第 1 章　程序设计方法与常用算法

1.1　程序设计与算法

1.1.1　算法

算法是解决问题方法的精确描述，但是并不是所有问题都有算法，有些问题经研究可行，则相应有算法；而有些问题不能说明可行，则表示没有相应算法，但这并不是说问题没有结果。例如，猜想问题，有结果，然而目前还没有算法。上述所谓“可行”，就是算法的研究。

1. 算法的性质

(1) 解题算法是一有穷动作序列。

(2) 动作序列仅有一个初始动作。

(3) 序列中每个动作的后继动作是确定的。

(4) 序列的终止表示问题得到解答或问题没有解答。

2. 待解问题的表述

待解问题表述应精确、简练、清楚，使用形式化模型刻划问题是最恰当的。例如，使用数学模型刻划问题是最简明、严格的，一旦问题形式化了，就可依据相应严格的模型对问题求解。

3. 算法分类

根据待解问题刻划的形式模型和求解要求，算法可以分成两大类：数值的和非数值的。数值的算法是以数学方式表示的问题求数值解的方法，例如代数方程计算、矩阵计算、线性方程组求解、函数方程求解、数值积分、微分方程求解等；非数值的算法通常为求非数值解的方法，例如排序查找、模式匹配、排列模拟、表格处理、文字处理等。将算法分成两大类将有利于算法的设计。

4. 算法设计

算法设计的任务是对各类具体问题设计良好的算法及研究设计算法的规律和方法。常用的算法设计方法有：数值算法(如迭代法、递归法、插值法等)；非数值算法(如分治法、贪婪法、回溯法等)。

5. 算法分析

算法分析的任务是对设计出的每一个具体的算法，利用数学工具，讨论各种复杂度，以探讨某种具体算法适用于哪类问题，或某类问题宜采用哪种算法。算法的复杂度分时间复杂度和空间复杂度。设问题规模以某种单位由 1 增至 n，研究解决该问题的具体算法。在运行算法时所耗费的时间为 $f(n)$(即 n 的函数)，实现算法所占用的空间为 $g(n)$(也为 n 的函数)，则称 $O(f(n))$和 $O(g(n))$为该算法的复杂度。$O(f(n))$与 $O(g(n))$描述了算法的有关性能，所以可用来宏观地评价算法的质量。

1.1.2 数据类型的概念

抽象数据类型是研究解决问题的又一个侧面，也就是算法加工处理的对象的形式。

1. 数据

早期认为数据是物质数量的凭据(或表示)。随着人们处理对象的变化，那些抽象概念(如名称、姓名、颜色等)、语言文字、图象信号以及已被认识的自然现象的表示均为数据。例如，数值计算处理的对象：整数、实数；书报编辑处理的对象：字符、图形；事务处理的对象：报表、文件。所有这些对象都是数据。

2. 数据结构

简单地说，数据结构是数据构造的逻辑表示形式。从学科意义上说，数据结构是指构造数据的方法和在所构造数据上构造相应操作的方法，以及对这些方法的研究。

3. 数据类型

数据类型本质上定义了一组值组成的集合，以及相应运算的集合。或者说，一个数据类型定义了变量或表达式可以取值的范围，以及可以施于它们的运算。

数据类型的意义：

(1) 无论常量、变量、函数或表达式，它们都有且仅有一个类型。

(2) 类型决定了变量、函数和表达式的取值集合以及可施于的运算的集合。

(3) 每个值属于且仅属于一个类型。

(4) 任何常量、变量、函数或表达式所表示的值的类型，都可由其形式或上下文推知。

因此，可以利用类型信息来避免及查明程序中一些无意义的构造并发现错误，而且使程序人员能按照所要解决的问题，通过类型组织数据，而不是根据所使用的计算机。

1.1.3 结构程序设计

1. 程序

程序是对所要解决问题的各个对象和处理规则的描述，或者说是数据结构和算法的描述，因此有人称：数据结构＋算法＝程序。

2. 程序设计

程序设计就是设计、编制和调试程序的过程。

3. 结构程序设计

结构程序设计就是利用逐步精化的方法，按一套程式化的设计准则进行程序的设计。由这种方法产生的程序是结构良好的，所谓“结构良好”是指：

(1) 易于保证和验证其正确性；

(2) 易于阅读、易于理解和易于维护。

按照这种方法或准则设计出的程序称为结构化的程序。

“逐步精化”是对一个复杂问题，不是一步编成一个可执行的程序，而是分步进行。第一步编出的程序抽象级最高；第二步编出的程序抽象级比第一步低；依此类推，第 i 步编出的程序抽象级比第 $i-1$ 步低；直到最后，第 n 步编出的程序即为可执行的程序。

所谓“抽象程序”是指程序所描述的解决问题的处理规则，是由那些“做什么”操作组成，而不涉及这些操作“怎样做”以及解决问题的对象具有什么结构，不涉及构造的每个局部细

节。这一方法的原理就是：对一个问题(或任务)，程序人员应立足于全局，考虑如何解决这一问题的总体关系，而不涉及每一局部细节。在确保全局的正确性之后，再分别对每一局部进行考虑，每个局部又将是一个问题或任务，因而这一方法是自顶向下的，同时也是逐步精化的。采用逐步精化方法的优点是：

(1) 便于构造程序。由这种方法产生的程序，其结构清晰、易读、易写、易理解、易调试、易维护；

(2) 适用于大任务、多人员设计，也便于软件管理。

逐步精化方法有多种具体做法，例如流程图方法、基于过程或函数的方法。

[**例 1.1**] 求两个自然数，其和是 667，最小公倍数与最大公约数之比是 120：1(例如(115,552)、(232,435))。

[**解**] 两个自然数分别为 m 和 $667-m$($2\leqslant m\leqslant 333$)。

处理对象：m(自然数)，l(两数最小公倍数)，g(两数最大公约数)。

处理步骤：对 m 从 2 到 333 检查 l 与 g 的商为 120，且余数为 0 时，打印 m 与 $667-m$。

第 0 层抽象程序：

```
main()
{int m,l,g;
  for (m=2;m<=333;m++){
    l=lcm(m,667-m);
    g=gcd(m,667-m);
    if((l==g*120) && (l%g==0))
        printf("\n%5d%5d",m,667-m);
    }
}
```

第二层考虑函数 gcd(最大公约数)、lcm(最小公倍数)的细化。

最大公约数问题是对参数 a、b，找到一个数 i 能整除 a 与 b，i 就是 gcd 的函数值。

```
int gcd (int a,int b)
{ int i;
  for (i=a;i>=1;i--)
      if(! ((a%i)||(b%i)))
      return(i);
}
```

而最小公倍数的计算是：若干个 b 之和，若能被 a 整除，则该和便是 a、b 的最小公倍数。

```
int lcm(int a,int b)
{int i;
  i=b;
  while (i%a) i+=b;
  return(i);
}
```

[**例 1.2**] 正三角形 ABC 的边 BC、CA、AB 上分别有点 A_1、B_1、C_1，使 $AC_1=2C_1B$，

$BA_1=2A_1C$，$CB_1=2B_1A$。验证由线段 AA_1、BB_1、CC_1 所交成的三角形 $A_2B_2C_2$ 的面积是三角形 ABC 面积的 $\frac{1}{7}$，见图 1.1。

[**解**]

处理对象：已知点 A、B、C；辅助点 A_1、B_1、C_1、A_2、B_2、C_2；直线 AA_1、BB_1、CC_1。

处理步骤：输入 A、B、C 三点坐标值；由点 A、B 计算 C_1 的坐标值；由点 B、C 计算 A_1 的坐标值；由点 C、A 计算 B_1 的坐标值；由点 A、A_1 计算线段方程的系数；由点 B，B_1 计算线段方程的系数；由点 C、C_1 计算线段方程的系数；由线段 AA_1、CC_1 计算点 B_2 的坐标值；由线段 BB_1、CC_1 计算点 A_2 的坐标值；由线段 AA_1、BB_1 计算点 C_2 的坐标值；验证 ABC 面积为 $A_2B_2C_2$ 面积的 7 倍。

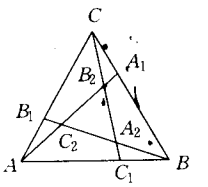

图 1.1　三角形面积比

第 0 层抽象程序为：

```
#include<stdio.h>
#include<math.h>
#define eps 1e-6
typedef struct {float x,y;}point;
typedef struct {float a,c;}line;
main()
{point a,b,c,a1,b1,c1,a2,b2,c2;
  line aa1,bb1,cc1;
  readpoint(&a);readpoint(&b);readpoint(&c);
  intermedium(a,b,&c1);
  intermedium(b,c,&a1);
  intermedium(c,a,&b1);
  straightline(a,a1,&aa1);
  straightline(b,b1,&bb1);
  straightline(c,c1,&cc1);
  intersection(aa1,cc1,&b2);
  intersection(bb1,cc1,&a2);
  intersection(aa1,bb1,&c2);
  if(abs(area(a,b,c)-7*area(a2,b2,c2))<eps)
    printf("\nO. K. This proposition is all right.\n");
  else
    printf("\nI find an error on this proposition.\n");
}
```

第一层的分析与细化：

(1) intermedium 计算两点间某一点的坐标。设点为 r，原两点为 p、q，则 $pr=\frac{2}{3}pq$，见图 1.2。于是这一操作的细化为：

```
void intermedium (point p,point q,point *r)
{r->x=p.x+2*(q.x-p.x)/3;
```

```
r->y=p.y+2*(q.y-p.y)/3;
}
```

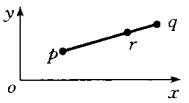

图 1.2　计算线段分点

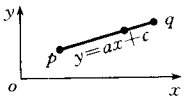

图 1.3　计算直线方程系数

(2) Straightline 为计算线段(pq)的直线方程 $y=ax+c$ 的系数(见图 1.3)。

$$\begin{cases} p_y=ap_x+c \\ q_y=aq_x+c \end{cases}$$

(设 $p_x\neq q_x$,即 pq 与 y 轴不平行)

$$\begin{cases} a=\dfrac{p_y-q_y}{p_x-q_x} \\ c=p_y-ap_x \end{cases}$$

这一操作的细化为:

```
void straightline(point p,point q,line * pq)
{pq->a=(p.y-q.y)/(p.x-q.x);
  pq->c=p.y-pq->a * p.x;
}
```

(3) intersection 为计算两条直线的交点 r,两直线的方程为(见图 1.4):

$$\begin{cases} y=a_1x+c_1 \\ y=a_2x+c_2 \end{cases}$$

其交点为该方程组的解　(设 $a_1\neq a_2$,即两直线不平行):

$$\begin{cases} x=\dfrac{c_2-c_1}{a_1-a_2} \\ y=a_1x+c_1 \end{cases}$$

这一操作的细化为:

```
void intersection (line pq,line mn,point * r)
{r->x=(mn.c-pq.c)/(pq.a-mn.a);
  r->y=pq.a * r->x+pq.c;
}
```

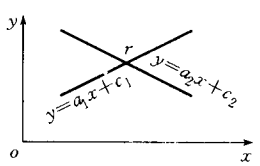

图 1.4　计算两直线交点

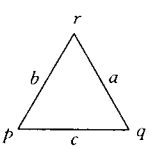

图 1.5　计算三角形面积

(4) area 为三角形的面积。设三角形三边长为 a、b、c(见图 1.5),由海伦公式 $A=\sqrt{l(l-a)(l-b)(l-c)}$ 计算,其中 $l=(a+b+c)/2$。

这一操作的细化为：

```
float area(point p,point q,point r)
{float l,a,b,c;
  a=distance(q,r);
  b=distance(r,p);
  c=distance(p,q);
  l=(a+b+c)/2;
  return(sqrt(l*(l-a)*(l-b)*(l-c)));
}
```

这里还有一个计算两点距离的函数需要细化。

(5) readpoint 为读一个点坐标值的操作。

```
vold readpoint(point * a)
{scanf("%f %f",&(a->x),&(a->y));
}
```

第二层分析与细化：

经第一层细化后，还剩下(4)中的一个操作：计算两点间距离，由公式 $S=\sqrt{(p_x-q_x)^2+(p_y-q_y)^2}$ 十分容易计算，故 distance 细化为：

```
float distance(point p,point q)
{return(sqrt((p.x-q.x)*(p.x-q.x)+(p.y-q.y)*(p.y-q.y)));
}
```

将它们连在一起，完整的程序便是：

```
#include<stdio.h>
#include<math.h>
#define eps 1e-6
typedef struct{float x,y;}point;
typedef struct{float a,c;}line;
void intermedium (point p,point q,point * r)
{r->x=p.x+2*(q.x-p.x)/3;
 r->y=p.y+2*(q.y-p.y)/3;
}
void straightline(point p,point q,line * pq)
{pq->a=(p.y-q.y)/(p.x-q.x);
 pq->c=p.y-pq->a*p.x;
}
void intersection(line pq,line mn,point * r)
{r->x=(mn.c-pq.c)/(pq.a-mn.a);
 r->y=pq.a*r->x+pq.c;
}
float distance(point p,point q)
```

```
{return(sqrt((p.x-q.x)*(p.x-q.x)+(p.y-q.y)*(p.y-q.y)));
}
float area(point p,point q,point r)
float l,a,b,c;
 a=distance(q,r);
 b=distance(r,p);
 c=distance(p,q);
 l=(a+b+c)/2;
 return(sqrt(l*(l-a)*(l-b)*(l-c)));
}
void readpoint(point *a)
{scanf("%f %f",&(a->x),&(a->y));
}
main()
{point a,b,c,a1,b1,c1,a2,b2,c2;
 line aa1,bb1,cc1;
 readpoint(&a);readpoint(&b);readpoint(&c);
 intermedium(a,b,&c1);
 intermedium(b,c,&a1);
 intermedium(c,a,&b1);
 straightline(a,a1,&aa1);
 straightline(b,b1,&bb1);
 straightline(c,c1,&cc1);
 intersection(aa1,cc1,&b2);
 intersection(bb1,cc1,&a2);
 intersection(aa1,bb1,&c2);
 if(abs(area(a,b,c)-7*area(a2,b2,c2))<eps)
   printf("\nO.K. This proposition is all right.\n");
 else
   printf("\nI find an error on this proposition.\n");
}
```

以上两例的设计都是“自顶向下”、“逐步精化”的。算法可以如此，数据结构也可以如此。用这种方法设计的程序，使程序易于阅读，利于保证程序正确，因此具有良好的程序设计风格。

将函数(或过程)调用作为“做什么”操作，而将相应说明看成是“怎样做”操作细节的叙述，这在现代程序设计中是十分重要的方法。

1.2 常用数值计算算法

1.2.1 迭代法

迭代法适用于方程(或方程组)求解，是使用间接方法求方程近似根的一种常用算法。设方程 $f(x)=0$，该方法将方程表示为等价形式：$x=g(x)$，或一般地将 $f(x)$ 拆成两个

函数 f_1、f_2，即 $f(x)=f_1(x)-f_2(x)=0$，因而有 $f_1(x)=f_2(x)$。其中 $f_1(x)$是这样一个函数，对于任意数 c，容易求出 $f_1(x)=c$ 的精确度很高的实根。迭代法求解算法如下：

(1) 首先选一个 x 的近似根 x_0，从 x_0出发，代入右面函数，并解方程 $f_1(x)=f_2(x_0)$得到下一个近似根 x_1。

(2) 将上次近似根 x_1代入右面函数，并解方程 $f_1(x)=f_2(x_1)$，得到又一个近似根 x_2。

(3) 重复(2)的计算，得到一系列近似根 $x_0,x_1,x_2,\cdots,x_i,x_{i+1},\cdots,x_n,\cdots$。

若方程有根，这数列收敛于方程的根。若满足 $|x_n-x_{n-1}|<\varepsilon$，则认为 x_n 是方程的近似根。

[**例 1.3**] 用迭代法求方程 $f(x)=\sqrt{1+2x^2}+\ln x-\ln(1+\sqrt{2+x^2})+3=0$的根，其流程图如图1.6所示。

[**解**] $f_1(x)=\ln x$

$f_2(x)=\ln(1+\sqrt{2+x^2})-\sqrt{1+2x^2}-3$

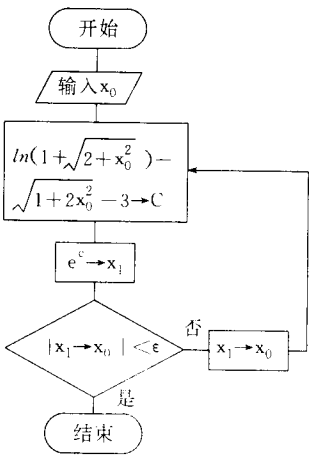

图 1.6 用迭代法求方程的根

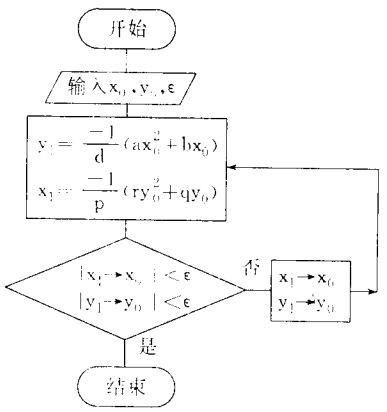

图 1.7 用迭代法求方程组的根

[**例 1.4**] 用迭代法求方程组的根，其流程图如图1.7所示。

$$\begin{cases} ax^2+bx+dy=0 & a、b、d\neq 0 \\ px+qy+ry^2=0 & p、q、r\neq 0 \end{cases}$$

[**解**]

$$\begin{cases} y=\dfrac{-1}{d}(ax^2+bx) \\ x=\dfrac{-1}{p}(ry^2+qy) \end{cases}$$

使用迭代法应注意：

(1) 如果方程无解，数列必不收敛，因而迭代的重复为"死循环"，所以使用迭代算法前应考察方程是否有解。另外，应对重复次数给予一定控制，以防死循环。

(2) 当方程有解时，若迭代公式选择不当，或初始近似根选择不当，也会导致迭代失败。例如，迭代公式中出现除数为0、ln0或 $\mathrm{tg}\,\dfrac{\pi}{2}$等。

(3) 有时为了算法的效率，在一定精确度的条件下，利用误差理论，预先确定迭代重复次数，而不采用$|x_n-x_{n-1}|<\epsilon$的方法。通常采用混合算法，例如计算$\sqrt{x}$，先利用契比雪夫多项式(二次的)计算$\sqrt{x}$的初始根，再用迭代公式$y_{i+1}=\frac{1}{2}\left(y_i+\frac{x}{y_i}\right)$迭代两次，便可达到$10^{-12}$的精确度。

1.2.2 递推法

递推法实际上是需要抽象为一种递推关系，然后按递推关系求解。递推法通常表现为两种方式：一是从简单推到一般；二是将一个复杂问题逐步推到一个已知解的简单问题。这两种方式反映了两种不同的递推方向，前者往往用于计算级数，后者与"回归"配合成为一种特殊的算法——递归法。

由简单推到一般的方法，例如，为得到某项(组)数值，依据前面各项(组)的值推出。通常用于计算级数第n项的值。

[例 1.5] 按递增次序生成集合M的最小的n个数。M定义如下：

(1) $1\in M$

(2) $x\in M\Rightarrow y=2*x+1\in M\wedge y=3x+1\in M$

(3) 再无别的数属于M

[解] 设n个数在数组m中，$2x+1$与$3x+1$均作为一个队列，从两队列中选一排头(数值较小者)送入数组m中，所谓"排头"就是队列中尚未选入m的第一个小的数。这里用$p2$表示$2x+1$这一列的排头，用$p3$表示$3x+1$这一列的排头。

[程序]

```
#include<stdio.h>
#define s 100
main()
{int m[s];
 int n,p2,p3,i;
  m[1]=p2=p3=1;
  scanf("%d",&n);
  for(i=2;i<=n;i++)
   if(2*m[p2]==3*m[p3])
          {m[i]=2*m[p2]+1;p2++;p3++;
          }
      else if(2*m[p2]<3*m[p3])
          {m[i]=2*m[p2]+1;p2++;
          }
      else
          {m[i]=3*m[p3]+1;p3++;
          }
  printf("\n");
  for (i=1;i<=n;i++)
          { printf("%4d",m[i]);
            if (!(i%10))printf("\n");
```

1.2.3 递归法

递归是一种特别有用的工具，不仅在数学中广泛应用，在日常生活中也常常遇到。例如一个画家在作图1.8所示的一幅画便是一种递归的图形。

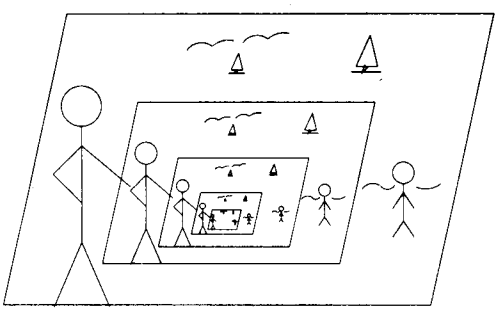

图 1.8 一种递归图形

在数学中几个熟知的递归定义：

(1) $n!=\begin{cases}1 & \text{当 } n=0\text{时} \\ n(n-1)! & \text{当 } n\neq 0\text{时}\end{cases}$

(2) 树结构：

(a) O 是树(空树)；

(b) 若 $t1$ 和 $t2$ 是树，则 $\begin{array}{ccc} & O & \\ t1 & & t2 \end{array}$ 是树。

递归算法包括“递推”和“回归”两部分。

递推 就是为得到问题的解，将它推到比原问题简单的问题的求解。

例如，$n!=f(n)$，为计算 $f(n)$，将它推到 $f(n-1)$，即 $f(n)=n\cdot f(n-1)$，这就是说，为计算 $f(n)$，将问题推到计算 $f(n-1)$，而计算 $f(n-1)$ 比计算 $f(n)$ 简单，因为 $f(n-1)$ 比 $f(n)$ 更接近于已知解 $0!=1$。

使用递推应注意：

(1) 递推应有终止之时，例如，$n!$，当 $n=0$ 时，$0!=1$ 为递推的终止条件。所谓“终止条件”，就是在此条件下问题的解是明确的，缺少终止条件便会使算法失败。

(2) “简单问题”表示离递推终止条件更为接近的问题。简单问题与原问题其解的算法是一致的，其差别主要反映在参数上。例如，$f(n-1)$ 与 $f(n)$ 其参数差1。参数的变化，使问题递推到有明确解的问题。

应很好把握设计与阅读递归算法。

回归 指当“简单问题得到解后，回归到原问题的解上来。例如，当计算完 $(n-1)!$ 后，回归计算 $n\cdot((n-1)!)$，即得到 $n!$ 的值。

使用回归应注意：

(1) 回归到原问题的解时，算法中所涉及的处理对象应是关于当前问题的，即递归算法

所涉及的参数与局部处理对象是有层次的。当解一问题时，有它的一套参数与局部处理对象。当递推进入一“简单问题”时，这套参数与局部对象便隐蔽起来，在解“简单问题”时，又有它自己的一套。但当回归时，原问题的一套参数与局部处理对象又活跃起来了。

(2) 有时回归到原问题已得到问题解，回归并不引起其它动作。

[例 1.6] 计算 $n!$

[解] 根据公式

$$n!=\begin{cases}1 & \text{当 } n=0\\ n\cdot(n-1)! & \text{当 } n\neq 0\end{cases}$$

设计一个函数，参数为 n。

[函数]

```
int factorial(int n)
{if(! n)return(1);
  else
    return(n * factorial(n-1));
}
```

以计算3!为例(图1.9)：

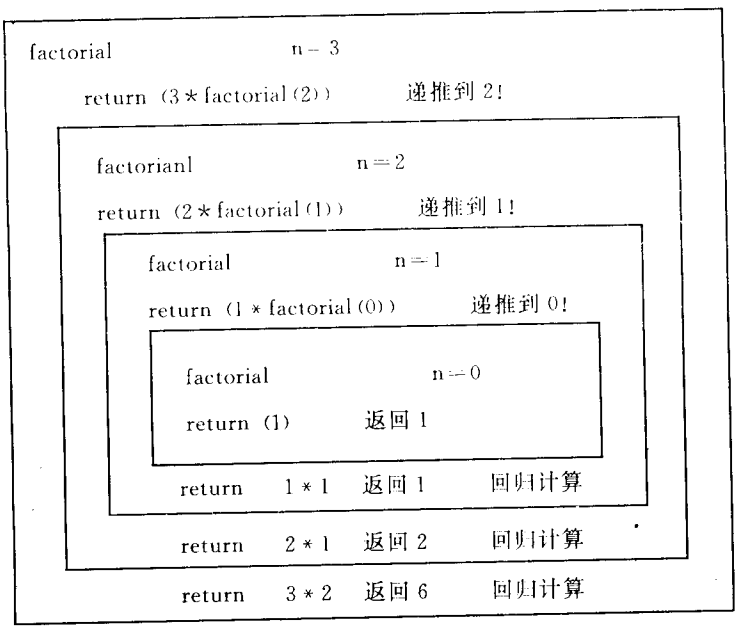

图 1.9

[例 1.7] 斐波那契(Fibonacci)数计算如图1.0所示。

[解] 斐波那契数列的计算公式：

$$\begin{cases}Fib_0=0\\ Fib_1=1\\ Fib_n=Fib_{n-1}+Fib_{n-2} & \text{当 } n>1\text{时}\end{cases}$$

相应程序为：

[函数]

```
int fib(int n)
{if (!n) return(0);
  else
    if (n==1)return(1);
  else return(fib(n-2)+fib(n-1));
  }
```

以 fib (4) 看斐波那契数列的第5项(图1.11)：

数学上早已证明递归算法可以转换成迭代算法，因此选择递归算法还是迭代算法，应视问题而定。例如，$n!$也可以这样设计：

[函数]

```
int factorial(int n)
{int i,f;
  f=1;
  for(i=1;i<=n,i++)
    f*=i;
  retnrn(f);
}
```

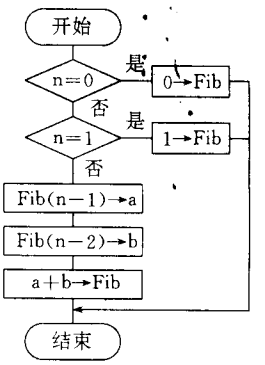

图 1.10 计算斐波那契数

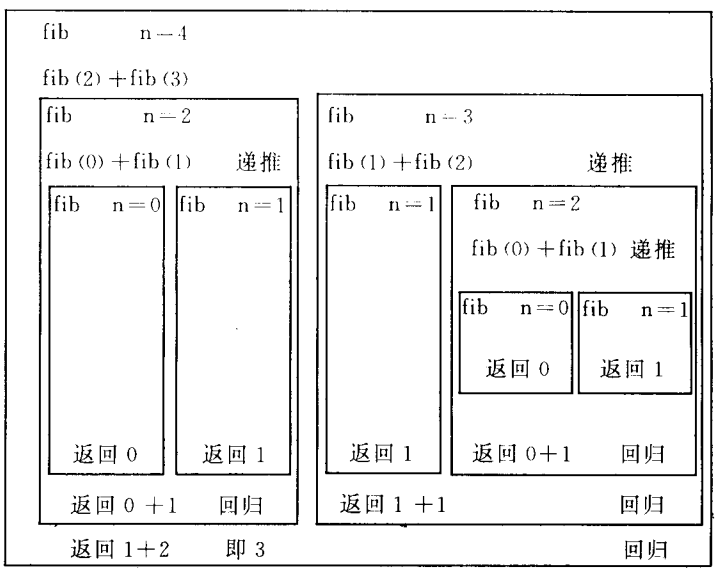

图 1.11

1.2.4 插值法

插值法也称内插法。在实际问题中出现的函数 $f(x)$，往往只知道它在某区间中若干点的函数值，这时作出适当的特定函数，使得在这些点上取已知值，并且在这区间内其它各点上就用这特定函数所取的值作为函数 $f(x)$的近似值，这方法称为“插值法”。如果这特定函

数是多项式，就称之为“插值多项式”或“内插多项式”。

例如，函数 $f(x)$ 可用两种不同插值多项式：

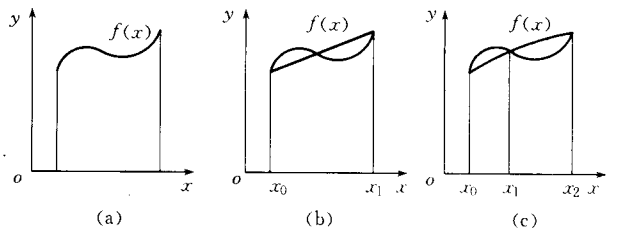

图 1.12　插入法的例

图1.12(b)是“线性插值”，即用直线代替原函数，$y=y_0+\frac{y_1-y_n}{x_1-x_0}(x-x_0)$。用这个多项式来计算 $f(x)$，在$[x_0,x_1]$上的近似值。

图1.12(c)是“二次插值多项式”，即用二次曲线代替原函数，$y=y_0+\frac{y_1-y_0}{x_1-x_0}(x-x_0)+\left(\frac{y_2-y_1}{(x_2-x_1)(x_2-x_0)}-\frac{y_1-y_0}{(x_1-x_0)(x_2-x_0)}\right)(x-x_0)(x-x_1)$。用这个多项式来计算 $f(x)$ 在$[x_0,x_2]$上的近似值。

插值多项式随着插值节点的增加，多项式表示的曲线与函数 $f(x)$ 表示的曲线越接近，但其复杂度也越高。为此，常采用分段插值。

设单变量函数 $f(x)$，已知 n 个节点 $x_0,x_1,\cdots,x_n$，及其对应的函数值 $y_k=f(x_k)(k=0,1,2,\cdots,n)$，要求插值多项式过各节点的常见插值方法有：

(1) 若节点不等距，常使用差商插值多项式。

(2) 若节点为等距，常使用差分公式，具体有牛顿第一、第二插值公式、斯特林插值公式、贝塞尔插值公式等。

(3) 以上两类插值方法的计算很有规律，但不直观，应用上有时不方便，于是还有拉格朗日插值多项式。

(4) 为保证插值曲线光滑，有样条内插公式。

[例 1.8]　差商插值多项式。

设函数 $f(x)$ 的 $n+1$ 个节点 $x_0,x_1,x_2,\cdots,x_n$，及其对应的函数值 $y_0,y_1,y_2,\cdots,y_n$（即 $y_k=f(x_k)$ $(k=0,1,2,\cdots,n)$。对于插值区间$[\min\limits_{0\leqslant i\leqslant n}(x_i),\max\limits_{0\leqslant i\leqslant n}(x_i)]$上任一点 x，其函数值 $f(x)$ 可用 $p_n(x)$ 近似地表示。

$$p_n(x)=y_0+y_{0,1}(x-x_0)+y_{0,1,2}(x-x_0)(x-x_1)+\cdots$$
$$+y_{0,1,\cdots,n}(x-x_0)(x-x_1)\cdots(x-x_{n-1})$$

其中 $y_{0,1},y_{0,1,2},\cdots,y_{0,1,2,\cdots,n}$ 分别为$\{y_0,y_1\cdots y_n\}$的一阶差商、二阶差商、…、n 阶差商。其计算公式为：

$$y_{i,i+1}=\frac{y_{i+1}-y_i}{x_{i+1}-x_i}\qquad(i=0,1,2,\cdots,n-1)$$

$$y_{i,i+1,i+2}=\frac{y_{i+1,i+2}-y_{i,i+1}}{x_{i+2}-x_i}\qquad(i=0,1,2,\cdots,n-2)$$

$$y_{0,1,2,\cdots,n}=\frac{y_{1,2,\cdots,n}-y_{0,1,\cdots,n-1}}{x_n-x_0}$$

或者用差商表表示：

x_k	y_k	一阶差商	二阶差商	三阶差商		n 阶差商
x_0	y_0					
x_1	y_1	$y_{0,1}$				
x_2	y_2	$y_{1,2}$	$y_{0,1,2}$			
x_3	y_3	$y_{2,3}$	$y_{1,2,3}$	$y_{0,1,2,3}$		
⋮	⋮	⋮				
x_{n-1}	y_{n-1}	$y_{n-2,n-1}$	$y_{n-3,n-2,n-1}$	$y_{n-4,n-3,n-2,n-1}$		
x_n	y_n	$y_{n-1,n}$	$y_{n-2,n-1,n}$	$y_{n-3,n-2,n-1,n}$		$y_{0,1,2,\cdots,n}$

［**解**］ 差商 $y_{0,1}$，$y_{0,1,2}$，$y_{0,1,2,3}$，$\cdots$，$y_{0,1,2,\cdots}$，分别用 z_1，z_2，z_3，$\cdots$，z_n 表示。先读入节点值及相应函数值，形成插值多项式，然后对读入在插值区间内一点 x 的值，计算其近似的函数值，其流程图如图1.13所示。

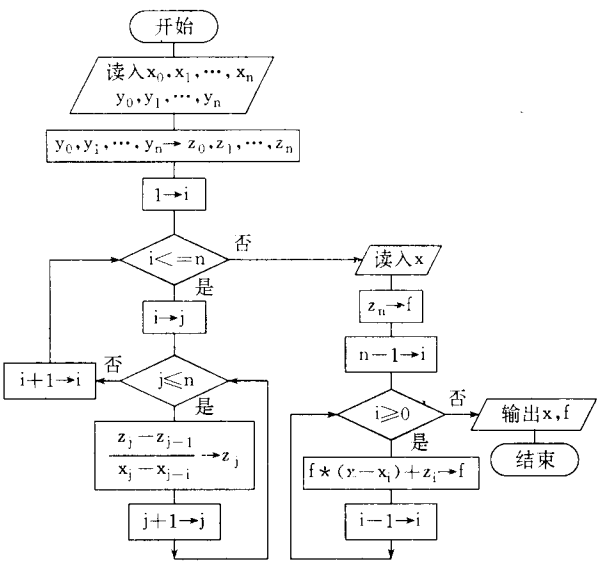

图 1.13　差商插值多项式

［**例 1.9**］ 等距节点插值公式(差分公式)。

已知函数 $f(x)$ 在等距节点 $x_k=x_6+kh(k=0,\pm1,\pm2,\cdots,\pm n)$ 的值为 $y_k=f(x_k)(k=0,\pm1,\pm2,\cdots,\pm n)$，

其向前差分为：

$\Delta y_i=y_{i+1}-y_i(i=0,\pm1,\pm2,\cdots,\pm(n-1))$（一阶差分）

$\Delta^2 y_i=\Delta y_{i+1}-\Delta y_i(i=0,\pm1,\pm2,\cdots,\pm(n-2))$（二阶差分）

$\cdots$

$\Delta^k y_i=\Delta^{k-1}y_{i+1}-\Delta^{k-1}y_i(i=0,\pm1,\pm2,\cdots,\pm(n-k))$（$k$ 阶差分）

显然 $\Delta^k y_i=\sum_{j=0}^{k}(-1)^j\binom{k}{j}y_{i+k-j}$

其中$\binom{k}{j}$为二项系数。

而向后差分为：

$\nabla y_i=y_i-y_{i-1}=\Delta y_{i-1}$

$\nabla^k y_i=\nabla(\nabla^{k-1}y_i)=\Delta^k y_{i-k}$

差 分 表

⋮	⋮	⋮				
x_{-3}	y_{-3}					
		Δy_{-3}				
x_{-2}	y_{-2}		$\Delta^2 y_{-3}$			
		Δy_{-2}			$\Delta^4 y_{-4}$	
x_{-1}	y_{-1}		$\Delta^2 y_{-2}$	$\Delta^3 y_{-3}$		
		Δy_{-1}			$\Delta^4 y_{-3}$	
x_0	y_0		$\Delta^2 y_{-1}$	$\Delta^3 y_{-2}$		
		Δy_0			$\Delta^4 y_{-2}$	
x_1	y_1		$\Delta^2 y_0$	$\Delta^3 y_{-1}$		
		Δy_1			$\Delta^4 y_{-1}$	
x_2	y_2		$\Delta^2 y_1$	$\Delta^3 y_0$		
		Δy_2			$\Delta^4 y_0$	
x_3	y_3	⋮	$\Delta^2 y_2$	$\Delta^3 y_1$		
⋮	⋮				$\Delta^4 y_1$	

（1）牛顿第一插值公式（向前插值）。节点：$x_k=x_0+kh(k=0,1,2,\cdots,n)$；插值点：$x=x_0+uh$，$u=\frac{x-x_0}{h}(0<u<1)$。牛顿第一插值公式：

$$N_1(x)=y_0+\frac{u}{1!}\Delta y_0+\frac{u(u-1)}{2!}\Delta^2 y_0+\cdots+\frac{u(u-1)\cdots(u-(n-1))}{n!}\Delta^n y_0$$

余项公式：

$$R_n(x)=\frac{u(u-1)\cdots(u-n)}{(n+1)!}h^{n+1}f^{n+1}(\xi)\quad(x_0<\xi<x_0+nh)$$

图1.14中用 z_i 表示 $\Delta^i y_0(i=1,2,\cdots,n)$，$z_0=y_0$。

（2）牛顿第二插值公式（向后插值）。节点：$x_k=x_0-kh(k=0,1,2,\cdots,n)$；插值点：$x=x_0-uh$，$u=\frac{x_0-x}{h}$。牛顿第二插值公式：

$$N_2(x)=y_0+u\Delta y_{-1}+\frac{u(u+1)}{2!}\Delta^2 y_{-2}+\cdots+\frac{u(u+1)\cdots(u+n-1)}{n!}\Delta^n y_{-n}$$

余项公式：

$$R_n(x)=\frac{u(u+1)\cdots(u+n)}{(n+1)!}h^{n+1}f^{(n+1)}(\xi)$$

流程图如图1.14所示，该程序留给读者作为习题

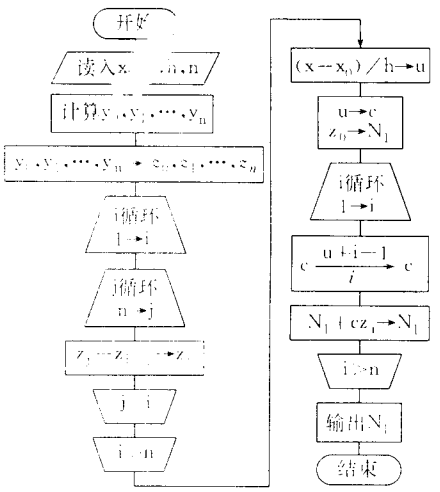

图 1.14 牛顿第一插值公式

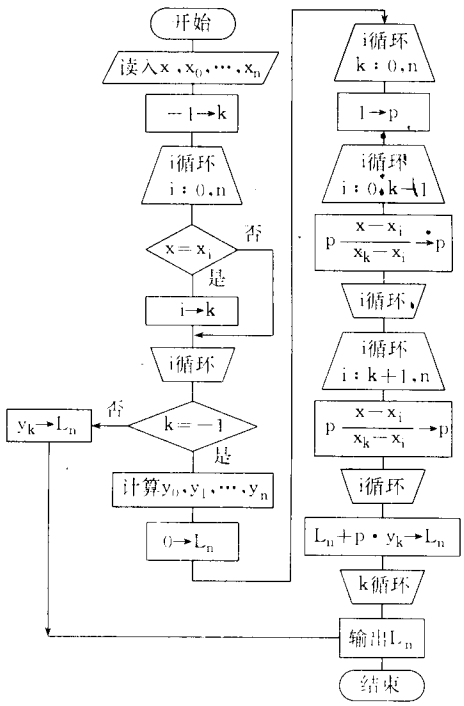

图 1.15 拉格朗日插值多项式

[例 1.10] 拉格朗日插值多项式。

单变量函数 $f(x)$的 $n+1$个节点 $x_0,x_1,x_2,\cdots,x_n$，及其对应函数值 $y_k=f(x_k)$ $(k=0,2,\cdots,n)$。对于插值区间内任一点 x，拉格朗日多项式

$$L_n(x)=\sum_{k=0}^{n}\prod_{\substack{i=0\\i\neq k}}^{n}\frac{(x-x_i)}{(x_k-x_i)}y_k$$

可近似地计算函数值 $f(x)$，流程图见图1.15。

1.2.5 差分法

联系未知函数的差分和自变量的方程称为差分方程。求微分方程的数值解时，常将其中微分用相应的差分代替，所得方程就是差分方程。通过差分方程求微分方程近似解的方法称为“差分法”。

在插值法中已引进了差商的概念，设 $y=f(x)$，差商 $y_{i,i+1}=\frac{y_{i+1}-y_i}{x_{i+1}-x_i}$ 为一阶差商；$y_{i,i+1,i+2}=\frac{y_{i+1,i+2}-y_{i,i+1}}{x_{i+2},-x_i}$ 为二阶差商；而 n 阶差商为 $y_{0,1,2,\cdots,n}=\frac{y_{1,2,\cdots,n}-y_{0,1,\cdots,n-1}}{x_n-x_0}$。多元函数 $fg(x,y)$ 的一阶差商为 $\frac{f(x+h,y)-f(x,y)}{h}$ 或 $\frac{f(x,y+h)-f(x,y)}{h}$；二阶差商为

$\frac{f(x+h,y)-2f(x,y)+f(x-h,y)}{h^2}$ 或 $\frac{f(x,y+h)-2f(x,y)+f(x,y-h)}{h^2}$，或 $\frac{f(x,y+h)-2f(x,y)+f(x-h,y)}{h^2}$等。

而差分则是 $y_{i+1}-y_i$、$f(x+h,y)-f(x,y)$、$f(x,y+h)-f(x,y)$等。

用差分代替微分，或用差商代替导数或偏导数，这在积分的数值计算或微分方程数值解法中常常使用。

[**例 1.11**] 差分法解常微分方程边值问题。

设二阶线性常微分方程边值问题：

$$\begin{cases}\dfrac{\partial^2 y}{\partial x^2}+p(x)\dfrac{\partial y}{\partial x}+q(x)y=f(x)\\ x=a, \quad y=a\\ x=b, \quad y=\beta\end{cases}$$

[**解**] 将区间$[a,b]$分成 n 等份，步长 $h=\dfrac{b-a}{n}$，分点 $x_0=a, x_1=a+h, \cdots, x_k=a+kh, \cdots, x_n=b$ 称为节点。用差商代替微商，边值问题化为如下差分方程组求解：

$$\begin{cases}\dfrac{y_{k+1}-2y_k+y_{k-1}}{h^2}+p_k\dfrac{y_{k+1}-y_{k-1}}{2k}+q_k y_k=f_k \qquad (k=1,2,\cdots,n-1)\\ y_0=\alpha\\ y_n=\beta\end{cases}$$

其中，$y_k=y(x_k)$，$p_k=p(x_k)$，$q_k=q(x_k)$，$f_k=f(x_k)$。

该方程组有 $n+1$ 个未知量 $y_0, y_1, y_2, \cdots, y_n$，方程个数也是 $n+1$ 个。整理合并后为：

$$\begin{cases}y_0=a\\ a_k y_{k-1}+b_k y_k+c_k y_{k+1}=d_k \quad (k=1,2,\cdots,n-1)\\ y_n=\beta\end{cases}$$

其中，

$$\begin{cases}a_k=l-\dfrac{h}{2}p_k\\ b_k=-2+h^2 q_k\\ c_k=l+\dfrac{h}{2}p_k\\ d_k=h^2 f_k\end{cases} \qquad (k=1,2,\cdots,n-1)$$

这组线性方程组可用消元法、迭代法或追赶法求解，从略。

1.2.6 常用算法举例

1. 实根的近似计算

[**例 1.12**] 秦九韶法。

[**解**] 设方程 $f(x)=a_0x^n+a_1x^{n-1}+a_2x^{n-2}+\cdots+a_{n-1}x+a_n$ 至少有一个正的实根。

算法步骤：

(1) 首先找到 r_0，使 $f(r_0)\cdot f(r_0+1)<0$