是不连续且随机（Random）存放的，其优点是数据的插入

或删除都相当方便，有新数据加入就向系统申请一块内存空间，而数据被删除后，就可以把这块内存空间还给系统，加入和删除都不需要移动大量的数据。其缺点就是设计数据结构时较为麻烦，并且在查找数据时也无法像静态数据（如数组）那样可随机读取数据，必须按序查找到该数据为止。

日常生活中有许多链表的抽象运用，例如可以把“单向链表”想象成火车（见图 2-11），有多少人就挂多少节对应的车厢，当假日人多时，需要较多车厢时就可多挂些车厢，人少时就把车厢数量减少，十分具有弹性。

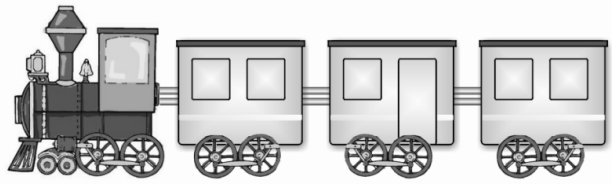

图 2-11

在动态分配内存空间时，最常使用的就是“单向链表”（Single Linked List）。一个单向链表节点基本上是由数据字段和指针两个元素所组成的，指针将会指向下一个元素在内存中的地址，如图 2-12 所示。

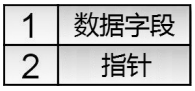

图 2-12

在“单向链表”中第一个节点是“链表头指针”，指向最后一个节点的指针设为 NULL，表示它是“链表尾”，不指向任何地方。例如，列表 A={a, b, c, d, x}，其单向链表的数据结构如图 2-13 所示。

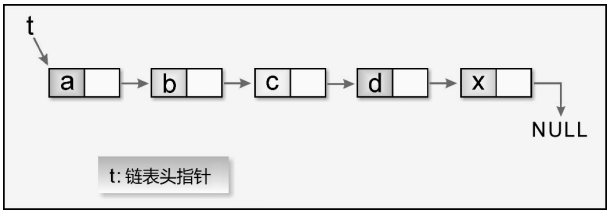

图 2-13

由于单向链表中所有节点都知道节点本身的下一个节点在哪里，但是对于前一个节点却没有办法知道，所以在单向链表的各种操作中，“链表头指针”就显得相当重要，只要存在链表头指针，就可以遍历整个链表、进行加入和删除节点等操作。注意，除非必要，否则不可移动链表头指针。

2.2.3　堆栈

堆栈（Stack）是一群相同数据形态的组合，所有的动作均在顶端进行，具有“后进先出”（Last In First Out，LIFO）的特性。所谓“后进先出”的概念，其实就如同自助餐中餐盘从桌面往上一个一个叠放，顾客取用时则从最上面的餐盘开始拿，如图 2-14 所示，这就是典型堆栈概念的应用。

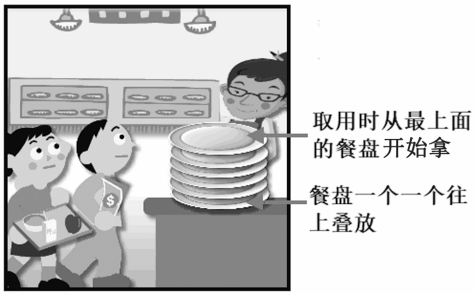

图 2-14

堆栈是一种抽象型数据结构（Abstract Data Type，ADT），具有下列特性（参考图 2-15）：

（1）只能从堆栈的顶端存取数据。

（2）数据的存取符合“后进先出”的原则。

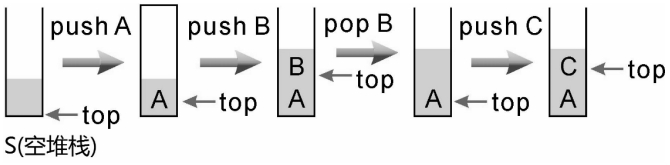

图 2-15

堆栈的基本运算如表 2-2 所示。

表 2-2　堆栈的基本运算

运算	说明
create	创建一个空堆栈
push	把数据存压入堆栈顶端，并返回新堆栈
pop	从堆栈顶端弹出数据，并返回新堆栈
empty	判断堆栈是否为空堆栈，是则返回 true，不是则返回 false
full	判断堆栈是否已满，是则返回 true，不是则返回 false

堆栈 push 和 pop 的操作示意图如图 2-16 所示。

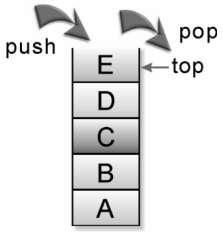

图 2-16

2.2.4　队列

队列（Queue）和堆栈都是有序列表，也属于抽象型数据类型（ADT），所有加入与删除的动

作都发生在不同的两端，并且符合“First In First Out”（先进先出）的特性。队列的概念就好比乘坐火车时买票的队伍，先到的人自然可以优先买票，买完票后就从前端离去准备乘坐火车，而队伍的后端又陆续有新的乘客加入，如图 2-17 所示。

图 2-17

队列在计算机领域的应用也相当广泛，如计算机的模拟（Simulation）、CPU 的作业调度（Job Scheduling）、外围设备联机并发处理系统（Spooling）的应用与图形遍历的广度优先搜索法（BFS）。堆栈只需一个顶端 top，指针指向堆栈顶端；而队列则必须使用 front 和 rear 两个指针分别指向队列前端和队列尾端，如图 2-18 所示。

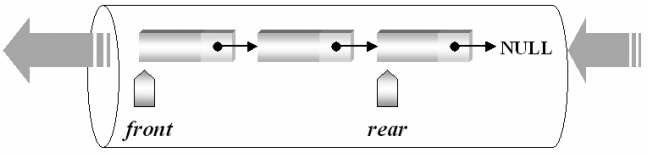

图 2-18

队列是一种抽象数据结构，具有下列特性：

（1）具有先进先出（FIFO）的特性。

（2）拥有两种基本操作，即加入与删除，而且使用 front 与 rear 两个指针分别指向队列的前端与末尾。

队列的基本运算如表 2-3 所示。

表 2-3　队列的基本运算

运算	说明
Create	建立空队列
Add	将新数据加入队列的尾端，返回新队列
Delete	删除队列前端的数据，返回新队列
Front	返回队列前端的值
Empty	若队列为空集合，则返回 true，否则返回 false

2.3 树结构

树结构（或称为树形结构）是一种日常生活中应用相当广泛的非线性结构，包括企业内的组织结构、家族的族谱、篮球赛程等。另外，在计算机领域中的操作系统与数据库管理系统都是树结构，比如 Windows、UNIX 操作系统和文件系统均是树结构的应用。图 2-19 所示是 Windows 的文件资源管理器，就是以树结构来存储各种文件的。

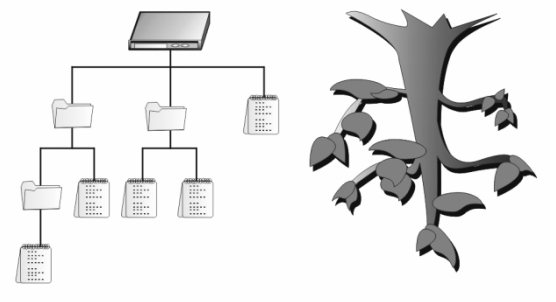

图 2-19

例如，在年轻人喜爱的大型网络游戏中，需要获取某些物体所在的地形信息，如果程序是依次从构成地形的模型三角面寻找，往往就会耗费许多运行时间，非常低效。因此，程序员一般会使用树结构中的二叉空间分割树（BSP tree）、四叉树（Quadtree）、八叉树（Octree）等来代表分割场景的数据，如图 2-20 和图 2-21 所示。

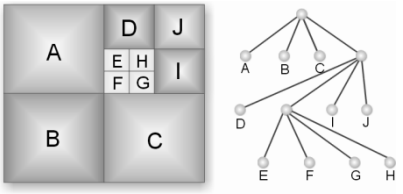

图 2-20

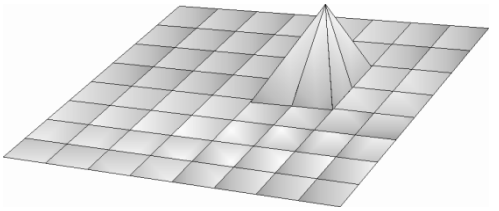

图 2-21

2.3.1 树的基本概念

树（Tree）是由一个或一个以上的节点（Node）组成的。树中存在一个特殊的节点，称为树根（Root）。每个节点都是一些数据和指针组合而成的记录。除了树根，其余节点可分为 $n \geqslant 0$ 个互斥的集合，即 $T_1, T_2, T_3, \ldots, T_n$，其中每一个子集合本身也是一种树结构，即此根节点的子树。在图 2-22 中，A 为根节点，B、C、D、E 均为 A 的子节点。

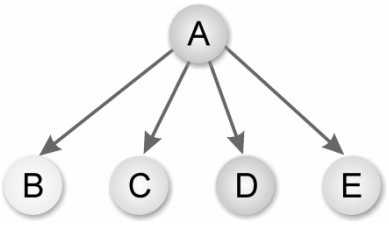

图 2-22

一棵合法的树，节点间虽可以互相连接，但不能形成无出口的回路。例如，图 2-23 就是一棵不合法的树。

树还可组成森林（Forest）。也就是说，森林是由 n 个互斥树的集合（$n \geqslant 0$）移去树根形成的。图 2-24 所示就是包含了 3 棵树的森林。

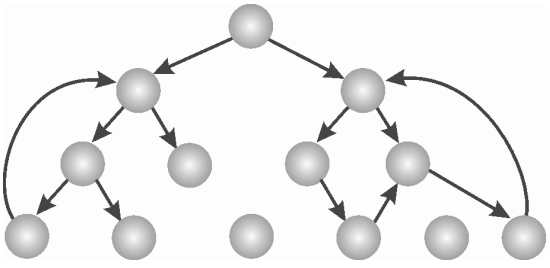

图 2-23

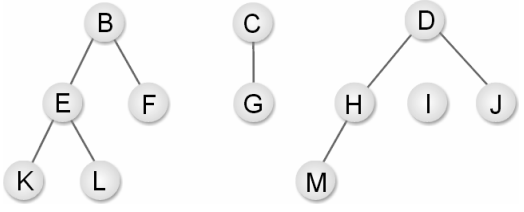

图 2-24

2.3.2　树结构专有名词的简介

在树结构中，有许多常用的专有名词，本小节将以图 2-25 中这棵合法的树来为大家详细介绍。

- 度数（Degree）：每个节点所有子树的个数。例如，图 2-25 中节点 B 的度数为 2，D 的度数为 3，F、K、I、J 等的度数为 0。
- 层数（Level）：树的层数，假设树根 A 为第一层，那么 B、C、D 节点的层数为 2，E、F、G、H、I、J 的层数为 3。
- 高度（Height）：树的最大层数。图 2-25 所示的树的高度为 4。
- 树叶或称终端节点（Terminal Node）：度数为零的节点就是树叶。例如，图 2-25 中的 K、L、F、G、M、I、J 就是树叶；图 2-26 则有 4 个树叶节点，即 E、C、H、I。

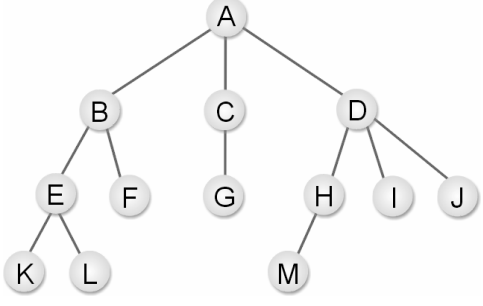

图 2-25

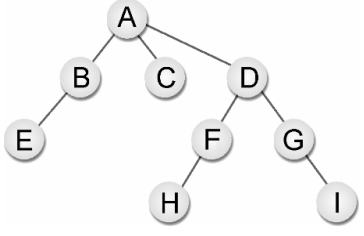

图 2-26

- 父节点（Parent）：与一个节点连接的上一层节点。在图 2-25 中，F 的父节点为 B，而 B 的父节点为 A。通常我们在绘制树形图时，会将父节点画在子节点的上方。
- 子节点（Children）：与一个节点连接的下一层节点。还是看图 2-25，A 的子节点为 B、C、D，而 B 的子节点为 E 、F。
- 祖先（Ancestor）和子孙（Descendent）：所谓祖先，是指从树根到该节点路径上所包含的节点，而子孙则是在该节点往下追溯子树中的任一节点。在图 2-25 中，K 的祖先为 A、B、E 节点，H的祖先为 A、D 节点，节点 B 的子孙为 E、F、K、L。
- 兄弟节点（Sibling）：有共同父节点的节点。在图 2-25 中，B、C、D 为兄弟节点，H、I、J 也为兄弟节点。
- 非终端节点（Nonterminal Node）：树叶以外的节点，如图 2-25 中的 A、B、C、D、E、H 等。
- 同代（Generation）：在同一棵树中具有相同层数的节点，如图 2-25 中的 E、F、G、H、I、J，或是 B、C、D。
- 森林（Forest）：*n* 棵（$n \geqslant 0$）互斥树的集合。例如，图 2-27 为包含 3 棵树的森林。

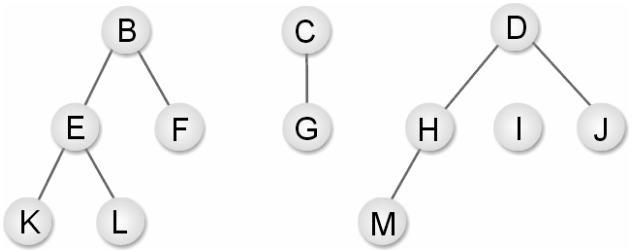

图 2-27

2.3.3 二叉树

一般树结构在计算机内存中的存储方式是以链表（Linked List）为主的。对于 *n* 叉树（*n*-way 树）来说，因为每个节点的度数都不相同，所以我们必须为每个节点都预留存放 *n* 个链接字段的最大存储空间。每个节点的数据结构如下：

请大家特别注意，这种 *n* 叉树十分浪费链接存储空间。假设此 *n* 叉树有 *m* 个节点，那么此树共有 *n***m* 个链接字段。另外，因为除了树根外，每一个非空链接都指向一个节点，所以得知空链接个数为 *n***m* − (*m*-1) = *m**(*n*−1) + 1，而 *n* 叉树的链接浪费率为 $\frac{m*(n-1)+1}{m*n}$。因此，我们可以得出以下结论：

- *n*=2 时，2 叉树的链接浪费率约为 1/2;
- *n*=3 时，3 叉树的链接浪费率约为 2/3;
- *n*=4 时，4 叉树的链接浪费率约为 3/4;

……

因为当 $n=2$ 时，它的链接浪费率最低，所以为了改进存储空间浪费的缺点，我们经常使用二叉树（Binary Tree）结构来取代其他树结构。

二叉树（又称为 Knuth 树）是一个由有限节点所组成的集合。此集合可以为空集合，或者由一个树根及其左右两个子树所组成。简单地说，二叉树最多只能有两个子节点，就是度数小于或等于 2。其计算机中的数据结构如下：

二叉树和一般树的不同之处整理如下：

（1）树不可为空集合，但是二叉树可以。

（2）树的度数为 $d \geqslant 0$，但二叉树的节点度数为 $0 \leqslant d \leqslant 2$。

（3）树的子树间没有次序关系，二叉树则有。

下面我们来看一棵实际的二叉树（见图 2-28）。

图 2-28 是以 A 为根节点的二叉树，且包含了以 B、D 为根节点的两棵互斥的左子树和右子树，如图 2-29 所示。

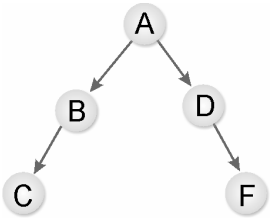

图 2-28

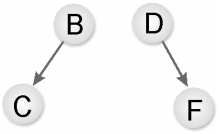

图 2-29

以上这两棵左右子树都属于同一种树结构，不过却是两棵不同的二叉树结构，原因就是二叉树必须考虑前后次序的关系，这点大家要特别注意。

2.4　图论简介

树结构描述节点与节点之间“层次”的关系，图结构（见图 2-30）讨论两个顶点之间“连通与否”的关系。在图中连接两顶点的边如果填上加权值（成本），则称这类图为“网络”。

图 2-30

图论（Graph Theory）起源于 1736 年，是一位瑞士数学家欧拉（Euler）为了解决“哥尼斯堡”问题所想出来的一种数据结构理论，这就是著名的“七桥问题”（见图 2-31）。简单来说，就是有七座横跨四个城市的大桥。欧拉所思考的问题是这样的，“是否有人在只经过每一座桥梁一次的情况下，把所有地方都走过一次而且回到原点。”

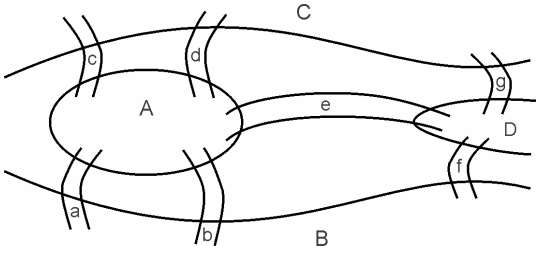

图 2-31

欧拉当时使用的方法就是以图结构来进行分析的。他以顶点表示城市，以边表示桥梁，并定义连接每个顶点的边数为该顶点的度数。于是可以用图 2-32 所示的简图来表示“哥尼斯堡桥梁”问题。

最后欧拉得出一个结论：“当所有顶点的度数都为偶数时，才能从某顶点出发，经过每条边一次，再回到起点。”也就是说，在图 2-32 中每个顶点的度数都是奇数，所以欧拉所思考的问题是不可能发生的，这个就是有名的“欧拉环”（Eulerian Cycle）理论。

但是，如果条件改成从某顶点出发，经过每条边一次，不一定要回到起点，即只允许其中两个顶点的度数是奇数，其余必须为偶数，符合这样的结果就称为欧拉链（Eulerian Chain），如图 2-33 所示。

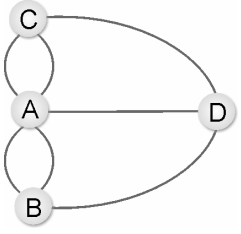

图 2-32

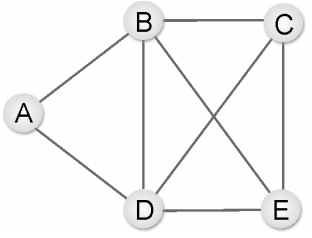

图 2-33

图的定义

图是由“顶点”和“边”所组成的集合，通常用 $G = (V, E)$ 来表示，其中 V 是所有顶点组成的集合，而 E 代表所有边组成的集合。图的种类有两种：一种是无向图；另一种是有向图。无向图以（V_1, V_2）表示其边，有向图则以<V_1, V_2>表示其边。

1．无向图

无向图（Graph）是一种边没有方向的图，即具有相同边的两个顶点没有次序关系，例如（V_1, V_2）与（V_2, V_1）代表的是相同的边，如图 2-34 所示。

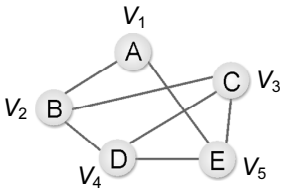

图 2-34

```
V={A,B,C,D,E}
E={(A,B),(A,E),(B,C),(B,D),(C,D),(C,E),(D,E)}
```

2．有向图

有向图（Digraph）是一种每一条边都可使用有序对<V_1, V_2>来表示的图，并且<V_1, V_2>与<V_2, V_1>是表示两个方向不同的边，而所谓<V_1, V_2>，是指 V_1 为尾端指向为头部的 V_2，如图 2-35 所示。

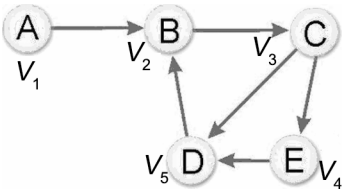

图 2-35

```
V={A,B,C,D,E}
E={<A,B>,<B,C>,<C,D>,<C,E>,<E,D>,<D,B>}
```

2.5　哈希表

哈希表是一种存储记录的连续内存，通过哈希函数的应用，可以快速存取与查找数据。基本上，所谓哈希法（Hashing）就是将本身的键（Key），通过特定的数学函数运算或使用其他的方法，转换成相对应的数据存储地址，如图 2-36 所示。注：哈希法所使用的数学函数称为“哈希函数”（Hashing Function）。另外，Key 在不混淆“键-值对”（Key-Value Pair）时也可以称之为键值。

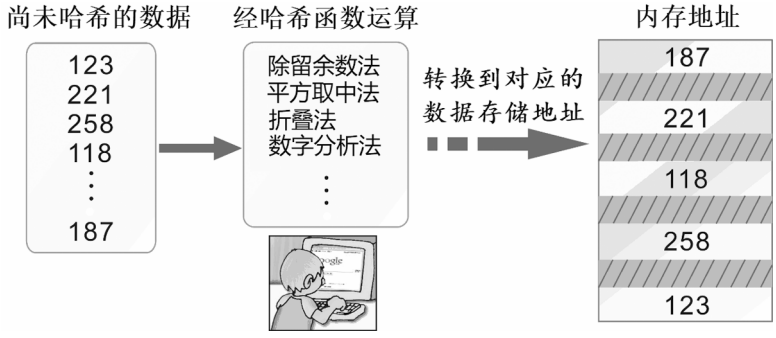

图 2-36

先来了解一下有关哈希函数的相关名词：

- Bucket（桶）：哈希表中存储数据的位置，每一个位置对应唯一的地址（Bucket Address）。桶就好比存在一个记录的位置。
- Slot（槽）：每一个记录中可能包含多个字段，而 Slot 指的就是“桶”中的字段。
- Collision（碰撞）：两个不同的数据经过哈希函数运算后对应到相同的地址。
- 溢出：如果数据经过哈希函数运算后所对应的 Bucket 已满，就会使 Bucket 发生溢出。
- 哈希表：存储记录的连续内存。哈希表是一种类似数据表的索引表格，其中可分为 *n* 个 Bucket，每个 Bucket 又可分为 *m* 个 Slot，如表 2-4 所示。

表 2-4　哈希表

	索引	姓名	电话
Bucket→	0001	Allen	07-772-1234
	0002	Jacky	07-772-5525
	0003	May	07-772-6604
		↑slot	↑slot

- 同义词（Synonym）：当两个标识符 I1 和 I2 经过哈希函数运算后所得的数值相同，即 f(I1) = f(I2)，就称 I1 与 I2 对于 f 这个哈希函数是同义词。
- 加载密度（Loading Factor）：标识符的使用数目除以哈希表内槽的总数，即

α（加载密度）= n（标识符的使用数目）/ [s（每一个桶内的槽数） * b（桶的数目）]

α 值越大，表示哈希空间的使用率越高，碰撞或溢出的概率也会越高。

- 完美哈希（Perfect Hashing）：没有碰撞也没有溢出的哈希函数。

在设计哈希函数时应该遵循以下原则：

（1）避免碰撞和溢出的发生。

（2）哈希函数不宜过于复杂，越容易计算越佳。

（3）尽量把文字的键值转换成数字的键值，以利于哈希函数的运算。

（4）所设计的哈希函数计算得到的值尽量能均匀地分布在每一桶中，不要过于集中在某些桶中，这样既可以降低碰撞又能减少溢出。

课后习题

1. 解释抽象数据类型。

2. 简述数据与信息的差异。

3. 数据结构主要是表示数据在计算机内存中所存储的位置和模式，通常可以分为哪三种类型？

4. 试简述一个单向链表节点字段的组成。

5. 简要说明堆栈与队列的主要特性。

6. 什么是欧拉链理论？试绘图说明。

7. 解释下列哈希函数的相关名词。

（1）桶（Bucket）

（2）同义字

（3）完美哈希

（4）碰撞

8. 一般树结构在计算机内存中的存储方式是以链表为主的，对于 n 叉树来说，我们必须取 n 为链接个数的最大固定长度，试说明为了改进存储空间浪费的缺点为何经常使用二叉树结构来取代树结构。

第 3 章

排序算法

排序（Sorting）算法几乎可以说是最常使用到的一种算法，其目的是将一串不规则的数据按照递增或递减的方式重新排列。随着大数据和人工智能（Artificial Intelligence，AI）技术的普及和应用，企业所拥有的数据量都在成倍增长，排序算法成为不可或缺的重要工具之一。即使在大家爱玩的各种电子游戏中，排序算法也无处不在。例如，在游戏中，在处理多边形模型中隐藏面消除的过程时，不管场景中的多边形有没有挡住其他的多边形，只要按照从后到前的顺序光栅化图形就可以正确地显示出所有可见的图形。其实就是可以沿着观察方向，按照多边形的深度信息对它们进行排序处理，如图 3-1 所示。

图 3-1

提　示

光栅处理的主要作用是将 3D 模型转换成能够被显示于屏幕的图像，并对图像进行修正和进一步美化处理，让展现在眼前的画面能更加逼真与生动。

人工智能的概念最早是由美国科学家 John McCarthy 于 1955 年提出的，目标是使计算机具有类似人类学习解决复杂问题与进行思考的能力。简单地说，人工智能就是由计算机所仿真或执行的具有类似人类智慧或思考的行为，如推理、规划、解决问题及学习等能力。

3.1　认识排序

排序（Sorting）功能对于计算机相关领域而言是一项非常重要并且普遍的工作。所谓排序，就是指将一组数据，按特定规则调换位置，使数据具有某种顺序关系（递增或递减）。用以排序的依据被称为键（Key，或键值）。通常，键值的数据类型有数值类型、中文字符串类型以及非中文字符串类型三种。

在比较的过程中，如果键值为数值类型，就直接以数值的大小作为键值大小比较的依据；如果键值为中文字符串，就按照该中文字符串从左到右逐字进行比较，并以该中文内码（例如：中文繁体 BIG5 码、中文简体 GB 码）的编码顺序作为键值大小比较的依据。假设该键值为非中文字符串，则和中文字符串类型的比较方式类似，仍然按照该字符串从左到右逐字比较，不过是以该字符串的 ASCII 码的编码顺序作为键值大小比较依据的。

在排序的过程中，数据的移动方式可分为“直接移动”和“逻辑移动”两种。“直接移动”是直接交换存储数据的位置，而“逻辑移动”并不会移动数据存储的位置，仅改变指向这些数据的辅助指针的值，如图 3-2 和图 3-3 所示。

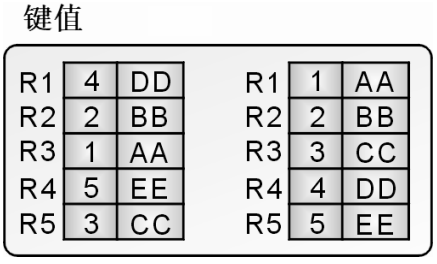

图 3-2

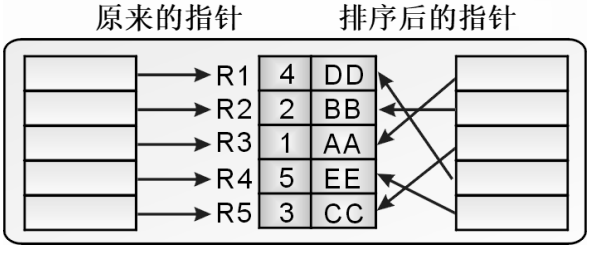

图 3-3

两者之间的优缺点在于直接移动会浪费许多时间，而逻辑移动只要改变辅助指针指向的位置就能轻易达到排序的目的。例如，在数据库中，可在报表中显示多个记录，也可以针对这些字段的特性进行分组并排序与汇总，这就属于逻辑移动，而不是直接改变数据在数据文件中的位置。数据在经过排序后会有以下好处。

（1）数据容易阅读。

（2）数据利于统计和整理。

（3）可大幅减少数据查找的时间。

排序的各种算法称得上是数据科学这门学科的精髓所在。每一种排序方法都有其适用的情况与数据类型。

3.2 冒泡排序法

冒泡排序法又称为交换排序法，是从观察水中气泡变化构思而成的，原理是从第一个元素开始，比较相邻元素的大小，若大小顺序有误，则对调后再进行下一个元素的比较，就仿佛气泡从水底逐渐升到水面上一样。如此扫描过一次之后就可确保最后一个元素位于正确的顺序，接着逐步进行第二次扫描，直到完成所有元素的排序关系为止。

下面使用数列（55,23,87,62,16）来演示从小到大的排序过程。这样大家就可以清楚地知道冒泡排序法的具体流程了。

原始数据如图 3-4 所示。

图 3-4

① 第一次扫描会先拿第一个元素 55 和第二个元素 23 进行比较，如果第二个元素小于第一个元素，则进行互换；接着拿 55 和 87 进行比较，就这样一直比较并互换，到第 4 次比较完后即可确定最大值在数组的最后面，如图 3-5 所示。

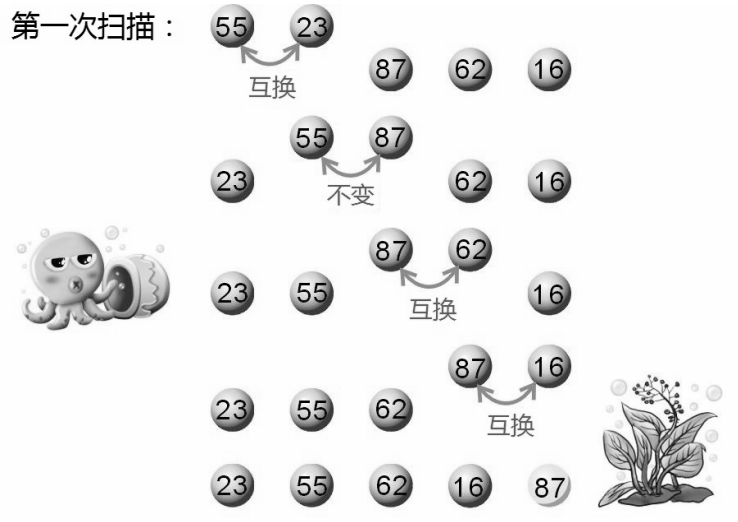

图 3-5

② 第二次扫描也是从头比较，但因为最后一个元素在第一次扫描就已确定是数组中的最大值，所以只需比较 3 次即可把剩余数组元素的最大值排到剩余数组的最后面，如图 3-6 所示。

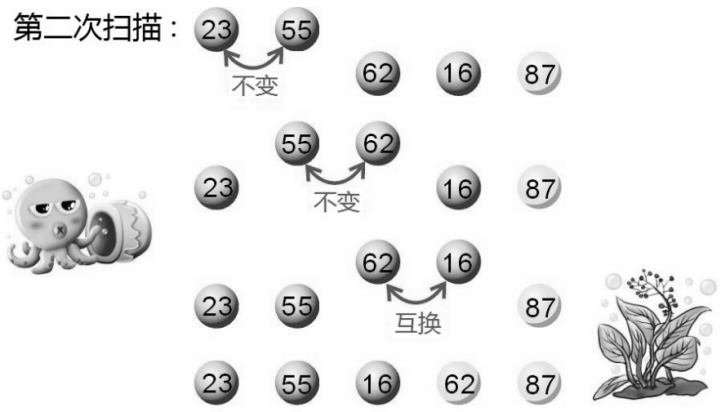

图 3-6

③ 第三次扫描只需要比较两次，如图 3-7 所示。

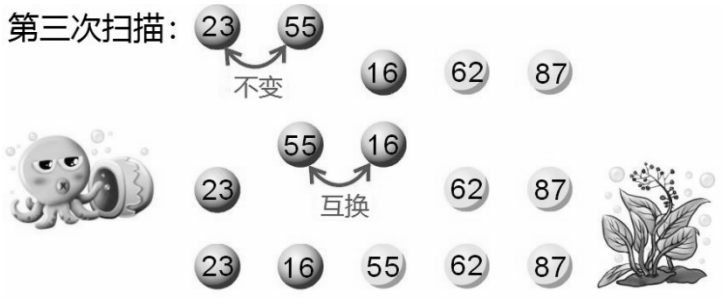

图 3-7

④ 第四次扫描完成后就完成了所有的排序，如图 3-8 所示。

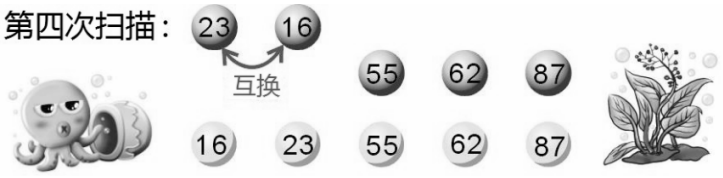

图 3-8

由此可知，5 个元素的冒泡排序法必须执行 5-1 次扫描，第一次扫描需要比较 5-1 次，第二次扫描比较 5-1-1 次，以此类推，共比较 4+3+2+1=10 次。

【范例程序：CH03_01.c】

设计一个 C 程序，使用冒泡排序法来对以下数列进行排序，并输出逐次排序的过程：

```
16,25,39,27,12,8,45,63
```

```
01    #include <stdio.h>
02    #include <stdlib.h>
03
04    int main()
05    {
```

```
06      int i,j,tmp;
07      int data[8]={16,25,39,27,12,8,45,63};      /* 原始数据 */
08      printf("冒泡排序法：\n 原始数据为：");
09      for (i=0;i<8;i++)
10         printf("%3d",data[i]);
11      printf("\n");
12
13      for (i=7;i>=0;i--)                  /* 扫描次数 */
14      {
15         for (j=0;j<i;j++)                /*比较、交换次数*/
16         {
17              if (data[j]>data[j+1])     /* 比较相邻两数，若第一个数较大则交换 */
18              {
19                 tmp=data[j];
20                 data[j]=data[j+1];
21                 data[j+1]=tmp;
22              }
23         }
24         printf("第 %d 次排序后的结果是：",8-i); /*把各次扫描后的结果打印出来*/
25         for (j=0;j<8;j++)
26              printf("%3d",data[j]);
27         printf("\n");
28      }
29      printf("最终排序的结果为：");
30      for (i=0;i<8;i++)
31         printf("%3d",data[i]);
32      printf("\n");
33
34      system("pause");
35      return 0;
36  }
```

【执行结果】参考图 3-9。

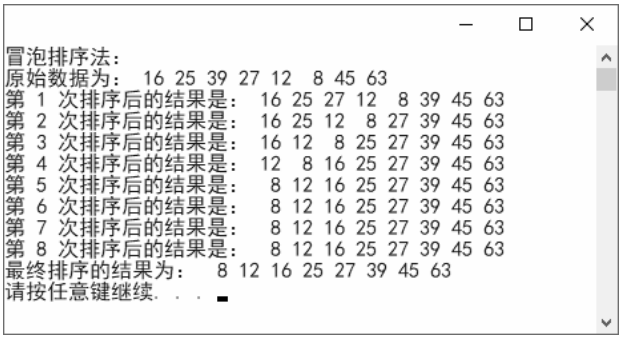

图 3-9

3.3 选择排序法

选择排序法（Selection Sort）也算是枚举法的应用，就是反复从未排序的数列中取出最小的元素，加入到另一个数列中，最后的结果即为已排序的数列。选择排序法可使用两种方式排序，即在所有的数据中，若从大到小排序，则将最大值放入第一个位置；若从小到大排序，则将最大值放入

最后一个位置。例如，一开始在所有的数据中挑选一个最小项放在第一个位置（假设是从小到大排序），再从第二项开始挑选一个最小项放在第 2 个位置，以此重复，直到完成排序为止。

下面我们仍然用数列（55,23,87,62,16）从小到大的排序过程来说明选择排序法的演算流程。原始数据如图 3-10 所示，排序过程如图 3-11 到图 3-14 所示。

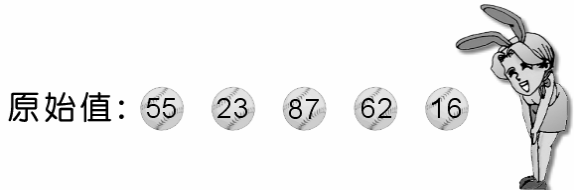

图 3-10

① 首先找到此数列中的最小值，并与数列中的第一个元素交换，如图 3-11 所示。

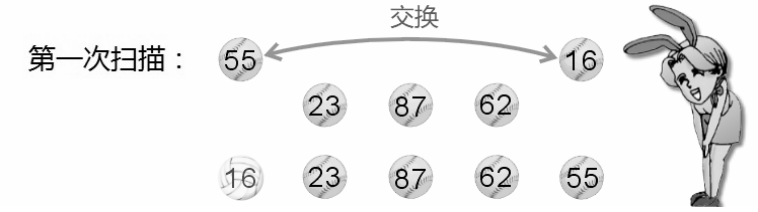

图 3-11

② 从第二个值开始找，找到此数列中（不包含第一个）的最小值，再与第二个值交换，如图 3-12 所示。

图 3-12

③ 从第三个值开始找，找到此数列中（不包含第一、二个）的最小值，再与第三个值交换，如图 3-13 所示。

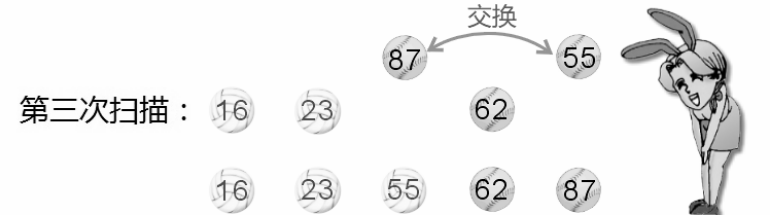

图 3-13

④ 从第四个值开始找，找到此数列中（不包含第一、二、三个）的最小值，再与第四个值交换，如图 3-14 所示。

图 3-14

【范例程序：CH03_02.c】

设计一个 C 程序，并使用选择排序法对以下数列进行排序：

```
16,25,39,27,12,8,45,63
```

```
#include <stdio.h>
#include <stdlib.h>
void select (int *);     /*声明排序法子程序*/
void showdata (int *);   /*声明打印数组子程序*/

int main()
{
   int data[8]={16,25,39,27,12,8,45,63};
   printf("原始数据为：");

   showdata(data) ;
   printf("--------------------------------------\n");
   select (data);
   printf("最终排序的结果为：");
   showdata(data) ;

   return 0;
}
void showdata (int data[])
{
   int i;
   for (i=0;i<8;i++)
      printf("%3d",data[i]);
   printf("\n");
}

void select (int data[])
{
   int i,j,tmp;
   for(i=0;i<7;i++)    /*扫描 5 次*/
   {
      for(j=i+1;j<8;j++)  /*从 i+1 比较起，比较 5 次*/
      {
         if(data[i]>data[j])  /*比较第 i 个和第 j 个元素*/
         {
            tmp=data[i];
            data[i]=data[j];
            data[j]=tmp;
         }
      }
      showdata(data);
```

```
42        }
43        printf("\n");
44    }
```

【执行结果】参考图 3-15。

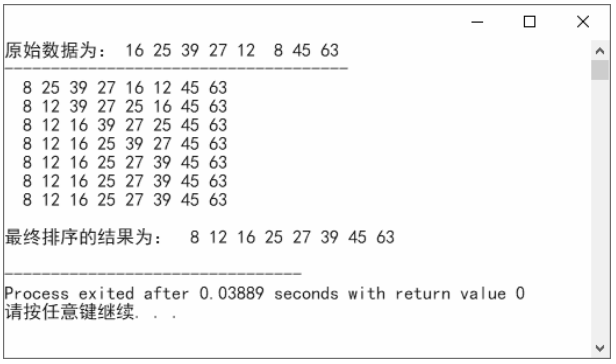

图 3-15

3.4　插入排序法

插入排序法（Insert Sort）是将数组中的元素逐一与已排序好的数据进行比较，先将前两个元素先排好，再将第三个元素插入适当的位置，也就是说这三个元素仍然是已排序好的，接着将第四个元素加入，重复此步骤，直到排序完成为止。可以看作是在一串有序的记录 R_1,R_2,…,R_i 中，插入新记录 R，使得 i+1 个记录排序妥当。

下面我们仍然用数列（55,23,87,62,16）从小到大的排序过程来说明插入排序法的演算流程。在图 3-16 中，在步骤二以 23 为基准与其他元素比较后，将其放到适当位置（55 的前面），步骤三是将 87 与其他两个元素比较，接着 62 在比较完前三个数后插到 87 的前面，以此类推，将最后一个元素比较完后就完成了排序。

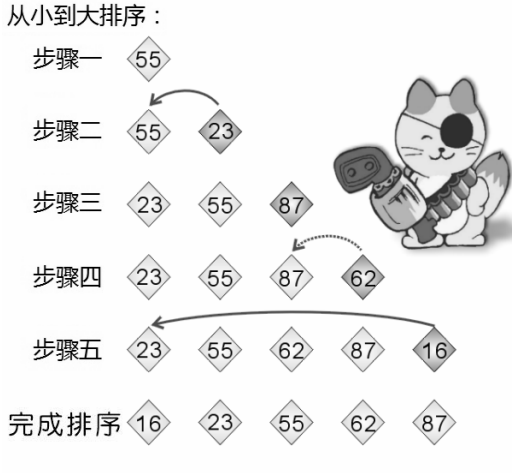

图 3-16

【范例程序：CH03_03.c】

设计一个 C 程序，输入以下的数列，并使用插入排序法对它们进行排序：

```
16,25,39,27,12,8,45,63
```

```
#include <stdio.h>
#define SIZE 8           /*定义数组大小*/
void inser (int *);      /*声明插入排序法子程序*/
void showdata (int *);  /*声明打印数组子程序*/
void inputarr (int *,int);  /*声明输入数组子程序*/

int main(void)
{
   int data[SIZE];
   inputarr(data,SIZE);     /*把数组名及数组大小传给子程序*/
   printf("您输入的原始数据是：");
   showdata (data);
   inser(data);
   printf("最终排序的结果为：");
   showdata (data);
   system("pause");
   return 0;
}

void inputarr(int data[],int size)
{
   int i;
   for (i=0;i<size;i++)      /*利用循环输入数组数据*/
   {
      printf("请输入第 %d 个元素：",i+1);
      scanf("%d",&data[i]);
   }
}
void showdata(int data[])
{
   int i;
   for (i=0;i<SIZE;i++)
      printf("%3d",data[i]);   /*打印数组数据*/
   printf("\n");
}
void inser(int data[])
{
   int i;     /*i 为扫描次数*/
   int j;     /*以 j 来定位比较的元素*/
   int tmp;   /*tmp 用来暂存数据*/
   for (i=1;i<SIZE;i++)  /*扫描循环次数为 SIZE-1*/
   {
      tmp=data[i];
      j=i-1;
      while (j>=0 && tmp<data[j])  /*如果第 2 个元素小于第 1 个元素*/
      {
         data[j+1]=data[j];        /*就把所有元素往后推一个位置*/
            j--;
      }
      data[j+1]=tmp;               /*最小的元素放到第 1 个位置*/
```

```
51            printf("第 %d 次扫描: ",i);
52            showdata(data);
53        }
54    }
```

【执行结果】参考图 3-17。

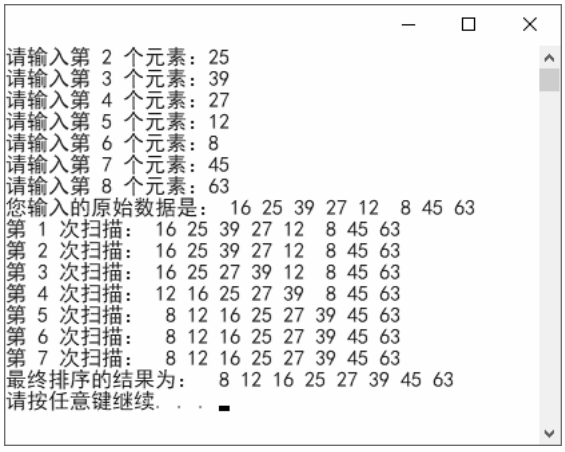

图 3-17

3.5　希尔排序法

我们知道当原始记录的键值大部分已排好序的情况下插入排序法会非常有效率，因为它不需要执行太多的数据搬移操作。“希尔排序法”是 D. L. Shell 在 1959 年 7 月所发明的一种排序法，可以减少插入排序法中数据搬移的次数，以加速排序的进行。排序的原则是将数据区分成特定间隔的几个小区块，以插入排序法排完区块内的数据后再渐渐减少间隔的距离。

下面我们用数列（63,92,27,36,45,71,58,7）从小到大的排序过程来说明希尔排序法的演算流程（参考图 3-18~图 3-23）。数据排序前的初始顺序如图 3-18 所示。

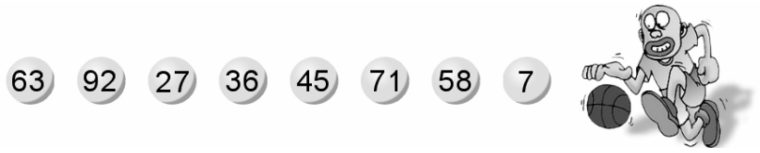

图 3-18

① 首先将所有数据分成 *Y*（8 div 2）份，即 *Y*=4，称为划分数。注意，划分数不一定是 2，质数最好，但为了方便计算，我们习惯选 2。因此，一开始的间隔设置为 8/2，如图 3-19 所示。

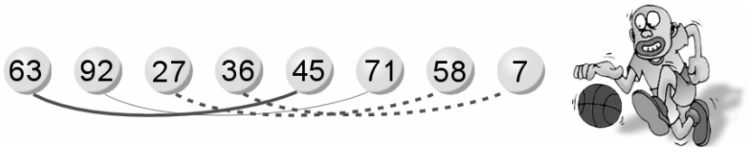

图 3-19

② 如此就可以得到 4 个区块，分别是(63，45)(92，71)(27，58)(36，7)，再分别用插入排序法排序为 (45，63)(71，92)(27，58)(7，36)。在整个队列中，数据的排列如图 3-20 所示。

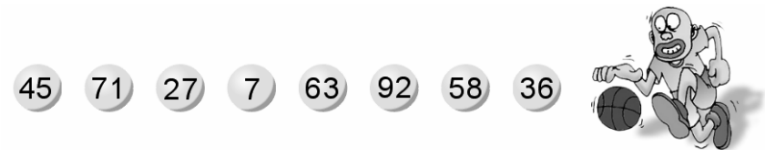

图 3-20

③ 接着缩小间隔为(8/2)/2，如图 3-21 所示。

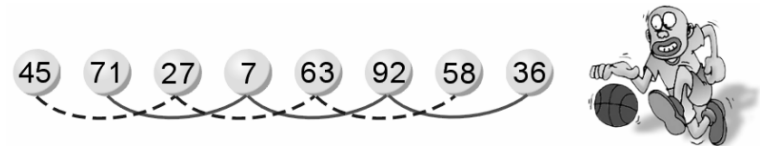

图 3-21

④ 再分别用插入排序法对(45, 27, 63, 58)(71, 7, 92, 36)进行排序，得到如图 3-22 所示的结果。

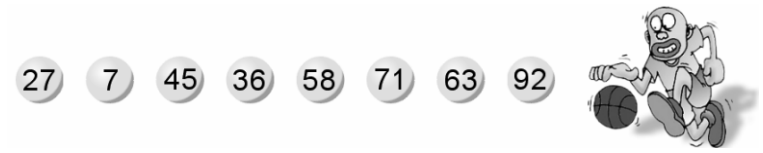

图 3-22

⑤ 再以((8/2)/2)/2 的间距进行插入排序，即对每一个元素进行排序，得到如图 3-23 所示的结果。

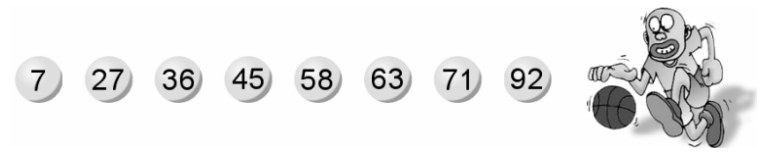

图 3-23

【范例程序：CH03_04.c】

设计一个 C 程序，并使用希尔排序法对以下数列进行排序：

```
16,25,39,27,12,8,45,63
```

```
01   #include <stdio.h>
02   #include <stdlib.h>
03   #define SIZE 8
04
05   void shell (int *,int);  /*声明排序法子程序*/
06   void showdata (int *);   /*声明打印数组子程序*/
07
08   int main(void)
09   {
10      int data[SIZE]={16,25,39,27,12,8,45,63};
```

```
11      printf("原始数据是:          ");
12      showdata (data);
13      printf("-------------------------------------------\n");
14      shell(data,SIZE);
15      printf("最终排序的结果为:    ");
16      showdata (data);
17      system("pause");
18      return 0;
19  }
20
21  void showdata(int data[])
22  {
23      int i;
24      for (i=0;i<SIZE;i++)
25          printf("%3d",data[i]);
26      printf("\n");
27  }
28
29  void shell(int data[],int size)
30  {
31      int i;       /*i 为扫描次数*/
32      int j;       /*以 j 来定位比较的元素*/
33      int k=1;     /*k 打印计数*/
34      int tmp;     /*tmp 用来暂存数据*/
35      int jmp;     /*设置间距位移量*/
36      jmp=size/2;
37      while (jmp != 0)
38      {
39          for (i=jmp ;i<size ;i++)
40          {
41              tmp=data[i];
42              j=i-jmp;
43              while(tmp<data[j] && j>=0)  /*插入排序法*/
44              {
45                  data[j+jmp] = data[j];
46                  j=j-jmp;
47              }
48              data[jmp+j]=tmp;
49          }
50          printf("第 %d 次的排序结果: ",k++);
51          showdata (data);
52          printf("-------------------------------------------\n");
53          jmp=jmp/2;    /*控制循环数*/
54      }
55  }
```

【执行结果】参考图 3-24。

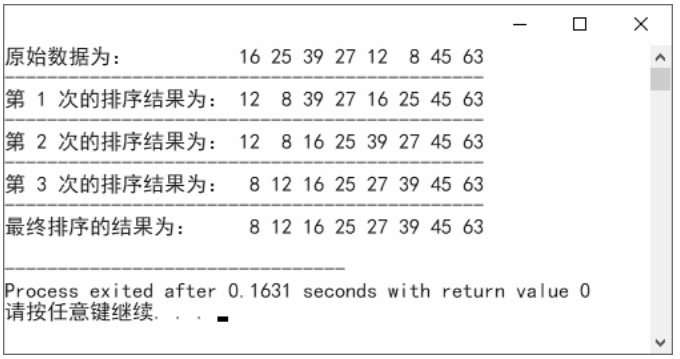

图 3-24

3.6 合并排序法

合并排序法（Merge Sort）是针对已排序好的两个或两个以上的数列（或数据文件），通过合并的方式将其组合成一个大的且已排好序的数列（或数据文件），步骤如下：

（1）将 *N* 个长度为 1 的键值成对地合并成 *N*/2 个长度为 2 的键值组。
（2）将 *N*/2 个长度为 2 的键值组成对地合并成 *N*/4 个长度为 4 的键值组。
（3）将键值组不断地合并，直到合并成一组长度为 *N* 的键值组为止。

下面我们用数列（38,16,41,72,52,98,63,25）从小到大的排序过程来说明合并排序法的基本演算流程，如图 3-25 所示。

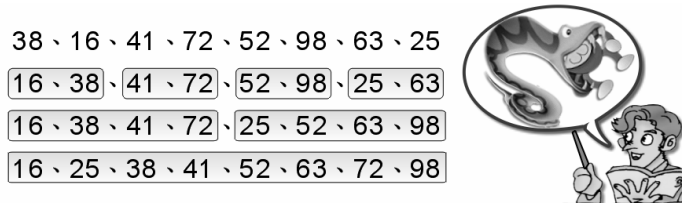

图 3-25

上面展示的是一种比较简单的合并排序，又称为 2 路（2-way）合并排序，主要是把原来的数列视作 *N* 个已排好序且长度为 1 的数列，再将这些长度为 1 的数列两两合并，结合成 *N*/2 个已排好序且长度为 2 的数列；同样的做法，再按序两两合并，合并成 *N*/4 个已排好序且长度为 4 的数列，以此类推，最后合并成一个已排好序且长度为 *N* 的数列。

现在将排序步骤整理如下：

步骤 01 将 *N* 个长度为 1 的数列合并成 *N*/2 个已排序妥当且长度为 2 的数列。
步骤 02 将 *N*/2 个长度为 2 的数列合并成 *N*/4 个已排序妥当且长度为 4 的数列。
步骤 03 将 *N*/4 个长度为 4 的数列合并成 *N*/8 个已排序妥当且长度为 8 的数列。
步骤 04 将 $N/2^{i-1}$ 个长度为 2^{i-1} 的数列合并成 $N/2^{i}$ 个已排序妥当且长度为 2^{i} 的数列。

3.7　快速排序法

快速排序（Quick Sort）是由 C. A. R. Hoare 提出来的。快速排序法又称分割交换排序法，是目前公认的最佳排序法，也是使用“分而治之”（Divide and Conquer）的方式，会先在数据中找到一个虚拟的中间值，并按此中间值将所有打算排序的数据分为两部分。其中小于中间值的数据放在左边，而大于中间值的数据放在右边，再以同样的方式分别处理左右两边的数据，直到排序完为止。操作与分割步骤如下：

假设有 n 项记录 $R_1,R_2,R_3,\ldots,R_n$，其键值为 $K_1,K_2,K_3,\ldots,K_n$。

步骤 01 先假设 K 的值为第一个键值。

步骤 02 从左向右找出键值 K_i，使得 $K_i>K$。

步骤 03 从右向左找出键值 K_j，使得 $K_j<K$。

步骤 04 若 $i<j$，则 K_i 与 K_j 互换，并回到步骤 02。

步骤 05 若 $i\geq j$，则 K 与 K_j 互换，并以 j 为基准点分割成左、右两部分，然后针对左、右两边执行步骤 01~05，直到左边键值等于右边键值为止。

下面示范使用快速排序法对数据进行排序的过程，原始数据参考图 3-26。

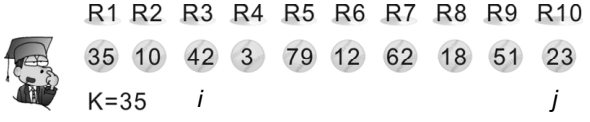

图 3-26

步骤 01 因为 $i<j$，所以交换 K_i 与 K_j，如图 3-27 所示，然后继续进行比较。

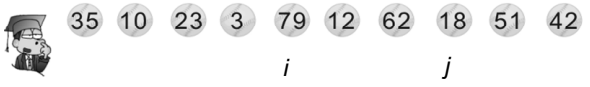

图 3-27

步骤 02 因为 $i<j$，所以交换 K_i 与 K_j，如图 3-28 所示，然后继续进行比较。

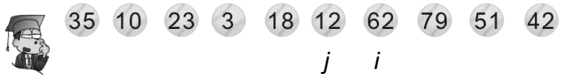

图 3-28

步骤 03 因为 $i\geq j$，所以交换 K 与 K_j，并以 j 为基准点分割成左、右两部分，如图 3-29 所示。

图 3-29

经过上述几个步骤，大家可以将小于键值 K 的数据放在左边；将大于键值 K 的数据放在右边。按照上述排序过程，对左、右两部分分别排序，过程如图 3-30 所示。

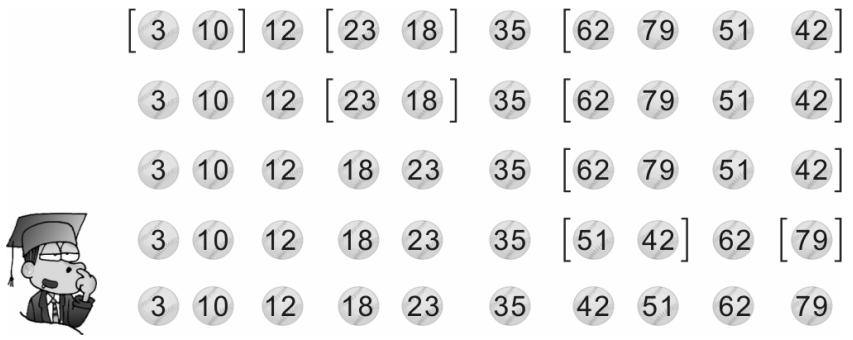

图 3-30

【范例程序：CH03_05.c】

设计一个 C 程序，使用快速排序法将输入的数字进行排序。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void inputarr(int*,int);
void showdata(int*,int);
void quick(int*,int,int,int);
int process = 0;
int main(void)
{
   int size,data[100]={0};
   srand((unsigned)time(NULL));
   printf("请输入数组大小(100 以下)：");
   scanf("%d",&size);
   printf("您输入的原始数据是：");
   inputarr (data,size);
   showdata (data,size);
   quick(data,size,0,9);
   printf("\n 最终的排序结果为：");
   showdata(data,size);
   system("pause");
   return 0;
}
void inputarr(int data[],int size)
{
   int i;
   for (i=0;i<size;i++)
      data[i]=(rand()%99)+1;
}
void showdata(int data[],int size)
{
   int i;
   for (i=0;i<size;i++)
      printf("%3d",data[i]);
   printf("\n");

}
void quick(int d[],int size,int lf,int rg)
{
   int i,j,tmp;
```

```
41      int lf_idx;
42      int rg_idx;
43      int t;
44                                  /*1:第一个键值为 d[lf]*/
45      if(lf<rg)
46      {
47          lf_idx=lf+1;
48          rg_idx=rg;
49  step2:
50          printf("[排序过程%d]=> ",process++);
51          for(t=0;t<size;t++)
52              printf("[%2d] ",d[t]);
53          printf("\n");
54          for(i=lf+1;i<=rg;i++)  /*2:从左向右找出一个键值大于 d[lf]者*/
55          {
56              if(d[i]>=d[lf])
57              {
58                  lf_idx=i;
59                  break;
60              }
61              lf_idx++;
62          }
63          for(j=rg;j>=lf+1;j--)  /*3:从右向左找出一个键值小于 d[lf]者*/
64          {
65              if(d[j]<=d[lf])
66              {
67                  rg_idx=j;
68                  break;
69              }
70              rg_idx--;
71          }
72          if(lf_idx<rg_idx)      /*4-1:若 lf_idx<rg_idx*/
73          {                          /*则 d[lf_idx]和 d[rg_idx]互换*/
74              tmp = d[lf_idx];
75              d[lf_idx] = d[rg_idx];
76              d[rg_idx] = tmp;
77              goto step2;            /*4-2:并继续执行步骤 2*/
78          }
79          if(lf_idx>=rg_idx)     /*5-1:若 lf_idx 大于等于 rg_idx*/
80          {                          /*则将 d[lf]和 d[rg_idx]互换*/
81              tmp = d[lf];
82              d[lf] = d[rg_idx];
83              d[rg_idx] = tmp;
84                                 /*5-2:并以 rg_idx 为基准点分成左右两部分*/
85                                     /*以递归方式分别为左右两部分进行排序*/
86              quick(d,size,lf,rg_idx-1);
87              quick(d,size,rg_idx+1,rg);     /*直至完成排序*/
88          }
89      }
90  }
```

【执行结果】参考图 3-31。

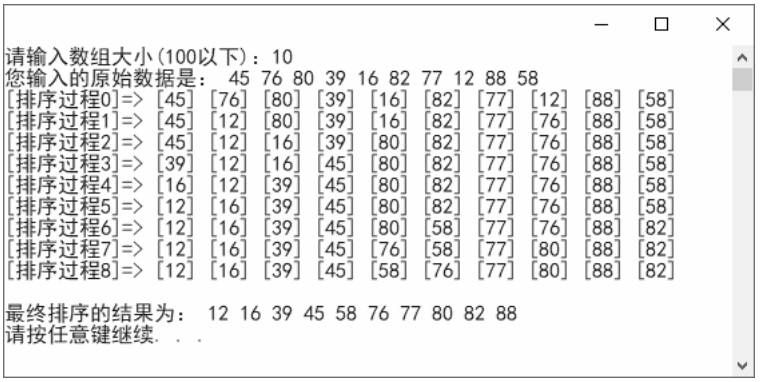

图 3-31

3.8 基数排序法

基数排序法与我们之前所讨论的排序法不太一样，并不需要进行元素之间的比较操作，而是属于一种分配模式排序方式。

基数排序法按比较的方向可分为最高位优先（Most Significant Digit First，MSD）和最低位优先（Least Significant Digit First，LSD）两种。MSD 法是从最左边的位数开始比较的，而 LSD 则是从最右边的位数开始比较的。直接看下面最低位优先（LSD）的例子，便可清楚地知道其工作原理。

在下面的范例中，我们以 LSD 将三位数的整数数据加以排序（按个位数、十位数、百位数来进行排序）。原始数据如下：

59	95	7	34	60	168	171	259	372	45	88	133

步骤 01 把每个整数按个位数字放到列表中。

个位数字	0	1	2	3	4	5	6	7	8	9
数据	60	171	372	133	34	95 45		7	168 88	59 259

合并后成为：

60	171	372	133	34	95	45	7	168	88	59	259

步骤 02 把每个整数按十位数字放到列表中。

十位数字	0	1	2	3	4	5	6	7	8	9
数据	7			133 34	45	59 259	60 168	171 372	88	95

合并后成为：

7	133	34	45	59	259	60	168	171	372	88	95

步骤 03 把每个整数按百位数字放到列表中。

百位数字	0	1	2	3	4	5	6	7	8	9
数据	7 34 45 59 60 88 95	133 168 171	259	372						

最后合并，即完成排序。

7	34	45	59	60	88	95	133	168	171	259	372

【范例程序：CH03_06.c】

设计一个 C 程序，自行输入数值数组的个数并输入这些数值，再使用基数排序法对这组输入数值进行排序。

```
/* 基数排序法，从小到大排序 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void radix (int *,int);/* 基数排序法子程序 */
void showdata (int *,int);
void inputarr (int *,int);
int main(void)
{
    int size,data[100]={0};
    printf("请输入数组大小(100 以下)：");
    scanf("%d",&size);
    printf("您输入的原始数据是：\n");
    inputarr (data,size);
    showdata (data,size);
    radix (data,size);
    system("pause");
    return 0;
}
void inputarr(int data[],int size)
{
    int i;
    srand((unsigned)time(NULL));
    for (i=0;i<size;i++)
        data[i]=(rand()%999)+1;/*设置 data 值最大为 3 位数*/
}
void showdata(int data[],int size)
{
    int i;
    for (i=0;i<size;i++)
        printf("%5d",data[i]);
    printf("\n");
}
void radix(int data[],int size)
```

```
35   {
36       int i,j,k,n,m;
37       for (n=1;n<=100;n=n*10)/*n 为基数，从个位数开始排序 */
38       {
39           int tmp[10][100]={0};/*设置暂存数组，[0~9 位数][数据个数]，所有内容均为 0 */
40           for (i=0;i<size;i++)/* 比对所有数据 */
41           {
42               m=(data[i]/n)%10;/* m 为 n 位数的值，如 36 取十位数 (36/10)%10=3 */
43               tmp[m][i]=data[i];/* 把 data[i]的值暂存于 tmp 中 */
44           }
45           k=0;
46           for (i=0;i<10;i++)
47           {
48               for(j=0;j<size;j++)
49               {
50                   if(tmp[i][j] != 0)/* 因为一开始设置 tmp ={0}，故不为 0 者即为 */
51                    {                  /* data 暂存在 tmp 中的值，把 tmp 中的值 */
52                       data[k]=tmp[i][j];  /* 放回 data[ ]中 */
53                       k++;
54                    }
55                 }
56           }
57           printf("经过%3d 位数排序后：",n);
58           showdata(data,size);
59       }
60   }
```

【执行结果】参考图 3-32。

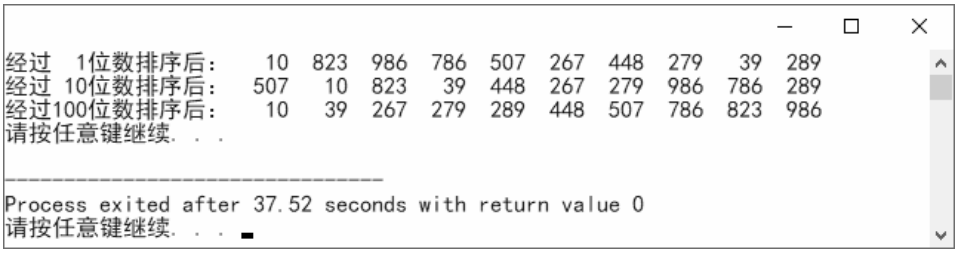

图 3-32

课后习题

1. 排序的数据是以数组数据结构来存储的。在下列排序法中，哪一个的数据搬移量最大？
   （A）冒泡排序法　　（B）选择排序法　　（C）插入排序法
2. 举例说明合并排序法是否为稳定排序。
3. 待排序的键值为 26、5、37、1、61，试使用选择排序法列出每个回合排序的结果。
4. 在排序过程中，数据移动可分为哪两种方式？试说明两者之间的优劣。
5. 简述基数排序法的主要特点。
6. 下列叙述正确与否？试说明原因。

（1）无论输入数据为何，插入排序的元素比较总次数都会比冒泡排序的元素比较总次数少。

（2）若输入数据已排序完成，再利用堆积排序，则只需 $O(n)$时间即可完成排序。其中，n 为元素个数。