# 定义

一个函数直接或间接调用自己。

注：递归很容易造成死循环，比如A调用B，B又调用A这种很隐蔽的递归。

# 原理

## 函数调用

当在一个函数的运行期间调用另一个函数时，在运行被调函数之前，系统需要完成三件事：

1. 将所有的实际参数，返回地址（下一个函数地址）等信息传递给被调函数保存；
2. 为被调函数的局部变量（也包括形参）分配存储空间；
3. 将控制转移到被调函数的入口。

从被调函数返回主函数之前，系统也要完成三件事：

1. 保存被调函数的返回结果；
2. 释放被调函数所占的存储空间；
3. 依照被调函数保存的返回地址将控制转移到调用函数。

当有多个函数相互调用时，按照“后调用先返回”的原则，上述函数之间信息传递和控制转移必须借助“栈”来实现，即系统将整个程序运行时所需的数据空间安排在一个栈中，每当调用一个函数时，就在栈顶分配一个存储区，进行压栈操作，每当一个函数退出时，就释放它的存储区，进行出栈操作，当前运行的函数永远在栈顶位置。

注：递归函数分为调用和回退阶段，递归的回退顺序是它调用顺序的逆序。

A函数调用A函数和A函数调用B函数在计算机看来是没有任何区别的，只不过用我们思维理解比较奇怪。

## 满足条件

递归需要满足的三个条件：

1. 递归必须得有一个明确的中止条件；
2. 该函数所处理的数据规模必须在递减；
3. 这个转换必须是可解的。

# 循环和递归

所有的循环都可以用递归实现，所有的递归不一定可以用循环实现。

递归特点：

1. 结构清晰，代码量少，易于理解；
2. 速度慢；
3. 存储空间大（大量的递归调用会建立函数的副本，会消耗大量的时间和内存，而迭代则不需要此种代价）。

循环特点：

1. 不易理解；
2. 速度快；
3. 存储空间小。

注：递归是一种效率较低的算法（主要是涉及栈和参数的各种操作，会对CPU和内存造成资源浪费），不到万不得已不要使用递归算法。

# 递归与分治

分治思想在算法设计中是非常常见的，当一个问题规模较大且不易求解的时候，就可以考虑将问题分为几个小的模块，逐一解决。

采用分治思想处理问题，其各个小模块通常具有与大问题相同的结构，这种特性使得可以使用递归技术。

# 尾递归

尾调用（Tail Call）：一个函数的最后一步是函数调用。

编译器只能对尾递归的情况进行优化（优化就是释放之前不用的内存，非尾递归后面的运算需要使用前面变量的内存，无法优化），尾递归可以节省函数调用的空间。

可以将非尾递归形式的函数转换为尾递归形式：

static int factorial(int n) {

if(n<=1)

return 1;

return n\*factorial(n-1);

}

static int factorialTail(int n,int computed) {//computed表示已经计算出的结果

if(n<=1)

return computed;

return factorialTail(n-1, (n-1)\*computed );

}

这样编译器就可能将尾递归优化掉，这样空间复杂度就变为O(1)。但是并不是所有的编译器和编译语言都支持尾递归优化。

# 应用

树和森林都是以递归的方式定义的

树和图的很多算法都是以递归来实现的

很多数学公式就是以递归方式实现的，比如斐波拉切序列

注：在数据库开发中，where条件语句的解析Item就是不断递归实现的

## 求阶乘

long f(long n)

{

if(1==n)

{

return 1;

}

else

{

return f(n-1)\*n;

}

}

## 1+2+3+…+100的和

long sum(int n)

{

if(1==n)

{

return 1;

}

else

{

return n+sum(n-1);

}

}

## 任意长度字符串反转

**题目：**编写一个递归函数，实现将输入的任意长度的字符串反向输出的功能，例如输入字符串abcd，则输出字符串dcba。

**分析：**

要将一个字符串反向地输出，一般采用的方法是将该字符串存放到一个数组中，然后将数组元素反向的输出即可。但是这里要求输入是任意长度，所以不用递归的话，实现起来比较麻烦（可以采用动态申请内存）。

递归需要有一个结束的条件（这是必须的），那么我们可以将“#”作为一个输入结束的条件。

代码：

void print()

{

char a;

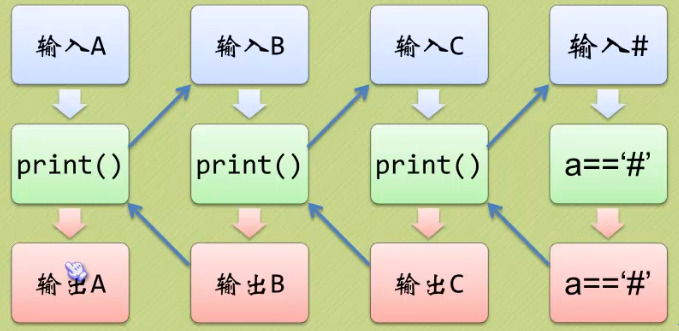
scanf("%c",&a);

if(a!='#') print();

if(a!='#') print("%c",a);

}

假设输入字符串：ABC#



## 折半查找

折半查找法是一种常用的查找方法，该方法通过不断缩小一半查找的范围，直到达到目的，所以效率比较高。

从算法的折半查找过程可以看出，这实际上也是一个递归的过程：因为每次都将问题的规模减小至原来的一半，而缩小后的子问题和原问题类型保持一致。

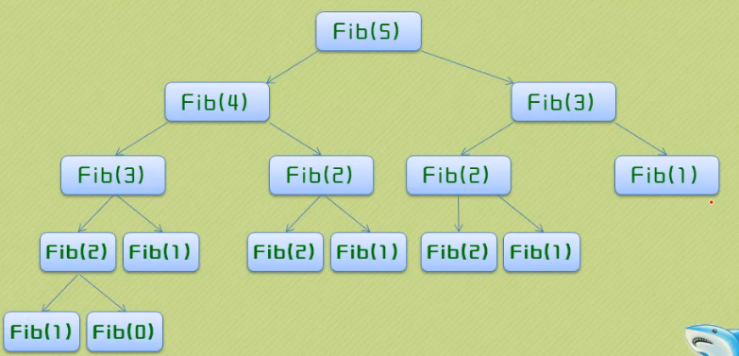
折半查找的递归实现：

## 斐波那契数列

0,当n=0

F(n) = 1,当n=1

F(n-1)+F(n-1),当n>1



**代码：**

int Fib(int i)

{

if(i<2)

{

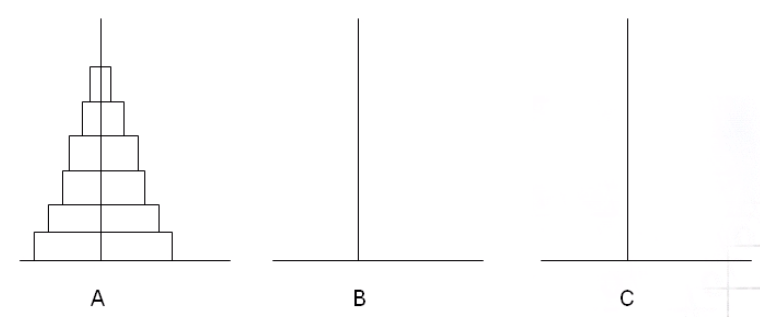
return i == 0 ? 0:1;

}

return Fib(i-1) + Fib(i-2);

}

## 汉诺塔



如何把A上面的N个盘子借助B移动到C上，要求：

1. 一次只能移动一个盘子；
2. 移动过程当中大盘子永远不能放在小盘子上面

## 八皇后问题

## 走迷宫