文章编号：2096-1618(2017)增-0005-03

浅谈递推与递归在程序设计中的合理应用

高正杰，　尹兴宝
(成都信息工程大学，四川 成都 610225)

摘要：总结了递推法与递归法的本质特征，具体分析使用递推法和递归法求解各类问题时可能出现的算法复杂度，在对两者进行比较的基础上，提出了问题求解时选择递推算法还是递归算法的一般性原则。

关　键　词：递推；递归；程序设计；关系；算法

0　引言

递推和递归在计算机科学和数学中是很重要的工具，在程序设计语言中用来定义句法，在数据结构中用来解决表或树形结构的搜索和排序问题，数学家在研究组合问题时也要用到递归和递推。递归和递推是非常重要的课题，在计算方法、运筹学模型、行为策略和图论的研究中，从理论到实践，都得到了广泛的应用。但是他们的概念经常容易被混淆。因此，深刻理解递归与递推的含义，掌握其中的规律，对于设计高质量的算法程序是非常重要的。

1　递推

1.1　递推的定义

一个状态可以和前面的状态联系起来，具体表示为：一个数的序列 $H_1, H_2, H_3 \cdots\cdots H_n$，其中 H_n 可用 $H_1 \cdots\cdots H_{n-1}$ 来表示[1]。

2　递归

2.1　递归的定义

若一个对象部分地包含它自己，或用它自己给自己的定义，则称这个对象是递归的；若一个过程直接或间接地调用自己本身，则称这个过程是递归的过程。

2.2　递归算法及其结构

递归算法作为计算机程序设计中的一种重要的算法，是比较难理解的算法之一。递归过程由于实现了自我的嵌套执行，使得该过程的执行变得复杂起来，其执行流程如下图所示：

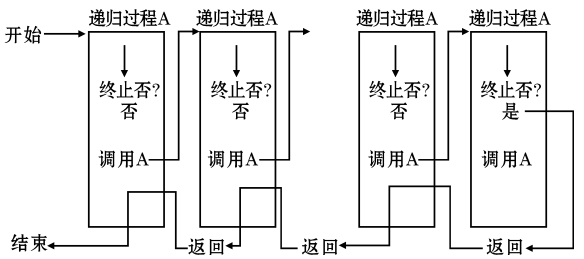

图 1　递归过程的执行流程

从图 1 中可以看出，递归过程的执行总是一个过程体未执行完，就带着本次执行的结果又进入另一轮过程体的执行，如此反复，不断深入，直到某次过程的执行遇到终止递归调用的条件成立时，才不再深入，然后执行本次的过程体余下的部分，如此反复，直到回到起始位置上，才最终结束整个递归过程的执行，得到相应的执行结果。递归算法的程序设计的核心就是参照这种执行流程，设计出一种适合“逐步深入，而后又逐步返回”的递归调用模型，以解决实际问题。

3　经典问题引入

3.1　斐波那契数列问题

已知 Fibonacci 数列的定义为

$$\begin{cases} F_1 = F_2 = 1 \\ F_n = F_{n-2} + F_{n-1} \quad (n > 2) \end{cases}$$

3.2　汉诺塔问题

有 A、B、C 3 根柱子和 64 个大小有别的盘子，现在所有盘子都从小到大地叠放在 A 柱上。要求每次只能搬动一个盘子，把所有盘子搬到 C 柱上，搬动的过程中可以利用 B 柱临时存放盘子，但任何时候任何柱子上都必须保证大盘子在下，下盘子在上。

收稿日期：2016-03-16

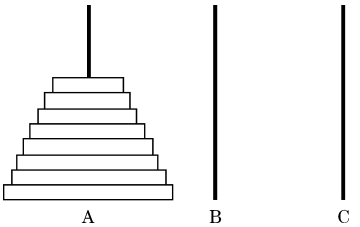

图 2 汉诺塔问题

4 递推法——在有向图中搜索路径

从递推法的定义可以知道,这种方法总是从一个确定的初始状态出发,经过一系列的状态转换,最终达到某一终止状态而使问题得到解决,求解过程中涉及的各个状态及其相互关系构成了一个有向图。

以 Fibobacci 数列问题为例,可以定义每个状态由数列的第 i 项和第 i+1 项构成,记作(F_i,F_{i+1}),问题的初始状态就是(F_1,F_2),对于一个问题中给定的整数值 n,终止状态是(F_{n-1},F_n),其状态转换图如图 3 所示。

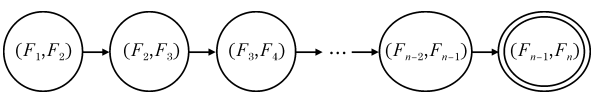

图 3 递推法求解 Fibonacci 问题

从图 3 可以看出,由某一状态转到下一状态只有唯一的可能性,从初始状态到终止状态只有唯一的路径,并且经过若干次状态转换后必然能达到问题要求的终止状态。因此,用递推法求解 Fibonacci 问题非常简单、明了、直接,其算法复杂度很小,为 $O(n)$。

但是,用递推法求解 Hanoi 塔问题时就遇到了很大的困难。Hanoi 塔问题的一个状态可以用各个柱子上放有盘子的情况来表示,初始状态是[(64,63,…,2,1),(),()],而终止状态也不难描述,即[(),(),(64, 63, …, 2, 1)]。困难在于状态转换不像 Fibonacci 数列问题中只有唯一的可能性,而是有两种或两种以上的可能,如图 4 所示。整个状态转换图非常庞大,图中包含了 3^n 个状态,终止态只是这些大量状态中的一个,而问题的解决则依赖于在如此庞大的有向图中,用探索的方法找打一条从初始状态到终止状态的路径。因此要用遍历算法在有向图中查找某一状态(节点),算法的复杂度为 $O(3^n)$,当 $n=64$ 时,世界上最快的计算机也难将这项工作完成。

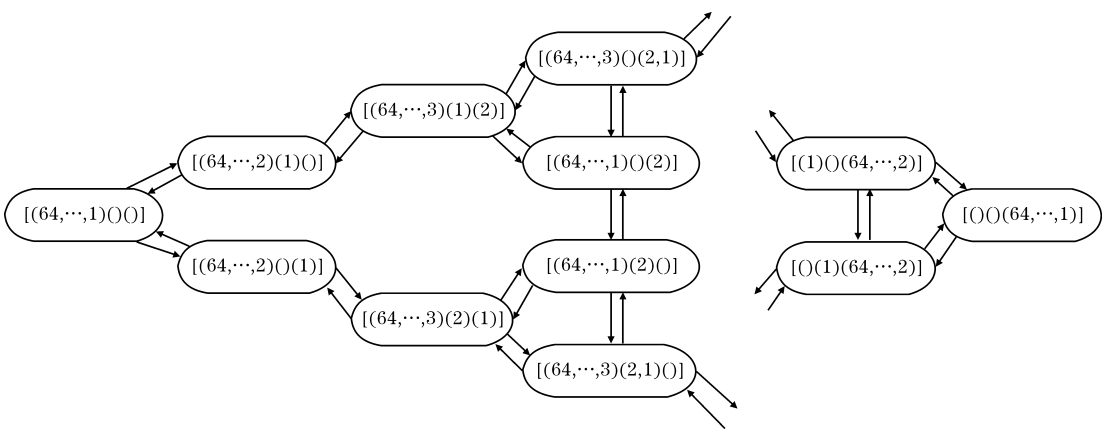

图 4 递推法求解 Hanoi 问题对应的有向图

5 递归法——对树进行后序遍历

与递推法不同,递归法是把问题的处理作为树来看待,以问题的结果作为根节点,把待求解的问题分解成几个子问题,每个子问题作为根的一个子节点,如果树的某个节点需要再次分解,则再往下链接几个子节点。一个节点及其所有兄弟节点对应的子问题解决以后,它们可以共同地解决其父节点对应的问题,整个问题的求解过程就是对整棵树的后序遍历过程。

在 Fibonacci 问题中,树根是 F_n,每个节点 F_i 都有两个子节点 F_{i-1} 和 F_{i-2},如图 5 所示。这是一颗二叉树,层数为 n,树中节点数目为 2^n-1,因而后序遍历的算法复杂度为 $O(2^n)$,这显然比递推法求解同一问题要差得多。当然,图 5 中的节点有严重的重复设置的现象,这是造成算法复杂度过大的根本原因。

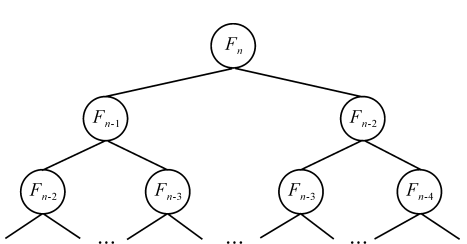

图 5 递归求解 Fibonacci 问题对应的树

在 Hanoi 问题上,递归方法明显地优于递推方法,图 6 是 Hanoi 塔问题对应的树,用(n, p_1, p_2, p_3)描述

问题“把 $n(n>1)$ 个盘子以 p_3 柱作为过渡从 p_1 柱搬到 p_2 柱”,这种节点需要进行分解,图中用长方形节点表示。另一种节点是<u, v>,表示“把 u 柱最上面的一个盘子搬到 v 柱”,这已经是简单操作,不需要再次分解,是树的叶节点,图中以圆形节点表示。因此树根可以描述成(64, A, C, B),它包含 3 个子节点:(63, A, B, C)、<A, C >(63, B, C, A),其中中间一个节点<A, C>是叶节点,而另外的两个节点还需要进一步分解,是中间节点。

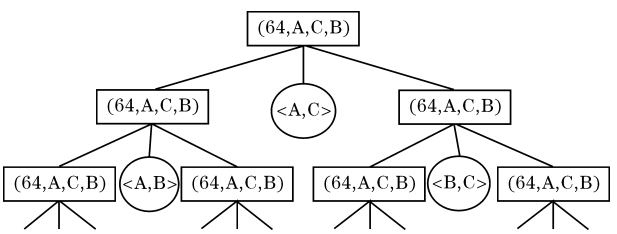

图 6　递归求解 Hanoi 问题对应的树

表面上看,Hanoi 塔问题还可以有其他分解方案,比如把根节点分解成<A, B>、(63, A, C, B)和<B, C >,意为先把最小的盘子从 A 搬到 B,然后把剩下的 63 个盘子从 A 搬到 C,再把最小的盘子从 B 搬到 C,但这种分解方案导致第二步“把剩下的 63 个盘子从 A 搬到 C”时违反了问题的规则,因为此时 B 柱上有一个最小的盘子,任何盘子搬到 B 上暂时过渡都会违反“上小下大”的规则。全面考虑之后发现,对根节点的分解方案是唯一的。推而广之,每一个子问题的进一步分解也只存在唯一的方案,整个树的结构是唯一的。

因此,可以计算出图中中间节点和叶节点的树木各为 2^n-1,n 是盘子的数目。因此,对树的遍历算法复杂度是 $O(3^n)$ 相对显然要低一些。

6　递推与递归的比较

人们在现实生活中遇到问题时一般会有两种思维模式:一种是从最简单的情形出发,利用各种方法进行尝试,看能否比较容易地推导出结果;另一种方法是把复杂的问题先分解成几个小问题,如果分解出的几个小问题都能求得答案,就可以把各个小问题的答案组合成原问题的解;如果分解出的几个子问题中,仍然有比较复杂的问题,那就将其再次分解,直到最后分解出的子问题可以直接解得,然后再按分解的逆过程对答案进行组合还原,直至求出原问题的解。这两种思维模式分别是递推和递归。尽管人们总是习惯遇到问题先用递推方法去是试探,但在试探多次未果的情况下就会另辟蹊径,此时往往就会采用递归的思路。所以说,递归与递推二者均有存在的价值。

递推是利用问题本身所具有的递推关系对问题进行求解的一种方法,往往以循环的形式出现,编写程序时必须明确写出循环内每一步的具体实现。与之相反,递归算法只需要写出问题的分解形式和组合答案的方法,至于如何递归求解各个子问题则由系统帮助解决。因此用递归算法编写的程序会很简洁,但是这样的程序不易于理解,因为递归的具体过程对程序员来说是透明的。

递推与递归的一个相似之处在于,他们都是设计同类问题的相似重复,重复操作在达到一定条件后就终止。而重复正是计算机的主要特长。

在程序设计中要使用递归算法必须考虑这样几个方面:

(1)用计算机语言描述如何将问题分解,并且让计算机记住分解出的是几个子问题。

(2)计算机对分解出的每个子问题如何处理,对分解出的简单问题如何求解,同类型的问题如何重复。

(3)当分解出的问题满族什么条件时不再继续分解。

(4)计算机如何把各个子问题的答案逐次组合成最终答案。

关于递归所使用到的存储机制——堆栈,以及递归与栈之间的关系,笔者不再一一赘述,详见参考文献[2]。

7　结束语

递归算法与递推算法在思维方式是相反的,递推算法比递归算法更符合人的思维习惯。在实际应用递归算法解决问题时,算法的复杂度会比较大,需要大量的时间和空间,而且递归方法在理解和掌握上有一定的困难。但不可否认,它是一种很有价值的问题求解方法。另一方面,针对实际问题时通常需要把递推方法和递归方法结合起来使用,可以大大提高解决问题的效率,对提高程序设计水平也将大有裨益!

参考文献:

[1]　Kenneth H. Rosen. Discrete Mathematics and Its Applications[M]. 北京:机械工业出版社,2007:329.

[2]　严蔚敏,吴伟民. 数据结构(C 语言版)[M]. 北京:清华大学出版社,2007:54-58.