**递归**

**定义：程序直接或间接调用自身的算法称为递归（ recursion）。**

递归通常把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解，递归策略只需少量的程序就可描述出解题过程所需要的多次重复计算，大大地减少了程序的代码量。

递归的能力在于用有限的语句来定义对象的无限集合。一般来说，递归需要有边界条件、递归前进段和递归返回段。当边界条件不满足时，递归前进；当边界条件满足时，递归返回。

适合用递归方案解决的问题：①有基本事件（边界条件） ②一个或多个比原问题更接近某基本事件的更小问题。

**递归应用**

递归算法一般用于解决三类问题：

(1)数据的定义是按递归定义的。如Fibonacci函数。

(2)问题解法按递归算法实现。

这类问题虽则本身没有明显的递归结构，但用递归求解比迭代求解更简单，如Hanoi问题、n阶台阶问题、Spock先生难题等。

(3)数据的结构形式是按递归定义的。

如二叉树、广义表等，由于结构本身固有的递归特性，则它们的操作可递归地描述。

**递归的缺点**

递归算法解题相对常用的算法如普通循环等，运行效率较低。因此，应该尽量避免使用递归，除非没有更好的算法或者某种特定情况。、因为在递归调用的过程当中系统为每一层的返回点、局部量等开辟了栈来存储，所以递归次数过多容易造成栈溢出等。

因此能用迭代完成的问题就不用递归去做（基本所有的递归都可以用迭代完成，只是迭代的代码量较大）。

**设计递归算法**

1.确定递归公式。

2.确定边界条件。

例题1 斐波那契数列的第n项（0、1、1、2、3、5、8、13、21、34、）

**static** **int** Fibonacci(**int** n){

**int** x=0;

**if**(n==2) x=1;

**if**(n>2)

x=*Fibonacci*(n-1)+*Fibonacci*(n-2);

**return** x;

}

例题2 楼梯有n阶台阶，上楼可以一步上1阶，也可以一步上2阶，编一程序计算共有多少种不同的走法。（n=3时3种方案，n=6时13种方案）

**static int step\_n(int n){**

**int x=1;**

**if(n==2) x=2;**

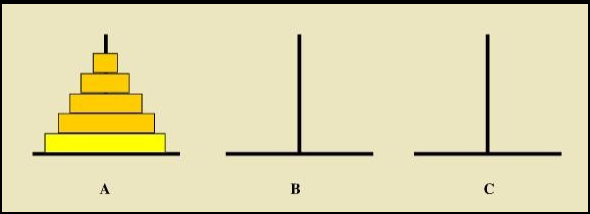
**if(n>2) x=step\_n(n-1)+step\_n(n-2);**

**return x;**

**}**

例题3 汉诺塔问题(Hanoi)

已知有三根针分别用A, B, C表示，在A中从上到下依次放n个从小到大的盘子，现要求把所有的盘子从A针全部移到B针，移动规则是：可以使用C临时存放盘子，每次只能移动一块盘子，而且每根针上不能出现大盘压小盘，找出移动次数最小的方案。（n=4是15种，n=6时63种）



**static** **int** Hanoi(**int** n){

**int** x=1;

**if**(n>1) x=2\**Hanoi*(n-1)+1;

**return** x;

}

例题4 Spock 先生的难题（n件事情中选择k件）

来自外星系的Spock 先生要从太阳系的n颗行星中选取k颗进行访问，共有多少种选法。

**static** **int** Spock(**int** n,**int** k){

**if**(k>n) **return** 0;

**if**(k==n) **return** 1;

**if**(k<n){

**if**(k==0) **return** 1;

**else** **return** *Spock*(n-1,k)+*Spock*(n-1,k-1);

}

**return** 0;

}

例题5 折半查找（查找有序数组中的元素）

**static** **int** binarySearch(**int** []s,**int** first,**int** last,**int** value){

**int** index;

**if**(first > last) index=-1;

**else**{

**int** mid=(first+last)/2;

**if**(value==s[mid]) index=mid;

**else** **if**(value<s[mid]) index=*binarySearch*(s,first,mid-1,value);

**else** index=*binarySearch*(s,mid+1,last,value);

}

**return** index;

}

例题6 快速排序

快速排序的思想是：先从数据序列中选一个元素,并将序列中所有比该元素小的元素都放到它的右边或左边，再对左右两边分别用同样的方法处之直到每一个待处理的序列的长度为1,处理结束。

**static** **int** partition(**int** []s,**int** left,**int** right){

**int** point =s[left];

**while** (left<right){

**while** (left<right && s[right]>=point)

right--;

**if**(left<right) s[left++]=s[right];

**while** (left<right && s[left]<=point)

left++;

**if**(left<right) s[right--]=s[left];

}

s[left]=point;

**return** left;

}

**static** **void** quicksort(**int** []s,**int** left,**int** right){

**int** mid;

**if**(left<right){

mid=*partition*(s,left,right);

*quicksort*(s, left, mid-1);

*quicksort*(s, mid+1, right);

}

}

例题7 二叉树的创建及遍历

**private** Node root;

// 内部节点类

**private** **class** Node{

**private** Node left;

**private** Node right;

**private** **int** data;

**public** Node(**int** data){

**this**.left = **null**;

**this**.right = **null**;

**this**.data = data;

}

}

**public** BinaryTree(){

root = **null**;

}

//递归创建二叉树

**public** **void** buildTree(Node node,**int** data){

**if**(root == **null**){

root = **new** Node(data);

}**else**{

**if**(data < node.data){

**if**(node.left == **null**){

node.left = **new** Node(data);

}**else**{

buildTree(node.left,data);

}

}**else**{

**if**(node.right == **null**){

node.right = **new** Node(data);

}**else**{

buildTree(node.right,data);

}

}

}

}

//前序遍历

//根节点 > 左子树 > 右子树

**public** **void** preOrder(Node node){

**if**(node != **null**){

System.*out*.print(node.data+" ");

preOrder(node.left);

preOrder(node.right);

}

}

//中序遍历

//左子数 > 根节点 > 右子树

**public** **void** inOrder(Node node){

**if**(node != **null**){

inOrder(node.left);

System.*out*.print(node.data+" ");

inOrder(node.right);

}

}

//后序遍历

//左子树 >右子树 > 根节点

**public** **void** postOrder(Node node){

**if**(node != **null**){

postOrder(node.left);

postOrder(node.right);

System.*out*.print(node.data+" ");

}

}

**public** **static** **void** main(String[] args) {

**int**[] a = {2,4,12,4,21,6,111};

BinaryTree bTree = **new** BinaryTree();

**for** (**int** i = 0; i < a.length; i++) {

bTree.buildTree(bTree.root, a[i]);

}

System.*out*.print("前序遍历结果： ");

bTree.preOrder(bTree.root);

System.*out*.println();

System.*out*.print("中序遍历结果： ");

bTree.inOrder(bTree.root);

System.*out*.println();

System.*out*.print("后序遍历结果： ");

bTree.postOrder(bTree.root);

}

}

例题8 广义表

广义表(Lists，又称列表)是一种非线性的数据结构，是线性表的一种推广。即广义表中放松对表元素的原子限制，容许它们具有其自身结构。它被广泛的应用于人工智能等领域的表处理语言LISP语言中。在LISP语言中，广义表是一种最基本的数据结构，就连LISP 语言的程序也表示为一系列的广义表。

**练习：**

**用递归的方法完成下列问题**

**1.求数组中的最大数。**

**2.有一对雌雄兔，每两个月就繁殖雌雄各一对兔子.问n个月后共有多少对兔 子。**

**3.已知：数列1,1,2,4,7,13,24,44,...求数列的第 n项.。**

**4.结合快速排序写出归并排序。**