Recursion 递归与递归计算

Introduction to Recursion

• 递归是一种重复

• 递归是一个模块通过调用自身实现算法步骤重复的过程

• 每个递归调用都基于不同的、通常更简单的实例

• 递归是一种分而治之、自上而下的解决问题的办法

递归函数是"自己调用自己"的函数，无论是采用直接或间接调用方式。间接递归意味着函数调用另一个函数（然后可能又调用第三个函数等），最后又调用第一个函数。因为函数不可以一直不停地调用自己，所以递归函数一定具备结束条件。

递归函数基于了这样一个事实：每次调用函数时，都会重新建立动态变量。这些变量，以及返回时需要的调用者地址，都存储在栈中，每次函数递归都会造成栈上新增一块数据。程序员要确保栈的空间够大，足以容纳递归的中间处理过程。

递归函数是采用逻辑方式来实现自然递归规律的算法，例如二元搜索技术，或者树状结构导航（navigation）。

4 Fundamental Rules

①Base Case : Always have at least one case that can be solved without recursion.

②Make Progress : Any recursive call must progress towards a base case.

③Always Believe : Always assume the recursive call works.

④Compound Interest Rule : Never duplicate work by solving the same instance of a problem in separate recursive calls.

Basic Form

```
void recurse(){
    recurse();//Function calls itself
}
int main(){
    recurse();//Sets off the recursion
}
```

Recursive Algorithms

Once you've figured out a recursive definition for a function, one can immediately turn it into a recursive algorithm in languages (such as Java) which can handle recursion.

一旦确定了函数的递归定义，就可以立即将其转换为语言中的递归算法，它可以处理递归。

```
#include <stdio.h>
int fac(int n){
   if(n<2){
       return 1;
   }
   else{
       return n*fac(n-1);
   }
}
int main(){
    printf("%d",fac(5));
}
```

(a) Sequence of recursive calls.

(b) Value returned from each recursive call.

How dose it work?

①The module calls itself.

②New variables and parameters are allocated storage on the stack.

③Function code is executed with the new variables from its beginning. It does not make a new copy of the function. Only the arguments and local variables are new.

④As each call returns, old local variables and parameters are removed from the stack.

⑤Then execution resumes at the point of the recursive call inside the function.

Recursion Trees

A key tool for analyzing recursive algorithms is the recursion tree.

Build a Recursion Tree:

• root = the initial call
• Each node = a particular call
Each new call becomes a child of the node that called it

• A tree branch (solid line) = a call-return path between any 2 call instances

Annotation

递归：在数学上的理解，就是函数自己调用自己

迭代：迭代就是将别人的输出作为自己的输入，得到最终的结果。从函数表现形式，就是自己调用别人。这个别人可以是自己。

递归是迭代的特殊形式。这个不难理解。

树形递归，就是指递归函数的时间复杂度是指数级别。比如，自己调用自己两次以上。

线性递归，就是指递归函数的时间复杂度是线性级别。比如，自己调用自己一次以上。

循环的3要素：循环终止条件，循环体，循环变量

递归分为两个过程，递推和回归。递推则是把复杂问题逐步分解为最简单的问题，回归就是把最简单的问题完成后逐渐回溯到最原先的复杂问题。

回溯是一种算法思想，就是指当前算法进行下去以无意义时，回退上一步重新寻找新的解题思路。类似孙子兵法的三十六计走回上策。

递归的第一阶段，递进在计算机看来就是一个不停建立函数调用栈的过程。

递归的第二阶段，回归的过程其实本质上就是一个迭代的过程。我们可以看待函数栈顶部的函数返回值回作为函数栈中下- - 一个函数的输入值。整个过程会把原先递进过程建立的函数栈不停的退栈，直到回归最初建立的函数（函数栈的栈底函数）。

尾递归，是指，如果一个函数中所有递归形式的调用都出现在函数的末尾，我们称这个递归函数是尾递归的。即，递归调用后返回的结果直接return；

尾递归，不会消耗栈空间，所以尾递归时间与空间复杂度和迭代的性能一样。

线性递归，就是大家平常说的递归，线性递归函数的最后一步操作不是递归操作，将最终条件代入计算。在每次递归调用时，递归函数中的参数，局部变量等都要保存在栈中，当数据量很大的时候，会造成栈溢出。

尾递归，也就是线性迭代，尾递归函数的最后一步操作是递归，也即在进行递归之前，把全部的操作先执行完，这样的好处是，不用花费大量的栈空间来保存上次递归中的参数、局部变量等，这是因为上次递归操作结束后，已经将之前的数据计算出来，传递给当前的递归函数，这样上次递归中的局部变量和参数等就会被删除，释放空间，从而不会造成栈溢出。但是很多编译器并没有自动对尾递归优化的功能，也即当编译器判断出当前所执行的操作是递归操作时，不会理会它究竟是线性递归还是尾递归，这样也就不会删除掉之前的局部变量和参数等。另外，尾部递归一般都可转化为循环语句。

迭代算法可以转化为尾递归。尾递归算法也可以转化为递归算法。编译器就是这么干的，编译器经常会为了减少函数开销，把尾递归算法转化为迭代算法。而树形递归和线性递归就无法做到这一点。尾递归也叫线性迭代。

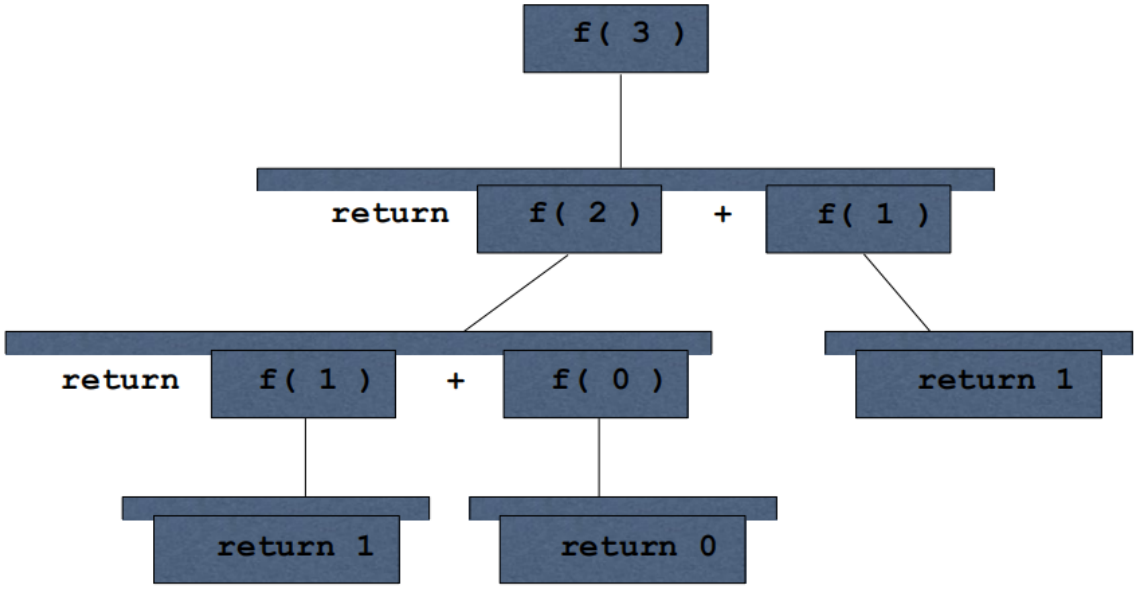
从时间复杂度来划分，可以分为树形递归和线性迭代。
尾递归也是一种特殊的线性递归，特殊在于他直接把递归函数的返回值直接return, 而一般的线性递归则需要把返回值带入表达式运算后再return。

Examples

Fibonacci

```
long fibonacci(long n){
    if(n==0||n==1) return n;
    else return fibonacci(n-1)+fibnonacci(n-2);
}
```

Set of recursive calls to function fibonacci:



Euclid's algorithm

Euclid's algorithm makes use of the fact that gcd(x,y) = gcd(y, x mod y)

$$\gcd(x,y) = \begin{cases} x, \text{ if } y = 0 \\ \gcd(y, x \bmod y), \text{otherwise} \end{cases}$$

```
int gcd(int x,int y){
    if(y==0) return x;
    else return gcd(y,x%y);
}
```

Running time: apparently we don't have the exponential method explosion problem as with binomial and
Fibonacci. Iterative version before is equivalent so this is an O (n) algorithm in terms of the length of largest input (assuming O(1) mod operation)

Hanoi Towers

```
void move(int n,int s,int a,int t){
    if(n==0){
        //
    }
    else{
        move(n-1,s,t,a);
        printf("move %d from %d to %d\n",n,s,t);
        move(n-1,a,s,t);
    }
}
```

Base Case:

No disks to be moved

Progress:

Move n-1 disks from source to the aux

Move nth form source to target

Move n-1 disks from aux to target

Square Roots by Newton's Method

```
#include <stdio.h>
#include <math.h>
const double eps=10e-3;
double pfg(double x,double g){
    //printf("%f\n",g);
    if(fabs(g*g-x)<eps){
        return g;
    }
    else{
        return pfg(x,(g+x/g)/2);
    }
}
```

guess	Quotient	Average
1	2/1->2	(1+2)/2->1.5
1.5	2/1.5->1.33	(1.5+1.33)/2->1.4167
1.4167	2/1.4167->1.4118	(1.4167+1.4118)/2->1.4142

Recursion vs Iteration

• In the **iterative** case, the program variables provide a complete description of the state of the process at any point. If we stopped the computation between steps, all we would need to do to resume the computation is to supply the interpreter with the values of the three program variables.
• In the **recursive** case, there is some additional “hidden” information, maintained by the interpreter and not contained in the program variables, which indicates “where the process is” in negotiating the chain of deferred operations.

• **Repetition**
  Iteration: explicit loop
  Recursion: repeated function calls
• **Termination**
  Iteration: loop condition fails
  Recursion: base case recognized
• Both can have infinite loops

- Balance

  Choice between performance (iteration) and good software engineering (recursion)