在不断地发展中，也在不断地受到专业人士的关注。

很多计算机工作者认为，程序设计的实质就是通过分析问题，确定数学模型和算法，然后再选择一个好的数据结构。即

$$\text{程序} = \text{算法} + \text{数据结构}$$

因此，建议读者在学习“数据结构”这门课程时应做到以下几点：

（1）牢记典型算法的基本思想和大致的求解步骤；

（2）注意各种结构之间的相互联系和区别；

（3）使用C语言按照算法编制程序，上机调试；

（4）分析遇到的问题，并研究解决问题的方法。

学习本门课程，要有前驱课程的准备。这本教材中的算法都是用C语言编写的，几乎不用做大的改动就能上机运行，所用的编程环境为Turbo C。建议读者在学习过程中一定要牢牢掌握数据结构中的一些基本概念和典型算法的基本思想，并注重上机实验的过程，它可以加深对所学算法的理解和掌握，同时也有助于提高C语言编程能力。

1.2 基本概念

要学好“数据结构”这门课程，必须要明确各种概念及相互之间的联系。本节只介绍一些主要基本概念，其他的相关术语将在以后各章中陆续讲述。

1. 数据

数据(data)是对客观事物的符号表示。在计算机学科中，数据是指所有能输入到计算机中，并能被计算机程序所处理的符号的总称。因此上，除了日常所习惯的符号，如除数字构成的整数和实数、字母构成的串之外，还有标点符号、键盘符号，甚至于图形、图像、声音等也都是数据的表示形式。

2. 数据元素和数据项

数据元素(data element)是描述数据的基本单位。**数据项**(data item)是描述数据的最小单位。在计算机中表示数据时，都是以一个数据元素为单位的，如一个整数表示的一个数据元素、一条记录表示的一个数据元素等。用一条记录表示一个数据元素时，这条记录中一般还会有多个描述记录属性的小项，称为数据项。比如，描述一台自行车的记录中可以包括车型、颜色、出厂日期、材质等小项；描述一个班级的记录可以包括班级名、人数、男女生比例、教室、班委会成员和团支部成员等小项。通常，数据项是不可再分的数据。

3. 数据对象

数据对象(data object)是性质相同的一类数据元素的集合。数据是非常广泛的一个概念，是用来描述千变万化的客观世界的。从中取出一部分，而这部分元素具有共同的性质，这些数据元素就可以组成一个数据对象，实质上，数据对象是数据的一个子集。如整数集、字符集、由记录组成的文件等。

4. 数据结构

数据结构(data structure)是由数据和结构两部分组成。其中,数据部分是指数据元素的集合;结构就是关系,结构部分是指数据元素之间关系的集合。所以,数据结构就是指数据元素的集合及数据元素之间关系的集合。概括地讲,数据结构就是指相互之间有一种或多种特定关系的数据元素的集合。在计算机上要处理数据,就要保存数据及它们之间的关系。在这里,关系就是结构。通常,数据之间有如下几种关系。

(1) **集合结构**:结构中的数据元素除了"同属于一个集合"的关系外,再无其他关系。

(2) **线性结构**:结构中的数据元素之间存在"一对一"的邻接关系。比如,生产过程中的流水作业、体育比赛中的接力赛跑等。

(3) **树状结构**:结构中的数据元素之间存在"一对多"的关系。比如,多米诺骨牌、政府组织机构等。

(4) **图状结构或网状结构**:结构中的数据元素之间存在"多对多"的关系。比如,城市交通图、电话网等。

图 1.1 是上述几种结构的关系图。其中,树状结构和图状结构又称为非线性结构。由于集合中数据元素之间关系是非常松散的,因此,常用其他几种结构来描述集合。数据结构依据抽象描述方式和机内存储形式可分为逻辑结构和物理结构两大类。

(1) **逻辑结构**(logical structure)是以抽象的数学模型来描述数据结构中数据元素之间的逻辑关系。通常用二元组来描述这种关系:

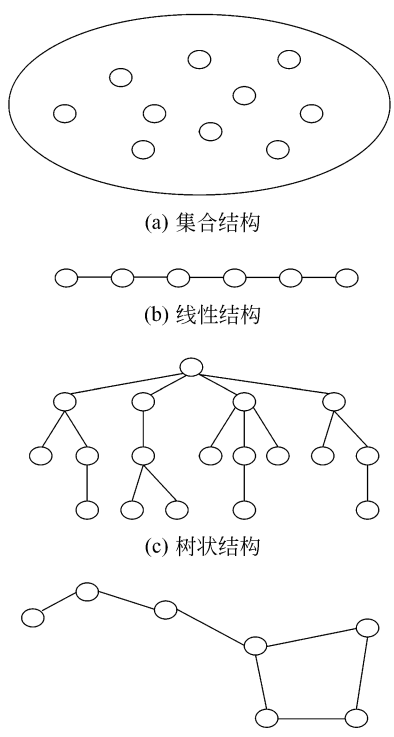
Data_Structure=(D,R)

其中,D 是数据元素的有限集,R 是 D 上的关系的有限集。例如,复数的逻辑结构可以用如下的二元组来描述:

Complex=(D,R)

其中,$D=\{x|x$ 为实数$\}$,$R=\{\langle x,y\rangle|x,y\in D,x$ 为实部,y 为虚部$\}$。其中,$\langle x,y\rangle$表示一个**有序偶**,即 x 和 y 有顺序关系,$\langle x,y\rangle$不等价于$\langle y,x\rangle$。常用(x,y)表示**无序偶**,即 x 和 y 无顺序关系,(x,y)等价于(y,x)。若在 D 中任取两个实数,如 5 和 3,则通过关系$\langle 5,3\rangle$可以表示一个复数 5+3i,而通过关系$\langle 3,5\rangle$可以表示一个复数 3+5i。

(2) **物理结构**(physical structure)又称存储结构,是数据结构在计算机内的存储表示,也称内存映象。一个存储在内存中的数据元素又称为**结点**,数据元素中的每个数据项又称为**域**。因此,数据元素或结点可以看成是数据元素在计算机中的映象。数据元素可以存放到内存某个单元,那么如何存储数据元素之间的关系呢?在计算机内部主要是顺序存



(a) 集合结构

(b) 线性结构

(c) 树状结构

(d) 图状结构

图 1.1　4 种基本结构关系图

储和链式存储这两种结构来表示数据元素之间的逻辑关系。顺序存储结构的特点是用物理地址相邻接来表示数据元素在逻辑上的相邻关系；链式存储结构的特点是逻辑上相邻接的数据元素在存储地址上不一定相邻接，数据元素逻辑上的相邻关系是通过指针来描述的，常用它来描述树状结构和图状结构在计算机内的存储。除这两种存储结构外，在计算机内还有索引存储结构和散列存储结构等，其实质都是通过顺序存储结构和链式存储结构复合而成。

逻辑结构和物理结构是描述数据结构的密不可分的两个方面。任何一个算法的设计取决于选定的逻辑结构，而算法的实现取决于依托的存储结构。

5. 数据类型

在客观世界中，任何数据元素都应该有自身的取值范围和所允许进行的运算操作。比如，车是一个数据对象，车族中有各种各样的汽车、火车、三轮车、自行车等，如果给车安装一对翅膀那就超出了车的范围，变成了飞机。车能进行的操作是安装、行驶、载人、运货、比赛等，如果让车到天上去飞，那就太难为它了。因此，数据类型(data type)就是一个值的集合和定义在这个值集上的一组操作的总称。例如，C语言中的基本整数类型(signed int)，它的值集是－32 768～32 767，在这个值集上能进行的操作有加、减、乘、除和取余数等，而在实数类型(float)上就不能进行取余数的操作。按值的不同特性，数据类型又可分为不可分解的**原子类型**及可分解的**结构类型**。比如C语言中的整型、实型、字符型就属于原子类型，而数组、结构体和共用体类型就属于结构类型，可由其他类型构造得到。

6. 抽象的数据类型

抽象的数据类型(abstract data type，ADT)是指一个数学模型以及定义在该模型上的一组操作。抽象数据类型的定义仅取决于它的一组逻辑特性，而与其在计算机内部如何表示和实现无关，即不论其内部结构如何变化，只要它的数学特性不变，都不影响其外部的使用。抽象的数据类型可以细分为如下3种类型。

(1) **原子类型**(atomic data type)：其值是不可分的。

(2) **固定聚合类型**(fixed_aggregate data type)：其值由确定数目的成分按某种结构组成。

(3) **可变聚合类型**(variable_aggregate data type)：其值由不确定数目的成分构成。

一个抽象的数据类型的软件模块通常包含定义、表示和实现3个部分。

7. 多型数据类型

多型数据类型(polymorphic data type)是指其值的成分不确定的数据类型。

1.3 算法描述与分析

解决实际问题要找出解决问题的方法。要用计算机解决实际问题，就要先给出解决问题的算法，再依据算法编制程序完成要求。所谓**算法** (algorithm)就是对求解问题步骤

的一种描述，也称为**算法设计**。描述算法的工具有许多，比如自然语言、框图、计算机语言程序、伪代码、类计算机语言等。本书主要采用 C 语言函数来描述算法，但在书写算法时可能会忽略一些细节，如变量定义、交换两个变量的值等。读者只要补充这些细节，再添加上主函数(main)就可以上机调试算法程序了。

1. 算法的 5 个重要特性

(1) **有穷性**：一个算法必须总是(对任何合法的输入值)在执行有穷步之后结束，且每一步都可在有穷的时间内完成。这也是算法与程序的最主要区别，程序是可以无限地循环下去的，如操作系统的监控程序在机器启动后就一直在监测着操作者的鼠标动作和输入的命令。

(2) **确定性**：算法中的每一条指令都必须有明确的含义，不应使读者产生二义性。并且，在任何条件下，算法只有惟一的一条执行路径，即对于相同的输入只能得到相同的输出。

(3) **可行性**：一个算法是可以被执行的，即算法中的每个操作都可以通过已经实现的基本运算执行有限次来完成。

(4) **有输入**：根据实际问题需要，一个算法在执行时可能要接收外部数据，也可能无需外部输入。所以，一个算法应有零个或多个输入，这取决于算法本身要实现的功能。

(5) **有输出**：一个算法在执行完成后，一定要有一个或多个结果或结论。这就要求算法一定要有输出，这些输出是同输入有着某些联系的量。

2. 算法的评价

通常，解决同一个问题，不同的人有不同的想法，即使是同一个人，在不同的时间里可能对同一个问题的理解也不完全相同。算法是依据个人的理解和想法人为设计出来的求解问题的步骤，不同的人或同一个人在不同的时间里设计出来的算法也不尽相同，那么哪种算法设计得好呢？如何评价一个算法的好与不好呢？通常，在算法设计时应该考虑如下几个方面。

(1) **正确性**：这是算法设计的最基本要求，算法应该严格地按照特定的规格说明设计，要能够解决给定的问题。但是，“正确”一词的含义在通常的用法中有很大的区别，大体上可分为以下 4 个层次：

① 依据算法所编制的程序中不含语法错误。

② 程序对于几组输入数据能够得到满足规格说明要求的结果。

③ 程序对于经过精心挑选较为苛刻的几组输入数据也能够得到令人满意的结果。

④ 程序对于所有符合要求的输入数据都能得到正确的输出。

对于大型软件需要进行专业测试，一般情况下，通常以第 3 个要求作为衡量算法正确性的标准。

(2) **可读性**：设计算法的主要目的是解决实际问题，在设计实现一个项目时，往往不是一个人去独立完成。为了达到可读性的要求，在设计算法时，一般要使用有一定意义的标识符给变量、函数等起名，达到“见名知意”。其次，可以在算法的开头或指令的后面加

注释来解释算法和指令的功能。

（3）**健壮性**：当输入不合法数据时，算法能作出相应的响应或进行适当地处理，避免带着非法数据执行，导致莫名其妙的结果。

（4）**高效率**：依据算法编制的程序运行速度较快。

（5）**低存储**：依据算法编制的程序运行时所需内存空间较少。

对于一个系统设计人员来说，前3项很容易实现。在使用软件时，人们更加注重于软件的运行速度，而后两点恰恰是影响速度的主要标志。

3. 时间复杂性

算法的执行时间需通过依据算法所编制的程序在计算机上运行时所消耗的时间来度量。度量一个程序执行的时间通常有两种方法。

（1）**事后统计法**：统计依据算法所编制的程序在计算机上运行时所消耗的时间。但是，同一个程序在不同类型的机器上运行所需的时间不一定相同，所以这种统计是片面的。

（2）**事先估算法**：根据每条指令的执行时间估算依据算法编制的程序在计算机上运行时所消耗的时间。但是，因为每种类型机器的指令集不同，执行的时间也不尽相同，这种方法也离不开具体的机器软硬件环境和设备。

显然，以具体的时间单位作为计算程序执行时的时间度量是不科学的。所以在计算算法的执行时间时，应该抛开具体机器软硬件环境和设备，而使用指令的执行次数作为时间单位更合理些。在算法中，可以使用基本语句的执行次数作为算法的时间度量单位。可以认为，一个特定算法时间性能只依赖于问题的规模（通常用 n 来表示），或者说它是关于问题规模 n 的一个函数 $f(n)$，当问题规模 n 趋近于无穷大时的时间量级就称为算法的渐近时间复杂性，简称**时间复杂性**或称**时间复杂度**。记作 $T(n)=O(f(n))$，即 $T(n)$ 是 $f(n)$ 的同阶无穷大。

【例1.1】 分析如下程序段的时间性能。

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

分析：

```
s=0; 执行1次
i=1; 执行1次
i<=n; 执行n+1次
s=s+i; 执行n次
i++; 执行n次
```

总的执行次数为 $3(n+1)$ 次，因此，该算法的时间复杂性为 $T(n)=O(3(n+1))=O(n)$。

【例1.2】 分析如下程序段的时间性能。

```
x=0;y=10;
```

```
while(y<100)
{ if(x==10) x=1;
  x++; y+=x;
}
```

分析：

x=0；y=10 各执行 1 次；循环部分语句的执行过程取决于 x 值和 y 值的变化，见表 1.1。

表 1.1　例 1.2 的时间性能统计表

x 值	0	1	2	3	4	5	6	7	8	9	10	2	3	4	5	6	7	8	
y 值	10	11	13	16	20	25	31	38	46	55	65	67	70	74	79	85	92	100	总次数
语句	当 x 值和 y 值变化时，各语句执行情况统计(1：执行，0：未执行)																		
y<100	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	18
x==10	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	17
x=1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
x++	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	17
y+=x	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	17

所以，总的执行次数为 2+18+17+1+17+17=72 次。这与问题规模 n 无关，因此，该算法的时间复杂性可以写成 $T(n)=O(n)$，是线性的。

如果这样去估算算法的时间性能，那就太不合适了。在分析算法时间性能时，常用最基本语句的执行次数来估算。所谓最基本语句通常是指最深层循环体中的语句，也是执行频度最快的语句。它的执行次数反映了整个算法的基本时间性能。如例 1.1 中的 s=s+i 和 i++均被执行了 n 次，所以 $T(n)=O(n)$；例 1.2 中的 x==10、x++及 y+=x 都被执行了 17 次，所以 $T(n)=O(1)$。

实际上，算法的时间量级有多种形式，见表 1.2，其对应的函数曲线见图 1.2。

表 1.2　算法的时间量级分类表

名　称	时间复杂度 $T(n)$	说　明
常量阶	$O(1)$	与问题规模无关的算法
线性阶	$O(n)$	与问题规模相关的单重循环
平方阶	$O(n^2)$	与问题规模相关的二重循环
立方阶	$O(n^3)$	与问题规模相关的三重循环
指数阶	$O(e^n)$	较为复杂
对数阶	$O(\log_2 n)$	折半查找算法
复合阶	如 $O(n\log_2 n)$	堆排序算法
其他	不太确定	过于复杂

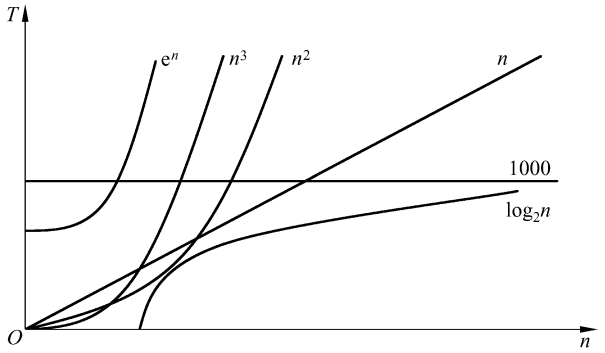


图 1.2 常见函数的增长率

【例 1.3】 讨论起泡排序算法的时间复杂性。

```
void bubblesort(int a[],int n)                    /* 0单元作为临时存储空间 */
{ int i,j,change;                                 /* change为交换标志 */
  for(i=change=1;change&&i<n;i++)
    for(j=1,change=0;j<=n-i;j++)
      if(a[j]>a[j+1])
      { a[0]=a[j];a[j]=a[j+1];a[j+1]=a[0];
        change=1;                                 /* 有交换,则进行下一趟比较 */
      }
}
```

这一算法的基本操作是内层循环的比较语句：$a[j]>a[j+1]$，而该语句的执行次数既受到与问题规模 n 有关的循环控制变量 i 和 j 影响，也受到是否进行下一次循环的条件变量 *change* 的影响，这与输入的数据有关。若输入的数据是已经有序的，则外循环仅进行一次，所以 $T(n)=O(n)$；若输入的数据是杂乱无章的，则外循环可能会执行 $n-1$ 次，则比较语句的执行次数为 $(n-1)+(n-2)+\cdots+(n-i)+\cdots+1=n(n-1)/2$，所以，$T(n)=O(n^2)$。

通过上例可以看出，一个算法时间复杂性可能存在最好情况和最坏情况，通常要以算法的平均时间复杂性来进行算法分析。但是算法的平均时间复杂性取决于各种数据出现的概率，难以进行分析，所以，往往以借助于最坏时间复杂性来进行算法分析与评价。

4. 空间复杂性

与时间复杂性类似，空间复杂性也是关于问题规模 n 的一个函数，当问题规模 n 趋近于无穷大时的空间量级就称为算法的渐进空间复杂性，简称**空间复杂性**。记作

$$S(n) = O(f(n))$$

那么，算法的空间需求有哪些呢？大体上，依据算法所编制的程序除了需要存储空间来寄存程序本身所用的指令、常数、变量和输入数据外，也需要一些对数据进行操作的工作单元和存储一些为实现计算所需信息的辅助空间。通常，程序所占空间变化不大，所以在此主要考虑算法的辅助空间需求。

【例 1.4】　将一维数组中的元素逆置存放。

完成这一题目可有几种方法，下面列举出 3 种实现本要求的算法，从中分析哪种更好些。

方法 1：

```
RevArray1(int a[],int n)
{ int i,j,t;
  for(i=0,j=n-1;i<j;i++,j--)
  { t=a[i];a[i]=a[j];a[j]=t;}
}
```

分析：

由于基本语句就是循环内的交换语句，共执行了 $n/2$ 次，所以

$$T(n) = n/2 = O(n)$$

又因为算法的辅助空间只是 i,j,t 3 个临时变量的空间，所以

$$S(n) = 3 = O(1)$$

方法 2：

```
RevArray2(int a[],int n)
{ int i,t;
  for(i=0;i<n/2;i++)
  { t=a[i];a[i]=a[n-i-1]; a[n-i-1]=t;}
}
```

分析：

由于基本语句就是循环内的交换语句，共执行了 $n/2$ 次，所以

$$T(n) = n/2 = O(n)$$

又因为算法的辅助空间只是 i,t 两个临时变量的空间，所以

$$S(n) = 2 = O(1)$$

方法 3：

```
RevArray3(int a[],int n)
{ int i,j, * b;
  b=(int *)malloc(sizeof(int) * n);
  for(i=0,j=n-1;i<n;i++,j--)
    b[j]=a[i];
  for(i=j=0;i<n;i++,j++)
    a[i]=b[i];
  free(b);
}
```

分析：

由于基本语句就是循环内的赋值语句，共执行了 $2n$ 次，所以

$$T(n) = 2n = O(n)$$

而算法的辅助空间是一个与问题规模同量级的一维数组空间，另外再加上两个控制变量 i 和 j，共 $n+2$ 个，所以

$$S(n) = n+2 = O(n)$$

从上述分析可以看出，方法 2 最好，方法 3 最差，方法 1 最简单。

在进行算法设计时，有时很难兼顾算法的时间性能和空间性能。因此，在进行算法设计时应该综合起来进行考虑，分析面临的问题，分析要迫切解决的是时间需求还是空间需求，或者不需要考虑这两个因素，而只要算法简单些。

本章介绍了数据结构的一些基本概念和算法分析方法，随着课程的进行，后面要用到这些最基本知识，读者一定要注意领会这些基本内容。

习 题 1

1. 回答下列问题。

(1) 什么叫数据、数据元素、数据项和数据对象？

(2) 什么叫数据结构？分为哪几类？具体都是什么？

(3) 什么叫数据类型？什么叫抽象的数据类型？

(4) 什么叫算法？算法有哪几个基本特性？

(5) 如何评价一个算法的好与坏？

2. 举例说明常见的时间复杂性有哪些。

3. 写出下列算法总的语句执行次数。

(1)
```
y=5;x=1;
while(y<=10)
  if(x==5)
  { x=1;y+=x;}
  else x++;
```

(2)
```
x=0;
for(i=0;i<10;i++)
  for(j=0;j<=i;j++)
    x=x+1;
```

4. 判断如下时空性能的计算是否正确(其中 n 为问题规模，K 为常数)。

(1) $O(1)=O(2)=\cdots=O(100)$(　　)

(2) $O(1)+O(2)=O(1)$(　　)

(3) $O(1)+O(n)=O(n)$(　　)

(4) $O(1)\times n=O(n)$(　　)

(5) $O(1)\times K=O(n)$(　　)

(6) $O(n)\times K=O(K\times n)=O(n)$(　　)

(7) $O(n)\times O(n)=O(n^2)$(　　)

(8) $O(n)+O(n)=O(n)$(　　)

(9) $O(n)+O(m)=\max(O(n),O(m))$　m 也是问题规模(　　)

(10) $O(n^P)=K_P\times n^P+K_{P-1}\times n^{P-1}+\cdots+K_1\times n^1+K_0$　P 和 $K_i(0\leqslant i\leqslant P)$也是常数(　　)

5. 分析如下各种算法的时空性能。

(1) 计算 n 个实数的平均值,并找出其中的最大数和最小数。

```
float ave=0,max,min;
float calave(float a[],int n)
{ int i;
  max=min=a[0];
  for(i=0;i<n;i++)
  { ave+=a[i];
    if(max<a[i]) max=a[i];
    if(min>a[i]) min=a[i];
  }
  return ave/n;
}
```

(2) 将一个(有 m 个字符)字符串中在另一个(有 n 个字符)字符串中出现的字符删除。

```
int found(char *t,char *c)
{ while(*t&&*t!=*c) t++;
  return *t;
}
delchar(char *s,char *t)
{ char *p,*q;
  p=s;
  while(*p)
    if(found(t,*p))
    { q=p;
      while(*q) *q++=*(q+1);
    }
    else p++;
}
```

(3) 用递归法求 $n!$ 值。

```
fun(int n)
{ int s;
  if(n<=1) s=1;
  else s=n*fun(n-1);
  return s;
}
```