# 31递归和分治思想1

## 递归

妹子，甲鱼哥今天给你讲一个故事吧，从前我有个小弟，酷爱探险，有一次他进了一个山洞，然后又出来，然后又进去，然后又出来，然后又进去，然后又出来。。。。。。后来他很开心~

艹，你说什么呢？

妹子悟性真高^\_^

事实上递归就跟鸡生蛋蛋又生鸡的道理一样，只有等哪一天鸡不想生蛋了，做了绝孕手术或者用上了杜蕾斯，这个递归就算结束了。

## 斐波那契数列的递归实现

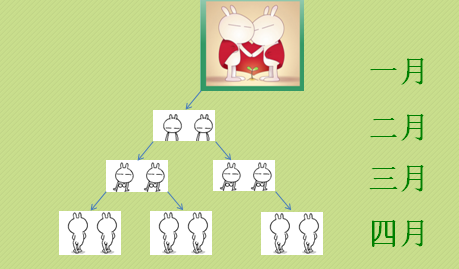
插句话：Sierpinski三角形源代码放在论坛，有需要的朋友可以去下载。

斐波那契(Fibonacci)数列的递归实现。

斐老跟小甲鱼有个共同爱好，就是老爱拿交配说事儿，不同的是小甲鱼注重过程和细节，斐老更关心结果，下边就有他讲的一个故事：

如果说兔子在出生两个月后，就有繁殖能力，一对兔子每个月能生出一对小兔子来。假设所有兔子都不会死去，能够一直干下去，那么一年以后可以繁殖多少对兔子呢？

## 坑爹的兔子



## 斐波那契数列的迭代实现

我们都知道兔子繁殖能力是惊人的，如下图：

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 所经过的月数 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 兔子的总对数 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 |

我们可以用数学函数来定义：

0，当n=0

F(n) = 1，当n=1

F(n-1)+F(n-2)，当n>1

课间练习：假设我们需要打印出前40位斐波那契数列数，我们不妨一起考虑下用迭代如何实现？

递归事实上就是函数自己调用自己，我们先一起看下代码的实现，然后再来分析：

int Fib(int i)

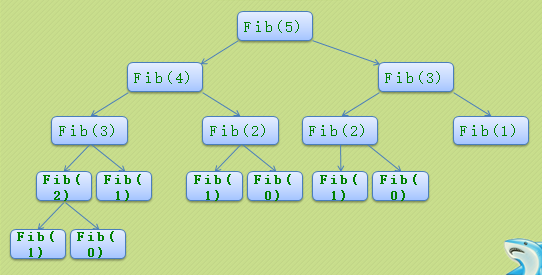
{

if( i < 2 )

return i == 0 ? 0 : 1;

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

}



迭代实现：

#include <stdio.h>

int main()

{

int i;

int a[40];

a[0] = 0;

a[1] = 1;

printf("%d %d ", a[0], a[1]);

for( i=2; i < 40; i++ )

{

a[i] = a[i-1] + a[i-2];

printf("%d ", a[i]);

}

return 0;

}

递归实现

#include <stdio.h>

int Fib(int i)

{

if( i < 2 )

{

return i == 0 ? 0 : 1;

}

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

}

int main()

{

int i;

for( i=0; i < 40; i++ )

{

printf("%d ", Fib(i));

}

return 0;

}

# 32递归和分治思想2

## 递归定义

在高级语言中，函数自己调用和调用其他函数并没有本质的不同。我们把一个直接调用自己或通过一系列的调用语句间接地调用自己的函数，称作递归函数。

不过，写递归程序最怕的就是陷入永不结束的无穷递归中。切记，每个递归定义必须至少有一个条件，当满足这个条件时递归不再进行，即函数不再调用自身而是返回值。

比如之前我们的Fbi函数结束条件是：i < 2。

对比了两种实现斐波那契的代码，迭代和递归的区别是：迭代使用的是循环结构，递归使用的是选择结构。

使用递归能使程序的结构更清晰、更简洁、更容易让人理解，从而减少读懂代码的时间。

但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(消耗系统栈)，而迭代则不需要此种付出，所以递归的效率相比迭代低。

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。

举个例子，计算n的阶乘n!

**1 n = 0**

**n! =**

**n\*(n-1) n > 0**

这样我们就不难设计出递归算法：

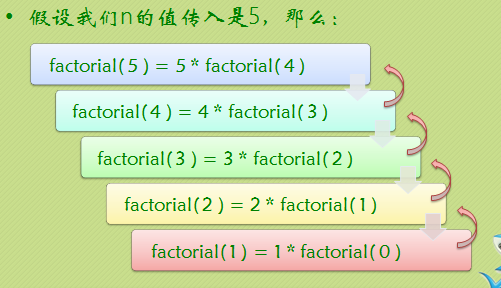
int factorial( n )

{

if( 0 == n ) return 1;

else return n \* factorial( n - 1 );

}



## 实例分析

题目要求：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FishC,则输出字符串ChsiF（可以使用栈来实现）。

应用递归的思想有时可以很轻松地实现一些看似不太容易实现的功能，例如这道题。

要将一个字符串反向地输出，童鞋们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。这道题要求输入是任意长度，所以不用递归的话，实现起来会比较麻烦（当然你可以用之前我们讲过的动态申请内存那招）。

我们说过，递归它需要有一个结束的条件，那么我们可以将“#”作为一个输入结束的条件。

void print()

{

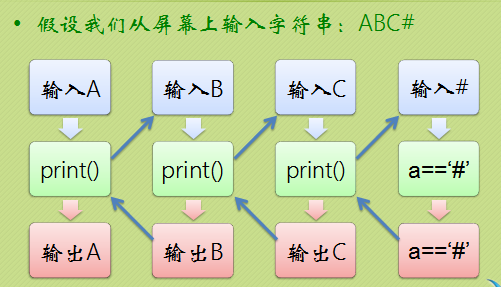
char a;

scanf(“%c”, &a);

if( a !=‘#’) print();

if( a !=‘#’) printf(“%c”, a);

}



## 分治思想

分而治之的思想古已有之，秦灭六国，统一天下正是采取各个击破、分而治之的原则。

而分治思想在算法设计中也是非常常见的，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成几个小的模块，逐一解决。

分治思想和递归算是有亲兄弟的关系了，因为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，这种特性也使递归技术有了用武之地。我们接下来通过实例来讲解。

## 折半查找算法的递归实现

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。

因为这个在《零基础入门学习C语言》等基础教程中已经详细讲解过，小甲鱼这里就通过[文字教程](http://bbs.fishc.com/)简单给大家回顾下算法的主要思路。

从算法的折半查找的过程我们不难看出，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。

作为课后题大家实现并体验下分治思想的妙处~

下节课我们将讨论递归算法和分治思想最有名的典型例题：汉诺塔问题

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// By 小甲鱼，http://www.fishc.com

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <stdio.h>

int bin\_search( int key[], int low, int high, int k )

{

int mid;

if( low > high )

{

return -1;

}

else

{

mid = (low + high) / 2;

if( key[mid] == k )

{

return mid;

}

if( k > key[mid] )

{

return bin\_search( key, mid+1, high, k ); // 在序列的后半部分查找

}

else

{

return bin\_search( key, low, mid-1, k ); // 在序列的前半部分查找

}

}

}

int main()

{

int str[11] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89};

int n, addr;

printf("请输入待查找的关键字: ");

scanf("%d", &n);

addr = bin\_search(str, 0, 10, n);

if( -1 != addr )

{

printf("查找成功，可喜可贺，可口可乐! 关键字 %d 所在的位置是: %d\n", n, addr);

}

else

{

printf("查找失败!\n");

}

return 0;

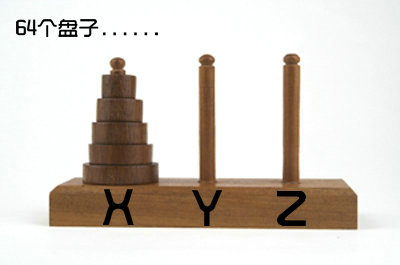
}

# 33汉诺塔

## 汉诺塔

一位法国数学家曾编写过一个印度的古老传说：在世界中心贝拿勒斯的圣庙里，一块黄铜板上插着三根宝石针。印度教的主神梵天在创造世界的时候，在其中一根针上从下到上地穿好了由大到小的64片金片，这就是所谓的汉诺塔。不论白天黑夜，总有一个僧侣在按照下面的法则移动这些金片：一次只移动一片，不管在哪根针上，小片必须在大片上面。

僧侣们预言，当所有的金片都从梵天穿好的那根针上移到另外一根针上时，世界就将在一声霹雳中消灭，而梵塔、庙宇和众生也都将同归于尽。



这其实也是一个经典的递归问题。

我们可以做这样的考虑：

先将前63个盘子移动到Y上，确保大盘在小盘下。

再将最底下的第64个盘子移动到Z上。

最后将Y上的63个盘子移动到Z上。

这样子看上去问题就简单一点了，但是关键在于第1步和第3步应该如何执行呢？

我们先一起来体验一下这个游戏：

汉诺塔游戏.swf

在游戏中，我们发现由于每次只能移动一个圆盘，所以在移动的过程中显然要借助另外一根针才行。

也就是说第1步将1~63个盘子借助Z移到Y上，第3步将Y针上的63个盘子借助X移到Z针上。那么我们把所有新的思路聚集为以下两个问题：

问题一：将X上的63个盘子借助Z移到Y上；

问题二：将Y上的63个盘子借助X移到Z上。

解决上述两个问题依然用相同的方法：

问题一的圆盘移动步骤为：

先将前62个盘子移动到Z上，确保大盘在小盘下。

再将最底下的第63个盘子移动到Y上。

最后将Z上的62个盘子移动到Y上。

问题二的圆盘移动步骤为：

先将前62个盘子移动到X上，确保大盘在小盘下。

再将最底下的第63个盘子移动到Z上。

最后将X上的62个盘子移动到Y上。

那我们是不是发现了什么？

Hanoi.c

#include <stdio.h>

// 将 n 个盘子从 x 借助 y 移动到 z

void move(int n, char x, char y, char z)

{

if( 1 == n )

{

printf("%c-->%c\n", x, z);

}

else

{

move(n-1, x, z, y); // 将 n-1 个盘子从 x 借助 z 移到 y 上

printf("%c-->%c\n", x, z); // 将 第 n 个盘子从 x 移到 z 上

move(n-1, y, x, z); // 将 n-1 个盘子从 y 借助 x 移到 z 上

}

}

int main()

{

int n;

printf("请输入汉诺塔的层数: ");

scanf("%d", &n);

printf("移动的步骤如下: \n");

move(n, 'X', 'Y', 'Z');

return 0;

}

# 34八皇后问题

## 八皇后问题

八皇后问题，是一个古老而著名的问题，是回溯算法的典型例题（这节课小甲鱼先用递归算法来解）。

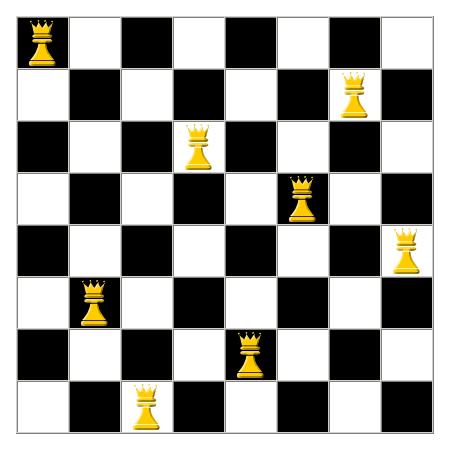
该问题是十九世纪著名的数学家高斯1850年提出：

在8X8格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。

高斯先生当年由于没有学好计算机编程，没日没夜地计算呀，得出结论是76种，硬生生把自己给“搞死”了！对了，当年还没有计算机......

正确的结果应该是92种，小甲鱼这节课和大家一起边敲代码边总结思路！

## 这是其中一种解法



#include <stdio.h>

int count = 0;

int notDanger( int row, int j, int (\*chess)[8] )

{

int i, k, flag1=0, flag2=0, flag3=0, flag4=0, flag5=0;

// 判断列方向

for( i=0; i < 8; i++ )

{

if( \*(\*(chess+i)+j) != 0 )

{

flag1 = 1;

break;

}

}

// 判断左上方

for( i=row, k=j; i>=0 && k>=0; i--, k-- )

{

if( \*(\*(chess+i)+k) != 0 )

{

flag2 = 1;

break;

}

}

// 判断右下方

for( i=row, k=j; i<8 && k<8; i++, k++ )

{

if( \*(\*(chess+i)+k) != 0 )

{

flag3 = 1;

break;

}

}

// 判断右上方

for( i=row, k=j; i>=0 && k<8; i--, k++ )

{

if( \*(\*(chess+i)+k) != 0 )

{

flag4 = 1;

break;

}

}

// 判断左下方

for( i=row, k=j; i<8 && k>=0; i++, k-- )

{

if( \*(\*(chess+i)+k) != 0 )

{

flag5 = 1;

break;

}

}

if( flag1 || flag2 || flag3 || flag4 || flag5 )

{

return 0;

}

else

{

return 1;

}

}

// 参数row: 表示起始行

// 参数n: 表示列数

// 参数(\*chess)[8]: 表示指向棋盘每一行的指针

void EightQueen( int row, int n, int (\*chess)[8] )

{

int chess2[8][8], i, j;

for( i=0; i < 8; i++ )

{

for( j=0; j < 8; j++ )

{

chess2[i][j] = chess[i][j];

}

}

if( 8 == row )

{

printf("第 %d 种\n", count+1);

for( i=0; i < 8; i++ )

{

for( j=0; j < 8; j++ )

{

printf("%d ", \*(\*(chess2+i)+j));

}

printf("\n");

}

printf("\n");

count++;

}

else

{

for( j=0; j < n; j++ )

{

if( notDanger( row, j, chess ) ) // 判断是否危险

{

for( i=0; i < 8; i++ )

{

\*(\*(chess2+row)+i) = 0;

}

\*(\*(chess2+row)+j) = 1;

EightQueen( row+1, n, chess2 );

}

}

}

}

int main()

{

int chess[8][8], i, j;

for( i=0; i < 8; i++ )

{

for( j=0; j < 8; j++ )

{

chess[i][j] = 0;

}

}

EightQueen( 0, 8, chess );

printf("总共有 %d 种解决方法!\n\n", count);

return 0;

}

# 35字符串

## 字符串

以前的计算机刚被发明的时候，主要作用是做一些科学和工程的计算工作，科学家发明计算机的时候压根儿不可能想到后人还可以用来看毛片。

刚开始的计算机都是处理数值工作，后来引入了字符串的概念，计算机开始可以处理非数值的概念了（当然原理还是用数值来模拟非数值，通过ASCII码表）。

鉴于大家对字符串已经有足够的了解，我们这几节就需要讲点不一样的东西才能满足大家的口味，我们会谈谈BF算法和KMP算法。

我们先来研究下“串”这样的数据结构：

定义：串（String）是由零个或多个字符组成的**有限**序列，又名叫字符串。

一般记为 s =“a1a2a3......an”（n>=0）

串可以是空串，即没有字符，直接由 ””表示（注意里边没有空格哦~），或者可以用希腊字母Φ来表示（读fai，四声）。

子串与主串，例如“FishC”是“FishC.com”的子串，反之则倒过来。

关于子串和主串，小甲鱼在做课件搜集资料的时候还发觉一些有意思的：“over”是“lover”的子串，“end”是“friend”的子串，“lie”是“believe”的子串。可见造字者对这世态是看得是清清楚楚明明白白真真切切！

相比起来中国的汉字比较有内涵，例如讲男女同睡在一张床上，我们可以仅仅只用“非、羽、臼、日”四个字来概括。

自己发挥想象力哈~

## 字符串的比较

字符串比较大小跟传统的数字比较有点差别，很容易我们可以知道2比1要大，可要是“FishC”和“fishc.com”呢？要怎么比较？比长短？比大小？

比大小！没错，比的就是字符串里每个字符的ASCII码大小，因为‘F’== 70 ‘f’== 102，‘f’>‘F’，所以“fishc.com”>“FishC”

其实这样的比较大小没有多大意义，字符串的比较我们更重视是否相等！

## 字符串的存储结构

字符串的存储结构与线性表相同，也分顺序存储结构和链式存储结构。

字符串的顺序存储结构是用一组地址连续的存储单元来存储串中的字符序列的。

按照预定义的大小，为每个定义的字符串变量分配一个固定长度的存储区，一般用定长数组来定义。

与线性表相似，既然是固定长度的存储区，就存在一个空间分配不灵活的问题，那么会考虑用链式存储结构。

不同的是字符串我们一般都是连在一起表述的，“断章取义”的情况并不多，所以习惯上我们通常还是会直接定义一个足够长度的存储区来存储的。

## BF算法

事实上，BF算法并不是“BoyFriend”算法的意思，他有一个很黄很暴力的原名：Brute Force

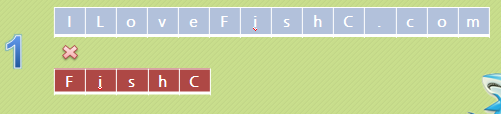
BF算法属于朴素的模式匹配算法，它的核心思想是：

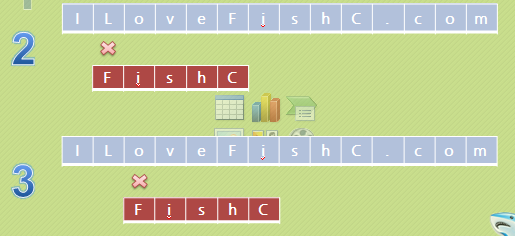
有两个字符串S和T，长度为N和M。首先S[1]和T[1]比较，若相等，则再比较S[2]和T[2]，一直到T[M]为止；若S[1]和T[1]不等，则T向右移动一个字符的位置，再依次进行比较。

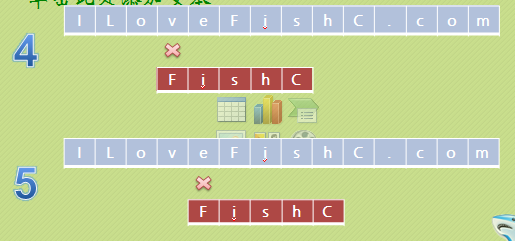
该算法最坏情况下要进行M\*(N-M+1)次比较，时间复杂度为O(M\*N)。

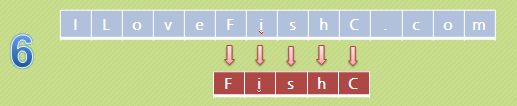
在这里S是主串，T是子串，这种子串的定位操作通常称作串的模式匹配。

假设我们要从主串S=“IloveFishC.com”找到T=“FishC”这个子串的位置，按照BF算法，我们需要进行下边的步骤：









* 课后作业：
  + 用你熟悉的一门语言写出BF算法的实现形式，参考答案详见课件与源代码。
  + 考虑为什么BF算法很黄很暴力，但效率低下？

Bf.c

// 返回子串T在主串S中第pos个字符之后的位置

// 若不存在，则返回0

// T非空，1 <= pos <= strlen(S)

// 注意：我们这里为了表述方便，字符串使用了第一个元素表示长度的方式。

int index( String S, String T, int pos )

{

int i = pos; // i用于主串S中当前位置下标

int j = 1; // j用于子串T中当前位置下标

while( i <= S[0] && j <= T[0] ) // i或j其中一个到达尾部即终止搜索！

{

if( S[i] == T[i] ) // 若相等则继续下一个元素匹配

{

i++;

j++;

}

else // 若失配则j回溯到第一个元素从新匹配

{

i = i-j+2; // i回溯到上次匹配首位的下一个元素，这是效率低下的关键！

j = 1;

}

}

if( j > T[0] )

{

return i - T[0];

}

else

{

return 0;

}

}