**[算法中的递归分析和分治法的原理](http://www.cnblogs.com/kubixuesheng/p/4355060.html)**

分析递归算法三种方法

替换法、迭代法、通用法（master method）

作用：分析递归算法的运行时间

分治算法

将一个问题分解为与原问题相似但规模更小的若干子问题，递归地解这些子问题，然后将这些子问题的解结合起来构成原问题的解。这种方法在每层递归上均包括三个步骤：

divide（分解）：将问题划分为若干个子问题

conquer（求解）：递归地解这些子问题；若子问题Size足够小，则直接解决之

Combine（组合）：将子问题的解组合成原问题的解

其中的第二步很关键：递归调用或直接求解  （递归终结条件），且有的算法“分解”容易，有的则“组合”容易

分治法举例：

归并排序

①分解：把n个待排序元素划分为两个Size为n/2的子序列

②求解：递归调用归并排序将这两个子序列排序，若子序列长度为1时，已自然有序，无需做任何事情（直接求解）

③组合：将这两个已排序的子序列合并为一个有序的序列

显然，分解容易（一分为二），组合难。

快速排序

刚刚分析过了，快速排序是枢轴记录划分，也就是分解难，但是组合易。   A[1…k-1] ≤ A[k] ≤ A[k+1…n]

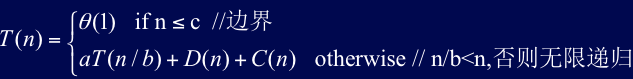
分治算法分析

设T(n)是Size为n的执行时间，若Size足够小，如n ≤ C (常数)，则直接求解的时间为θ(1)

①设完成划分的时间为D(n)

②设分解时，划分为a个子问题，每个子问题为原问题的1/b，则解各子问题的时间为aT(n/b)

③设组合时间C(n)



一般地，递归的求解划分，而解递归式时可忽略细节

①假定函数参数为整数，如http://images.cnitblog.com/blog2015/682679/201503/311620576703808.png

②边界条件可忽略，当n较小时，T(n）= θ(1)

因为这些细节一般只影响常数因子的大小，不改变量级。求解时，先忽略细节，然后再决定其是否重要！

分析的方法

替换法

猜测解，用数学归纳法确定常数C，证明解正确，关键步骤是用猜测的解代入到递归式中。

做出好的猜测（没有一般方法，只能凭经验）

①与见过的解类似，则猜测之。

②先证较宽松的上、下界，减小猜测范围。

细节修正

有时猜测解是正确的，但数学归纳法却不能直接证明其细节，这是因为数学归纳法不是强大到足以证明其细节。这时可从猜测解中减去一个低阶项以使数学归纳法得以满足

避免陷阱

与求和式的数学归纳法类似，证明时渐近记号的使用易产生错误。

变量变换

有时改动变量能使未知递归式变为熟悉的式子。例如：

迭代法

展开：无须猜测，展开递归式，使其成为仅依赖于n和边界条件的和式，然后用求和方法定界。需要关注：

1、达到边界条件所需的迭代次数

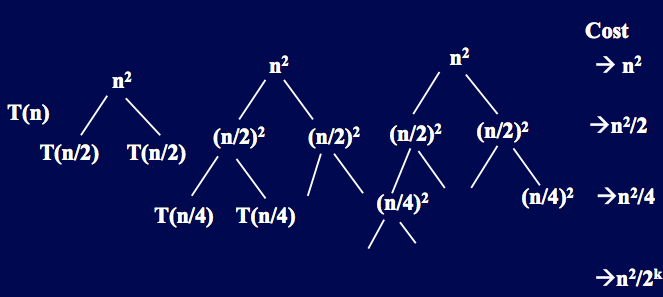
2、迭代过程中的和式。若在迭代过程中已估计出解的形式，亦可用替换法

3、当递归式中包含floor和ceiling函数时，常假定参数n为一个整数次幂，以简化问题。

递归树

使展开过程直观化

例： T(n)=2T(n/2)+n^2   （不妨设n=2k）



The master method（通用法，万能法）

可迅速求解

T(n)=aT(n/b)+f(n)   //常数a ≥1, b>1, f(n)渐近正

意义：将Size为n的问题划分为a个子问题，每个子问题Size为n/b。每个子问题的时间为T(n/b)，划分和combine的时间为f(n)。

注意：n/b不一定为整数，应为⌊n/b⌋或⌈n/b⌉，不会影响渐近界。

关于递归和循环的理解与比较

递归通俗的说就是一个函数调用函数自己（本身），这个调用过程叫递归，递归是一把双刃剑（有时方便，有时不好），如果需要处理重复的需要多次计算的问题，通常可以选择用递归或者循环两种方式，但是递归的执行效率不如循环语句。

注意：必须设置终止递归的条件检测，否则慎用。

[复制代码](javascript:void(0);)

void up\_and\_down(int);//函数原型声明

int main()

{

up\_and\_down(1);//调用递归函数  
  
 system("pause");  
return 0;

}

void up\_and\_down(int n)

{

printf("level%d, n地址=%p\n", n, &n);

if (n < 4)

{

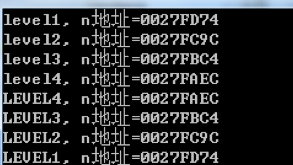
up\_and\_down(n + 1);

}

printf("LEVEL%d, n地址=%p\n", n, &n);

}

[复制代码](javascript:void(0);)



首先，main函数用参数1调用递归函数，递归函数形参n=1，打印语句level1……然后，n<4，故函数本身使用参数n+1=2,第二次调用自己，这样就打印了level2……

以此类推，当执行到第四级调用，n=4，if失效，不再调用函数，而是执行了第二句打印，先输出LEVEL4……此时第四级调用结束，控制权返回给了主调函数，也就是第三级主调函数，此函数中上一句是if语句，已经执行完毕，然后继续执行第二句打印语句，输出LEVEL3……第三级调用结束，返回控制权给调用函数（也就是第二级主调函数），然后第二级函数开始继续执行，以此类推，打印LEVEL2,1……

**递归的基本原理**

每一级递归都使用自己这一级的私有变量n，同级调用时的地址和返回的地址是一样的。好好揣摩！

这是函数自己在一层层的往深度调用自己，然后一层层的往回返，每到一层，就继续执行接下来的语句（故调用开始的地址和返回的地址一样），而每一级递归都是用自己的局部变量。也就是第一级的n不同于第二级的n，这样子，函数逐步调用然后逐步返回直到main函数里。

递归函数里，递归语句之前的语句和各级被调的递归函数执行顺序一致，而递归语句之后的语句和被调的递归函数执行顺序相反（这一特点针对涉及反向顺序的编程问题很有用）

递归函数必须包含可以终止的条件，因为递归可以替代循环，故必须有终止

**尾递归**

最简单的递归：递归语句放到函数末尾，恰在return语句前，叫做tail recursion（尾递归），因为出现在函数尾部，作用相当于一条循环语句。

[复制代码](javascript:void(0);)

//计算阶乘（递归和循环）

#include <stdio.h>

#include <stdlib.h>

//计算阶乘

int factorial(int);

int loopFactorial(int);

int main()

{

int num;

printf("输入1-12的整数，q退出\n");

while (scanf\_s("%d", &num))

{

if (num < 0)

{

printf("error!输入1-12的整数!");

}

else if (num > 12)

{

printf("输入1-12的整数!");

}

else

{

printf("\n%d的阶乘=%d", num, factorial(num));

printf("\n%d的阶乘=%d", num, loopFactorial(num));

}

printf("\n输入1-12的整数");

}

system("pause");

return 0;

}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

//循环计算阶乘

int factorial(int n)

{

int temp;

for (temp = 1; n > 1; n--)

{

temp \*= n;

}

return temp;

}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

//使用递归计算阶乘

int loopFactorial(int n)

{

int temp;

if (n > 0)

{

temp = n \* loopFactorial(n - 1);//属于尾递归，如n>0那么这就是最后一句

}

else

{

temp = 1;//必须要有递归结束判断条件！

}

return temp;

}

[复制代码](javascript:void(0);)

注意:整型范围，32位机器，int类型最大到21多亿，再大的话，就要用long long或者double类型，一般来说，选择循环比较好些，递归每次调用都要有自己的变量集合，占据内存大，每次都要存储新的变量集合到堆栈，这样速度慢，但是递归（最简单的是尾递归）比较简单。一些情况还是要用。