中国地质大学(武汉)实验教学系列教材

数据结构及算法分析实践指导

SHUJU JIEGOU JI SUANFA FENXI SHIJIAN ZHIDAO

吴让仲
王　瑾　◎编著
张晓锋

中国地质大学出版社
ZHONGGUO DIZHI DAXUE CHUBANSHE

数据结构及算法分析实践指导

吴让仲　王　瑾　张晓锋　**编著**

中国地质大学出版社
ZHONGGUO DIZHI DAXUE CHUBANSHE

图书在版编目(CIP)数据

数据结构及算法分析实践指导/吴让仲，王瑾，张晓锋编著．—武汉：中国地质大学出版社，2014.6

ISBN 978-7-5625-3340-5

Ⅰ．数…

Ⅱ．①吴…②王…③张…

Ⅲ．①数据结构-高等学校-教材②算法分析-高等学校-教材

Ⅳ．TP311.12

中国版本图书馆 CIP 数据核字(2014)第 050872 号

数据结构及算法分析实践指导　　**吴让仲　王　瑾　张晓锋　编著**

责任编辑：王凤林　　责任校对：代　莹

出版发行：中国地质大学出版社(武汉市洪山区鲁磨路 388 号)　　邮编：430074

电　话：(027)67883511　　传　真：(027)67883580　　E-mail：cbb@cug.edu.cn

经　销：全国新华书店　　Http://www.cugp.cug.edu.cn

开本：787 毫米×1 092 毫米　1/16　　字数：435 千字　印张：17

版次：2014 年 6 月第 1 版　　印次：2014 年 6 月第 1 次印刷

印刷：武汉市珞南印务有限责任公司　　印数：1—1 000 册

ISBN 978-7-5625-3340-5　　定价：32.00 元

如有印装质量问题请与印刷厂联系调换

前 言

“数据结构及算法分析实践指导”是一门计算机软件和算法理论相结合的课程，对于培养学生将实际问题转化为用计算机描述语言实现问题的能力具有重要的意义。由于前期学生学习了C语言，故该实验教材用C语言编写，教材使用的全部程序代码在Microsoft Visual Studio C++6.0(SP6)平台中调试通过。对于C语言和数据结构的关系，笔者认为，如果将C语言比喻为汉语，数据结构则为用汉语写的诗歌。

教材中共讲述了多种典型的数据结构，如线性表、堆栈和队列、数组、字符串、树和图等，在这些数据结构的学习中穿插讲述它们的建立、插入、删除、查找和排序算法，分析各算法的时间复杂度。在各章讲解各种数据结构及其应用范围和抽象数据类型。在解决实际问题时，先分析实际问题的数据，采用一种合适的逻辑结构来描述，选择最佳的数据结构来物理实现，最后编写基于数据结构的各种操作算法的代码。

在具体上机的过程中，严格将工程模块化设计，至少包括模块的头文件、实现文件和应用文件(或测试文件)，将模块的数据类型定义和操作算法函数的声明部分放在头文件中，将函数的定义部分放在实现文件中，将模块的测试和应用代码放在应用文件中，一般应用文件中包含有C语言的主函数main。对于工程中的实现文件，因其是工程的核心文件，在实际应用中一般将模块的实现文件转换为动态链接库(DLL)，或其他的二进制形式的文件，不提供实现文件的源代码，起到保护知识产权的作用。

第一章概论，简述如何利用数据结构知识将实际问题转化为用计算机解决的问题，数据结构上机平台Microsoft Visual Studio C++6.0(SP6)，尤其着重介绍调试工具Debug的使用方法，为后面章节中调试程序打好基础。介绍算法的定义和算法时间复杂度的表示方法以及分析算法复杂度的技巧，传授获取算法实际运行时间的经验。

第二章线性表，介绍第一种线性结构的数据结构，每个数据最多只有一个前驱和最多只有一个后继，分别用数组和指针实现顺序表和链表，讲述如何在Debug中观察变量的地址和变量中存储的值，最后讲述了一个经典的问题即约瑟夫环(Josephus Problem)的实现过程。

第三章堆栈和队列，讲述了另两种线性表堆栈和队列，堆栈是一种先进后出的线性表(FILO)，队列是一种先进先出的线性表(FIFO)，讲述这两种线性表在日常生活中的广泛应用，以及如何用数组和链表实现堆栈与队列。在本章的最后，演示了用堆栈实现编译系统中判断表达式中符号时匹配的算法，并通过上机实现。

第四章字符串，回顾了C语言中关于字符串操作的库函数，讲述字符串的匹配算法，字符串的匹配算法有传统的朴素匹配算法和快速匹配的KMP算法。本章重点论述KMP算法的思想，Next数组的求取，为什么KMP算法源串指针不需要回溯等问题。在本章的实验部分介绍了KMP算法的模块化设计方法和调试技巧。

第五章数组，学习最后一种线性数据结构，数组所有元素的数据类型是一样的，它们在内存中的地址是连续分配的，占有连续的内存空间。在本章中讲解了同学们最头痛的问题，数组

与指针的关系，用多个具体的实例演示内存空间的数据如何用数组形式和指针形式访问，通过学习加深同学们对数组和指针的理解。在本章中还简述了一些特殊的数组和稀疏矩阵，在实验部分使用三元组实现了稀疏矩阵的转置算法。

第六章树，讲解第一种非线性数据结构，树中数据元素之间的关系不再是一对一的关系，而是一对多的关系，体会树的递归定义。介绍了二叉树、满二叉树、完全树、搜索树、平衡树以及各种树的特点、树遍历算法。传授树的表示和实现的方法，树的查找、插入算法思想。本章最后讲述堆的概念，它使用数组来实现的一种树结构，其他的树结构一般使用指针来实现。在实验部分，演示了堆的建立和删除过程，并在调试工具中观察中间结果。

第七章图，简述图的基本概念和使用范围、图的 ADT 以及图的邻接矩阵表示法、图的遍历方法和最小生成树算法。在本章的实验部分介绍最短路径算法(Dijkstra)，如何从图的邻接矩阵中求取单元多目标的最短路径，对于具体实际图给出该算法的应用。在本章实验部分，给出了 Dijkstra 算法的调试方法和实验数据。

第八章查找和排序，介绍基于各种数据结构的各种查找和排序算法，分析它们的时间复杂度。论述查找和排序算法是相融合的，互相包含互相配合，排序的目的是为了查找，有的查找算法中也包括了排序。简述了哈希算法、哈希表和哈希函数的设计以及处理冲突的方法。在实验部分，演示了一个系统软件的建立过程，提供菜单功能，将系统中的各种排序和查找算法连接起来，并介绍了用另一种计算算法实践运行时间的方法。

第九章综合实验，在本章中综合运用前面所讲的各种数据结构模块功能，开发学生信息管理系统，将链表、查找、排序、图和堆栈等功能模块融合在一起。按照软件工程的工作流程，严格实现了模块化程序设计，使同学们体会并各自开发一个独立模块，最后合并到系统，提高(或培养)同学们的团队合作精神。

附录提供了一些数据结构设计的题目，编写这些题目有助于同学们培养解决实际问题的能力。同时提供了一份数据结构报告的样例，为同学们写报告作参考。

在多年数据结构及算法分析教学过程中，上机实习一直是同学们的弱点。为了帮助学生能较快地把理论知识应用到实践中去，加深对理论知识的理解，同时提高程序设计的能力，编写了这本上机指导书。笔者认为数据结构上机学习不是靠一日之功，而是要练内功，算法设计要像喝茶，慢慢品尝，调试技巧更需要长年累月积累经验。

通过实践指导学习，培养同学们针对具体应用问题来选择、设计和实现相应的数据结构的能力，锻炼同学们“从问题到程序”的应用软件设计技能，培养团队合作精神，为将来进行软件开发和研究打下扎实的基础。

经过数年的准备与数月撰写，书稿终于完稿。感谢在本书撰写中给予我帮助的妻子，在这半年中她提供了后勤保障，数个节假日陪我写书稿。感谢中国地质大学(武汉)机械与电子信息学院的吴奕凡、吴泽光、康坊、裴登科、罗敏、黄文君、宋佳珍和白雨晓等同学，在本书的排版过程中，他们从学生的角度，提出了宝贵的意见。

本书可以作为高校本科生数据结构及算法分析课程的实践教程，也可作为从事底层软件开发人员的参考书。由于水平有限，本书中仍难免有些错误和缺点，恳切请求广大读者尤其是从事算法设计与软件开发的相关人员提出批评和建议，鄙人不胜感激。

本书中所有的代码在 Microsoft Visual Studio C++6.0(SP6)中调试通过，可以在中国地质大学出版社网站下载。

此为试读，需要完整PDF请访问：www.ertongbook.com

目　录

第一章 概 论

数据结构及算法分析是一门以算法为核心，利用计算机解决具体的实际问题的课程。它是一门综合性较强的专业基础课程，该课程的体系结构由美国 Knuth 教授开创，是介于数学、计算机硬件和计算机软件三者之间的一门交叉学科。目前，国内外高校大多数理工专业开设了该课程，该课程有助于培养学生分析问题的技能、实践编程能力和优化算法的技巧。

第一节 数据的表示

数据结构及算法分析这门课，我们从名称中可以看出它主要包含两个方面的内容：一是数据结构；二是算法分析。

数据结构是研究实际问题中数据的逻辑结构、物理结构（ 也称为存储结构 ） 和基于数据的运算。例如，我们处理一个数据序列：111，20，90，89，10，20，100，120，169，200，对于这个数据序列提出一系列需要处理的问题。

(1)找出该数据序列中的最小值和最大值。

(2)求出序列的平均值、方差。

(3)按升序或降序顺序输出该数据序列。

(4)在数据序列中插入或删除某个数据。

当然，上述问题数目的多少可以按照实际的需求进行增减。一般来说，对应上述每个问题，我们采用一个功能函数进行处理。编写函数要尽可能符合泛性，功能函数能完成一些通用性问题。如果用户提出的问题不能用我们提供的某个函数来解决，也可借助几个函数联合解决。也就是说，要求编写的函数具有通用性。

目前计算机普及率很高，在人们生活和生产的各个环节中，通常采用计算机来处理实际中的具体问题，利用计算机解决具体实际问题的过程一般包括以下几个步骤。

一、Step1 分析问题

在分析阶段，需要深入生产实践环节，了解问题的需求，得出问题的已知条件和需要解出的结果。

二、Step2 数据表示

在数据表示阶段，我们要利用计算机解决具体问题，首先我们要弄清实际问题数据的逻辑结构，譬如是线性的还是非线性的，常见的数据逻辑结构见表 1 - 1；其次需要将数据存储在计算机内存或其他介质中，即数据对应的物理结构或存储结构。常见的存储结构有两种：一是用顺序结构，将数据储存在内存地址空间连续分布的数组中；二是用链式结构，用链式组织数据存储，数据在内存中不要求顺序存储，即不要求连续的存储单元，数据可以存储在内存地址空

间不连续的零散内存单元，见表 1－2。

表 1－1 常见数据的逻辑结构

逻辑结构	常见类型
线性结构	顺序表、线性表、堆栈和队列等
非线性结构	树、图等

表 1－2 数据的存储结构

存储结构	特点
顺序结构	用数组存储数据，需要连续内存单元
链式结构	用链式结构存储，不需要连续内存单元，使用指针来实现

三、Step3 算法设计

数据存储在计算机中后，需要设计相应的处理实际问题的过程，即算法。譬如上例中提出的 4 个问题，编写对应的 4 个算法。对于同一实际问题，可能有不同的解决算法，如何衡量算法的好坏，将在后面的章节中讲解。

四、Step4 软件测试

完成数据的表示和实现算法后，需要测试算法正确性，一般我们使用计算机高级语言(例如 C 语言)来实现算法，通常一个算法对应一个函数。算法函数设计完成后，再设计测试样例，观察算法运行结果，给出判断(算法函数正确与否)。

五、Step5 软件发布

软件经测试确定正确后，需要打包发布，提供安装程序和使用手册。具体打包发布的方法将在后面章节中介绍。

对于数据表示问题，我们还是回到上面的例子。要表示上述数据序列的物理存储结构，可以使用顺序表或链表，如图 1－1 所示。

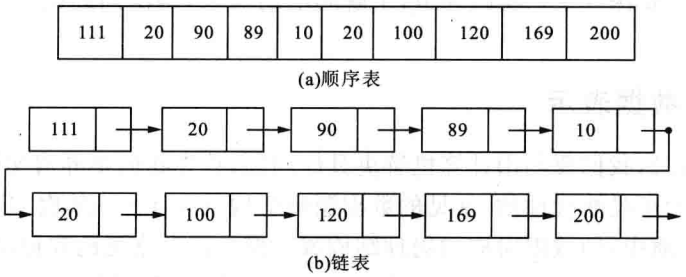

图 1－1 数据物理存储结构图

第二节 算法的衡量

解决一个具体的问题，不同的人有不同的解决方法，那到底谁的算法要好些呢？一般的想法是，看看谁的算法运行的时间较短，运行时间越短的算法我们认为要好些，这种比较算法好与坏的方法称为事后统计法。但是，程序的运行时间还依赖于计算机的硬件和编译系统，算法在硬件配置高的计算机上运行比在配置低的计算机上要快些，所以只从运行时间的快与慢来判断算法的好与坏，这种方法有欠妥当，需要用另外的更加合理的方法来衡量。下面我们要讲另外一种衡量算法好与坏的方法，事先估计法，首先介绍与有关算法相关的一些知识。

一、算法的定义

算法是人们解决问题的步骤，是语句的有限集合。一个算法应符合如下的几项原则。

(1)输入。算法可以有输入，也可以没有输入。

(2)输出。算法必须有一个输出，不能没有输出，也就是算法必须有结果。

(3)确定性。组成算法的每个语句的功能必须是确定的，不能含糊不清。这是因为算法可以用人类的自然语言来描述，且可能存在二义性，如果算法用自然语言定义，每个语句的意思必须是唯一的。

(4)有限性。算法的所有可能情况必须在有限的步骤内完成，不能出现算法需要无穷步的情况。

(5)可行性。算法可以被人们用纸和笔进行静态跟踪，并且算法在计算机当前的配置上可以实现，也就是算法需要的资源目前的计算机系统可以满足，不能出现运行时间需要数月、数年或内存需要数 G、数 T 字节的情况。

算法 1：这是一个古老的算术问题，用 100 元钱买 100 只鸡，其中公鸡 3 元一只，母鸡 2 元一只，小鸡 5 角一只，请给出各种解决的方案。一般简单的算法设计，设公鸡、母鸡和小鸡的数目分别为 i, j 和 k 只，我们可以得到两个方程：

$$i+j+k=100 \tag{1-1}$$

$$3i+2j+0.5k=100 \tag{1-2}$$

用 C 语言来实现，采用三层 for 循环，循环变量为公鸡、母鸡和小鸡的数目，显然它们都小于等于 100，只要满足上述的两个方程，就得到我们所需的解决方案。

```
for(i=0;i<=100;i++)
  for(j=0;j<=100;j++)
    for(k=0;k<=100;k++)
      if(i+j+k==100 &&3*i+2*j+0.5*k==100)
        printf("i=%d,j=%d,k=%d",i,j,k);
```

核心语句是最内层循环的语句：

```
if(i+j+k==100 &&3*i+2*j+0.5*k==100)
  printf("i=%d,j=%d,k=%d",i,j,k);
```

显然该语句执行的次数为 100*100*100=100 万次。

算法 2：算法 1 的有些步骤是多余的，可以优化，譬如公鸡数不能多于 33 只(100/3=33)，那

公鸡最多有多少只呢？显然，不买母鸡，只买公鸡和小鸡可以得出最多买公鸡的数目，可以求得公鸡最多 20 只，同时小鸡 80 只，它们需要的钱数共是 20*3+80*0.5=100，符合要求。外层循环次数减小到 20，比原来的 100 小了 5 倍。同样购买母鸡最多的数目也可用上面介绍的方法求得，值为 33.3，向上取整数为 34，第二层循环次数调整为 34 - i。

还有，将方程式(1 - 1)和式(1 - 2)化简为一个方程，将方程式(1 - 1)代入方程式(1 - 2)，消去循环变量 k，得方程(1 - 3)，三层循环简化为二层循环。

$$3*i+2*j+(100-i-j)*0.5=100 \tag{1-3}$$

```
for(i=0;i<=20;i++)
  for(j=0;j<=34 - i;j++)
    if(3*i+2*j+(100 - i - j)*0.5==100)
      printf("i=%d,j=%d,k=%d",i,j, 100 - i - j);
```

核心语句是最内层循环的语句：

```
if(3*i+2*j+(100 - i - j)*0.5==100)
  printf("i=%d,j=%d,k=%d",i,j, 100 - i - j);
```

该语句执行的次数为 525 次，算法 2 要比算法 1 快约 1500 次。

从上面两个算法可以看出，两种方案都可以正确给出结果，但是算法 2 要比算法 1 好，正如有的教材称算法是程序的灵魂，程序的效率由算法的优劣决定。

算法可以用某种计算机编程语言描述，如 C\C++、汇编和 Java 语言等，也可以用人类自然语言、流程图或伪代码描述。不管算法用什么方式描述，只要满足算法的几项准则即可。下面列举一个用伪代码描述的算法。

算法 3：选择排序。将 n(n>1)个整数序列，整数序列存储在数组 list[n]中，将它们按照升序方式排序。

```
for (i=0;i<n;i++)
{
    找出 list[i]到 list[n - 1]中的最小数 list[min];
    交换 list[i] 和 list[min];
}
```

二、算法复杂度的渐近表示法

前面我们谈过使用事后统计法的运行时间的长短来衡量算法的好坏，该种方法是不可取的，因为算法的运行时间还与计算机的硬件和编译系统有关。一般，衡量算法的运行时间采用事先估计法，估计一下算法的语句执行多少次。事先估计法不再依赖于计算机的硬件配置和软件环境，用于衡量算法的复杂度更具有科学性。

事先估计法，往往需要以下几点假设来支撑。

(1)组成算法的语句是顺序执行的。

(2)算法的每一条语句是简单的，也就是说每条语句执行的时间是一样的，都为一个执行时间单位。也就是说没有考虑分析问题各步骤的复杂性，譬如将加减法和乘除法语句执行的时间都认为是一个单位时间。而在计算机系统中，加减法实际执行时间要比乘除法实际执行的时间要少，但是为了考虑问题的方便，我们将它们执行的时间都归一化为一个时间单位。

此为试读，需要完整PDF请访问：www.ertongbook.com

(3)为了讨论问题的简单性,我们认为算法涉及的数值类型都是整型,并且计算机的内存足够运行我们设计的算法,即认为解决问题的数据可以一次性装入计算机的内存中,不需要辅助存储空间。

算法时间复杂度表示上常常用大写字母 T 来表示,记为 T(N),式中 N 为数据规模。

当然算法复杂度的衡量还可从空间上估计,估计算法所需要的内存空间,记为 S(N),N 为数据规模,在该教材中我们只讨论算法的时间复杂度,没有涉及到算法的空间复杂度。

下面我们看看两个算法。

算法 4:使用重复语句计算数据序列的和,数据序列存储在数组 list 中,计算前面 n 个数的和,设计算法如下:

```
1:float  sum (float  list[ ],  int  n)
2:{  /*add a list of numbers*/
3:   float  tempsum=0;
4:   int  i;
5:   for (i=0;i<n;i++)
6:        tempsum+=list [ i ];
7:   return  tempsum;
8:}
```

下面分析上述算法中各行语句执行的次数:第 3 行语句为变量赋值语句,执行 1 次;第 4 行语句属于变量定义语句,在源代码编译过程中完成,执行的时候不计执行时间。第 5 行语句是 for 循环语句,包含 3 条分句(i=0;i<n;i++),执行 n+1 次,当 n=0,1,2,…,n 时执行;第 6 行语句为赋值语句,是 for 循环的循环体语句,执行 n 次;第 7 行语句为函数返回值语句,执行 1 次。整个算法中的语句一共执行 1+(n+1)+n+1=2n+3 次,算法时间复杂度 T(N)=2n+3。

上面算法语句执行次数也可以用 debug 工具来跟踪统计,如图 1-2 所示,在语句 float tempsum=0 前设置一个断点,用单步跟踪工具进行调试,可以发现语句 float tempsum =0 执行一次,语句 int i 没有执行,for(i=0;i<n;i++)语句执行 6 次,temp+=list[i]执行 5 次,re -

```
#include <stdio.h>
float  sum ( float  list[ ],  int  n );
void main()
{
    float Array[5]={1.0,2.0,3.0,4.0,5.0},fSum=0.0;
    fSum=sum(Array,5);
    printf("The Sum is %f.\n",fSum);
}
float  sum ( float  list[ ],  int  n )
{  /* add a list of numbers */
   float  tempsum = 0;
   int  i ;
   for ( i = 0; i < n; i++ )
       tempsum  += list [ i ] ;
   return  tempsum;
}
```

图 1-2 debug 调试

turn tempsum 执行 1 次。在跟踪调试的过程中，黄色箭头指向算法中某行语句一次，该语句就需要执行一次。

算法 5：对于上面算法 4 我们采用递归来实现，代码如下：

```
1:float  rsum (float  list[ ],  int  n)
2:{  /*add a list of numbers*/
3:   if (n)
4:        return rsum(list, n - 1)+list[n - 1];
5:   return   0;
6:}
```

第 3 行语句执行 n+1 次，其中 n 次(n=n，n - 1，…，2，1)条件为真，1 次(n=0)条件为假；第 4 行语句为函数返回值语句，在条件为真时执行，共执行 n 次；第 5 行语句为函数返回值语句，执行 1 次。整个算法中语句一共执行(n+1)+n+1=2n+2 次，所以算法时间复杂度T(N)=2n+2。

比较上面两个算法，表面上我们会认为算法 5 的时间复杂度要比算法 4 的要小，因为算法 5 的时间复杂度为 2n+2，要小于算法 4 的时间复杂度 2n+3，譬如当 n=10 时，2n+2=22 小于 2n+3=23，效率要高。但是当数据的规模 n 趋向于无穷大时，两者无限接近，认为两者的时间复杂度一样。

这样我们提出算法复杂度的渐近表示法，随数据规模 N 的增长，考虑算法复杂度是关于数据规模 N 的一个什么关系式。

下面看看一个比较直观的例子，假设一个算法的时间复杂度为 $T_{p1}(N)=c_1N^2+c_2N$，另一个算法的时间复杂度为 $T_{p2}(N)=c_3N$，式中 c_1，c_2 和 c_3 为常数。到底哪个算法快些呢？

不妨令 $c_1=1$，$c_2=0$，$c_3=10000$，当 $N\leqslant 10000$ 时，$T_{p1}(N)\leqslant T_{p2}(N)$；当 $N>10000$ 时，$T_{p1}(N)>T_{p2}(N)$。显然，无论 c_1，c_2 和 c_3 取什么常数，我们总能找到一个值 n_0，当 $N>n_0$ 时，确保 $T_{p1}(N)>T_{p2}(N)$。因为 $T_{p1}(N)$是关于 N 的二次方关系式，$T_{p2}(N)$是关于 N 的一次方关系式，二次方的增长趋势肯定要比一次方的快。

下面给出算法时间复杂度的 4 种形式定义：

$T(N)=O(f(N))$if there are positive constants c and n_0 such that $T(N)\leqslant c\cdot f(N)$for all $N\geqslant n_0$

$T(N)=\Omega(g(N))$if there are positive constants c and n_0 such that $T(N)\geqslant c\cdot g(N)$for all $N\geqslant n_0$. (lower bound)

$T(N)=\Theta(h(N))$if and only if $T(N)=O(h(N))$and $T(N)=\Omega(h(N))$

$T(N)=o(p(N))$if $T(N)=O(p(N))$and $T(N)\neq\Theta(p(N))$

算法时间复杂度定义有 4 种，分别记为 O，Ω，Θ 和 o，我们重点讨论 O，其他的 3 个可以自己研究。O 读作“大 O”，定义中文意思是存在正的常数 c 和 n_0，确保当 $N\geqslant n_0$ 时，$T(N)<cf(N)$。例如，前面算法 4，时间复杂度 T(N)=2n+3，存在正的常数 c=3 和 $n_0=3$，当 $N\geqslant n_0$ 时，$T(N)<3N=3f(N)$，即时间复杂度为 O(N)。依次类推，可以得 $2n+3=O(N)=O(N^{k\geqslant 1})=O(2^N)=\cdots$。但在实际研究中，我们取最靠近 2n+3 的一个，即 T(N)的上限——O(N)，在实际应用中，算法 4 的时间复杂度记为 T(N)=O(N)，而不是其他的几种形式。

时间复杂度渐近表示法的三项法则。

(1)如果两个时间复杂度分别为 T1 和 T2，则它们的和与积计算为：

if T1 (N)=O(f (N))and T2 (N)=O(g(N)), then

① $T1(N)+T2(N)=\max(O(f(N)), O(g(N)))$

② $T1(N)*T2(N)=O(f(N)*g(N))$

式①常用于分支结构，时间复杂度取复杂度大的一个分支；式②应用于循环嵌套，嵌套循环的时间复杂度取决于各层时间复杂度的积。

(2)如果时间复杂度是关于 N 的 k 阶多项式，则 $T(N)=\Theta(N^k)$，忽略低次项和常数项。

(3)如果算法的时间复杂度 $T(N)=O(\log^k N)$，无论 k 为何值，算法的时间复杂度增长都慢。如果我们将算法复杂度设计为关于数据规模的对数级(表 1-3)，该算法通常是一个很好的算法。

表 1-3 关于时间复杂度与数据规模 N 的各种级别的具体值

Input size n							
Time	Name	1	2	4	8	16	32
1	constant	1	1	1	1	1	1
$\log_2 n$	logarithmic	0	1	2	3	4	5
n	linear	1	2	4	8	16	32
$n\log_2 n$	log linear	0	2	8	24	64	160
n^2	quadratic	1	4	16	64	256	1024
n^3	cubic	1	8	64	512	4096	32768
2^n	exponential	2	4	16	256	65536	4294967296
n !	factorial	1	2	24	40326	2092278988000	26313×10^{33}

其中:constant 为常数级，logarithmic 为对数级，linear 为线性级，log linear 为对数线性级，quadratic 为平方级，cubic 为立方级，exponential 为指数级，factorial 为阶乘级。

表 1-3 列出了时间复杂度关于数据规模 N(N 分别取 1,2,4,8,16 和 32)各种级别的具体值，可以看出当 N 值不大时，立方级别以下时间复杂度还是可行的，最好是常量和对数级别，最不好的是阶乘级别。

绘制时间复杂度关于数据规模 N 的各种级别曲线，可以很直观地看出它们的增长趋势，如图 1-3 所示。从图中可以看出对数级、线性级和线性对数级增长速度慢，指数级增长快，设计算法时尽可能设计在平方级以下。

关于时间复杂度计算的一般规则如下。

(1)如果算法由单层循环构成，时间复杂度等于循环次数乘以循环体语句个数。

(2)如果算法由多层循环组成，时间复杂度等于各层循环次数与最内层循环体语句条数的乘积。

(3)顺序语句时间复杂度等于语句条数和。

(4)分支语句时间复杂度等于时间复杂度最大的一个分支的时间复杂度。

下面看看一些关于时间复杂度的例子。

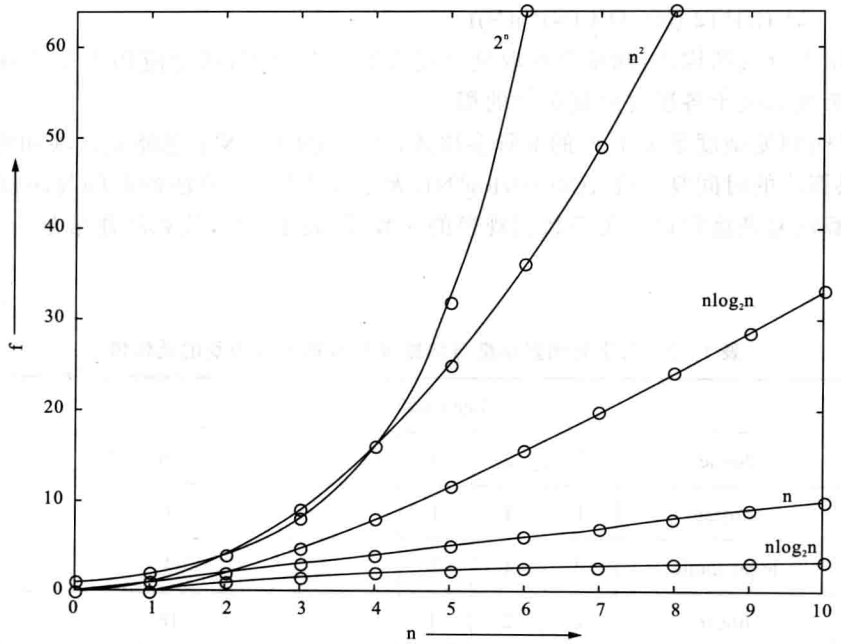

图 1-3　时间复杂度曲线图

1. 常量级

```
s=0;
```

语句频度为 1，时间复杂度为 O(1)。

```
for(j=1;j<=10000;++j)
{
  ++x;  s+=x;
}
```

核心语句 s+=x 执行频度为 10000，是一个常数，所以算法时间复杂度为 O(1)。

2. 对数级

```
s=0;
for(j=1;j<=n;j*=2)
  s++;
```

核心语句 s++执行频度为 $\log_2 n$，所以算法时间复杂度为 $O(\log_2 n)$。

3. 线性级

```
s=0;
for(j=1;j<=n;++j)
  s++;
```

核心语句 s++执行频度为 n，所以算法时间复杂度为 O(n)。

4. 对数级

```
s=0;
for(j=1;j<=n;j*=2)
  for(k=1;k<=n;++k)
    s++;
```

核心语句 s++执行频度为 $O(n\log_2 n)$，所以算法时间复杂度为 $O(n\log_2 n)$。

5. 平方级

```
s=0;
for(j=1;j<=n;++j)
  for(k=1;k<=n;++k)
    s++;
```

核心语句 s++执行频度为 n^2，所以时间复杂度为 $O(n^2)$。

6. 立方级

例：计算两个 n 阶矩阵 a 和 b 乘法，存放在矩阵 c 中：

```
for(i=0;i<n;i++)                          // 执行(n+1)次
  for(j=0;j<n;j++)                        // 执行 n(n+1)次
  { c[i][j]=0;                            // 执行 n² 次
    for(k=0;k<n;j++)                      // 执行 n²(n+1)次
      c[i][j]=c[i][j]+a[i][k]*b[k][j];    // 执行 n³ 次
  }
```

说明：各语句行后注释部分中数字是该语句执行的次数，算法中语句一共执行 $(n+1)+n(n+1)+n^2+n^2(n+1)+n^3=2n^3+2n^2+2n+1$ 次，是关于 n 的 3 阶多项式，算法时间复杂度为 $O(n^3)$。

第三节 上机环境

该教材采用英文版 Visual Studio C++6.0+SP6 作为软件开发平台，为了方便编写和调试程序的方便，建议安装 VC 助手和 MSDN。

启动 VC，VC 主界面如图 1-4 所示，主界面中包括标题栏、工具栏、工作空间、编辑区、输出区和状态栏，关于 VC 界面的介绍，同学们可以参考其他书籍，在这里只介绍与数据结构上机关系最密切的一部分。

下面介绍数据结构在 VC 环境中的上机步骤。

一、上机步骤

(1)打开 VC 主界面，点击 File(文件)菜单，选择 New(新建)菜单项，出现 New 对话框，在对话框中选择 Projects(工程)选项，在 Project 内容中选择 Win32 Console Application(Win32 控制台应用程序)，建立一个基于 Win32 控制台的应用项目，也就是常说的具有 DOS 界面的程序。对于建立其他类型工程，同学们可以参考其他书籍。在 Project Name(工程名称)编辑框中输入

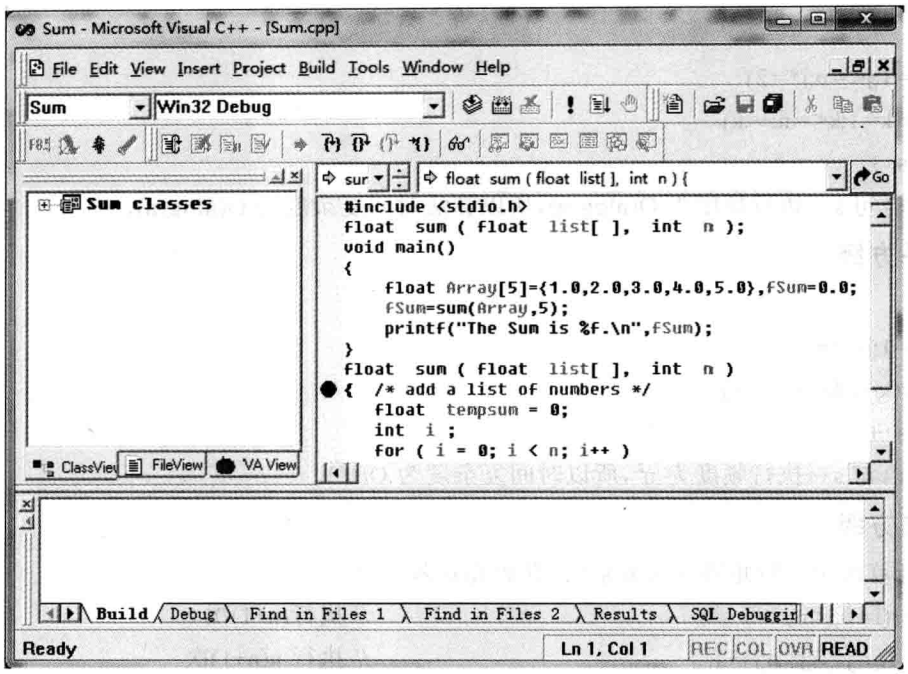

图 1-4 VC 主界面

项目的名称，在 Locate(位置)编辑框中输入项目存放的路径，也可点击 Locate 编辑框右边的 ... 按钮，在弹出的 Choose Directory(选择目录)对话框中选择路径，如图 1-5 和图 1-6 所示。

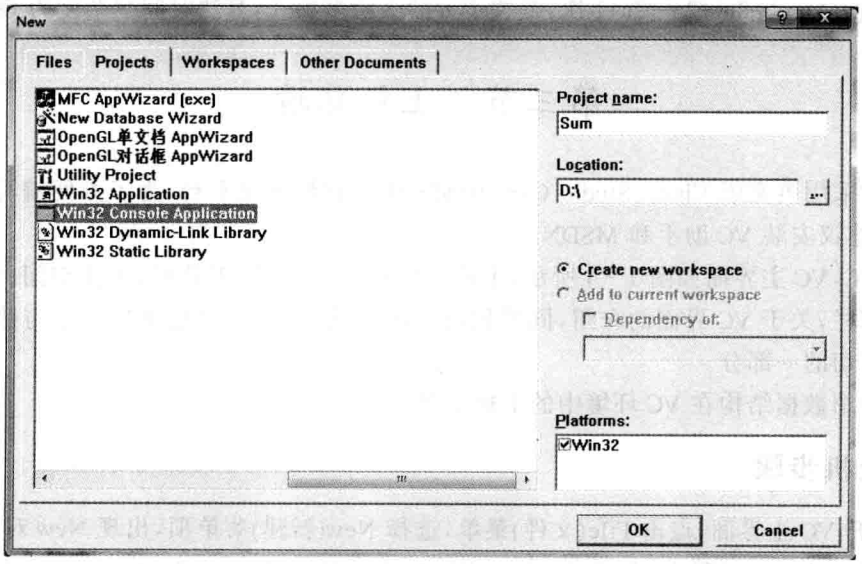

图 1-5 New 对话框

此为试读，需要完整PDF请访问：www.ertongbook.com