递归算法的非递归化研究

Recursive Algorithm Transform into Non-recursive Algorithm

孟 林 李 忠

(宜宾师范高等专科学校计科系 四川宜宾644007)

Abstract The method of transforming a recursive algorithm into non-recusive algorithm is discussed in this paper, which is based on the typical questions: Fibonacci series, Ackermann-functuin, Hanoi tower problem and Traversing binary tree etc. in the same time, the characteristics and executing efficiency of recursive algorithm and non-recursive algothm are also discussed.

Keywords Recursion, Algorithm, Non-recursive algorithm

1 引言

在工程实际中,有许多概念是用递归来定义的,数学中的许多函数也用递归来表达。一个递归算法的执行过程类似于多个函数的嵌套调用,只是主调函数和被调函数是同一个函数而已,在执行过程中,信息的传递和控制的转移必须通过栈来实现,这就导致空间耗费大,执行效率较低,尤其是当递归深度较深时,不但耗费的空间大而且执行的效率也相当低,这是递归算法的一个致命的缺点。同时,有些语言不允许递归调用,此时也需要将递归算法非递归化。如何将一个递归算法转化为一个非递归算法呢?这值得我们去探索。

2 递归算法的非递归化及执行过程比较

下面以 Fibonacci 数列、Ackermann 函数、Hanoi 塔、遍历二叉树等典型问题为例阐述转化方法与过程,其算法均以类 C 语言形式进行描述。

问题1(Fibonacci 数列) 有雌雄一对兔子,假定过两个月便可繁殖雌雄各一的一对小兔。问过 n 个月后共有多少对兔子?

求解:设满 n 个月时兔子对数为 F_n,其中当月新生兔数目设为 N_n 对。第 n-1个月留下的兔子数目设为 O_n 对。则有:$F_n=N_n+O_n$。其中 $O_n=F_{n-1}$,$N_n=O_{n-1}=F_{n-2}$。即第 n-2个月的所有兔子到第 n 个月都有繁殖能力了。从而,有 F_n 满足的递归关系式:

$F_n=F_{n-1}+F_{n-2}$,$F_1=F_2=1$

由此可得:

算法一 求解 Fibonacci 数列的递归算法

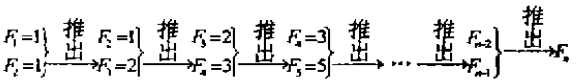
```
int Fibonacci1(int n){
    if(n==1||n==2)z=1;
    else z=Fibonacci1(n-1)+Fibonacci1(n-2);
    return z;
}
```

对于数值计算中的具有显示表达式的这类递归关系问题,可以转化为递推算法求解,求解 Fibonacci 序列的非递归算法见算法二。

算法二 求解 Fibonacci 序列的非递归算法

```
int Fibonacci2(int n){
    f1=1;f2=1;
    for(i=3;i<=n;i++){t=f1+f2;f1=f2;f2=t;}
    return f2
}
```

算法二的执行过程为:



算法一的执行过程远比算法二的执行过程复杂得多,实际上,算法一可表述为如图1所示的二叉树形式(其中,F_i 表示 Fibonacci1(i),i=1,2,3,…,n)。因此,为了计算 F_n,必须作深度为 N-1的递归调用,直到 $F_1=1$ 为止。

由上述可知,对于有些问题,既可以归纳为递归算法,又可以归纳为递推算法,但递推算法和递归算法的实现方法是大不一样的。递推是从初始条件出发,逐次推出所需求的结果;而递归则是从算法本身到达递归世界。

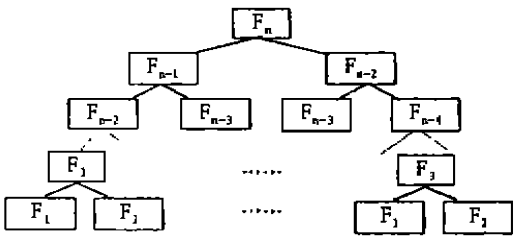


图1

问题2(Ackermann 函数):Ackermann 函数的定

义如下

$$akm(m,n)=\begin{cases} n+1 & m=0 \\ akm(m-1,1) & m\neq 0,n=0 \\ akm(m-1,akm(m,n-1)) & m\neq 0,n\neq 0 \end{cases}$$

按上述定义可写出计算 Ackermann 函数的递归算法，见算法三。

算法三 计算 Ackermann 函数的递归算法

```
int Ackermann1(int m,int n){
   if(m==0) return n+1;
   else if(n==0) return(Ackermann(m-1,1));
   else return(Ackermann(m-1,Ackermann(m,n-1)));
}
```

也可以借助于栈，而不用递归来求解该问题，见算法四。

算法四 计算 Ackermann 函数的非递归算法

计算 Ackermann 函数的非递归算法需要用到以下类型的栈结构。

```
typedef struct node{/*定义栈中数据元素结构*/
   int mval;
   int nval;
}node;
typedef struct stack{/*定义附设顺序栈结构*/
   node Arr[MAX];
   int top;/*栈顶指针*/
}stack;
int akm(int m,int n){
int akmval;
stack S;
S.top=0;
S.Arr[S.top].mval=m;S.Arr[S.top].nval=n;
do{while(S.Arr[S.top].mval){
       while(S.Arr[S.top].nval){
       S.top++;
       S.Arr[S.top].mval=S.Arr[S.top-1].mval;
       S.Arr[S.top].nval=S.Arr[S.top-1].nval-1;
   }
   S.Arr[S.top].mval--;S.Arr[S.top].nval=1;
   }
   if(S.top>0){
       S.top--;
       S.Arr[S.top].mval--;
       S.Arr[S.top].nval=S.Arr[S.top+1].nval+1;
   }
}while(S.top!=0||S.Arr[S.top].mval!=0);
akmval=S.Arr[S.top].nval+1;S.top--;
return akmval;
}
```

问题3(n 阶 Hanoi 塔) 假设有三个分别命名为 X，Y，Z 的塔座，在塔座 X 上有 n 个直径大小各不相同、依小到大编号为1，2，…，n 的圆盘(图2)。现要求将塔座 X 上的 n 个圆盘移至塔座 Z 上并仍按同样的顺序叠放，圆盘移动时必须遵循下列规则：1)每次只能移动一个圆盘；2)圆盘可以插在 X，Y，Z 中的任一塔座上；3)任何时刻都不能将一个较大的圆盘压在较小的圆盘之上，如何实现移动圆盘的操作呢？

求解：解该问题的最容易想到的方法是利用递归，其步骤如下：

当 n=1时，只要将编号为1的圆盘从塔座 X 直接移至塔座 Z 上即可；当 n>1时，需利用塔座 Y 作辅助塔座，先将塔座 X 上的编号为1，2，…，n−1的 n−1个圆盘借助于塔座 Z 移至塔座 Y 上，再将塔座 X 上的编号为 n 的圆盘移至塔座 Z 上，最后将塔座 Y 上的 n−1 个圆盘借助于塔座 X 移至塔座 Z 上即可。按此思想可得解此问题的如算法五所示的递归算法。

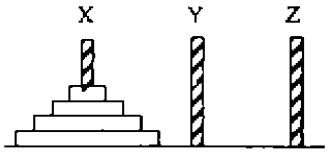


图2 4阶 Hanoi 塔的初始

算法五 求解 Hanoi 塔问题的递归算法：

```
void Hanoi(int n,char x,char y,char z){
    if (n==1)move(x,1,z);/*将编号为1的圆盘从 x 移到
      z*/
    else{
        Hanoi(n-1,x,z,y);/*将 x 上编号为1至 n-1的圆
        盘移到 y,z 作辅助塔*/
        move(x,n,z);/*将编号为 n 的圆盘从 x 移到 z*/
        Hanoi(n-1,y,x,z);/*将 y 上编号为1至 n-1的圆
        盘移到 z,x 作辅助塔*/
    }
}
```

对于这类非数值计算的递归问题也可以借助于栈，仿照递归算法执行过程中递归工作栈的状态变化状况，而直接写出其非递归算法，见算法六。

解 Hanoi 塔问题的非递归算法需要用到以下类型的栈结构

```
typedef struct node{/*定义栈中数据元素结构*/
    int nval;
    char chx,
    char chy;
    char chz;
}node;
typedef struct stack{/*定义附设顺序栈结构*/
    node arr[MAX];
    int top;
}stack;
```

算法六 Hanoi 塔问题的非递归算法：

```
void hanoi(int n,char x,char y,char z){
  stack S;char temp;
  static int flag[MAX];
  S.top=0;
  S.arr[S.top].nval=n;S.arr[S.top].chx=x;
  S.arr[S.top].chy=y;S.arr[S.top].chz=z;
  do{
     while(S.arr[S.top].nval!=1){
        S.top++;
        S.arr[S.top].nval=S.arr[S.top-1].nval-1;
        S.arr[S.top].chx=S.arr[S.top-1].chx;
        S.arr[S.top].chy=S.arr[S.top-1].chz;
        S.arr[S.top].chz=S.arr[S.top-1].chy;
     }
     flag[S.top]=S.arr[S.top].nval;
     printf("\n%d%c->%c",S.arr[S.top].nval,S.arr[S.
       top].chx,S.arr[S.top].chz);
     if(S.top>0){
        printf("\n%d:%c->%c",S.arr[S.top].nval+1,S.
          arr[S.top].chx,S.arr[S.top].chy);
        printf("\n%d:%c->%c",S.arr[S.top].nval,S.arr
```

```
            [S.top].chz,S.arr[S.top].chy);
        if(S.arr[S.top].nval=flag[S.top])
          while(S.arr[S.top].nval=flag[S.top]){flag[S.
          top]=0;S.top--;}
        else{flag[S.top]=0;S.top--;}
        flag[S.top]=S.arr[S.top].nval;
        if(S.top>0)
        printf("\n%d:%c->%c,S.arr[S.top-1].nval,S.arr
        [S.top].chx,S.arr[S.top].chy);
        temp=S.arr[S.top].chx;S.arr[S.top].chx=S.arr
          [S.top].chz;
        S.arr[S.top].chz=S.arr[S.top].chy;S.arr[S.
          top].chy=temp;
      }
}while(S.top>0);
return;
}
```

问题4(遍历二叉树):给定一棵二叉树,如何沿某条搜索路径周游二叉树,对树中每个结点访问一次且只访问一次?

求解:通常遍历一棵二叉树有三种方法,即前序遍历、中序遍历、后序遍历。而所谓的后序遍历,是指:若二叉树非空,则依次进行以下操作:(1)后序遍历左子树;(2)后序遍历右子树;(3)访问根结点。

按上述描述很容易写出后序遍历的递归算法,见算法七,其中,二叉树采用如下形式的存储结构:

```
typedef struct BiTNode{
   TelemType data;/*结点数据元素*/
   Struct BiTNode *lchild,*rchild;/*左右孩子指针*/
}*BiTree;
```

算法七 后序遍历二叉树的递归算法:

```
postorder1(BiTree t){/*后序遍历二叉树t*/
   if(t){   /*二叉树非空*/
       postorder1(t->lchild);/*后序遍历左子树*/
       postorder1(t->rchild);/*后序遍历右子树*/
       visite(t);/*访问根结点*/
   }
}
```

仿照算法六,也可以直接写出解该问题的借助于栈的非递归算法(在此从略)。对于该问题,还可以采用以下存储结构:

```
typedef struct BiTNode{
   TelemType data;/*结点数据元素*/
   int mark;/*标志域,该域取值为0、1、2,初始时均为0*/
   Struct BiTNode *parent,*lchild,*rchild/*双亲,左孩
     子、右孩子指针*/
}*BiTree;
```

而设计出不借助于栈的非递归算法,见算法八。

算法八 后序遍历二叉树的非递归算法。

```
postorder2(BiTree t){/*后序遍历二叉树t*/
  p=t;
  while(p)
    switch(p->mark){
    case0:/*p所指向的结点未曾访问*/
          p->mark=1;/*标识p所指向的结点为从双亲
          结点而来*/
          if(p->lchild)p=p->lchild;/*左子树非空,访问
          左子树*/
          break;
    case 1./*p所指向的结点左子树已访问结束,右子树未
          曾访问*/
          p->mark=2;/*标识p所指向的结点为从左子
          树遍历结束而来*/
          if(p->lchild)p=p->lchild;/*左子树非空,访问
          左子树*/
          break;
    case 2./*p所指向的结点左、右子树均已访问结束,结
          点p未曾访问*/
          p->mark=0;/*标识p所指向的结点为从右子
          树遍历结束而来*/
          visite(p);/*访问根结点*/
          p=p->parent;/*回溯到双亲结点;
          break;
    }
}
```

限于篇幅,不再叙述解问题2、问题3及问题4的算法的执行过程。

3 递归算法及非递归算法的比较

由上述的例子可以看出,递归算法的最大优点是:结构简练、清晰,可读性强,而且它的正确性容易得到证明,特别是在许多复杂的问题中,很难找到从初始条件推出所需结果的全过程,此时,设计递归算法要比设计非递归算法容易得多,且利用允许递归调用的语言进行程序设计时,递归算法很容易转化为语言程序,能给用户编制程序和调试程序带来很大方便。但另一方面,一个递归算法的执行过程类似于多个函数的嵌套调用,只是主调函数和被调函数是同一个函数而已,在执行过程中,信息的传递和控制的转移必须通过栈来实现,这就导致空间耗费大,执行效率较低,尤其是当递归深度较深时,不但耗费的空间大而且执行的效率也相当低,有时甚至超出人们的可接受限度,这是递归算法的一个致命的缺点。而设计得好的非递归算法的执行效率就高得多(尽管上述的算法四、算法六也用到了栈,但却省去了递归算法执行过程中的控制转移以及控制转移过程中的参数传递等过程)。同时,有些语言不允许递归调用,此时必须将递归算法进行非递归化,将其转化为非递归算法后问题才能得到解决。

结束语 递归问题在工程实际中是极为常见的,在解决这类问题时可以用递归算法,也可以用非递归算法,二者各有所长。在具体问题中,应根据具体情况来选择,一般情况下,非递归算法的执行效率要高于递归算法的执行效率(尤其当问题的规模较大时),而递归算法的可读性比非递归算法的可读性好,算法的正确性更容易得到证明,尽管递归算法要用到栈,但在使用允许递归调用的语言进行程序设计时,栈的管理是由系统自己管理的,而不需要用户去管理递归工作栈。

参考文献

1 徐士良编著.计算机常用算法(第二版).清华大学出版社,1995

2 卢彦芝编著.C语言常用算法与子程序.清华大学出版社,1991

3 周培德编著.算法设计与分析.机械工业出版社,1992

4 严蔚敏等编著.数据结构(C语言版).清华大学出版社,1997