第1章 基础知识

知识要点

(1) 理解数据结构的基本术语
(2) 掌握抽象数据类型与数据结构的关系
(3) 能够描述数据结构的逻辑结构
(4) 了解算法增长率的概念；
(5) 掌握渐近分析的表示方法；
(6) 能够对程序、算法或问题的上下限作出估算；
(7) 掌握时间复杂度与空间复杂度的分析方法；

1.1 数据结构的基本概念

使用计算机解决实际问题，大致需要以下三个步骤：

（1）分析实际问题，抽象出一个适当的数学模型。

（2）为此数学模型设计一个合适的数据结构。

（3）设计和实现具体操作的算法。

数据结构涉及一些基本概念和术语:

（1）**数据(Data)**：指**计算机能接受和处理的一切对象**。如整数、实数、复数是数据，字符、文字、表格、图形、图像、声音、程序等也是计算机能够接受和处理的数据。总之，**存储在计算机中可用二进制表示的内容**都是数据。

（2）**数据元素(data element)**：指**在计算机中作为一个整体考虑和处理的对象，也是组成数据的基本单位**。一般来说，数据元素是由若干**数据项**(data item)组成的，数据项是具有独立含义的数据最小单位。

如，学籍管理系统中，“学生”是数据元素，而“学生”是由学号、姓名、性别、年龄、家庭住址、联系方式等数据项组成。

（3）**数据对象(data object)**：指**性质相同的数据元素的集合**。

如：

自然数的数据对象是集合N={1,2,3,…},

字符数据对象是集合Letter={A,B,C,…,Z,a,b,…z}

学生数据对象是集合S={Student1,Student2,Studengt3...}。

（4）**数据结构(Data Structure）**：是**研究数据及数据之间关系的一门学科**，它包括三个方面：

①**数据的逻辑结构**(logic structure)，即依据实际情况描述数据元素之间的逻辑关系。

②**数据的存储结构**(storage structure)或称物理结构(physical structure)，即数据在计算机中的表示和存储方式。

③**数据的操作(operation)实现**，即在相应数据结构存储下所提供的基本操作的算法实现。

几种逻辑结构可用一个层次图描述，如图1-1所示

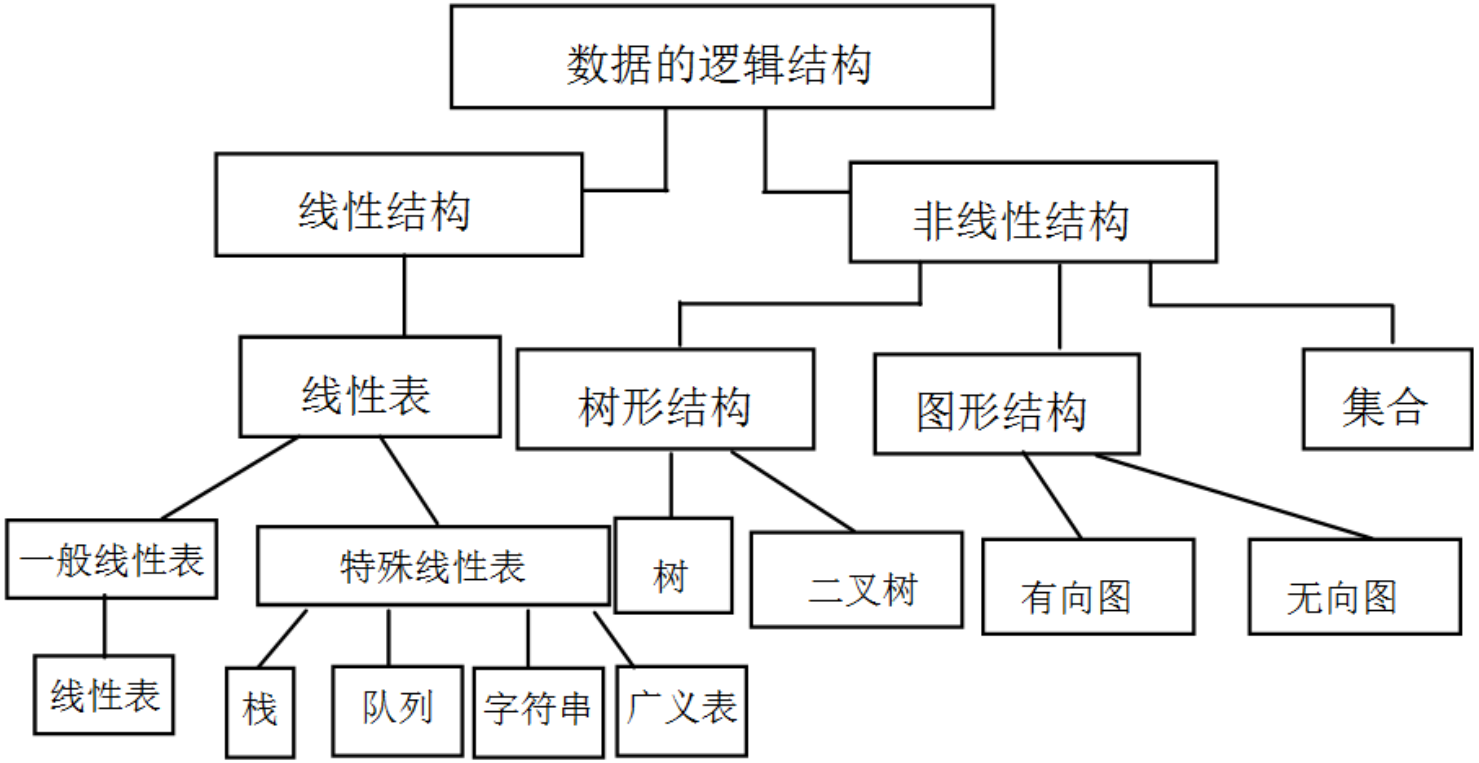


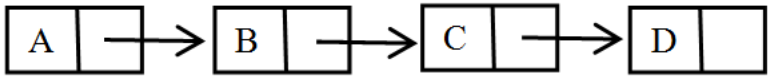
图1-1 数据逻辑关系图

数据的逻辑结构强调的是数据元素之间的逻辑关系，反映了数据的特性。

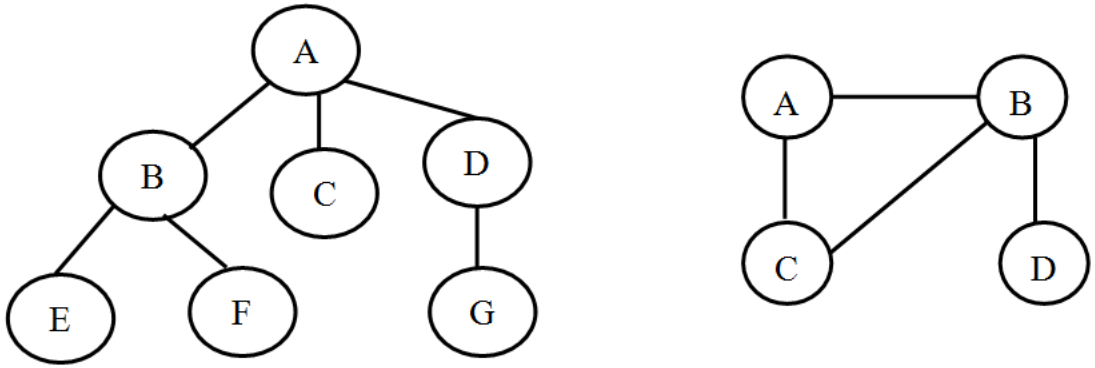
任何数据结构都是**由数据集和关系集组成**，**DS=(D，R)**

其中，D表示数据元素集合

R表示数据元素之间的关系集合。



(a)线性表 L



（b）树 T　　(C)图 G

图 1-2 数据的逻辑结构示例

图1-2(a)描述了线性结构---线性表L。L用数据集与关系集表示如下：L=(D,R)

其中：D={A,B,C,D}

R={<A,B>,<B,C>,<C,D>}

图1-2(b)描述了非线性结构---树T。T用数据集与关系集表示如下：T=(D,R)

其中：D={A,B,C,D,E,F,G}

R={<A,B>,<A,C>,<A,D>,<B,E>,<B,F>,<D,G>}

图1-2(c)描述了非线性结构---图G。G用数据集与关系集表示如下：G=(D,R)

其中：D={A,B,C,D}

R={(A,B),(A,C),(B,C),(B,D)

数据在计算机中的存放有四种基本的存储方法：

（1）顺序(Sequential)存储方法：就是把每个数据元素，按某种顺序存放在一段连续的存储单元中。其局限性：一是需要足够大的连续存储空间，不能有效利用零碎小块；二是事先无法得知所需存储空间的大小，预留过大过小都不合理。

（2）链式(Linked)存储方法：就是把每个数据元素，按结点结构可以分别零散地存放在存储单元中。这种方法就是将结点所占的存储单元分为两部分：一部分存放结点本身的元素信息，另一部分存放此结点的逻辑前驱或后继结点所对应的存储地址，称为指针项。指针项也可以有多个。

（3）索引(Index)存储方法：按关键字段建立索引表，用结点的索引号来确定结点的存储地址，而把每个结点的元素数据按一定规律顺序存放在存储单元中。

（4）散列(Hash)存储方法：设计散列函数，每个结点的存储单元位置通过设计的散列函数计算得到。

当为解决实际问题而选择一个数据结构时需以下三步：

（1）分析问题，确定解决问题需满足的资源限制。一种算法如能在所需求的资源限制内将问题解决好，则称该算法是有效率的；

（2）确定满足资源限制的基本操作，例如对数据结构中的数据元素进行插入、删除、查找和定位等；

（3）选择满足需求的数据结构，每个数据结构都与代价和效益紧密相联，每一个问题都有时间和空间资源的限制。

1.2 抽象数据类型

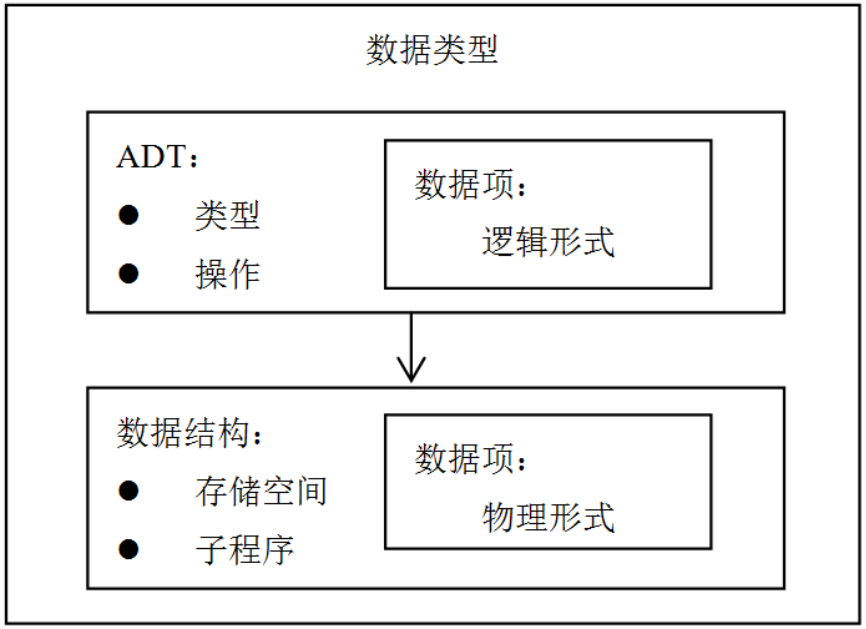


图1-3 数据项、抽象数据类型和数据结构之间的关系

ADT的表示和实现概括起来应该有三种情况：

- ✓ 第一种情况是面向过程的表示和实现，C语言。
- ✓ 第二种情况是面向模块结构的表示和实现，如Turbo Pascal4.0版中的Unit。
- ✓ 第三种情况是面向对象技术的表示和实现，如Visual C++、Visual Basic、Java等。

抽象数据类型可用以下三元组表示：**(D，R，P)**。

其中：D表示数据对象，R是D上的关系，P是对D的基本操作集。

ADT抽象数据类型名｛

数据对象：〈数据对象的定义〉

数据关系：〈数据关系的定义〉

基本操作：〈基本操作的定义〉

｝ADT抽象数据类型名

其中数据对象和数据关系的定义用伪码描述，基本操作的定义格式为：

基本操作名（参数表）

初始条件：〈初始条件描述〉

操作结构：〈操作结构的功能描述〉

表 1-1 手机通讯录的数据结构

姓名	移动电话	办公电话	E-mail	QQ
张一	13002900101	029-11111111	zhyi@163.com	129159164
王二	13802900202	029-22222222	wer@sina.com	204554897
李三	13902900303	029-33333333	lisan@qq.com	705897623
赵四	13002911101	029-44444444	zhsi@126.com	422215108
…	…	…	…	…

手机通讯录的抽象数据类型描述如下：

ADT Address_list{

数据对象： $D=\{a_i|a_i \in ElemSet,\ i=1,\ 2,\ \ldots,\ n;\ n\geq 0\}$

数据关系： $R=\{<a_i,a_{i+1}>|\ a_i,a_{i+1} \in D,\ i=1,\ 2,\ \ldots,\ n\}$

基本操作： 添加新联系人（指定人）；

删除联系人（指定人）；

排序联系人（指定数据项）；

修改联系人（指定人）；

查询联系人（指定数据项）；

等等；

}ADT Address_list

1.3 问题、算法和程序

- **问题**（problem）是一种计算机需要完成的任务。
- **算法**（algorithm）是计算机对特定问题求解步骤的一种描述，它是指令的有限序列。

算法的特性：

（1）有穷性：对任何合法的输入值，一个算法的执行步骤是有限的，且每一步都可在有穷时间内完成。

（2）确定性：算法中的每个步骤都应明确，算法的每一条指令必须有确切的含义，使读者理解时不会产生歧义。而且，算法在给定输入条件下都有且只有一条实际的执行路径，即相同的输入下得到相同的输出。

（3）可行性：指令可以在现在的计算技术基础上实现，即算法是有效的。

（4）输入/输出：任何算法都是对输入数据加工的过程，最后给出输出结果。

（5）通用性：算法要具有一般性，对一般的数据集合都要成立。

（6）可读性：具备良好可读性的算法有利于查错和理解。

（7）健壮性：当输入数据非法时，算法能适当的处理并作出反应，而不应出现死机状况或输出异常结果。

算法分析中的两个重要因素：

✓ **时间复杂度**：如何使算法执行速度尽可能快，所需时间尽可能短；

✓ **空间复杂度**：如何使执行算法运行所需的存储量尽可能小。

任何算法可以用自然语言、伪代码或者某种计算机语言来描述。

程序（program）是指一组指示计算机每一步动作的指令序列，通常用某种程序设计语言编写，运行于某种目标体系结构上。

首先要从“问题”中抽象出数学模型；然后设计一个求解此数学模型的“算法”；选择合适的数据结构，最后编出“程序”，进行调试直至得出解答。

1.4 概述

一个用高级程序语言编写的程序在计算机上运行时所消耗的时间取决于下列因素：

（1）算法采用的策略；

（2）问题的规模；

（3）编写程序所采用的计算机语言，通常，实现语言的级别越高，执行效率就越低；

（4）编译和运行环境。

渐近算法分析

渐近算法分析（asymptotic algorithm analysis），简称算法分析（algorithm analysis）——是一种估算方法，可以估算出当问题规模变大时，一种算法及实现程序的效率和开销。**时间复杂度和空间复杂度是渐进算法分析的两个指标。**

渐近时间复杂度（asymptotic time complexity）是指评估算法编制完成程序后，在计算机中运行所消耗的时间函数。一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数$f(n)$，算法的时间度量记作：

$$T(n)= O(f(n))$$

它表示随着问题规模n的增大，算法执行时间的增长率与$f(n)$的增长率相同。由于输入的规模是影响运行时间的最主要因素，所以经常把执行算法所需的时间T写成输入规模n的函数，记做$T(n)$（$T(n)$为非负值）。

算法增长率（**growth rate**）是指当输入的问题规模增长时，算法代价的增长速率。通过渐近复杂度分析，可以把某个算法实际的复杂度函数表示结果化简到某个简化函数类别上。

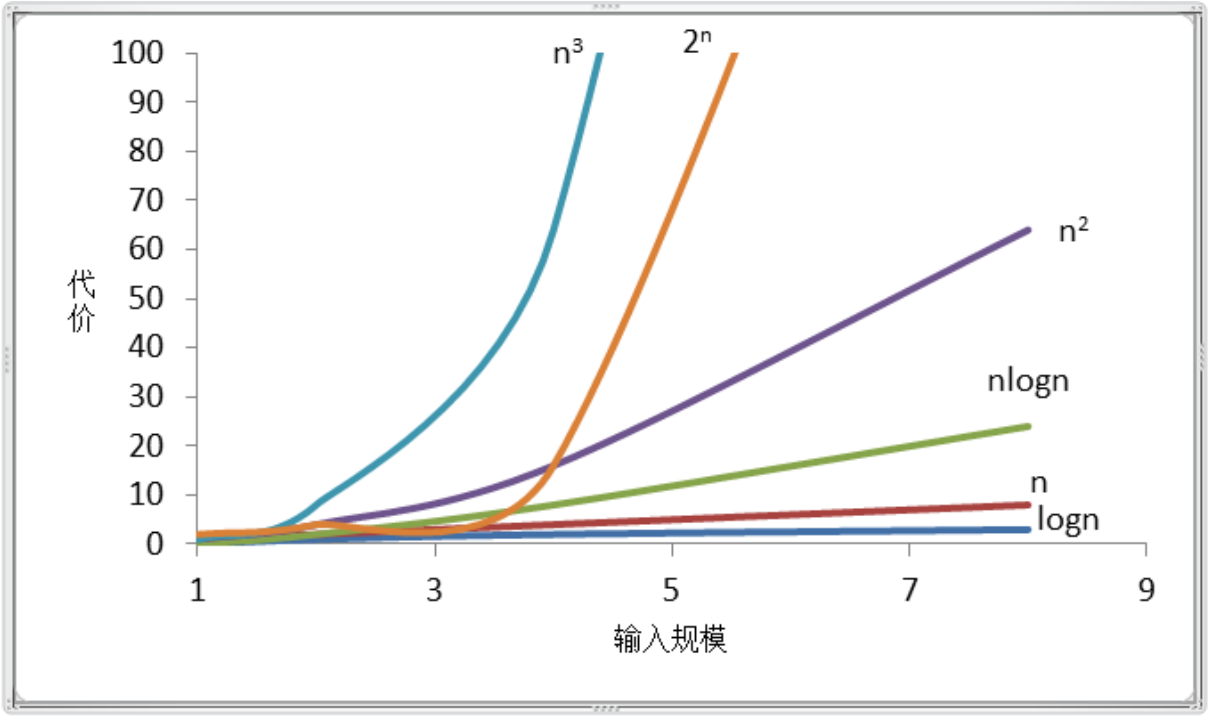


图 2-1 各种运行时间函数的曲线

- ✓ 标记为n的函数图像为直线，对应的时间复杂度为O(n)，称为**线性阶**。说明，当n增大时，算法的运行时间也以相同的比例增加。
- ✓ 标记为n^2的函数图像为抛物线，对应的时间复杂度为$O(n^2)$，称为**平方阶**。如果算法的运行时间函数中含有形如n^2的高次项，则称为二次增长率。
- ✓ 标有2^n的曲线属于指数增长率（exponential growth rate），对应的时间复杂度为$O(2^n)$，称为**指数阶**。
- ✓ 此外，还有**常数阶**O(1)，**对数阶**O(logn)等，它们都是由n出现的位置而得名。

渐近空间复杂度（asymptotic space complexity）是指算法编制成程序后，在计算机中运行时所占的**存储空间**函数。

存储空间包括：

- ✓指令、常数、变量；
- ✓输入数据；
- ✓辅助空间
- ✓运行栈区（静态、动态）等

空间复杂度作为算法所需存储空间的量度，记作：

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

1.5 时间复杂度

时间复杂度分析涉及两个阶段：

- 第一阶段是算法分析：对算法或数据结构进行分析，用函数T(n)测量复杂度，T(n)是关于输入数据大小n的函数；
- 第二阶段是渐近分析：分析函数T(n)，确定它属于哪种复杂度类型。

严格分析规则：

（1）可以假设一个时间单位.

（2）执行如下的操作之一，花费的时间计为1：

a. 赋值运算

b. 单一的输入/输出操作

c. 布尔操作，数据比较

d. 算术运算

e. 函数返回

f. 数组下标操作，指针引用

（3）选择语句（if，switch）运行时间：

条件计算 + 选择语句中运行时间最长的语句执行时间

（4）循环执行时间：

循环体执行时间 + 循环条件检查和更新操作时间 + 创建循环的时间

通常假设循环执行的最大可能迭代数量。

（5）函数调用的运行时间：

创建函数时间（1）+ 所有的参数计算时间 + 函数体执行时间

【例1-2】分析双重程序段的运行时间：

```
for (i = 0; i < n-1; i++) {
    for (j = 0;  j < i;  j++ ) {
        array[i][j] = 0;
    }
}
```

①

②

③

④

⑤

遵守规则4和2a，在进入循环前，执行时间为1

遵守规则4，2c和2d，外层循环每次迭代的执行时间为3

遵守规则4和2a，执行时间为1

遵守规则4，2c和2d，内层循环每次迭代的执行时间为2

每一个循环多做一次测试退出

遵守规则2a和2f，每次内层循环执行时间为3

总的时间

$$T(n) = 1 + \sum_{i=0}^{n-2}\left(4 + \sum_{j=0}^{i-1} 5 + 1\right) + 2 = \frac{5}{2}n^2 - \frac{5}{2}n + 3$$

时间复杂度函数的简化分析方法：

通常，可以采用简单分析方法计算程序或算法的运行时间，假设程序的每步执行时间为1。

程序步可以定义为一个语法或语义意义上的程序片段，该片段的执行时间独立于实例特征。

例1-2 程序段用简单分析方法描述

语句	步数	频率	总步数

for (i = 0; i < n-1; i++) {
for (j = 0; j < i; j++) {
array[i][j] = 0;
}
}
	1
	1
	1
	0
	0
		n
		1+2...+n-1
		0+1...+n-2
		0
		0
			n
			n(n-1)/2
			(n-1)(n-2)/2
			0
			0

total	n^2-n+1		

【例1-3】统计A[0..N-1]的元素和，该算法用简单分析方法描述如下所示。

语句	步数	频率	总步数
template<class T>			
T Sum(T a[], int n) {	0	0	0
T tsum=0;	1	1	1
for (int i = 0; i < n; i++)	1	n+1	n+1
tsum += a[i];	1	n	n
return tsum;	1	1	1
}	0	0	0
total			2n+3

【例1-4】递归算法统计A[0..N-1]的元素和，该算法用简单分析方法描述如下所示。

语句	步数	频率	总步数
template<class T>			
T Rsum(T a[], int n) {	0	0	0
if (n > 0)	1	n+1	n+1
return Rsum(a,n-1)+ a[n-1];	1	n	n
return 0;	1	1	1
}	0	0	0
total			2n+2

1.6 渐近分析

1.6.1 上限表示法（大O表示法）

算法运行时间的上限（upper bound）就是用以估计问题所需某种资源的复杂程度的界限函数，它表示该算法可能有的**最高增长率**。算法的上限还应与输入规模n有关。

为了表示起来更简单，采用**大O表示法**，给出函数$f(n)$的一个上限。

【定义：*O*符号】：设$f(n)$和$g(n)$是两个关于整数n的非负函数，若存在两个正常数c和n_0，对所有的$n>n_0$，有$f(n)\leq cg(n)$，则称$f(n)$在集合$O(g(n))$中，简记为$f(n)= O(g(n))$。其中，常数n_0是使上限成立的n的最小值，c是某个确定的常数。

若$f(n)=T(n)$，则对于问题的所有（如最差情况）输入，只要输入规模足够大（即$n>n_0$），算法总能在$cg(n)$步以内完成。

【例1-5】（线性函数）考察T(N) = 3N+ 2，T(N)是一个线性变化的函数。

（1）当n≥2时，3n+ 2≤3n+n＝4n，所以$T(n)= O(n)$，其中：$n_0=2$，c=4

（2）当n≥1时，3n+ 2≤3n+ 2n= 5n，所以$T(n)= O(n)$，其中：$n_0=1$，c=5

对于两组n_0和c的取值都满足n的大O定义，也即它们都是线性函数（对于一定的n）。

因此，用来满足大O定义的c和n_0 的值并不重要，因为只需说明$T(n)$在$O(g(n))$中即可。

【例1-6】（平方函数）考虑冒泡排序算法，T(n)= $c_1[n(n-1)]/2$。T(n)是一个平方函数。

如果比较数组中相邻元素需要的时间为c_1（c_1为正数），那么在最差情况下，对于$n>1$，$c_1[n(n-1)]/2 \le c_1n^2$。所以$T(n)= O(n^2)$，其中：$n_0=1$，$c=c_1$。

【例1-7】（立方函数）某一算法最差情况下

$T(n)=c_1n^3+c_2n^2+c_3n$，c_1，c_2，c_3为正数。若$n>1$，$c_1n^3+c_2n^2+c_3n \le (c_1+c_2+c_3)n^3$。取$c=c_1+c_2+c_3$，$n_0=1$，有$T(n) \le cn^3$，则$T(n)= O(n^3)$。

【例1-8】（常数函数）当T (n)是一个常数时，如T (n) = 10，对于T(n) = $10 \le 10 * 1$，所以T(n)= O(1)，其中：n0 =0，c=10。

【定理1-1】 如果$f(n)=a_m n^m + a_{m-1} n^{m-1}\ldots+ a_1 n+a_0$且$a_m > 0$，则$f(n)= O(n^m)$。

证明对于所有的n≥1有：

$$f(n) \le \sum_{i=0}^{m} |a_i| n^i$$

$$\le n^m \sum_{0}^{m} |a_i| n^{i-m}$$

$$\le n^m \sum_{0}^{m} |a_i|$$

即，若$f(n)$是一个关于n的多项式，则其算法复杂度取最高次幂：$f(n)= (n^m)$。

【定理1-2 [大O 比率定理] 】 对于函数$f(n)$和$g(n)$，若 $\lim_{x\to\infty} f(n)/g(n)$ 存在，则$f(n)= O(g(n))$当且仅当存在确定的常数c，有 $\lim_{x\to\infty} f(n)/g(n)\leq c$。

证明：如果$f(n)= O(g(n))$，则存在$c>0$及某个n_0，使得对于所有的$n\geq n_0$，有$f(n)/g(n)\leq c$，因此 $\lim_{x\to\infty} f(n)/g(n)\leq c$。接下来假定 $\lim_{x\to\infty} f(n)/g(n)\leq c$，它表明存在一个$n_0$，使得对于所有的$n\geq n_0$，有$f(n)\leq max\{1, c\} * g(n)$。

1.6.2 下限表示法（大Ω表示法）

同上限表示法一样，算法运行时间的下限也是估计问题所需某种资源的复杂程度的界限函数，它表示该算法可能有的**最小增长率**。与大O表示法类似，用符号大Ω来估算函数$f(n)$的下限值。

【定义：Ω符号】假设$f(n)$和$g(n)$是关于整数n的非负函数，若存在两个正常数c和n_0，对于$n \geq n_0$，有$f(n) \geq cg(n)$，则称$f(n)$在集合$\Omega(g(n))$中。简记为$f(n)=\Omega(g(n))$。

【例1-9】对于所有的n，有$T(n) = 3n+ 2 > 3n$，因此$T(n)=\Omega(n)$。同样地，$T(n) = 3n+ 3 > 3n$，所以有$T(n)=\Omega(n)$。因而$3n+ 2$和$3n+ 3$都是带有下限的线性函数。

【例1-10】某一算法平均情况下$T(n)= c_1n^3+c_2n^2+c_3n$，其中：c_1，c_2，c_3为正数，则有：

$$c_1n^3+c_2n^2+c_3n \geq c_1n^3 \quad (n>1)$$

因此，取$c=c_1$，$n_0=1$，有$T(n)\geq cn^3$，则$T(n)=\Omega(n^3)$。

【定理1-3】如果$f(n)=a_m n^m+\ldots+a_1 n+a_0$且$a_m>0$，则$f(n)=\Omega(n^m)$。（证明略）

因此，如果某种算法在$\Omega(n^3)$中，那也一定在$\Omega(n^2)$中，正如大O表示法，我们同样希望找到一个最“紧”（即最大）的下限。

【定理1-4 [大Ω比率定理]】对于函数$f(n)$和$g(n)$，若$\lim\limits_{x\to\infty} f(n)/g(n)$存在，则$f(n)=\Omega(g(n))$，对于确定的常数$c$，有$\lim\limits_{x\to\infty} g(n)/f(n)\leq c$。

1.6.3 Θ表示法

以上大O表示法和大Ω表示法是描述某一算法的界限函数。**当界限函数的上、下限相等时**，即如果一种算法既在O(h(n))中，又在Ω(h(n))中，则用Θ表示法，称其为Θ(h(n))。

【定义：Θ符号】 假设$f(n)$和$g(n)$是关于整数n的非负函数，$f(n)=\Theta(g(n))$当且仅当存在正常数c_1，c_2和某个n_0，使得对于所有的$n \geq n_0$，有$c_1 g(n) \leq f(n) \leq c_2 g(n)$。

1.6.4 化简法则

化简规则如下：

（1）**传递性**：**如果 $f(n)=O(g(n))$，且 $g(n)=O(h(n))$，则 $f(n)=O(h(n))$。**

如 $n=O(n^2)$ 且 $n^2=O(n^3)$，则 $n=O(n^3)$。该规则说明，如果g(n)是算法代价函数的一个上限，则g(n)的任意上限也是该算法代价的上限。**在实际问题中，总是试图找到一个最“紧”的上限。**

（2）**系数提取**：**如果 $f(n)=O(kg(n))$，对常量k>0，则 $f(n)=O(g(n))$。**

如 $5n=O(n)$，$100n^2=O(n^2)$。该规则说明大O表示法中的常数因子无关紧要。对于Ω和Θ表示法同样有这样的性质。

（3）**加法规则**：**如果$f_1(n)=O(g_1(n))$，且$f_2(n)=O(g_2(n))$，则$f_1(n)+f_2(n)= O(max(g_1(n),g_2(n)))$。**

如$10n= (n)$，$2n^2= (n^2)$，所以$10n+2n^2= (n^2)$。多项式在 (n 的最高次幂)中，如$7n^6-10n^4+n^3+12n+37= O(n^6)$。该规则说明并列程序段中只需要考虑一段程序中资源开销较大的部分。

（4）**乘法规则**：**如果$f_1(n)= O(g1(n))$，且$f_2(n)= O(g2(n))$，则$f_1(n)*f_2(n)= O(g1(n)*g2(n))$。**

函数	名称
1	常数
$\log_b(n)$	对数（如省略基数，默认以2为底）
n	线性
$n \log_b(n)$	n个logn
n^2	平方
n^3	立方
n^i n的更大常量次幂	大数次幂
2^n	2指数
3^n	3指数
i^n 更大常量的n次方	大数指数
n!	阶乘
n^n	n的n次方

低

高

图1-5 常见函数的大O复杂度级别

【例1-11】 分析下面三个程序片段的运行时间

① { x++; }

② for（i=1；i<＝n；i++）{ x++; }

③ for（j=1；j<＝n；j++）

for（k=1；k<＝n；k++）{ x++; }

这三段程序的时间复杂度分别为$\Theta(1)$，$\Theta(n)$和 $\Theta(n^2)$

【例1-12】 比较下面几段程序的算法复杂度分析

时间复杂度

① x=0;

for（i=0；i＜n；i++）

for（j=0；j＜n；j++）x++;

Θ(n^2)

② y=0;

for（k=0；k< n；k++）

for(i=0；i< k；i++) y ++;

Θ(n^2)

③ z=0;

for（i=1；i＜=n；i* =2）

for(j=1；j＜=n；j+ +) z + +;

Θ($n\log n$)

【例1-13】计算交换两个数据元素的算法的时间复杂度

```
void swap（Elem &x, Elem &y）{
    Elem temp;
    temp=x;
    x=y;
    y=temp;
}
```

此程序段中共执行了3次赋值运算，
所以运行时间为$T(n)=3$，$T(n)=O(1)$

【例1-14】两个N*N矩阵相乘的算法的时间复杂度

```
for（i=0；i< n；i++）{
    for(j=0；j< n；j++) {
        c[i][j]=0；
        for(k=0；k<n；k++)  c[i][j]= a[i][k] * b[k][j]；
    }
}
```

数组元素相乘的重复执行次数是n^3，
所以这段程序的时间复杂度$T(n)=\Theta(n^3)$

1.7 空间复杂度

空间复杂度指当问题的规模以某种单位从1增加到n时，解决这个问题的算法在执行时所占用的存储空间也以某种单位由1增加到f(n)，则称此算法的空间复杂度为f(n)。研究算法的空间复杂度，只需要分析除了算法程序和输入数据之外的额外空间。

- 算法所需要的空间主要由两部分构成

✓ 一是指令空间。就是指用来存储经过编译之后的程序指令所需的空间。

✓ 二是数据空间，就是指用来存储所有常量和所有变量值所需的空间。

任意程序P所面的空间复杂度S(P)可表示为：

$$S(P) = C + SP(L)$$

- ✓ C表示固定空间的需求
- ✓ Sp(L)表示可变空间的需求，包括复合变量所需的空间（这些变量依赖于所解决的具体问题），动态分配的空间（依赖于某一实例的特征L）及递归栈空间的大小（对于每个递归函数而言，该空间主要依赖于局部变量及形式参数所需要的空间，还依赖于递归的深度）。

【例1-16】 计算下列程序的空间复杂度

空间复杂度

① for（i=1；i<＝n；i++） {x++； }

Θ(1)

② float sum(float list[ ] , int n) {

float tempsum=0;

int i;

for (i=0; i < n; i++) tempsum += list[i];

return tempsum;

}

O(1)

③ float rsum(float list[ ], int n) {

if (n) return rsum(list, n-1) + list[n-1];

return 0;

}

O(n)

【例1-17】分析阶乘的递归函数的空间复杂度

```
int fact (int  n){
    if  (n<=1) return 1;
    return  n*fact(n-1);
}
```

C=1，Sp(n)= 4*max{n，1};
S(fact)=O(n)

【例1-18】分析顺序查找的空间复杂度

```
template<class T>
int SequentialSearch(T a[ ], const T &x, int n) {
    int i;
    for (i = 0; i < n && a[i] != x; i++);
    if (i == n) return -1 ;
    return i;
}
```

C=12，Sp(n)=1，S (SequentialSearch)=O(1)

1.8 C++语言基础

1.8.1 面向对象的概念

1.8.2 数据声明和作用域

1.8.3 输入/输出

1.8.4 函数

1.8.5 参数传递

1.8.6 函数重载

1.8.7 动态内存分配

1.8.8 C++的模板(template)