数据结构与算法5: 递归(Recursion)

分类： 数据结构与算法 2014-12-01 22:33 0人阅读 评论(0) 收藏

数据结构与算法5: 递归(Recursion)

写在前面

《软件随想录：程序员部落酋长Joel谈软件》一书中《学校只教java的危险性》一章提到，大学计算机系专业课有两个传统的知识点，但许多人从来都没搞懂过，那就是指针和递归。我也很遗憾没能早点熟练掌握这两个知识点。本节一些关键知识点和部分例子，都整理自教材或者网络，参考资料列在末尾。如果错误请纠正我。

思考列表:

1)什么程序具有递归解决的潜质?

2)递归还是非递归算法，怎么选择?

3)递归程序构造的一般模式

1.递归定义

首要引入递归定义这一概念。

通常我们定义一个新概念，总是使用已经定义过的或者意义显然的术语，二递归定义则是一种根据自身概念定义自身的定义。

例如阶乘函数的定义：

$$Fact(n) = \begin{cases} 1 & n = 0 \\ n * Fact(n-1) & n > 0 \end{cases}$$

例如斐波那契数列定义如下:

$$Fib(n) = \begin{cases} n & n < 2 \\ Fib(n-1) + Fib(n-2) & n \geq 2 \end{cases}$$

递归定义，主要有两个作用，一是产生新的元素，另一个是判断一个元素是否属于某个集合.

具有这种递归定义的函数，是很容易编程实现的。

2. 递归是怎么实现计算过程的？

假设我们讲上述阶乘函数转换为程序代码如下:

[cpp] view plain copy print ?

```cpp
//using recursion
int factr(int n)
{
```

阅读排行

文章存档

最新评论

JAVA学习脚印10:解惑java 中UTI
看连连: 多谢博主，看了好些资料，终于明白关于java中的utf-16的解释了

OpenGL学习脚印: 理解坐标系及
cmweiwei: 赞一个，学习到了新东西

JAVA学习脚印6: java方法调用机
逐梦行者: @u012890871:重写父类方法返回值，子类不是必须与父类一样的。java规范允许重写父类方法时...

JAVA学习脚印6: java方法调用机
sjy1203: 重写父类方法返回值不是必须一样的吗？

中文编码转换---6种编码30个方向
逐梦行者: @Shangdeone:头文件是EnCoding.h。这是我早期写的一个练习程序,这个头文件的命名习...

中文编码转换---6种编码30个方向
Shangdeone: 程序没法用，没有encoding.h头文件

汇编语言子程序设计 查找电话号
pdl2055: 很棒

最速梯度下降法及matlab实践
逐梦行者: @chilewh:抱歉，回答的晚了点。1.为什么使用符号表达式? 符号表达式不是必须的.我使用符号...

最速梯度下降法及matlab实践
chilewh: 我是直接复制你的代码的然后在界面运行的>> syms x1 x2;X = ;fx = (x1-2)^...

最速梯度下降法及matlab实践
chilewh: 作为菜鸟的我，运行你"2)一般函数的极小值点"里写的代码，出现了错误，我想问一下，

```cpp
    if ( n ==0) return 1;
    return n*factr(n-1);
}
```

我们调用fact(3)则返回6，这里有一个疑问，我好像什么都没做啊？

如果我们选用他的迭代实现：

[cpp] view plain copy print ?

```cpp
//using iteration
int facti(int n)
{
    int result = 1;
    for(int i = 1 ; i <= n;i++)
        result *= i;
    return result;
}
```

则感觉实在利用累乘计算我们的阶乘，一下子就比较清楚了。我们的疑问在于：递归实现中，是怎么执行计算过程的？

首先要了解系统中函数调用时大致情况(此处不详细学习，要想更深入和全面，请参考:Wiki Call stack)。在高级语言编写的程序中，调用函数和被调用函数之间的链接及信息交换通过栈来进行。（该段参考自【1】）

通常，在运行被调用函数之前，系统需要做3件事，包括:

- 将所有的实在参数、返回地址等信息传递给被调用函数保存
- 为被调用函数的局部变量分配存储区
- 将控制转移到被调用函数的入口

从被调用函数返回调用函数之前，系统也要完成3件事：

- 保存被调用函数的计算结果
- 释放被调用函数的数据区
- 依照被调函数保存的返回地址将控制转移到调用函数

归纳起来，就是函数调用的过程中处理要素包括：函数控制权的转接工作（利用函数入口地址和返回地址），局部变量的空间分配工作，实参和形参的传递，函数的返回结果传递。一个函数的状态由一个5元组决定,即function(入口地址，返回地址，形参值，局部变量值，返回值)。保存所有这些信息的数据区称为活动记录（activation record）或者叫做栈结构(stack frame),它是在运行时栈上分配空间的。活动记录，在函数开始执行的时候得到动态分配的空间，在函数退出的时候就释放其空间。main函数的活动记录比其他活动记录生命周期长。

这里注意的是，活动记录保存函数参数的时候，既可以值传递也可以传递地址，如果是值传递则保存数据元素的副本，如果是传递数组或者按引用则活动记录包含数组第一个元素的地址（数组首地址）或者该变量的地址。同时对于局部变量，活动记录只包含他们的描述符和指向存储它们的位置的指针。

简单的函数调用过程,例如如下代码:

[cpp] view plain copy print ?

```cpp
int main()
{

      int m,n;
      ...
 /*110*/ first(m,n);
 /*111*/ ...

}

int first(int s,int t)

{
```

是我在哪里少写了...

```

     int i;

    ...

    /*210*/ second(i);
    /*211*/ ...
}

int second(int d)

{

  int x,y

  ...

}
```

那么函数调用的过程中形成的活动记录栈的内容如下:

x,y
d
*211
?
i
s,t
*111
?
m,n
*main 返回地址
? 返回值

对于递归函数调用，表面上看，好像我们什么都没做，就完成了功能，实际上在函数递归调用的过程中，函数的活动记录不停的分配和回收，计算过程在进行着。

递归调用不是表面上的函数自身调用，而是一个函数的实例调用同一个函数的另一个实例。(参考自【2】)

我们将上面的阶乘函数重写，标上一个地址标号(这是一个粗略的标号，实际上底层的机器地址不是这样的):

[cpp] view plain copy print ?

```cpp
int main()
{
  /*102*/ int n = factr(3);
  /*103*/ std::cout << "Factorial of "<<n<<" : "<<factr(n)<<std::endl;

}
//using recursion
/*201*/ int factr(int n)
{
/*202*/  if ( n == 0)
/*203*/      return 1;
/*204*/  return n*factr(n-1);
}
```

则我们在main函数中调用fact(3)是执行的活动记录过程如下:

fact(0)	0	0	0	0	0	0
	*204	*204	*204	*204	*204	*204
	?	1	1	1	1	1
fact(1)	1	1	1	1	1	1
	*204	*204	*204	*204	*204	*204
	?	?	1*1=1	1	1	1
fact(2)	2	2	2	2	2	2
	*204	*204	*204	*204	*204	*204
	?	?	?	2*1=2	2	2
fact(3)	3 参数	3	3	3	3	3
	*102	*102	*102	*102	*102	*102
	?	?	?	?	3*2=6	3*2=6
main	n	n	n	n	n	n=6
	*main (返回地址)	*main	*main	*main	*main	*main
	?(返回值)	?	?	?	?	?

可以看出实际上递归调用时，系统不停的分配活动记录，调用从main()--->fact(3)--->fact(2)-->fact(1)--->fact(0) 一层层深入，然后再一层层回退，直到main函数中。由于活动记录中保存了函数局部变量，因此每次调用之间互补干扰，从一个被调用函数返回调用函数时能够保证计算出准确的结果，并返回上一层的函数，依此录栈是分析递归程序的一种好的方法。

3.递归类别

递归有许多级别和许多不同量级的复杂度。

我们从尾部递归与非尾部递归，直接递归和间接递归来分类了解。

简单的如，尾部递归。尾部递归，即那种在函数实现的末尾只使用一个递归调用。尾部递归的特点是，当进行调用时，函数中没有其他剩余的语句要执行，并且在这之前也没有其他直接或者间接的递归调用。

尾部递归示例程序:

[cpp] view plain copy print ?

```cpp
#include <iostream>
#include <list>
#include <string>
using namespace std;

void printListi(list<int>::iterator itCur,list<int>::iterator end);
void printListr(list<int>::iterator itCur,list<int>::iterator end);

int main(int argc,char** argv)
{
   list<int> iList;
   for(int i = 0;i < 10 ;i++)
       iList.push_back(i);
   if ( argc == 2 && string(argv[1]) == "-r")
      printListr(iList.begin(),iList.end());
   else
      printListi(iList.begin(),iList.end());
}

//using tail recursion
void printListr(list<int>::iterator itCur,list<int>::iterator end)
{
    if( itCur == end)
    {
      std::cout<<std::endl;
      return;
    }
    std::cout<<*itCur++<<" ";
    printListr(itCur,end);//at tail ,call itself
}
//using iteration
void printListi(list<int>::iterator itCur,list<int>::iterator end)
{
    while(itCur != end)
        std::cout<<*itCur++<<" ";
    std::cout<<std::endl;
}
```

这里使用尾递归或者迭代方式输出链表内容，可以看出尾递归只是一个变形的循环，可以很容易用循环来代替。在含有循环结构的语言中，不推荐使用尾部递归。

除了尾部递归，当然存在非尾部递归，例如:

[cpp] view plain copy print ?

```cpp
#include <iostream>
#include <string>

void reverse1();
void reverse2();
void reverse3();

int main(int argc,char** argv)
{
   std::cout<< "input somethind:"<<std::endl;
   reverse2();
   std::cout<<std::endl;
}
//all chars reverse,no newline
void reverse1()
{
   char ch;
   std::cin.get(ch);
   if( ch != '\n')
   {
      reverse1();
      std::cout.put(ch);
   }
}
//with newline at head line,other chars reverse
void reverse2()
{
   char ch;
   std::cin.get(ch);
   if( ch != '\n')
      reverse2();
   std::cout.put(ch);
}
//output only newlines
void reverse3()
{
   static char ch;//note the static
   std::cin.get(ch);
   if( ch != '\n')
      reverse3();
   std::cout.put(ch);
}
```

这里提供了三个版本的函数，旨在帮助理解递归调用特性。版本1，会对输入进行逆向输出。当然，可以利用栈或者缓存实现逆向输出的迭代版本。

上面的尾部和非尾部递归，都是直接递归，也就是函数自身调用自己。另外一种情形是，一个函数通过其他函数间接调用自身，例如f()-->g()-->f()这种间接形式。

还有所谓的嵌套递归，即函数不仅根据自身定义，而且还作为该函数的一个参数进行传递。例如Ackermann函数：

$$A(n,m)=\begin{cases} m+1 & n=0 \\ A(n-1,1) & n>0, m=0 \\ A(n-1,A(n,m-1)) & others \end{cases}$$

这种情形是很复杂的。

4 递归经典问题

4.1 Tower of Hanoi

汉诺塔问题已经将的很多了，这里给出其递归和迭代实现，然后讨论一些注意点。

迭代版本的实现，参考了http://www.ecse.rpi.edu的资料，然后自行整理的。迭代版本的实现方式如下:

将盘子从大到小编号1-n,将柱子编号0,1,2。规定:

1)每次只能移动一个盘，且只能将小盘放在大盘上面。

2)盘子偶数号码时，沿着逆时针方向移动即0-2-1-0;盘子奇数号码时，沿着顺时针移动，即0-1-2-0；

3)每次移动的盘子不能是上次移动过的盘子

注意，每次移动的过程中，要选择一个上次没有移动过的盘，那么剩下的两个盘子中，肯定有一个大和一个小些的，一般总是选择最小的一个，如果没有最小的一个则移动那个大的盘(例如另外一个没有移动的盘已经压在了刚刚移动过的盘子下面)。同时如果初始时移动盘子数目为奇数，则最终盘子在1号柱子，否则在2号柱子。

[cpp] view plain copy print ?

```cpp
// using recursion
void hanoiRecursion(int n,char src,char mid,char target)
{
    if(n == 0)
    {
      return; // do nothing
    }
    hanoiRecursion(n-1,src,target,mid); // move the up n-1 stack to mid
    ++g_stepCnt;
    cout<<"("<<n<<","<<src<<"-->"<<target<<")"
<<endl;// move the n-th stack to target
    hanoiRecursion(n-1,mid,src,target);// move the up n-1 stack from mid to target
}
// using iteration
// if n odd then final post is 1,else is 2
void hanoiIteration(int n)
{
   stack<int> dStack[3];

   if (n <= 0) return;

   // put n disk at stack 0
   for(int i = n;i > 0;i--)
     dStack[0].push(i);
   int lastItem = -1; //record last moved disk

   while(!dStack[0].empty() || !dStack[n%2+1].empty())
   {
      //pick the smallest and not the last moved disk to move
      int stackNum = 0,moveItem = n+1;
      for(int i = 0;i < 3;i++)
      {
         if(dStack[i].empty() || dStack[i].top() == lastItem)
             continue;
         if(dStack[i].top() < moveItem)
         {
             stackNum = i;
             moveItem = dStack[i].top();
         }
      }
      lastItem = moveItem;
      ++g_stepCnt;
      //move odd-numbered disk clockwise ,move even-numbered disk counter-clockwise
      int target = (moveItem % 2 == 0 )?(stackNum+2)%3:(stackNum+1)%3;
      cout<<"("<<moveItem<<","<<stackNum<<"-->"<<target<<")"<<endl;
      dStack[target].push(moveItem);
      dStack[stackNum].pop();
   }
}
```

两种实现方法移动的都是最少次数。

如何计算盘子移动次数？

一个性质是，只要盘子总数确定，不过从哪儿移动到哪个目的柱子，总共要移动的次数是一样的。

假设n个盘子移动次数为h(n),则我们可以计算如下:

$$
\begin{aligned}
h(n) &= h(n-1) + 1 + h(n-1) \\
&= 2 * h(n-1) + 1 \\
&= 2 * (2 * h(n-2) + 1) + 1 \\
&= 2^2 * h(n-2) + 2 + 1 \\
&= 2^2 * (2 * h(n-3) + 1) + 2 + 1 \\
&= 2^3 * h(n-3) + 2^2 + 2 + 1 \\
&= \cdots \\
&= 2^{n-1} * h(1) + 2^{n-2} + \cdots + 2 + 1 \qquad h(1) = 1 \\
&= \sum_{i=0}^{n-1} 2^i \\
&= 2^n - 1
\end{aligned}
$$

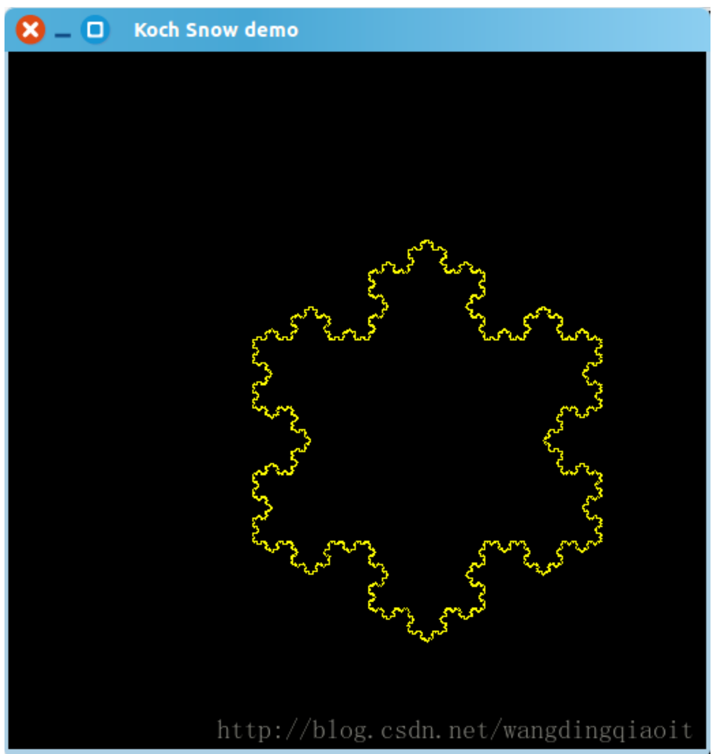
可以看出当n=64时，数目极其巨大，无法在有效时间内解决。

Hanoi塔的迭代版本可以使用位操作实现，不过好像技巧性比较强，有兴趣可以参考: How does this work? Weird Towers of Hanoi Solution.

4.2 Koch Snow

Koch雪花问题，设计到递归实现问题。关于其一般介绍可参考Wiki koch snow,这里主要讲述与递归实现相关的部分。

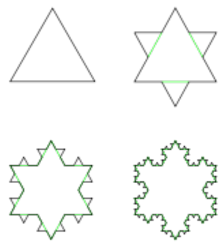
首先给出一个OpenGL绘制的效果图如下:



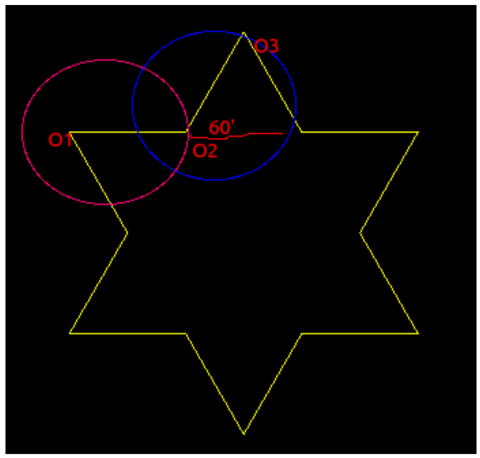
关于这个雪花的实现有3个重要的数学问题。

1)绘制的规律--利用转动角度和圆的坐标计算

首先发现一下规律。



上面分别为进行了0次分形，1,2,3次分形的图形，不管怎么分形有一个重要的性质(一开始我是没想到的，后来观察教材的实现代码是明白了)。



第一次从O1点计算出O2点，O2在其圆周上，利用圆心坐标公式:

[cpp] view plain copy print ?

```cpp
prevX = prevX + (side/3)*cos(angle*PI / 180.0);
prevY = prevY + (side/3)*sin(angle*PI / 180.0);
```

即可计算，然后从O2点计算出O3点，这次在O2圆周上且圆周夹角为+60，后面angle依次加上-120度，+60，这样第一条便就绘制结束。第二条边与第一条边之间angle加上-120度，第三条和第二条也是angle加上-120度。这个angle是全局的，这一点性质很重要。

2)周长是无限的

每进行一次细分，则边的数目是原来的4倍，同时边的长度是上次的1/3，因此有:

$$side = \frac{side}{3^n} \quad length = 3 * 4^n * \frac{side}{3^n} = 3 * (\frac{4}{3})^n$$

当n趋于无穷大时，长度不收敛，为无穷大。

3)面积是有限的

面积推导也比较简单，每次分形后，面积都在前面的基础上增加，而增加的部分就是向外扩展的小三角形的面积。设原始边长为a,则前几次分形的面积计算如下:

$$s_0 = s(a) = \frac{\sqrt{3}}{4}a^2 \quad s_1 = s_0 + 3 * s(\frac{1}{3}a)$$

$$s_2 = s_1 + 3 * 4 * s(\frac{1}{3^2}a) \quad s_3 = s_2 + 3 * 4 * 4 * s(\frac{1}{27}a)$$

通过发现规律，可以计算出n趋于无穷时的面积为:

$$
\begin{aligned}
s_n &= \sum_{i=1}^{n-1} s_i + 3*4^{n-1}*s(\frac{1}{3^n}a) \\
&= \frac{\sqrt{3}}{4}a^2 + 3*\frac{\sqrt{3}}{4}*\frac{1}{9}a^2 + 12*\frac{\sqrt{3}}{4}*\frac{1}{81}a^2 + \cdots + 3*4^{n-1}*\frac{\sqrt{3}}{4}*\frac{1}{9^n}a^2 \\
&= \frac{\sqrt{3}}{4}a^2 + \frac{3\sqrt{3}}{16}a^2\sum_{i=1}^{n}(\frac{4}{9})^n \\
&= \frac{\sqrt{3}}{4}a^2 + \frac{3\sqrt{3}}{20}a^2 \quad (\frac{4}{9} < 1) \\
&= \frac{2\sqrt{3}}{5}a^2
\end{aligned}
$$

由此可以看出，Koch雪花在有限的面积内，周长却无限大。

实际在利用OpenGL绘制Koch雪花时，只需要保存所有的点即可，然后一次渲染即可。

关键部分实现如下:

[cpp] view plain copy print ?

```cpp
//predefined variables
std::vector< glm::vec4 > vertexVec;//hold points
float prevX = 0.0f,prevY =0.0f;//the previous point
int angle = 0;

float side = 3.0f;
int level = 6;

//prepare snow data
void prepareData()
{

    float originX = 0,originY =0;
    vertexVec.push_back(glm::vec4(originX,originY,0,1));
    for(int i=0;i< 3;i++)
    {
        drawFourLine(side,level);
         angle += -120;
    }
}
//draw four lines
void drawFourLine(float side,int level)
{
    if (level == 0)
    {
          prevX = prevX + (side/3)*cos(angle*PI / 180.0);
          prevY = prevY + (side/3)*sin(angle*PI / 180.0);
          vertexVec.push_back(glm::vec4(prevX,prevY,0,1));
    }
    else
    {
        drawFourLine(side/3,level-1);
        angle += 60;
        drawFourLine(side/3,level-1);
         angle += -120;
         drawFourLine(side/3,level-1);
         angle += 60;
         drawFourLine(side/3,level-1);
    }
}
```

代码表明，我们实际上把一个雪花看做3个4段组成的，而一个4段的每一段又可以继续分为4段，特殊情况例如没有分形时只有一条边，而这条边可以看做一个4段的特殊情况。

4.2 全排列问题

曾经遇到过一个全排列的问题，即给定无重复值的字符串，给出其全排列，例如ab,全排列即为ab,ba.

实际上在用递归实现时，基本算法描述为:

[plain] view plain copy print ?

```
permutaion(input,output)
    如果只有一个字符，则将字符添加到output即可;
    否则:
        每次从input中取出一个不同的字符作为头部字符head
                  然后拿出剩下的部分leftPart进行排列(返回的是一个子串的排列的集合)
                  对剩余部分排列的每个结果的首部加上头部head,得到的结果添加到output
```

简单来讲，就是固定一个头部，然后让剩下的子串全排列，将头部和子串全排列的每个结果串链接起来，从而得到完整的全排列。
对于子串重复执行这个过程，直到遇到只有一个字符时，它不用排列了，直接返回即可。

算法实现为:

[cpp] view plain copy print ?

```cpp
// permutation the input string and save it to result
void permutation(string input,vector<string> &result)
{
   if(input.length() == 1)
   {
      result.push_back(input);
      return;
   }
   for(string::size_type i= 0;i < input.length();++i)
   {
        string leftPart = input;
        leftPart.erase(i,1);//get left part
        vector<string> strVec;
        permutation(leftPart,strVec);// use  left part to permutate
        // add this char with left part result
        for(vector<string>::iterator it = strVec.begin();it != strVec.end();++it)
           result.push_back(input[i] + *it);
   }
}
```

例如输入"abc",则输出结果为:

[plain] view plain copy print ?

```
Permutation of: abc ,kind: 6
abc
acb
bac
bca
cab
cba
```

用递归实现的还有很多程序，例如迷宫问题，8皇后问题等等，不再列举。

5. 递归与非递归选择

刚刚学习python的时候写过一个Fibonacci数列的程序，如下:

[python] view plain copy print ?

```python
def fibr(n):
    """get the n-th Fibonacci series number,using recursion"""
    global callCnt
    callCnt += 1 # count how many time function called
    if n < 2:
       return n
    return fibr(n-1)+fibr(n-2)
```

当然这个callCnt是之后加上去的。程序运行正常，可是我输入n=40,n=100的时候，程序好像死机了，然后我就开始责怪python效率低(刚开始我没有分析复杂度，确实错怪了Python：)。

我们看下实际情形(粗略的时间估计):

[plain] view plain copy print ?

```
~ python3 fibr.py 30
fib(30)=832040
    called 'recursive function fibr' 2692537 times
    consumed 1029.3409824371338 ms

**************************************************
~ python3 fibr.py 40
fib(40)=102334155
    called 'recursive function fibr' 331160281 times
    consumed 125293.18809509277 ms
```

现在知道实际上在**递归调用斐波那契数列时例如n=40时函数fibr调用了3亿多次，花了2分多钟才计算完毕:**

同样书写了一份迭代版本，去除程序中多余的时间统计和函数调用统计语句后，粗略地比较了c++/P

时间:

Fibonacci 数列计算粗略时间比较												
	n=20			n=30			n=40			n=45		
	递归	迭代	比值	递归	迭代	比值	递归	迭代	比值	递归	迭代	比值
C++	0m0.003s	0m0.003s	1	0m0.044s	0m0.003s	14	0m4.362s	0m0.003s	1454	0m48.595s	0m0.003s	16198
Python	0m0.076s	0m0.037s	2	0m0.696s	0m0.037s	18	1m23.854s	0m0.041s	2045	>8m17.686s	0m0.040s	>12425
比值	25	12		15	12		19	13		10	13	

通过分析上述数据，可以得出:迭代算法的效率要比递归效率高，C++编译型语言执行计算时比解释型语言Python要快10倍左右(这个比较不代表python在其他方面没有优势)。

下面要对Fabonacci数列的递归实现和迭代实现做简单分析。

对于递归实现，归纳总结得出:

编号	0	1	2	3	4	5	6	7	8	
Fib序列值	0	1	1	2	3	5	8	13	21	
调用函数次数	1	1	3	5	9	15	25	41	67	2Fib(n+1)-1次
加法次数	0	0	1	2	4	7	12	20	33	Fib(n+1)-1次

也就是说，对于Fib(n)，计算时执行函数调用2Fib(n+1)-1次，执行加法Fib(n+1)-1次。

而对于迭代实现:

[cpp] view plain copy print ?

```cpp
//using iteration
long long fibi(int n)
{
   if(n < 2)
       return n;
    else
    {
       long long last =0;
       long long cur = 1;
       for(;n > 1;n--)
       {
           long long tmp = cur;
           cur += last;
           last = tmp;
       }
       return cur;
    }
}
```

对于n>1,进入for循环，一共执行(n-1)次。每次循环中执行3次复制操作，隐含一个加法操作，那么一共需要3(n-1)次赋值和(n-1)次加法运算。

Fibonacci数列增长很快，迭代算法不需要递归调用的函数开销，同时加法和赋值操作也比递归版本少，因此，对

于Fibonacci数列使用递归算法是不恰当的，应该采取其迭代版本。

这个例子告诉我们，虽然递归算法很容易书写，但是具体应该用递归还是迭代实现，应该视情况而定;最好对迭代和递归版本实现的复杂度和开销进行分析，或者在实际机器上比较算法执行效率。

6.总结

对于递归算法，总结如下:

1)一般对如要解决的问题，如果能进行分解，且分解为一个和原问题具有相同特征，则可以利用递归实现。

例如移动Hanoi塔分解后就是将上面的(n-1)个塔从一个塔移动到另外一个塔；例如全排列问题，取出一个作为头部后，对于剩下的元素，同样要求出其全排列；对于迷宫问题，总是从当前位置开始，如果是结束位置则停止，否则尝试4个方向走出迷宫，每走到一个新位置，又作出同样的抉择。

2)递归程序的编写，有一个普遍的模式，即程序有一个基底或者叫做出口，另外的部分就是调用自身即可。请注意递归程序一定要选择好出口，否则就成了盗梦空间里回不来了。

出口部分，例如只有一个盘子的Hanoi塔，只需移动他即可；只有一个字符的全排列问题，只需要返回它即可;这些都是程序的出口。递归程序的一个模式就是:

[plain] view plain copy print ?

```
function(param)

   if 出口:

      处理并返回;

  否则:

        ...

        function(param)

      .....
```

3)是谁在背后支持我们的递归调用？ 一个是语言本身的支持，另一个是操作系统的运行时栈的支持以及可能的硬件支持。

递归函数避免不了递归调用时的栈开销，但这也并不意味着它的效率一定比迭代方法低。

对如一个问题，迭代实现和递归实现需要作出比较和分析，然后确定到底使用哪种算法实现。

如果想进一步了解，可以参考[4]上面的讨论。

最后，贴上StackOverflow上面的关于递归的一个挺有趣的解释:

[plain] view plain copy print ?

```
A child couldn't sleep, so her mother told her a story about a little frog,
    who couldn't sleep, so the frog's mother told her a story about a little bear,
         who couldn't sleep, so the bear's mother told her a story about a little wease
            who fell asleep.
        ...and the little bear fell asleep;
    ...and the little frog fell asleep;
...and the child fell asleep.
```

参考资料:

[1] 《数据结构》 严蔚敏 吴伟明 清华大学出版社

[2]《数据结构与算法 c++版 第三版》 Adam Drozdek编著 清华大学出版社

[3] Wiki Tower of Hanoi

[4] What is recursion and when should I use it?

[5] Wiki Koch snowflake

[6] a simpler iterative solution to the towers of hanoi problem

上一篇 Python3 面向对象OOP编程初步认识

顶 0
踩 0

主题推荐 数据结构 recursion 算法 递归 局部变量

猜你在找

数据结构学习笔记(12.递归的应用之八皇后回溯算法)
它处资料：数据结构学习笔记(12.递归的应用之八皇后
数据结构与算法中的“递归”——用回溯法求解8皇后问
【数据结构与算法】二叉树深度遍历（非递归）
【算法数据结构Java实现】递归的简单剖析及时间复杂
Java数据结构---递归算法
算法设计和数据结构学习_8(单链表的递归逆序)
【数据结构与算法】汉诺塔算法——C语言递归实玥
数据结构-----二叉树的中序遍历的非递归算法实现
数据结构与算法之递归算法 C++和PHP实现

查看评论

暂无评论

发表评论

用 户 名： ziyuanxiazai123
评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap