第3章

扫一扫

视频讲解

递　　归

扫一扫

思政教学案例

本章学习要点

- 递归的概念
- 递归的两个基本要素
- 通过典型应用学习递归算法设计策略
  - 阶乘
  - 汉诺塔
  - 全排列
  - 整数划分

“递”有更替、传送之意，而“归”有返回、趋向或集中之意，“递归”就是一个不断传递进去又回来的过程。在计算机范畴内，递归是在其定义中直接或间接调用自身的一种方法，如数据结构中对于二叉树的定义，就是一个递归定义——二叉树是一棵空树，或者是一棵由一个根节点和两棵互不相交的分别称作根的左子树和右子树组成的非空树；左子树和右子树又同样都是二叉树。

通过最后这句话，可以很直接地感受到这是一个递归的定义。类似的递归过程在数学中也很常见，如阶乘和斐波那契数列等。

3.1　引例：阶乘

问题描述　一个正整数的阶乘是求所有小于及等于该数的正整数的积，定义 0 的阶乘为 1。自然数 n 的阶乘写作 $n!$，也可以表示为

$$n! = \begin{cases} 1, & n = 0 \\ n(n-1)!, & n > 0 \end{cases}$$

从问题描述中可以看出，对于阶乘的定义是一个递归定义，每次计算都是用当前数值乘以前次阶乘的结果。由此可以得到利用递归进行直接求解的算法，用 Fact(n)函数表示对 $n!$ 的计算，当 $n=0$ 时，直接返回 1，否则返回 nFact($n-1$)。

以 $n=5$ 为例，递归计算过程如图 3-1 所示。先计算 Fact(5)，进入函数后，需要返回 5Fact(4)，这里又需要 Fact(4)的结果，则进入 4Fact(3)，以此类推，进入一个递推的过程，当进入 1Fact(0)时，计算 Fact(0)，因为 $n=0$，直接返回 1；然后进入回归阶段，首先得到 Fact(1)的结果，接着得到 Fact(2)、Fact(3)、Fact(4)，直到最后得到 Fact(5)的值为 120，通过递推和回归两个阶段完成整个的递归计算过程。

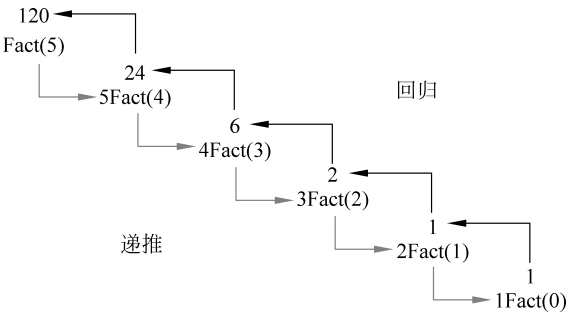

图 3-1　阶乘的递归执行过程

扫一扫

看彩图

扫一扫

视频讲解

3.2　递归的基本思想

在上述阶乘的求解过程中，通过递归的过程把一个不好解决的大问题(5!的求解)转化为一个或几个小问题(4!的求解)，再把这些小问题进一步分解成更小的小问题(3!的求解)，直至每个小问题都可以直接解决(0!的求解)。在递归过程中，计算如何传递，以及什么阶段开始返回是两个重点，这也对应了递归中的两个基本要素：边界条件和递归式。递归函数只有具备了这两个要素，才能在有限次计算后得到结果。

边界条件：确定递归到何时终止，也称为递归出口。

递归式：大问题是如何分解为小问题的，也称为递归体，通常以递推方程的形式给出。

在设计递归算法时，要牢记这两个基本要素。递归式是算法的主体执行过程，边界条件可以确保递归过程能够结束。

扫一扫

视频讲解

3.3　递归应用：汉诺塔问题

问题描述　设 A、B、C 是三个塔座，开始时，在 A 塔座上有一叠 n 个自下而上由大到小叠在一起的圆盘，这些圆盘按从小到大编号为 1，2，…，n，如图 3-2 所示。现要求将 A 塔座上的这一叠圆盘移动到 B 塔座上，并仍按同样的顺序叠放。

扫一扫

看彩图

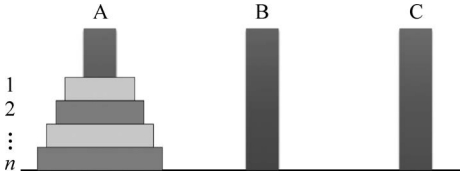

图 3-2　n 阶汉诺塔

移动规则是：①每次只能移动一个圆盘；②任何时刻都不允许将较大的圆盘压在较小的圆盘之上；③在满足前两条移动规则的前提下，可将圆盘移至 A、B、C 中任意塔座上。

问题分析　最简单的情况是当只有一个圆盘时，可以将这个圆盘直接从 A 塔座移动到 B 塔座；当不止一个圆盘时，需要利用 C 塔座实现辅助移动，可以先将 $n-1$ 个较小的圆盘依照规则从 A 塔座移动到 C 塔座；接着将剩下的最大圆盘从 A 塔座移至 B 塔座；最后将 $n-1$ 个较小的圆盘依照规则从 C 塔座移至 B 塔座。经过这样的 3 步后，就能够完成 n 个圆盘的移动。在此过程中，将 n 个圆盘移动问题分解为两次 $n-1$ 个圆盘移动问题；而对于 $n-1$ 个圆盘的移动，又可以利用上面的过程，去移动两次 $n-2$ 个圆盘，以此类推。

算法思想　结合上述分析过程，可以得到这样的一个递归算法，定义一个 Hanoi()函数，输入为 n，以及三个塔座 A、B、C。当 $n>0$ 时，首先将 $n-1$ 个圆盘从 A 塔座移动到 C 塔座，借助 B 塔座完成这个移动，接着直接将第 n 个圆盘从 A 塔座移动到 B 塔座，最后再将 $n-1$ 个圆盘从 C 塔座移动到 B 塔座，借助 A 塔座完成这个移动。

算法实现　具体实现如代码 3-1 所示，对应了算法思想中的 3 个步骤。

代码 3-1：汉诺塔问题的实现

```
def Move(n, A, B):                    #输出函数,第 n 个圆盘从 A 塔座移动到 B 塔座
    print(n, ':', A, '-->', B)

def Hanoi(n, A, B, C):                #递归函数
    if n > 0:                         #边界条件
        Hanoi(n - 1, A, C, B)
        Move(n, A, B)
        Hanoi(n - 1, C, B, A)

if __name__ == '__main__':
    A = 'A'
    B = 'B'
    C = 'C'
    n = int(input("请输入圆盘个数: "))
    Hanoi(n, A, B, C)
```

以 3 阶汉诺塔为例，分析递归的执行过程。初始状态如图 3-3 所示。

首先，执行 Hanoi(3,A,B,C)函数，进入函数后，3>0，执行 Hanoi(2,A,C,B)函数；2>0，执行 Hanoi(1,A,B,C)函数；1>0，执行 Hanoi(0,A,C,B)函数，此时不满足条件，直接返回，回到 Hanoi(1,A,B,C)函数的第 2 步执行 Move(1,A,B)函数，得到状态如图 3-4 所示。

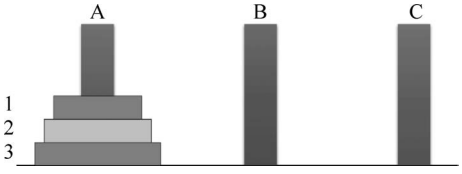

图 3-3　3 阶汉诺塔初始状态

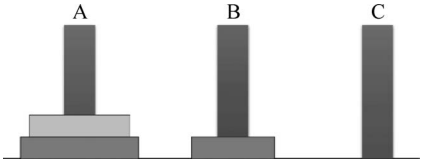

图 3-4　3 阶汉诺塔第 1 次移动

扫一扫 看彩图

接着执行 Hanoi(1,A,B,C)函数的第 3 步 Hanoi(0,C,B,A)函数，不满足条件，直接返回，此时 Hanoi(1,A,B,C)函数执行完成；进入 Hanoi(2,A,C,B)函数的第 2 步 Move(2,A,C)函数，得到状态如图 3-5 所示。

扫一扫 看彩图

接着执行 Hanoi(2,A,C,B)函数的第 3 步 Hanoi(1,B,C,A)函数，进入后会执行 Move(1,B,C)函数，得到状态如图 3-6 所示。

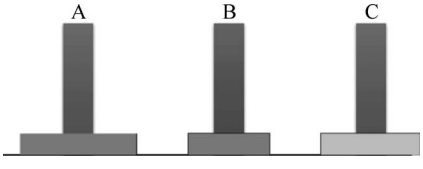

图 3-5　3 阶汉诺塔第 2 次移动

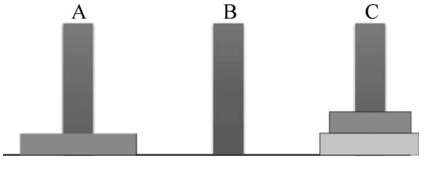

图 3-6　3 阶汉诺塔第 3 次移动

扫一扫 看彩图

此时，Hanoi(2，A，C，B)函数执行完毕，接着执行 Move(3，A，B)函数，得到状态如图 3-7 所示，以此类推，完整的执行调用过程如图 3-8 所示。

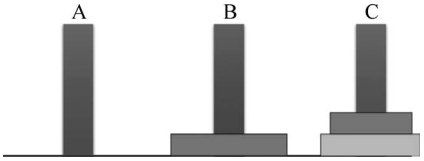

图 3-7　3 阶汉诺塔第 4 次移动

扫一扫

看彩图

整个执行过程并不是并行执行，而是逐步递进的执行过程，图 3-8 中函数的标号顺序表明该函数结束计算的顺序，标号越小说明函数越早结束计算。

算法分析　结合上述过程，可以得到代码 3-1 对应的递推式为 $T(n)=2T(n-1)+1$，计算后可得 $T(n)=2^n-1$，时间复杂度为 $O(2^n)$。由此可知，汉诺塔问题是一个难解问题，即使使用当今性能最佳的高性能计算机，当 n 接近 100 时，也无法在有效时间内求解。

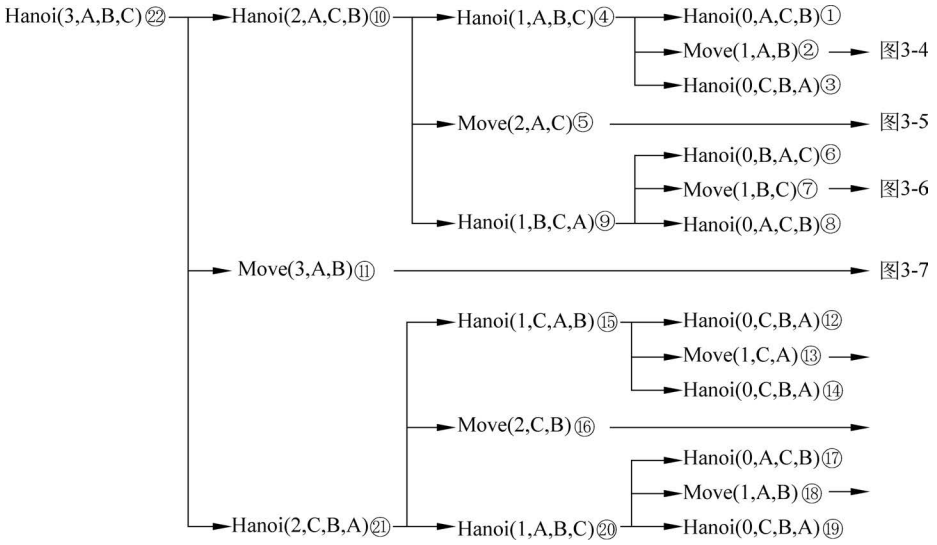

图 3-8　3 阶汉诺塔执行过程

扫一扫

视频讲解

3.4　递归应用：全排列

问题描述　设 $R=\{r_1,r_2,\cdots,r_n\}$ 为要进行排列的 n 个元素，各元素不重复，设计一个算法，输出这 n 个元素的全排列。

问题分析　以 $n=3$，$R=\{1,2,3\}$ 为例，所有全排列情况如下。

1　2　3
1　3　2
2　1　3
2　3　1
3　1　2
3　2　1

观察一下这样的排列，如以 1 开头，后面跟的就是 2 和 3 的两种排列，即除 1 外剩余元素进行全排列；后面以 2 开头的排列、以 3 开头的排列都具有同样的情况。可以看到，每个元素都会分别作为排列的第 1 个元素，除此之外，后面的排列就是除此元素以外剩余 $n-1$ 个元素的全排列，这种关系就是一种递归关系。

算法思想 为了更清楚地表达全排列的递归关系，做如下定义：设 $R_i=R-\{r_i\}$。集合 R_i 中元素的全排列记为 Perm(R_i)。(r_i)Perm(R_i)表示在全排列 Perm(R_i)的每个排列前加上前缀 r_i 得到的排列。

由此就能得到对于 R 的全排列，当 $n=1$ 时，Perm(r)=(r)，r 是集合中唯一的元素；当 $n>1$ 时，Perm(R)由(r_1)Perm(R_1)，(r_2)Perm(R_2)，…，(r_n)Perm(R_n)构成。

算法实现 根据递归关系，全排列的递归算法如代码 3-2 所示。根据前面的分析，需要由每个元素开始进行全排列，所以在循环中，依次将元素交换到第 1 个位置，然后进行 $k+1$ 到 len 的排列，排列完后，再恢复元素位置，对于 Perm()函数，执行 Perm(a,0,$n-1$)函数就实现了对数组 a 前 n 个元素的全排列。

代码 3-2：全排列的代码实现

```
"""
可直接调用排列函数 permutations()

from itertools import permutations
data = input().split()
for i in permutations(data):
    print("".join(i))
"""

def Perm(a, k, len):
    if k == len:                                #边界条件
        global a_i
        print("第{0}种排列为：{1}".format(a_i, "".join(a)))
        a_i += 1
    else:
        for i in range(k, len + 1):
            a[i], a[k] = a[k], a[i]             #交换 a[i]和 a[k]
            Perm(a, k + 1, len)
            a[i], a[k] = a[k], a[i]             #恢复现场

if __name__ == '__main__':
    a_i = 1
    a = list(input("请输入排列的字符串："))  #string 为不可变类型,因此转为 list
    Perm(a, 0, len(a) - 1)
```

以 $n=3$，$a[3]=\{1,2,3\}$为例，分析实现全排列的过程，如图 3-9 所示。深色线路为递归调用过程，浅色线路为递归返回过程。首先调用 Perm(a,0,2)函数，根据算法执行过程，在计算过程中，需要再调用 Perm(a,1,2)函数，以此类推。通过这样的调用和返回过程，就可以完成全排列。

算法分析 结合代码 3-2，可以得到全排列的递推式为 $T(n)=nT(n-1)+O(1)$，计算后可得其时间复杂度为 $O(n!)$，这是一个比 2^n 还大得多的时间复杂度，所以全排列问题也是一个难解问题。

扫一扫

看彩图

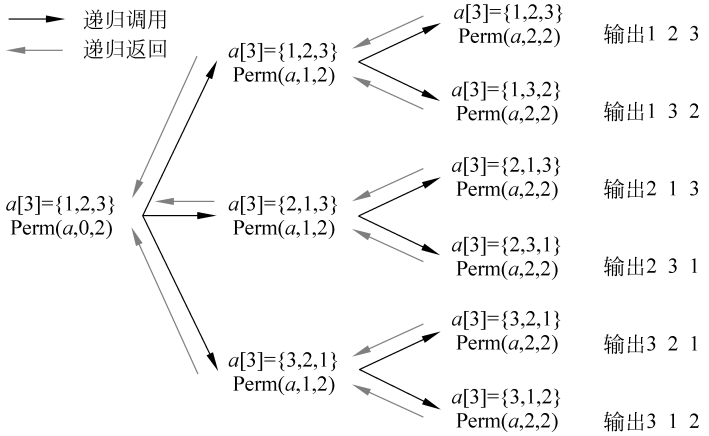

图 3-9 3 个元素的全排列的执行过程

扫一扫

视频讲解

3.5 递归应用：整数划分

问题描述 整数划分问题要求将一个正整数表示成一系列正整数之和。其中，正整数 n 的这样一种表示称为它的一个划分，不同划分个数称为正整数 n 的划分数。现给出一个正整数 n，要求出其划分数。

问题分析 以整数 6 为例，先给出可表示的正整数之和，分别为

$$6$$
$$5+1$$
$$4+2,4+1+1$$
$$3+3,3+2+1,3+1+1+1$$
$$2+2+2,2+2+1+1,2+1+1+1+1$$
$$1+1+1+1+1+1$$

上述划分一共有 11 种，可以将其分为 6 类，对应第 1 个数不同的情况。此外，在确定了第 1 个划分数之后，后续的数都不会超过第 1 个数。若第 1 个数确定为 3，则对 6 的划分就可以转化为对 6－3＝3 的划分，划分方法与对 6 的划分相同。最终，通过对每类划分个数的汇总得到总的划分数。

算法思想 针对上述过程，先定义一个 $q(n,m)$ 函数表示在正整数 n 的所有不同划分中，最大加数不大于 m 的划分个数，根据 n 和 m 的关系，可以得到以下几种情况。

当 $n=1$ 时，此时只有一种情况。

当 $m=1$ 时，表示所有加数最大为 1，也只会有一种情况。

当 $m>n$ 时，表示加数 m 比划分数 n 还大，这是不会出现的情况，因为 m 最大只会和 n 相等，所以此时 $q(n,m)$ 等价为 $q(n,n)$。

对于 $m=n$，可以等价为 n 本身的一种划分和 $m=n-1$ 时的划分之和。

最后，当 $n>m>1$ 时，其实存在两类情况，一类是加数不超过 $m-1$，即不包含 m，应该是 $q(n,m-1)$；另一类是包含加数 m，此时就应该是 $q(n-m,m)$。

由此，可以形成完整的递归关系为

$$q(n,m)=\begin{cases}1, & n=1 \text{ 或 } m=1\\ q(n,n), & n<m\\ 1+q(n,n-1), & n=m>1\\ q(n,m-1)+q(n-m,m), & n>m>1\end{cases}$$

结合递归关系，请读者思考并完成该问题的递归算法实现。整数划分问题还有另外一种提法：给定整数 n，输出它的每种划分，留作习题。

扫一扫

视频讲解

3.6　小结

递归作为一种重要的实现方式，普遍存在于各类算法的实现中。问题能否使用递归求解的重点在于该问题是否可以转换为规模更小的子问题进行求解。带着这样的分析思路，本章对阶乘问题、汉诺塔问题、全排列问题以及整数划分问题进行了详细讲解，重点介绍了递归的两个基本要素：边界条件以及递归式的设计。特别是边界条件，它是递归程序能够正常停止的关键，同时也是编程中容易出错的关键点，在程序实现过程中，设置正确的边界条件尤为重要。

扩展阅读

递　　归[①]

在数学与计算机科学中，递归(Recursion)是指在函数的定义中调用函数自身的方法。顾名思义，递归包含了两个意思：递和归，这正是递归思想的精华所在。递归问题可以分解为若干个规模较小、与原问题形式相同的子问题，这些子问题可以用相同的解题思路来解决，这些问题的演化过程是一个从大到小、由近及远的过程，并且会有一个明确的终点(临界点)，一旦到达了这个临界点，就不用再向更小、更远的地方走去。最后，从这个临界点开始，原路返回到原点，原问题得以解决。

在春秋战国时期曾子所作的《礼记·大学》中有："古之欲明明德于天下者，先治其国；欲治其国者，先齐其家；欲齐其家者，先修其身；欲修其身者，先正其心；欲正其心者，先诚其意；欲诚其意者，先致其知，致知在格物。物格而后知至，知至而后意诚，意诚而后心正，心正而后身修，身修而后家齐，家齐而后国治，国治而后天下平。"这就是早在中国上千年前递归思想的体现。

法国数学家爱德华·卢卡斯曾写过一个古老的印度传说：在贝拿勒斯(在印度北部)圣庙里的一块黄铜板上插着 3 根宝石针。印度教的主神梵天在创造世界时，在其中一根针上从下到上穿好了由大到小的 64 片金片，这就是所谓的汉诺塔。不论白天黑夜，总有一名僧侣在按照一定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。这就是最初的汉诺塔问题。

百余年来，很多研究者和数学爱好者对汉诺塔问题进行了扩展，包括双色汉诺塔问题、多柱汉诺塔问题等。

1. 双色汉诺塔问题

n 个大小不一的圆盘依半径从大到小自下而上地套在塔座 A 上。这些圆盘从小到大

① 参考来源：马昕宇. 正心、修身、齐家、治国、平天下[J]. 文史知识，1987(1)：120.
姚云霞. 对汉诺塔(Hanoi)问题的算法探索与研究[J]. 物联网技术，2013，3(7)：48-49.

编号为1,2,…,n,奇数号圆盘着红色,偶数号圆盘着蓝色。现在要借助塔座C将n个圆盘从塔座A全部转移到塔座B上,并仍按同样顺序叠置。在移动圆盘时应遵循以下条件:

(1) 每次只允许移动一个圆盘;

(2) 在转移过程中不允许大圆盘放在小圆盘上;

(3) 在转移过程中任何时刻都不允许将同色圆盘叠在一起;

(4) 在满足以上移动规则的前提下,可将圆盘移至A、B、C中的任意塔座上。

试设计一个算法,用最少的移动次数将塔座A上的n个圆盘移动到塔座B上,并仍按同样顺序叠置。

2. 四柱汉诺塔问题

四柱汉诺塔问题和三柱汉诺塔问题的唯一区别是增加了塔座D,目标是把塔座A上的n个圆盘通过塔座B和塔座C移动到塔座D上,在其他移动条件不变的情况下,使用最少的移动次数。

上述问题进一步可以扩展为多柱汉诺塔问题,有兴趣的读者可以尝试不同的汉诺塔问题。

从算法思想到立德树人

递归是一个不断重复相同计算的过程,在不断的重复中达到目标。当然,重复也不是一成不变,每次重复计算都在减小计算规模,向最终的目标逼近。

在我们的学习和科研工作中,很多时候也需要有递归的样子,有经得住寂寞完成重复性工作的耐心,在一遍遍的重复过程中,就会不断接近目标。正所谓“不积跬步,无以至千里;不积小流,无以成江海。骐骥一跃,不能十步;驽马十驾,功在不舍。锲而舍之,朽木不折;锲而不舍,金石可镂。”

习题3

扫一扫

习题