

递归算法设计及其非递归化研究

汤亚玲

(安徽工业大学 计算机学院, 安徽 马鞍山 243002)

摘要:递归作为一种算法设计思想在求解实际问题 and 程序设计中广泛应用,采用递归设计的算法具有思路清晰、易于描述复杂问题等优点。文中对递归算法的理论依据、设计思想、应用、递归的内部执行过程做了较为全面的探讨,并以火车进站问题为例,重点分析了如何根据问题的递归表达式扩充为递归算法。同时,对递归的非递归化作了较为深入的分析和探讨,并给出了实例源程序。理论分析和实践证明,在具体应用问题中,通过寻找问题对应的递归表达式,可以容易和准确地设计出求解的递归算法,提高算法设计效率。

关键词:递归;算法设计;递归表达式;栈

中图分类号:TP301.6

文献标识码:A

文章编号:1673-629X(2009)11-0085-04

Research on Recursive Algorithm Design and Its Non-recursive Form

TANG Ya-ling

(School of Computer, Anhui University of Technology, Maanshan 243002, China)

Abstract: Recursion as a kind of algorithm idea is widely used in solving reality questions and programing. Algorithm which is decribed by recursive mode has the advantages of clarity and decribing question easily. This paper does research on foundation, idea, application and executing procedure of recursion throughly, and as a example of question of the train pulling in it analyzes how to expand the recursion function to recursion algorithm in emphasis. Finally it does study furtherly on how to transform recursion algorithm into non-recursive algorithm and offers source code of examples. It proves that we can easily design the algorithm for the problem on finding out the recursion expression functions and improve designing efficiency by theory and pratice.

Key words: recursion; algorithm design; recursion expression function; stack

0 引言

在运用计算机求解实际问题的过程中,针对具体的问题往往会选择不同的、最适合解决问题的算法去完成。这其中,有一类问题规模较大,但可以把问题化解成求解若干个规模较小和原问题类似的子问题,并且可以一直这样做下去,直到规模最小的问题可以求解,这种情形下往往会选择递归思想来分析和求解该类问题。程序设计中,递归表现为过程或函数在其定义中直接或间接调用自身的一种方法。递归的优点在于仅需少量的代码就可描述出解题过程所需要的多次重复计算,大大地减少了程序设计的代码量,其能力在于用有限的语句来定义对象的无限集合,用递归思想描述的算法往往也十分简洁易懂。

1 递归的理论根源

人们对递归的研究源于数论,递归思想始于可计算性理论,在高级数理逻辑里有较为详细的定义和证明^[1-3]。早在1936年,Church提出一般初等函数都可以定义为递归函数,Turing也提出论点:一般所说的可计算函数,是可用Turing机所计算的函数,而Kleene则证明了,一般递归函数也就是可用Turing机可计算的函数,这就是后来著名的Church-Turing论点^[1,2]。实际上在设计算法时,可以对数论中对递归的论证加以简化,仅仅考虑简单的基于自然数集的递归问题(广义的递归论基于一般的序数,包括无穷序数)。一般地,简单递归函数可以定义如下:

设有函数 $f(e_1, e_2)$ 及 $g(e_1)$,构造递归函数 h 的形式定义:

$$h(u, 0) = C \quad (1)$$

$$h(u, S_x) = f(x, h(u, g(x))) \quad (2)$$

或者定义一个带参数 v 的递归函数 h :

$$h(v, u, 0) = C \quad (3)$$

$$h(v, u, S_x) = f(v, x, h(v, u, g(v, x))) \quad (4)$$

收稿日期:2009-03-28;修回日期:2009-06-29

基金项目:安徽省自然科学基金项目(2006KJ062B);安徽省优秀青年人才基金项目(2009SQRZ076)

作者简介:汤亚玲(1974-),男,硕士,副教授,研究方向为智能化信息处理、数据挖掘及网络数据库系统。

其中,在式(1)、(2)、(3)、(4)中的 e_1, e_2, u, v, S_x 定义域是集合论中的一般序数, C 代表确定的常数值。事实上,在数论中研究的递归理论还牵涉到很多复杂的数学定义和定理的证明,在此,不做深入讨论。而在计算机的算法设计中,需要的是一种简单易行的理论指导,可以把递归函数的定义简化为:

$$\text{Func}(n_0) = C_0 \quad (5)$$

$$\text{Func}(m) = \text{Func}(n_1) + \text{Func}(n_2) + \text{Func}(n_3) + \dots + \text{Func}(n_k) + C(m) \quad (6)$$

式(5)、(6)给出了在算法设计时对递归算法简化表达形式,是递归算法的形式表达函数。实际应用中,函数 Func 可能带两个或者两个以上的参数,其中: $m > n_i, i \geq 1, C(m)$ 表示常量,代表一个具体的操作,“+”表示功能的组合,该定义也是设计递归算法的重要理论依据。

算法设计中,递归的实现思想可分为基于归纳法的递归和基于分治法的递归,而基于分治法的递归实际上也是一种特殊归纳方式的递归,只是在计算机学科里将分治方法单独划分出来。同时,数学上重要的证明方法——数学归纳法创立的一个重要基础是基于自然数集的计算理论:从 $n = 0, 1, \dots, k-1$ 问题可解直至 $n = k$ 问题可解。从数学归纳法的一般求解过程来看,它是一个从 $0, 1, \dots, k-1$ 至 k 的计算过程,如果把这个过程倒过来看,就不难发现数学归纳法计算的逆过程和递归的求解过程有着惊人的相似,事实上,递归恰恰可以看成是反向运用了数学归纳法来解决实际问题。如要求解规模为 N 的问题,先计算规模为 $N-1$ 的子问题,直至某个规模为 $i (i \geq 0)$ 的问题可解。

2 递归算法设计

计算机科学中,递归是指函数、过程、子程序在运行过程中直接或间接调用自身而产生的重入现象,是一个过程或函数在其定义或说明中又直接或间接调用自身。它通常把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解。很多问题,经典的如 $N!$ 、Hanio 问题、八皇后问题、二叉树的遍历问题、 N 个数的全排列等等都可以使用递归写出简洁、思路清晰的算法程序代码。

下面介绍一种借用函数分解法来设计递归算法,说明递归形式函数的形式定义在实际问题中的应用。这里举一个较为复杂的例子,采用(5)、(6)两式去设计递归算法。

采用递归形式函数去解决实际的步骤可由以下几步去完成:

(1)分析问题,写出对应问题的递归形式函数。

(2)找出递归形式函数的收敛条件(或者称为递归结束条件)。

(3)将递归形式函数转化成对应的算法(或者对应的程序代码)。

看一个例子,火车进站问题:编号为 $A_1, A_2, A_3, \dots, A_n (A_1 < A_2 < A_3 < \dots < A_n)$ 的 N 列火车顺序开进一个栈式结构的站台。问 N 列火车的出站有多少种可能?

问题分析:进站过程中,火车的动作有两种,进站和出站。设某一时刻站台入口处有 i 列火车,站台内有 j 列火车,则下一动作有两种可能:(1)站台入口处第 1 列火车入站台;(2)站台内最顶上的火车出站台。

对于该问题,借助递归函数的定义方式,可以这样设计对应火车进站问题的递归函数:

$$F(0, j) = 1, F(0, 0) = 1, j > 0 \quad (7)$$

$$F(i, 0) = F(i-1, 1), i > 0 \quad (8)$$

$$F(i, j) = F(i-1, j+1) + F(i, j-1), i > 0, j > 0 \quad (9)$$

其中 $F(i, j)$ 表示站台外有 i 列火车,站台内有 j 列火车, $F(0, j)$ 表示站台外没有火车,站台内有 j 列火车, $F(i, 0)$ 表示站台内没有火车,有 i 列火车在站台外;(7)式对应于递归形式函数定义中式(5),而(8)和(9)式对应于一般递归形式函数定义中式(6);并且,(9)式表明,当站外有 i 列火车,站内有 j 列火车时,它的下一步动作可能有两种,一是站外一列火车进站,或者站内一列火车出站,也就是结果 $F(i-1, j+1)$ 和 $F(i, j-1)$,即(9)式所表达的。

根据式(7)、(8)、(9),很容易写出求火车进站问题对应的算法(C++语言描述):

```
int Train_into_Platform(i, j)
{
    if(i == 0) return 1; // 火车没有,或者全部在站台内
    else if(j == 0) return Train_into_Platform(i-1, 1);
    else return Train_into_Platform(i-1, j+1) + Train_into_Platform(i, j-1);
}
```

函数 $\text{Train_into_Platform}(N, 0)$ 的计算结果就是 N 列火车进站的可能数,并且结果与理论计算公式 $C_{2n}^n / n + 1$ 是一致的^[4,5]。如果适当扩充和改进函数 $\text{Train_into_Platform}(i, j)$ 的代码,很容易将 N 列火车进站具体排列情形求出来,扩充的原则是当满足(7)式的情形时,打印全部出站的火车序列,否则讨论二种情形的递归调用:

(1)当站台内没有火车时($F(i, 0)$),递归调用函数 $F(i-1, 1)$,它代表将火车队列最前面一列入站台,

然后递归调用自身。

(2) 当站外有火车时,并且站台内也有火车时,分两种情形讨论:

I) 站台外最前面的火车入站,递归调用自身,即 $F(i-1, j+1)$ 。

II) 站台内最顶上(栈式结构站台)火车出站,递归调用自身,即 $F(i, j-1)$ 。

具体编程时,定义辅助数据结构对象栈 S 和对象队列 In_Quene、Out_Quene,栈 S 模拟站台,In_Quene 模拟入站台火车队列,Out_Quene 模拟出站台火车队列,下面是主要的 C++ 源代码:

```
void Train_into_Platform(Stack s, Queue In_Quene, Queue Out_Quene)
{
    Stack s_backup; Queue In_Quene_Backup, Out_Quene_Backup;
    //定义局部备份对象 s_backup, In_Quene_Backup, Out_Quene_Backup
    long int x; //代表某一列火车编号
    if(s.empty_stack() && In_Quene.empty_queue()) //火车全部出站,打印出站序列
    {
        Out_Quene.Print(); // (7) 式中的  $F(0,0)=1$  情形
        return;
    }
    if(!s.empty_stack() && In_Quene.empty_queue()) //站台非空且入队列为空
    {
        while(!s.empty_stack()) // (7) 式中的  $F(0,j)=1$  情形,反复出站,直至站台内空
        {
            x = s.pop(); //入队列为空
            Out_Quene.add_queue(x); //将 x 加入出队列中
        }
        Out_Quene.Print(); //打印出站序列
        return;
    }
    if(s.empty_stack() && !In_Quene.empty_queue()) //情形 (1)
    {
        x = In_Quene.out_queue(); //取出入队列队头元素
        s.push(x); //将取出的元素压入栈中
        Train_into_Platform(s, In_Quene, Out_Quene);
    }
    if(!s.empty_stack() && !In_Quene.empty_queue())
    {
        s.backup_to(s_backup);
        In_Quene.backup_to(In_Quene_Backup); Out_Quene.backup_to(Out_Quene_Backup);
        //备份 s, In_Quene, Out_Quene
        x = In_Quene.out_queue(); //取出入队列队头元素至 x
        s.push(x); //将取出的元素 x 压入栈中
        Train_into_Platform(s, In_Quene, Out_Quene); // (2) 中的情形 a
        x = s_backup.pop();
    }
}
```

```
Out_Quene_Backup.add_queue(x);
```

```
Train_into_Platform(s_backup, In_Quene_Backup, Out_Quene_Backup); // (2) 中的情形 b
```

```
}
```

此时函数 Train_into_Platform 将打印出火车出站的所有可能排列。

可以看出,设计递归算法时,一般可以先找出该算法所对应的递归形式函数,再根据递归形式函数编写它所对应的源代码程序。此方法具有适应面广,思路清晰,编程效率高的优点。

3 递归的执行过程分析

递归的执行依赖系统堆栈的支持,系统栈保证了递归的正确执行,递归的执行过程主要分为两步,逐层深入递归调用和层层向上递归返回,在递归调用时需要做的工作有:

(1) 进行断点保存,局部变量、形式参数保存。

(2) 然后控制流程的转向递归调研的入口。

在本次递归调用结束后向上层调用返回时需要做的工作有:

(1) 保存本次调用的函数结果,恢复调用函数时的局部变量和形式参数。

(2) 根据递归调用时的断点地址将控制流程转到调用函数中递归调用的下一行代码处继续执行。

清华大学的数据结构教材(Pascal 语言版)对此过程有较为详细的阐述^[6],此处不再重复。分析递归的执行过程对理解递归和将非递归程序的非递归化具有很重要的指导意义。

4 非递归化

递归算法在执行时,存在多次进栈和出栈,流程的跳转和返回,甚至会出现重复计算,如 Hanio 问题、Fibonacci 数列中会出现性质相同或相似的一个子问题被多次计算,却无法利用已计算结果,因此,在对算法执行效率要求较高的情况下,需要编写非递归算法;或者在已有递归算法的基础上写出其对应的非递归算法。非递归化最重要的基础是理解递归的执行过程,在此基础之上,对于一般的递归算法,有两种方法可以对它进行非递归化。

一种是机械转换算法,它的根据是系统栈支持下的递归执行,通过自定义栈模拟递归的执行过程。可以写出一般递归算法的机械转换方法如下:

(1) 初始化自定义栈。

(2) 给每个递归函数的入口分别标以不同的语句标号。

(3)并在每个递归返回处设立一个语句标号。

(4)将递归函数中的每次递归调用用以下操作代替:

a)保存断点:将递归调用中的局部变量、形式参数与返回地址入栈。

b)准备数据:为被调用递归子程序准备所需的参数,计算实参值,赋给对应形参。

c)GoTo 递归函数的开始处。

d)如果函数有返回值,则用从栈顶取回该函数值的代码来替代该次递归调用。

(5)用以下操作替代返回语句:

a)如果栈不空,则依次执行如下操作,否则结束该递归函数,返回。

b)回传数据:若有变参或是函数,增设一个回传变量,将其值保存到回传变量中。

c)恢复断点:从栈顶取出返回地址和保存的局部变量的值,并退栈,按断点返回地址返回。

转换规则中陈述了对递归调用和返回语句的处理,其它语句保持不变。

上述转换规则可将任意的递归算法转换为等价的非递归算法^[6,7],但得到的算法不符合结构化编程的原则,可以分析代码得到其等价的流程图,进行优化,用循环语句替代 GoTo 语句,得到结构清晰的非递归程序。

第二种方法适合递归不是很复杂的情形,通过详细分析递归的执行过程,直接写出它等价的非递归算法。通过递归执行过程分析,可以看出,递归调用时,实际上是再次转向算法的开始处执行,实际上它是一个循环,当满足一定的条件时循环结束,分析算法中的每次递归调用及参数值的变化,同时规划出递归算法中所包含的循环部分,用循环结构去代替递归调用,构造非递归算法。

限于篇幅,给出一个简单的例子,二叉树的中序遍历,用递归描述其算法非常简洁,如下所示:

```
void InOrder(Binary_node * T)
{if(! T)return;
 In_Order(T->Lchild);
 Visit(T);
 In_Order(T->Rchild);
}
```

算法的思路清晰,在此不再赘述,通过机械转换算法得到的非递归算法如下:

```
void InOrder(Binary_node * T)
{if(! T) return;
 Stack * S= Init_Stack();
 L0://算法入口处
```

```
if(! T)if(! Empty_Stack(s))
    |pop(S,T,L);//返回地址出栈
    goto L;//转向地址 L 执行
    |
    else return;//T 为空,且栈空,所有树的结点访问完毕,
函数返回
if(T)
    |Push(S,T,L1);//局部量 T,返回地址 L1 入栈保存
    T=T->Lchild; //处理 T 的左子树
    goto L0;//转向函数入口
    L1://递归左子树返回地址
    Visit(T);
    Push(S,T,L2) //局部量 T,返回地址 L2 入栈保存
    T=T->Rchild; //处理 T 的右子树
    Goto L0;//转向函数入口
    |
    L2://原递归函数尾部返回出口
    pop(S,T,L);//返回地址出栈
    goto L;//转向返回地址 L 执行
    |
```

上面的算法使用了 goto 语句,程序的流程不太清晰,可以画出上面程序等价的流程图后进行优化并消除 goto 语句,也可以用第二种方法直接写出此算法的非递归程序。

分析二叉树的递归算法的执行过程可知,在二叉树非空的前提下,执行算法,首先访问根的左子树,再访问根,最后访问根的右子树;访问根的左子树时,先要访问左子树根的左子树,再访问左子树的根,其次再访问左子树根的右子树……,如此递归下去,一直到树的最左下结点被访问(向左下搜索时,将当前结点压入系统栈中保存,以便向上回退时调出),然后访问最左下结点的父结点,通过弹栈获得最左下结点的父结点,然后处理该父结点的右子树,依次类推,循环直到整颗树访问完毕。

根据上述对递归执行过程的分析,不难写出它对应的非递归算法,如下所示:

```
void InOrder(Binary_node * T)
{if(! T) return;
 Stack * S= Init_Stack();
 while(T || ! Empty_Stack(S))
 { while(T)//当指针 T 非空时入栈
    |Push(S,T)
    T=T->Lchild;
    |
    Pop(S,T);
    Visit(T);
    T=T->Rchild;
```

容,则粒计算研究恐难有科学性可言。而要把握这大致的确定性,不能不研究四个要素本身的问题。

粒计算的四个要素分别从不同的视角对粒计算进行研究,涵盖了粒计算学科的基本属性,它们既自成体系,又相互联系、相互依赖、相互制约,还功能互补、相互协同、缺一不可,形成了粒计算特有的一种“四位一体”结构模型,共同促进粒计算学科的良好发展。

5 结束语

文中讨论了要为粒计算建立一个自己的、很好的和统一的模型,必须要研究粒计算学科四个基本要素。但研究只是初步的,其进一步的完善还需要广大粒计算研究者的共同努力。下一步研究的重点将是以此四个基本要素为基础,构建粒计算学科的四面体结构模型,从而为建立粒计算统一的独特的模型框架奠定坚实的基础。

参考文献:

- [1] Zadeh L A. Some reflections on soft computing, granular computing and their roles in the conception, design and utilization of information/intelligent systems//Soft Computing [M]. Berlin: Springer - Verlag, 1998: 23 - 25.
- [2] Yao Y Y. A Unified Framework of Granular Computing [M]//in: Pedrycz W, Skowron A, Kreinovich V. Handbook of

Granular Computing. [s. l.]: Wiley, 2008: 401 - 410.

- [3] 姚一豫. 粒计算的艺术. 粒计算: 过去、现在和展望[M]. 北京: 科学出版社, 2007: 1 - 20.
- [4] Yao Y Y. Three perspectives of granular computing[J]. Journal of Nanchang Institute of Technology, 2006, 25(2): 16 - 21.
- [5] Yao Y Y. Perspectives of granular computing[C]//The 2005 IEEE International Conference on Granular Computing. Beijing, China: [s. n.], 2005: 85 - 90.
- [6] Yao Y Y. Granular Computing: Past, Present and Future [C]//2008 IEEE International Conference on Granular Computing. Hangzhou, China: [s. n.], 2008: 80 - 85.
- [7] 李 鸿. 粒集及其描述[J]. 宿州学院学报, 2006, 21(1): 90 - 93.
- [8] 李 鸿. 粒集理论: 粒计算的新模型[J]. 重庆邮电大学学报, 2007, 19(4): 397 - 404.
- [9] 李 鸿. 粒集的扩展与提升[J]. 计算机科学, 2008, 35(8A): 234 - 236.
- [10] 李 鸿. 粒集的提升及其描述[J]. 宿州学院学报, 2008, 23(5): 96 - 98.
- [11] Li H. Concept Granular System and Granular Concept Lattice [C]//In: Wang Guojun. Proceedings of The 9th International Conference for Young Computer Scientists (ICYCS2008, ZhangJiaJie, Hunan, China). Los Alamitos, California, USA: IEEE Computer Society, 2008: 1860 - 1865.

(上接第 88 页)

```

} //end while(T|| Empty_Stack(s))
//end InOrder()

```

5 结束语

递归在算法和程序设计中有中广泛的运用,递归方式描述的算法具有代码简洁,思路清晰的优点,是设计算法的强有力工具。对于一般满足递归特征的问题,可以根据通过先写出问题自身对应的递归形式函数((5)、(6)两式),然后可较容易扩充为它相应的递归算法程序,此方法也是设计一般递归算法重要技巧。一般地,递归程序的执行效率低于非递归程序^[4,7,8],递归的效率可采用画递归树的方法来分析^[8,9];在一些效率要求高的情况下,需要撰写非递归算法,可根据算法的复杂程度选择转换方法。但同时要认识到,非递归算法的效率虽高,但往往难于编写,容易出错;理论上,递归算法都可以转化为非递归算法,但存在一些算法很难非递归化,如复杂的间接递归,因此,要根据具体情形选择递归还是非递归。

参考文献:

- [1] Milner R. Functions as processes[M]//Mathematical Structures in Computer Science. University of Edinburgh. Berlin: Springer, 1992: 167 - 180.
- [2] Sangiorgi D. An investigation into functions as processes[C]//In: Proc. Math. Foundations of Program Semantics' 93. University of Edinburgh. Berlin: Springer, 1993: 143 - 159.
- [3] Hamilton. 数学家的逻辑[M]. 北京: 科学出版社, 1989.
- [4] 汤亚玲, 崔志明. 基于 C++ 递归方法在应用问题中的设计与实现[J]. 微机发展(现更名: 计算机技术与发展), 2003, 13(8): 90 - 92.
- [5] 李红卫, 徐亚平. 出栈序列的研究[J]. 计算机技术与发展, 2007, 17(10): 127 - 129.
- [6] 严蔚敏. 数据结构(Pascal 语言版)[M]. 北京: 清华大学出版社, 1993.
- [7] 秦 锋. 数据结构[M]. 合肥: 中国科学技术大学出版社, 2008.
- [8] Comen T H, Leiserson C E. Introduction to Algorithms[M]. 北京: 机械工业出版社, 2008.
- [9] 冯群强, 苏 淳, 胡治水. 均匀递归树的分支结构[J]. 中国科学(A 辑), 2005, 35(5): 569 - 584.