1.递归思想

树算法的核心无疑是递归思想和深度优先的问题。我们首先来分析怎么写递归代码，然后深入分析一些典型的二叉树问题。

1.1.递归的特征

递归，大部分人都知道怎么回事，但是代码就是写不出来，所谓”你讲的都对，但我就是不会“。递归的本质仍然是方法调用，不过是自己调用自己，系统给我们维护了不同调用之间的保存和返回等功能。

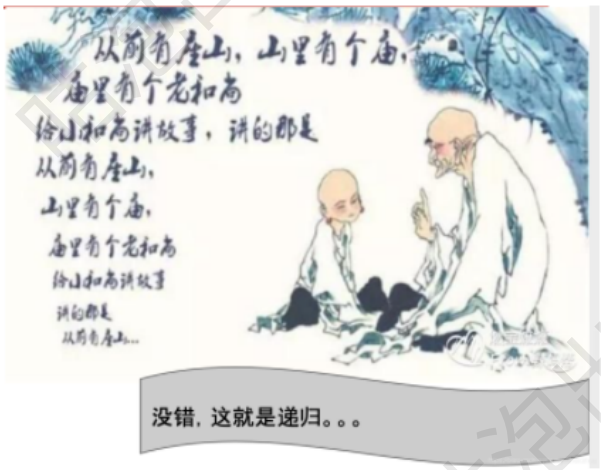
这种例子在现实中也有很多的，例如有一个笑话：

从前啊，有座山，山上有座庙，庙里有个老和尚和一个小和尚在讲故事，老和尚对小和尚说：

从前啊，有座山，山上有座庙，庙里有个老和尚和一个小和尚在讲故事，老和尚对小和尚说：

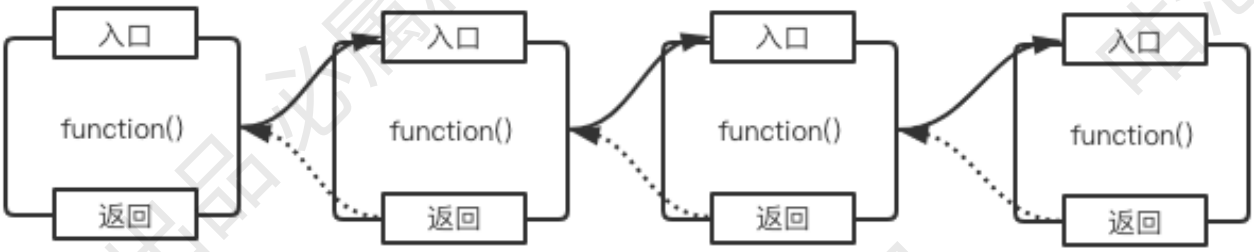
从前啊，有座山，山上有座庙，庙里有个老和尚和一个小和尚在讲故事，老和尚对小和尚说：

.....



再比如下面这张图：

如果看递归代码的结构，就像下面这个样子，前面的每一层都去一模一样地调下一层，不同的只是输入和输出的参数，这个递推过程是写递归的第一个核心问题。



当然这个过程不能一直持续下去，一定要在满足某个要求之后返回结果的，这就是递归的第二个核心问题：确定终止条件。

递归有两个基本的特征：

①执行时范围不断缩小，这样才能触底反弹。
②终止判断在调用递归的前面。

理解这几条特征可以辅助我们更好的理解递归，我们一条条来看：

【1】执行范围不断缩小

递归就是数学里的递推，设计递归就是努力寻找数学里的递推公式，例如阶乘的递推公式就是f(n)=n*f(n-1)，很明显一定是要触底之后才能反弹。再比如斐波那契数列的递归公式为f(n)=f(n-1)+f(n-2)，n也在不断缩小。这条规律可以辅助我们检查递推公式对不对。

再比如青蛙跳台阶的递推公式为：f(n)= f(n-1) + f(n-2)，哎！怎么和斐波那契数列一样？是的，就是一样。

范围缩小不一定只体现n的变化上，在树的递归中我们会大量使用类似这样的结构：

```
int leftDepth = getDepth(node.left); // 左
int rightDepth = getDepth(node.right); // 右
```

每递归一次，都将范围缩小到当前节点的左子树或者右子树，范围也是在不断缩小的。

【2】终止条件判断在递归调用的前面

终止判断条件至少要在递归的前面，看下面例子，很明显这样会一直递归下去，无法退出来，直到抛出堆栈溢出异常 (StackOverflowError)。

```
public void recursion(参数0) {
   recursion(参数1);
  if (终止条件) {
    return ;
  }
}
```

实际一个方法里的递归调用可能不止一次，还会加一些逻辑处理，比如下面这样，但是终止的条件仍然在前面。

```
     public void recursion(参数0) {
        if (终止条件) {
            return;
        }
        //可能有一些逻辑运算1
        recursion(参数1);
        // 可能有一些逻辑运算1
        recursion(参数2);
        // ......3
        recursion(参数n);
        // 可能有一些逻辑运算
    }
```

这一特点启示我们，可以先考虑清楚什么情况下终止，而且相关代码要写在靠前位置的，之后再考虑递归的逻辑，这样可以降低编写的难度。

1.2 如何写递归

明白了上面的道理，那么该怎么才能写出递归方法呢?

第一步：从小到大递推

递归该怎么写呢？递归源自数学里的归纳法，这个在高中数学中学过，大致就是你先猜测出存在递归关系，f(n)=δf(n-1)，然后你只要证明当n增加1时，f(n+1)=δf(n)也是成立就说明你的猜测是对的。不过我们写递归一般不需要证明，先选几个较小的值验一下，再选择几个比较大的验一下即可。

很明显，大部分从n=1，2，3或者只有一两个元素开始写最简单。例如斐波那契序列为 1 1 2 3 5 8，...，从n=3开始都满足f(n)= f(n-1) + f(n-2)，然后我们再选择某个比较大的n来验证即可。

我们仍然以阶乘和斐波那契数列为例来看。斐波那契数列的是这样一个数列：1、1、2、3、5、8、13、21、34....，即第一项 f(1) = 1,第二项 f(2) = 1.....，很明显就是第 n 项目为 f(n) = f(n-1) + f(n-2)。阶乘也一样：

```
n=1 f(1)=1
n=2 f(2)=2*f(1)=2
n=3 f(3)=3*f(2)=6
n=4 f(4)=4*f(3)=24
....
```

由此我们可以推测递推公式是f(n)=n*f(n-1)。

第二步：分情况讨论，明确结束条件

我们说过递归里终止条件一定是靠前的，而大部分递归的终止条件不过是n最小开始触底反弹时的几种情况。例如n=0，n=1等等，例如对于1+2+3+..累加的话，终止条件就是：

```
int f(int n){
    if(n ==1){
        return 1;
    }
    if(n ==2){
        return 2;
    }
}
```

对于阶乘，当 n = 1 时你就应该知道 f(1) = 1，也就是下面这样子：

```
// 算 n 的阶乘(假设n不为0)
int f(int n){
    if(n == 1){
        return 1;
    }
}
```

有时候需要考虑的终止条件不止一个，例如斐波那契数列的递推公式f(n) = f(n-1) + f(n-2)里，如果n=2时会出现f(2)=f(1)+f(0)，很明显这里是没有f(0)的，所以我们要将n==2也给限制住，所以结束条件是这样的：

```
int f(int n){
    if(n <=2){
        return 1;
    }
}
```

有些情况不一定是触底才开始反弹，而是达到某种要求就要停止，这样需要考虑的情况会比较多。解决这类问题最直接的方式就是枚举，将可能的情况列举一下，再逐步优化。只有列举清楚了才可能将终止条件写完整，所以在面试的时候千万不要上来就写，而应该先和面试官讨论你的设计方案，不要害怕与面试官讨论！假如有明显的缺陷他甚至会提醒你的，所以这也是借力打力的一个技巧。

确定终止条件对于递归至关重要，后面很多题目会花很大的篇幅来分析怎么判断终止条件，而一旦判断完毕，递推关系也就水到渠成了。

第三步：组合出完整方法

将递推公式和终止条件组合起来，变成完整的方法。递归经常能看到很多骚操作代码，不要迷信这些，先分情况逐个先写出来，之后再看能否精简优化，不要步子太大，否则会扯到Dan。这里的阶乘和斐波那契数列还是能比较容易找到的，继续完善我们的代码，如下：

```
// 算 n 的阶乘(假设n不为0)
int factorial(int n){
    if(n ==1){
        return n;
    }
    // 把 f(n) 的等价操作写进去
    return factorial(n-1) * n;
}
```

斐波那契数列的实现为：

```
int fibonacci(int n){
    // 1.先写递归结束条件
    if(n <= 2){
        return 1;
    }
    // 2.接着写等价关系式
    return fibonacci(n-1) + fibonacci(n - 2);
}
```

每次写递归的时候，你就强迫自己试着去从这几个方面来考虑，后面我们会不断通过题目来感受上面这个过程。上面还有个问题没有介绍，就是入参和出参怎么确定，这个没有定论，一般是根据题目要求先将能写的写出来，之后再逐步补充，后面在题目里再看。

第四步 从大到小 画图推演

递归有个问题经常让人晕：返回之后每层计算的参数是什么，这个问题经常是debug也会晕。

我们先思考一个问题，上面的阶乘，如果n=4会调几次上面的factorial()方法呢？很明显应该是4次，递归的特征就是“不撞南墙不回头”，n=4，3和2时会继续递归，而n=1时发现满足退出条件了，就执行return 1，不再递归，而是不断返回上一层并计算。

接着再看返回时每层参数的问题，递归本质上仍然是方法调用，所以可以按照方法调用的方式来验证写的对不对。下面这个图完整的表示了求阶乘的过程，你会发现递归不过是一个方法被调了好几次，每次n都在减小，这就是递进的过程。触底之后，也就是满足终止条件之后就开始返回了。递进的时候当前层的n被系统给保存了，而返回的时候会自动设置回来，因此每层的n自然是不一样的，所以此时就是重新拿到当前这一层n的值完成计算即可。

例如我们将f(4)阶乘的过程如下：

通过这个图你会发现所谓的递归与普通的方法调用没什么区别，只不过我们平时是f(n)里调用g(m)，这里是f(n)继续调用自己而已，没有任何神秘的。

2.递归与树

内容来自《树和递归》一章，待补充

3.递归与回溯

内容来自《树和回溯》一章，待补充

4.递归与动态规划

内容来自《树和动态规划》一章，待补充

5.总结

二叉树是我们算法面试的绝对重点，涉及的题目多，有些难度还挺大，如果不准备，面试时甚至都不知道怎么做。

经过上面的学习，我们也可以看到，很多题目都是有规律可循的，最大的规律就是二叉树的特征，以及其遍历方法。我们按照层次遍历、前序后序遍历、中序与搜索树三个章节分析了30多道高频题目的解决方法。很多题目我们给出了不止一种方法。本人认为这些方法大部分都应该掌握，至少理解用别的方法结解题的思路是什么。只有这样才能融会贯通，刷清楚一道题，10道题就不用做了。