递归函数的高效实现方法

赵建华

递归函数的适用范围和优缺点

- 分治法
  – 把一个比较大的问题分解为若干个比较小的问题，分别求解这些比较小的问题，再综合得到原问题的解。
- 如果比较小的问题和原问题具有同样的性质，那么适用递归接法
  – 要求最终能够把问题分解为能够直接解决的简单问题
- 优点
  – 简洁
  – 能够帮助思考
  – 和问题的结构有对应关系
- 缺点
  – 效率低下

递归的定义

- 递归
  - 若一个对象部分地包含它自己，或用它自己给自己定义，则称这个对象是递归的；
  - 若一个过程直接地或间接地调用自己，则称这个过程是递归的过程。
- 以下三种情况常常用到递归方法。
  - 定义是递归的
  - 数据结构是递归的
  - 问题的解法是递归的

定义是递归的（1）

例如，阶乘函数的定义

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

求解阶乘函数的递归算法

```
long Factorial(long n) {
    if (n == 0) return 1;     //可直接解答的情况
    else return n*Factorial(n-1);  //递归调用
}
```

可以简化成较小的问题

- 最大公约数

gcd(x,y) = (x==0 || y==0)? x+y : (

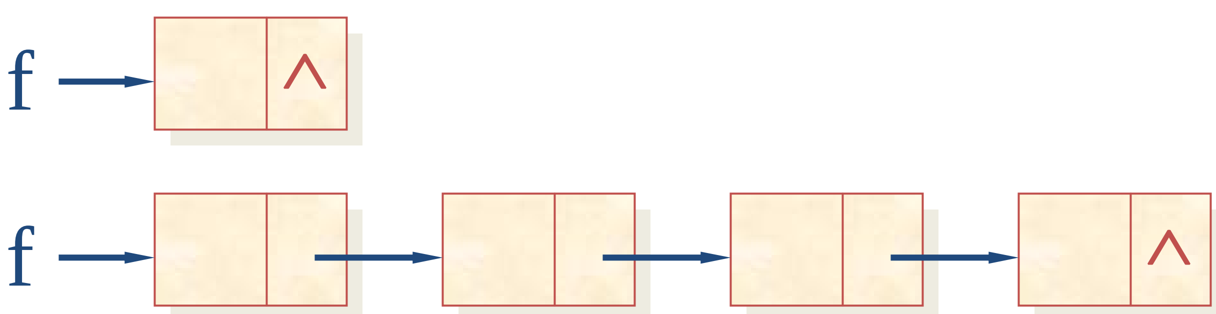
(x>y ? gcd(x-y,y) : gcd(x,y-z)

)

```
int gcd(int x; int y)
{
        if(x==0 || y==0) return false;
        if(x>y)
                return gcd(x-y,y);
        else
                return gcd(x,y-x);
}
```

数据结构是递归的（1）

- 数据结构由更小的、相似的数据结构组成。
  - 对这个数据结构的处理，分解成对较小部分的递归处理
- 例如，单链表结构
  - struct Node {int Data; struct Node *link;}
  - 一个指针f指向一个单链表，iff
    - f == NULL 或者
    - f != NULL 且f->link指向一个单链表



数据结构是递归的（2）

- 单链表f长度的定义
  - length(f) = (f==NULL)?0:1+length(f->link)
- p指向f单链表中某个结点
  - isNode(f,p) = (f==NULL)? false : (f==p) || isNode(f->link,p)

```
int Length(Node *f)
{
        if(f==NULL) return 0;
        return 1 + Length(f->link);
}
```

```
bool isNode(Node *f, Node *p)
{
        if(f==NULL) return false;

        if(f==p) return true;
        return isNode(f->link, p);
}
```

问题的解法是递归的

- 汉诺塔(Tower of Hanoi)问题
- 解法：假设要把n个盘子从X移动到Y，允许使用Z作为过渡
  - 如果 n = 1，将盘子直接从X移Y。
  - 如果n>1，分成三步

```
#include <iostream.h>
void Hanoi (int n, char A, char B, char C) {
//解决汉诺塔问题的算法
    if (n == 1) cout << " move " << A << " to  " << C << endl;
    else {
            Hanoi(n-1, A, C, B);
            cout << " move " << A << " to " << C << endl;
            Hanoi(n-1, B, A, C);
         }
}
```

递归函数的要求

- 不能无限制地调用本身
  - 必须有一个出口，化简为非递归情况，直接处理。
  - 必须保证分解之后的子问题要“小于”原来的问题，且最终能把一个问题分解为可直接处理的基本情况。

```
Procedure <name> ( <parameter list> )
{
        if ( < initial condition> )    //递归结束条件
        {
                直接处理并返回；
        }
        else   //递归
        {
                递归调用<name>过程，
                并进行某些综合处理，然后返回；
        }
}
```

递归的高效迭代实现

- 根据不同的情况, 可以采取不同的方法
  - 尾递归的迭代实现
  - 其它简单情况的迭代实现
  - 使用栈的迭代实现
- 其它情况
  - 大参数的处理
  - 重复计算的处理

尾递归

- 一个函数只在代码的最后调用自己，并且返回递归调用得到的值。
- 相当于在处理完成当前参数的情况之后，用新的参数再次运行自己的代码。

- 转换方法：
  - 使用变量来存放实在参数；
  - 迭代处理
    - 循环体中的代码就是原来的不包含递归调用的代码；
    - 在原来递归调用的地方，把实在参数赋予参数变量，并continue

尾递归的例子（1）

```
bool isNode(Node *f, Node *p)
{
        if(f==NULL) return false;

        if(f==p) return true;
        return isNode(f->link, p);
}

bool isNode(Node *f, Node *p)
{       Node *arg1, *arg2;
        arg1 = f; arg2 = p;
        for(;;)
        {
                if(arg1==NULL) return false;
                if(arg1==p) return true;
                arg1 = arg1->link; arg2 = arg2;
                continue;
        }
}
```

尾递归的例子（2）

```
int gcd(int x; int y)
{
        if(x==0 || y==0) return false;
        if(x>y)
                return gcd(x-y,y);
        else
                return gcd(x,y-x);
}
```

```
int gcd(int x; int y)
{
        if(x==0 || y==0) return false;
        if(x>y)
                { x = x-y;  continue;}
        else
                {y = y-x; continue;}
}
```

其它简单类型的递归

- 类似于尾递归
- 返回值会参与某些运算，但是这些运算满足交换率
- 可以增加一个变量，存放结果值

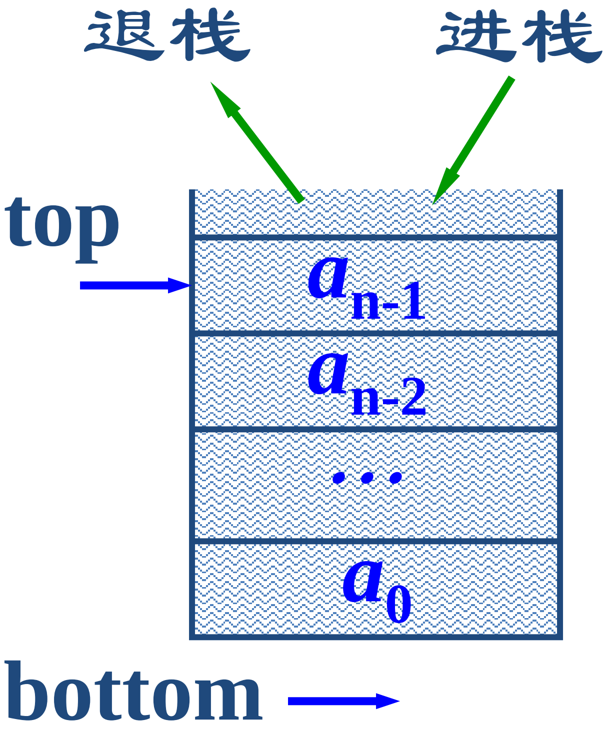
```
int Length(Node *f)
{
        if(f==NULL) return 0;
        return 1 + Length(f->link);
}

int Length(Node *f)
{
        int result = 0;

        for(;;)
        {
                if(f==NULL) return result;
                result ++;
                f = f->link;
        }
}
```

栈 (Stack)

- 只允许在一端插入（Push）和删除（Pop）的序列
  - 允许插入和删除的一端称为栈顶(top),
  - 另一端称为栈底(bottom)
- 特点：后进先出 (LIFO)
- 函数调用：
  - 后调用者先退出



一个简单的实现

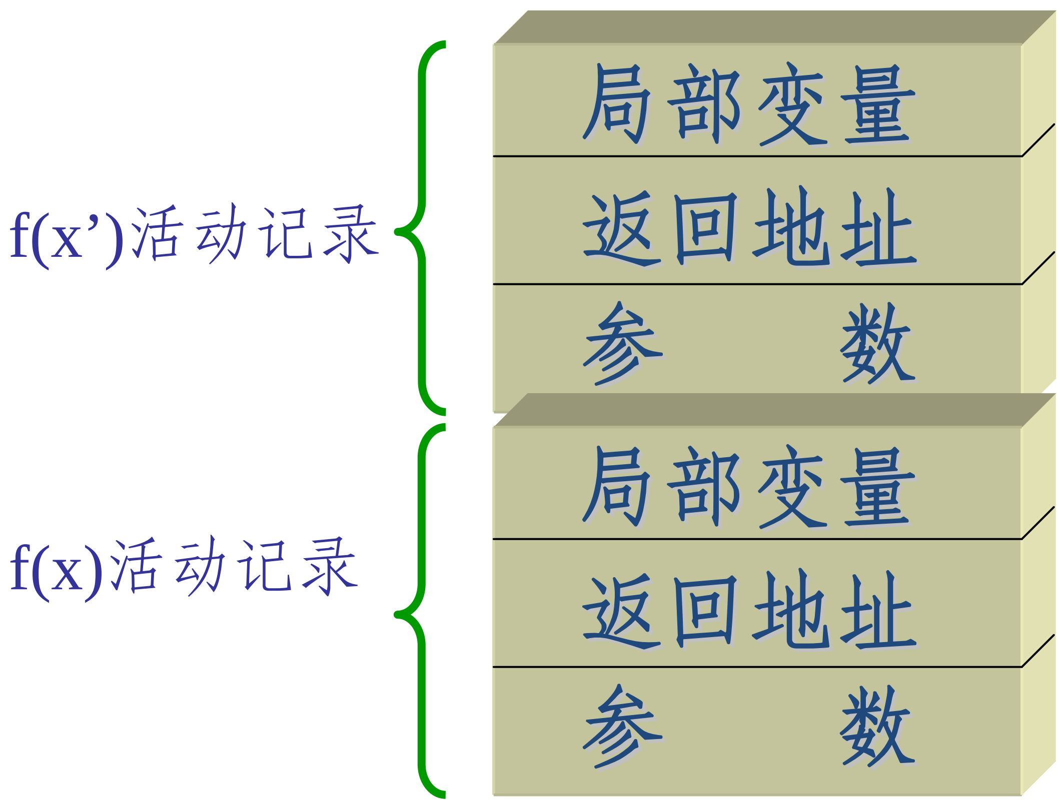
- 使用数组

```
struct Stack {T ele[M]; int cnt;};

void Pop(Stack* s, T& t)
    {cnt--; t = s->ele[cnt];}
void Push(Stack* s, T t)
    {s->ele[cnt] = t; cnt++;}
isEmpty()
    {return cnt == 0;}
```

函数调用的工作栈

- 每一次函数调用
  - 为函数的参数、局部变量等在栈中分配（Push）存储空间（称为活动记录）。
  - 跳转到函数开始处执行
- 当函数返回
  - 传递返回值，收回存储空间；
  - 跳转回调用者，从调用处继续执行
- 每次递归调用都是同样处理



使用栈的迭代实现方法

- 模拟递归栈，栈中保存
  - 局部变量
  - 参数
  - 返回值存放地址；
  - 当前处理进度（相当于返回后将执行地址）
- 使用入栈/出栈来模拟递归调用/返回
  - 调用：设置栈顶的进度标记；设置参数、返回值地址入栈
  - 返回：拷贝返回值，出栈
  - 其他计算过程：根据当前进度标记执行；

使用栈的Fib的迭代实现（1）

```
long Fib(long n) {
    long t1, t2;
    if (n <= 1) return n;
    else
    {
        t1=Fib(n-1);
        t2=Fib(n-2);
        return t1+t2;
    }
}
```

栈元素的类型

```
struct Node{
        long n;             //参数
        long t1;            //局部变量，存第一次调用Fib的返回值
        long t2;            //局部变量，存第二次调用Fib的返回值
        long* ret;          //指向存放返回值的地址
        int curPos;         //0表示刚开始执行；
                            //1表示调用了第一次Fib,
                            //2表示第二次调用Fib
}
```

使用栈的Fib的迭代实现（2）

```
long Fib(int n)
{   long ret;              //返回值
    stack<Node>  s; Node *w;
    Node record = {n,0,0,&ret,0};   //第一次递归调用，压栈
    s.push(bottom);
    while(!s.IsEmpty())
    {
        Node* curRec = s.GetTopPointer();
        //根据curRec->IP的值执行相应的代码
        … … … …
    }
    return ret;
}
```

```
switch(curRec->IP)
{case 0:
        if(curRec->n <= 1)                    // if (n <= 1) return n;
        {  *curRec->ret = curRec->n;  s.Pop();}
        else
        { curRec->IP=1;               //记住已经执行到第一次调用之前；
          s.Push({curRec->n-1,0,0,&curRec->t1,0});} //递归调用t1=Fib(n-1);
        break;
case 1:  //第一次调用返回
        curRec->IP=2;
        s.Push({curRec->n-2,0,0,&curRec->t2, 0}); //递归调用t2=Fib(n-2);
        break;
case 2:  //第二次递归调用返回，将t1+t2后返回
        *curRec->ret = curRec->t1 + curRec->t2; //return t1+t2;
        s.Pop(); //返回
}
```

```
long Fib(long n) {        long t1, t2;
        if (n <= 1) return n;
        else
        {       t1=Fib(n-1);
                t2=Fib(n-2);
                return t1+t2;
        }
}
```

重复计算的处理

- 在写递归函数时，我们通常不考虑效率
  - 同一个问题可能计算多次
  - 参数可能是一个体积很大的值
- 重复计算的解决方法：
  - 如果递归函数的参数范围有限，解决方法是设法把每个参数的值保存起来
  - 只计算一次, 计算完毕后把结果保存起来
  - 第二次计算通过查询完成

解决模式

设有递归函数

T f(S)

{

    … my code...

    return t ;

}

- 设参数的取值范围是集合S，值域是T
  - 使用数据结构Data来记录S→T的映射关系
  - 初始化时，S中所有的元素映射成特殊值nil

T f(S s)

{

在Data中查询s对应的值，如果不是nil，返回相应的值；

… my code …

将Data中s对应的值设置为t；

return t;

}

例子

```
long Fib(long n) {
    if (n <= 1) return n;
    else return Fib(n-1)+Fib(n-2);
}
```

```
long  results[M];
long  Fib(int n);
main()
{           int n;
            for(int i = 0; i<M; i++)
                        results[i] = -1;
            cin >> n;
            Fib(n);
}
```

```
long Fib(int n)
{
            if(results[n]>=0)
                        return results[n];
            if (n <= 1) {results[n] = n; return n;     }
            else
            {
                        results[n] = Fib(n-1)+Fib(n-2);
                        return Fib(n-1)+Fib(n-2);
            }
}
```

大参数的处理

- 在定义递归函数时，有时为了定义方便，参数可能是一个占用较大内存的值，比如一个列表、矩阵等。
- 此时可以适用全局变量+修改/复原操作来提高效率

幂集的递归解法

```
PowerSet(SetOfInt s, ListOfInt prefix)
//SetOfInt, ListOfInt都会占用很多内存，且拷贝时很低效。
{
        if(s为空)
        {
                输出prefix;
                return;
        }
        令x为s的第一个元素;
        ListOfInts newPrefix = prefix + x;
        SetOfInt s' = s – {x}
        PowerSet(s', newPrefix);
        PowerSet(s', prefix);
}
```

解决方法：使用全局数组S和pre保存参数，同时：

用L表示S的起始点，

preCnt表示prefix的长度；

l和cnt可以作为参数传递。

幂集的递归解法（2）

```
int S[MAX], pre[MAX];

PowerSet(int L, int preCnt)
{
    if(L>=N)
    {
        输出pre中0到preCnt-1的元素;
        return;
    }
    pre[preCnt] = S[L];
    PowerSet(L+1, preCnt+1);
    PowerSet(L+1, preCnt);
}
```

输出全部排列的方法

- 给定一个整数序列，输出其全部可能的排列
  - 输入：1, 2, 3
  - 输出：

1 2 3

1 3 2

2 1 3

2 3 1

3 1 2

3 2 1

递归解法：

```
P(ListOfInt lst, prefix)
{
        if(lst为空) {输出prefix；  return;}
        for(x in lst)
        {
                newprefix = prefix + x;
                newlst = lst – x;
                P(newlst, newprefix);
        }
}
```

输出全部排列的方法

- 注意：prefix的长度+lst的长度等于原来数据的长度。
- 可以使用一个数组来表示这两个参数
  - 前面半部分存放prefix，后面存放lst

```
int LST[MAX];
void P( int preLen)
{
        int j; //选取的元素
        if(preLen >= cnt)
        {    for(j = 0; j<preLen; j++)    cout << LST[j] << " ";    cout << endl;    return;  }

        for(j = preLen; j<cnt; j++)
        {
                int tmp = LST[j]; LST[j] = LST[preLen]; LST[preLen]=tmp;  //计算新参数
                        //newprefix = prefix + x;newlst = lst –x;
                P(preLen + 1);
                tmp = LST[j]; LST[j] = LST[preLen]; LST[preLen]=tmp; //复原
        }
}
```