第 3 章 递归算法

本章主要内容

- 递归算法简介；
- 线性与非线性递归；
- 问题与子问题；
- 递归与迭代；
- 多重递归；
- 经典递归；
- 优化递归。

递归算法是非常重要的算法，是很多算法的基础。递归算法不仅能使代码优美简练，容易理解解决问题的思路或发现数据的内部逻辑规律，而且具有很好的可读性。递归算法是分治算法思想的重要体现，或者说分治算法思想来源于递归：将规模大的问题逐步分解成规模小的问题，最终解决整个问题。与排序算法不同，许多经典的排序算法已经日臻完善，在许多应用中只需选择一种排序算法直接使用即可（见第 4 章 4.3 节），而对于递归算法，只有真正理解递归算法内部运作机制的细节，才能针对实际问题写出正确的递归算法。所以本书单独设一章讲解递归算法。

为了方便调试代码，本章的例子各自存放在一个单独的目录中，例如，例 3-5 存放在“C:\ch3\例子 5”中。通常用 ALG.py 文件存放例子用到的重要的函数（ALG 是 algorithm 的缩写）。

3.1 递归算法简介

一个函数在执行过程中又调用了自身，形成了递归调用，这样的函数被称为递归函数或递归算法。递归函数是一个递归过程，函数调用自身一次、就是一次递归。每一次递归又导致函数调用自身一次，形成下一次递归。结束递归需要条件，当这个条件满足时递归过程会立刻结束，即在某次递归中函数不再调用自身，结束递归。如果在某次递归中函数不再调用自身，那么此次递归就是函数最后一次调用自身，从递归开始到函数最后一次调用自身，函数被调用的总数记作 $R(n)$（也称递归总数为递归深度），那么 $R(n)$ 是依赖于一个正整数 n 的函数。

假设函数名是 f，下面进一步说明递归过程中的压栈和弹栈的细节。

递归函数 f 的递归过程是这样的：第 k 次调用 f 需要等待第 $k+1$ 次调用 f 结束执行后才能结束本次调用的执行。那么第 $R(n)$ 次（最后一次）调用 f 结束执行后，就会依次使得第 k 次调用 f 结束执行（$k=R(n)-1, R(n)-2, \cdots, 1$），如图 3.1 所示。

函数被调用时，函数的（入口）地址会被压入栈（栈是一种先进后出的结构）中，称为压栈操作，同时函数的局部变量被分配内存空间。函数调用结束，会进行弹栈，称为弹栈操作，同时释放函数的局部变量所占的内存。递归过程的压栈操作会导致栈的长度不断变大，而弹栈操作

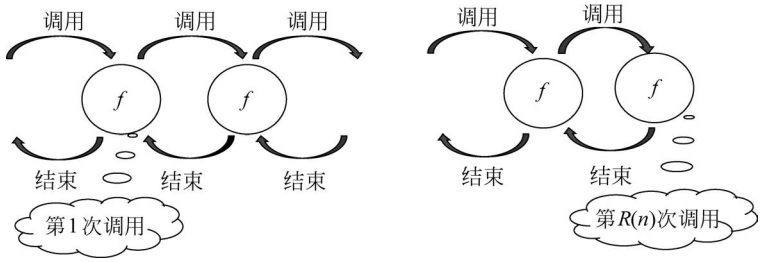


图 3.1　递归执行过程

会导致栈的长度不断变小，最终使栈的长度为 0，如图 3.2 所示，其中用函数的名字 f，表示函数的地址。

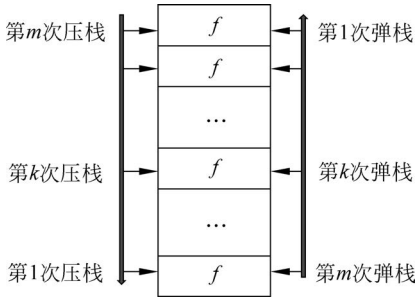


图 3.2　递归过程中的压栈、弹栈

1. 时间复杂度

递归函数是一个递归过程，从递归开始到递归结束，函数被调用的总数 $R(n)$ 是依赖于一个正整数 n 的函数。那么递归函数中基本操作被执行的总次数 $T(n)$ 就依赖于递归的总次数 $R(n)$ 和每次递归时基本操作被执行的总次数。因此要针对具体的递归函数计算其时间复杂度。

2. 空间复杂度

递归过程的压栈操作增加栈的长度，而弹栈操作减小栈的长度。需要注意的是，递归过程中压栈操作和弹栈操作可能交替地进行，直到栈的长度为 0（见例 3-2），所以需要计算出递归过程中某一时刻（某一次递归）栈出现的最大长度和每次递归中函数的局部变量所占的内存空间，即计算出栈的最大长度以及局部变量所占的全部内存空间，明确它们与依赖的正整数之间的关系，才能知道空间复杂度。大部分递归的空间复杂度通常是 $O(n)$，但也有的递归是 $\log_2 n$（见例 3-7）。

注意：递归会让栈的长度不断发生变化，如果栈的长度较大可能导致栈溢出，使得进程（运行的程序）被操作系统终止。

3.2　线性与非线性递归

▶ 3.2.1　线性递归

线性递归是指每次递归时函数调用自身一次。

例 3-1　判断一年的第 n 天是星期几。

为了知道一年的第 n 天是星期几，需要知道第 $n-1$ 天是星期几。本例中 ALG3_1.py 中

的函数 $f(n)$是一个递归函数，即 $f(n)$需要等待 $f(n-1)$返回的值，才能计算出自己的返回值，即才会知道第 n 天是星期几，这就形成了递归调用。$f(n)$函数可以返回一年的第 n 天是星期几，其中返回值是 0 表示星期日，返回 1 表示星期一，……，返回 6 表示星期六。

ALG3_1. py

```
def f(n,startWeekDay):
    if n == 1:
        return startWeekDay
    else:
        return (f(n-1,startWeekDay) + 1) % 7
```

递归函数 $f(n)$递归过程中的示意图如图 3.3 所示，向下方向的弧箭头表示函数被调用，向上方向的直箭头表示函数调用结束。图 3.4 示意了递归过程中压栈、弹栈操作产生的最长的栈。

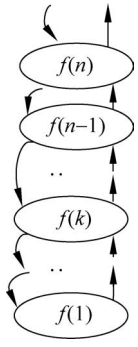


图 3.3 递归过程中函数的调用和结束

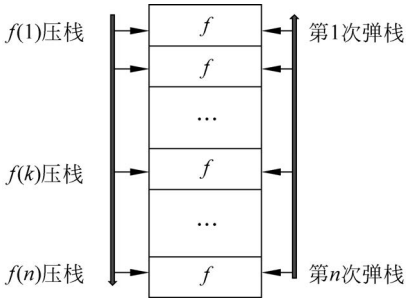


图 3.4 递归过程中最长栈的长度是 n

递归函数 $f(n)$的递归的总次数：

$$R(n)=n$$

每次递归只有两个基本操作，一个是关系表达式 $n=1$，另一个是 return 语句，所以递归函数 $f(n)$执行的基本操作总数是 $2\times n$，即递归结束后，执行的基本操作总数是 $2\times n$，因此 $f(n)$时间复杂度是 $O(n)$。

在递归的压栈操作过程中得到的栈的最大长度是 $R(n)=n$，每次递归只有两个局部变量：参数 n 和 startWeekDay，所占用的总内存是一个常量，例如 C，因此递归过程中占用的最大内存是 $C\times n$，所以空间复杂度是 $O(n)$。

可以把本例的递归算法类比为，如果你忘记了今天是星期几，就需要知道昨天是星期几，如此这般地向前翻日历，使得手中的日历越来越厚（相当于递归中的压栈，导致栈的长度在增加），直到翻到了某个日历页上显示了星期几，就结束翻阅日历（相当于结束压栈），然后一页一页地撕掉日历（相当于弹栈），撕掉日历的过程中，日历上出现了星期数，例如星期日、星期五等，即函数依次计算出自己的返回值。不断地撕掉日历退回到今天，就知道了今天是星期几。

```
如果元旦是星期一.
第 168 天是 星期日 .
```

图 3.5 使用递归算法输出星期

本例的 ch3_1. py 中，假设元旦是星期一，输出这一年的第 168 天是星期日，运行效果如图 3.5 所示。

ch3_1. py

```
from ALG3_1 import f
day = 168
```

```
m = f(day, 1)
print("如果元旦是星期一.")
days_of_week = ["星期日", "星期一", "星期二", "星期三", "星期四", "星期五", "星期六"]
print("第", day, "天是", days_of_week[m], ".")
```

▶ 3.2.2 非线性递归

非线性递归是指每次递归时函数调用自身 2 次或 2 次以上。

例 3-2 递归与 Fibonacci 序列。

Fibonacci 序列的特点是，前 2 项的值都是 1，从第 3 项开始，每项的值是前两项值的和。Fibonacci 序列如下：

```
1,1,2,3,5,8,13,21,…
```

本例 ALG3_2.py 中的函数 $f(n)$返回参数 n 指定的 Fibonacci 序列第 n 项的值。$f(n)$需要知道第 $n-1$ 项的值和第 $n-2$ 项的值，即需要 $f(n-1)$和 $f(n-2)$的返回值，这就形成了递归调用，即 $f(n)$是一个递归函数。

ALG3_2.py

```
def f(n):
    if n == 1 or n == 2:
        return 1
    elif n >= 3:
        return f(n-1) + f(n-2)
```

递归过程中，每次递归时函数调用自身两次，使得每次递归出现了两个递归“分支”，然后选择一个分支，继续递归，直到该分支递归结束，再沿着下一分支继续递归，当两个分支都递归结束，递归过程才结束，而且递归过程交替地进行压栈和弹栈操作，直至栈的长度为 0。

递归过程可以用一个二叉树来刻画，二叉树的结点数目恰好是递归总数，如图 3.6 所示。该二叉树至少有 2^k-1 个结点($k<n$)，k 随着 n 的增大而增大。例如，参数 n 的值是 6 时(即求第 6 项的值时)，调用函数的总次数(即递归的总次数)是 2^4-1($n=6$，$k=4$)。在递归过程中 f(n)被调用(压栈)和结束执行(弹栈)，其中向下方向的弧箭头表示函数被调用(压栈)，向上方向的直箭头表示函数结束(弹栈)。

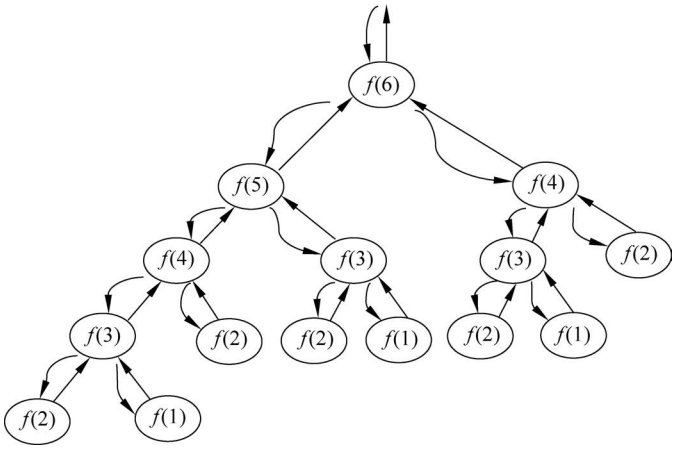


图 3.6 递归过程的二叉树示意图

k 随着 n 的增大而增大，在计算时间复杂度时可以设 $R(n)=2^n-1$，而每次递归的基本操

作只有一个加法操作和一个 return 语句，即每次递归有两个基本操作，因此 f(n)的执行过程(即递归过程)产生的基本操作的总次数 $T(n)$为

$$T(n)=R(n)=2\times(2^n-1)=2^{n+1}-2$$

所以 f(n)的时间复杂度是 $O(2^n)$。

递归过程是按照两个“分支”分别进行的，一个分支的递归结束(压栈、弹栈结束)，再进行另一个分支的递归。递归形成的二叉树至少有 2^k-1 个结点($k<n$)，那么树的高度(或叫深度)至少是 k(这是二叉树的简单规律)。递归过程中交替进行压栈和弹栈操作，因此递归过程中栈的最大长度大于 k 小于 n，由于 k 随着 n 的增大而增大，因此 $f(n)$的空间复杂度是 $O(n)$。图 3.6 中左侧的递归分支产生的压栈过程得到的最长栈如图 3.7 所示。

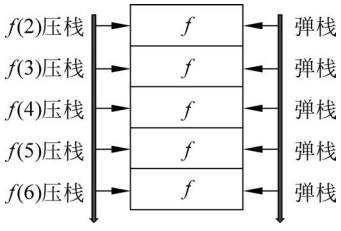


图 3.7 递归过程中的最长栈

注意：图 3.7 中 f(2)弹栈后，f(3)不马上弹栈，而是将 f(1)压栈，即 f(1)入栈。压栈、弹栈如此交替进行，直到递归结束。

本例的 ch3_2.py 中使用 ALG3_2.py 中的递归函数 $f(n)$ 输出 Fibonacci 序列的第 22 项，并计算了黄金分割的近似值，运行效果如图 3.8 所示。

```
Fibonacci序列第 22 项是 17711
黄金分割近似值: 0.618 (保留3位小数)
```

图 3.8 计算 Fibonacci 的某一项

ch3_2.py

```
from ALG3_2 import f
item = 22
print("Fibonacci 序列第",item,"项是", f(item))
golden_ratio = round(f(19)/f(20), 3)
print("黄金分割近似值:",golden_ratio,"(保留 3 位小数)")
```

Fibonacci 序列可以解释青蛙跳台阶问题。假设有 n 级台阶，青蛙每次只可以跳一个台阶或两个台阶，问青蛙完成跳 n 级台阶的任务(跳到最后一个台阶上算完成任务)一共有多少种跳法？

当 n 的值是 1 时，青蛙只有 1 种跳法，即跳一个台阶。当 n 的值是 2 时，青蛙有 2 种跳法：一种是每次跳一个台阶，另一种是每次跳两个台阶。当 n 的值是 3 时，青蛙可以选择先跳一个台阶，剩下的可能性跳法交给 $n-1$ 级台阶的情况；或者青蛙可以先跳两个台阶，剩下的可能性跳法交给 $n-2$ 级台阶的情况。即青蛙完成跳 n 级台阶的全部跳法总数($n\geqslant 3$)满足 Fibonacci 序列，所以 Fibonacci 序列的第 $n+1$ 项的值就是青蛙跳 n 级台阶的全部跳法的总数。

注意：青蛙跳台阶使用递归是合理的，理由是跳台阶的各种可能性和起始方式有关，比如跳 4 级台阶，1,1,2 和 2,2 是不同的跳法。

从 Fibonacci 序列中还可以计算出黄金分割数的近似值：

$$f(n)/f(n+1)(n=1,2,\cdots)$$

的极限是黄金分割数(0.618…，一个无理数)。

3.3 问题与子问题

一个问题的子问题就是数据规模比此问题的规模更小的问题。当一个问题可以分解成许多子问题时，就可以考虑用递归算法来解决这个问题。

例 3-3 计算 8＋88＋888＋8888＋…的前 n 项。

计算前 n 项和与计算前 $n-1$ 和就是问题和子问题的关系。如果解决了子问题，即算出了前 $n-1$ 项的和，那么前 n 项和的问题就解决了：前 n 项和等于前 $n-1$ 项的和乘以 10 再加上 $8\times n$。

本例 ALG3_3.py 中的递归函数 sum(a,n)返回形如

$$a+aa+aaa+aaaa+\cdots$$

的前 n 项和（a 是 1～9 中的某个数字）。递归函数 multi(n)返回 $n!$（n 的阶乘）。

ALG3_3.py

```
def sum(a, n):
    result = 0
    if n == 1:
        result = a
    elif n >= 2:
        result = sum(a, n-1) * 10 + n * a
    return result
def multi(n):
    result = -1
    if n == 1:
        result = 1
    elif n >= 2:
        result = multi(n-1) * n
    return result
```

不难计算出 sum(a,n)和 multi(n)的时间复杂度都是 $O(n)$。

```
8+88+888+…前5项和是98760
10的阶乘是3628800
```

图 3.9 计算连续和与阶乘

本例 ch3_3.py 使用 ALG3_3.py 中的递归函数计算了 8＋88＋888＋…的前 5 项的连续和以及 10！（10 的阶乘），运行效果如图 3.9 所示。

ch3_3.py

```
from ALG3_3 import sum,multi
a = 8
n = 5
m = 10
print(f"{a}+{a}{a}+{a}{a}{a}+…前{n}项和是{sum(a, n)}")
print(f"{m}的阶乘是{multi(m)}")
```

例 3-4 反转（倒置）字符序列。

本例中 ALG3_4.py 中的递归函数 reverse_string(s)返回参数 s 中的反转（倒置）字符序列。倒置长度为 n 的字符序列 $s_1s_2\cdots s_n$，这一问题的子问题是反转长度为 $n-1$ 的字符序列：$s_2s_3\cdots s_n$，将字符 s_1 放在反转后的字符序列 $s_ns_{n-1}\cdots s_2$ 的后面，变成 $s_ns_{n-1}\cdots s_1$，就解决了反转长度为 n 的字符序列问题。这一问题的子问题也可以是反转长度为 $n-1$ 的字符序列：$s_1s_2\cdots s_{n-1}$，将字符 s_n 放在反转后的字符序列 $s_{n-1}s_{n-2}\cdots s_1$ 的前面变成 $s_ns_{n-1}\cdots s_1$，就解决了反转长度为 n 的字符序列问题。

ALG3_4.py

```
def reverse_string(s):
    if len(s) <= 1:
        return s
    else:
        return reverse_string(s[1:]) + s[0]
```

不难计算出 reverse_string(s)的时间复杂度和空间复杂度都是 $O(n)$。

本例的 ch3_4.py 使用 ALG3_4.py 中的 reverse_string(s)得到一个字符序列的反转(倒置),并判断一个英文单词是否是回文单词(回文单词和它的反转相同),运行效果如图 3.10 所示。

```
GFEDCBA
racecar 是回文单词.
```

图 3.10 反转字符序列

ch3_4.py

```
from ALG3_4 import reverse_string
str1 = "ABCDEFG"
reversed_str1 = reverse_string(str1)
print(str1)
print(reversed_str1)
str2 = "racecar"
reversed_str2 = reverse_string(str2)
if str2 == reversed_str2:
    print(str2, "是回文单词.")
else:
    print(str2, "不是回文单词.")
```

3.4 递归与迭代

递归的思想是根据上一次操作的结果来确定当前操作的结果,比如当前结果与上一次结果相同或需要根据上一次结果来确定本次操作的结果。迭代的思想是根据当前操作的结果来确定下一次操作的结果。对于解决相同的问题,递归代码简练,容易理解解决问题的思路或发现数据的内部逻辑规律,具有很好的可读性。迭代代码可能比较复杂,处理数据的过程也可能比较复杂,所以迭代代码不如递归简练。对于递归算法能解决的问题,也可以用迭代算法解决。由于迭代不涉及函数的递归调用,所以通常情况下递归算法的空间复杂度会大于迭代的复杂度,当递归过程的递归总数(也称递归总数为递归深度)较大时会导致栈溢出。

例 3-5 计算圆周率的近似值。

下列无穷级数的和是圆周率的 1/4。

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots$$

本例的 ALG3_5.py 中的 recursion_method(n)函数和 iteration_method(n)函数都用来返回圆周率的近似值。recursion_method(n)函数使用的是递归,iteration_method(n)函数使用的是迭代。

ALG3_5.py

```
def recursion_method(n):
    sum = 0
    if n == 1:
        sum = 1
    elif n % 2 == 0:
        sum = recursion_method(n-1) - 1.0 / (2 * n - 1)
    else:
        sum = recursion_method(n-1) + 1.0 / (2 * n - 1)
    return sum
def iteration_method(n):
    sum = 0
    for i in range(1, n+1):
        if i % 2 == 0:
            sum -= 1.0 / (2 * i - 1)
        else:
```

```
            sum += 1.0 / (2 * i - 1)
    return sum
```

不难计算出基于递归的 recursion_method(n)函数的时间复杂度和空间复杂度都是 $O(n)$。基于迭代的 iteration_method(n)函数的时间复杂度是 $O(n)$，空间复杂度是 $O(1)$。

本例的 ch3_5.py 使用 ALG3_5.py 中的函数计算圆周率的近似值，运行效果如图 3.11 所示。

```
递归深度999,计算圆周率(保留6位小数): 3.142594
迭代次数9999999,计算圆周率(保留6位小数): 3.141593
```

图 3.11　计算圆周率的近似值

ch3_5.py

```
from ALG3_5 import recursion_method,iteration_method
n = 999
pi = recursion_method(n) * 4
print(f"递归深度{n},计算圆周率(保留 6 位小数):{pi:.6f}")
n = 9999999
pi = iteration_method(n) * 4
print(f"迭代次数{n},计算圆周率(保留 6 位小数):{pi:.6f}")
```

注意：递归算法可能导致栈溢出，在 ch2_3.py 中，当 n 的值较大时递归算法会导致栈溢出，例如，本机当 n 是 1000 就会导致栈溢出(这与 Java 或 C++的等效代码相差甚远)。

例 3-6　判断某个数是否是有序数组的元素值。

第 2 章的例 2-9 的 ALG2_9.py 中的 binary_search(array,number)函数使用迭代法判断 number 是否是有序数组 array 的元素值。本例的 ALG3_6.py 中的 binary_search_recursive (array,number,start,end)使用递归判断 number 是否是有序数组 array 的元素值，递归的时间复杂度是 $O(\log n)$，空间复杂度是 $O(n)$(时间复杂度和空间复杂度和例 2-9 中的迭代函数 binary_search(array,number)的相同)。

ALG3_6.py

```
def binary_search_recursive(array, number, start, end):
    if start > end:
        return -1
    else:
        mid = (start + end) // 2
        mid_value = array[mid]
        if number == mid_value:
            return mid
        elif number < mid_value:
            return binary_search_recursive(array, number, start, mid - 1)
        else:
            return binary_search_recursive(array, number, mid + 1, end)
```

二分法在处理数据时处理的数据量每次减少一半，递归的总次数 k 的最大可能取值就是使得数组的长度是 1(见例 2-9)，即

$$\frac{n}{2^k}=1,\quad k=\log_2 n$$

所以递归的总次数 $R(n)$的最大取值是 $\log_2 n$。由于每次递归的基本操作总数是一个常量，例如 C，因此递归过程的基本操作的总次数

$$T(n)=C\times\log_2 n$$

所以 binary_search_recursive(array,number,start,end)的时间复杂度是 $O(\log_2 n)$。

算法中影响空间复杂度的是一维数组 array 的长度 n,因此空间复杂度是 $O(n)$。

本例的 ch3_6.使用 binary_search_recursive(array, number, start, end)函数判断一些数是否是有序数组 array 的元素值,运行效果如图 3.12 所示。

```
-11 1 12 56 89 100 128 128 129 199 200 289
-11在数组中,数组索引位置是0
128在数组中,数组索引位置是6
11不在数组中
129在数组中,数组索引位置是8
289在数组中,数组索引位置是11
```

图 3.12 判断某个数是否在数组中

ch3_6.py

```
from ALG3_6 import binary_search_recursive
import array
number = [-11, 128, 11, 129, 289]
a = array.array('i',[128, 129, 199, 200, 289, -11, 1, 12, 56, 89, 100, 128])
a = sorted(a)
for num in a:
    print(num,end =" ")
print()
for num in number:
    index = binary_search_recursive(a, num, 0, len(a) - 1)
    if index == -1:
        print(f"{num}不在数组中")
    else:
        print(f"{num}在数组中,数组索引位置是{index}")
```

例 3-7 求两个正整数的最大公约数。

第 2 章的例 2-10 的 ALG2_10.py 中的 gcd(n,m)函数返回两个正整数 m 和 n 的最大公约数。本例中 ALG3_7.py 中的 gcd_recursive(n,m)函数使用的是递归。两者都是求两个正整数的最大公约数,但例 2-10 中的 gcd(n,m)函数(迭代法)的空间复杂度是 $O(1)$,这里的递归函数 gcd_recursive(n,m)的空间复杂度是 $O(\log_2 n)$。二者的时间复杂度都是 $O(\log_2 n)$。

本例的 gcd_recursive 函数的代码要比例 2-10 中的 gcd(n,m)函数能更简练地体现辗转相除:如果 $n\%m(n\geqslant m)$不是 0,那么 n 和 m 的最大公约数与 m 和 $n\%m$ 的最大公约数相同。如果 $n\%m$ 等于 0,二者的最大公约数就是 m。

ALG3_7.py

```
import math
def gcd_recursive(n, m):
    n = abs(n)
    m = abs(m)
    if n % m == 0:
        return m
    else:
        return gcd_recursive(m, n % m)
```

由于 $n\%m$ 小于 $n/2$,即辗转相除都会使得 n 的值减小至少二分之一(减少至少一半),那么,递归总次数 k 会满足:

$$\frac{n}{2^k}=1,\quad k=\log_2 n$$

即栈的最大长度是 $\log_2 n$,因此空间和时间复杂度都是 $O(\log_2 n)$。

```
6, 12的最大公约数6.
63, 42的最大公约数21.
```

图 3.13 求最大公约数

本例的 ch3_7.py 使用 ALG3_7.py 中的递归函数 gcd_recursive 输出两个正整数的最大公约数,运行效果如图 3.13 所示。

ch3_7.py

```
from ALG3_7 import gcd_recursive
a = 6
b = 12
print(f"{a},{b}的最大公约数{gcd_recursive(a, b)}.")
a = 63
b = 42
print(f"{a},{b}的最大公约数{gcd_recursive(a, b)}.")
```

3.5 多重递归

所谓多重递归，是指一个递归函数调用另一个或多个递归函数，我们称这样的递归函数为多重递归函数。

这里用一个数字问题来说明多重递归：求 n 位十进制数中含有偶数个 6 的数（即数的某位上是数字 6）的个数，但不要求输出含有偶数多个 6 的数。两位十进制数中含有偶数多个 6 的数的个数是 1 个（66 含有两个 6），含有奇数个 6 的数的个数是 17 个：

16，26，36，46，56，60，61，62，63，64，65，67，68，69，76，86，96

用 $a(n)$ 表示 n 位十进制数中含有偶数个 6 的数的个数，$b(n)$ 表示 n 位十进制数中含有奇数个 6 的数的个数。$c(n)$ 表示 n 位十进制数中不含有 6 的数的个数。

对于 $n>2$，有下列递推关系成立，这里的“=”是数学意义的等号：

$$a(n)=9\times a(n-1)+b(n-1)$$
$$b(n)=9\times b(n-1)+a(n-1)+c(n-1)$$
$$c(n)=9\times c(n-1)$$

非常容易证明上述等式，因为对于任意一个 $(n-1)$ 位十进制数，例如 $a_1a_2\cdots a_{n-1}$，如果 $a_1a_2\cdots a_{n-1}$ 中出现了偶数个 6，那么 n 位十进制数 $a_1a_2\cdots a_{n-1}p$（$p=1,2,3,4,5,7,8,9,0$），即 p 取 6 以外的其他数字，都出现了偶数个 6。如果 $a_1a_2\cdots a_{n-1}$ 中出现了奇数个 6，那么 n 位十进制数 $a_1a_2\cdots a_{n-1}6$ 就出现了偶数个 6。所以有

$$a(n)=9\times a(n-1)+b(n-1)$$

另外两个等式的论证道理类似。如果 $a(n)$，$b(n)$ 对应到递归函数，那么所对应的递归函数就都是多重递归函数。

例 3-8 求 n 位十进制数中含有偶数个 6、奇数个 6 以及不含有 6 的数的个数。

本例的 ALG3_8.py 中的 $a(n)$ 函数和 $b(n)$ 函数是多重递归函数，不难验证二者的时间复杂度都是 $O(2^n)$，空间复杂度都是 $O(n)$（验证方法和例 3-2 类似）。

ALG3_8.py

```
#返回n位十进制中出现偶数个6的数的个数
def a(n):
    result = 0
    if n == 1:
        result = 0
    elif n == 2:
        result = 1
    elif n > 2:
        result = 9 * a(n - 1) + b(n - 1)
    return result
#返回n位十进制中出现奇数个6的数的个数
```

```
def b(n):
    result = 0
    if n == 1:
        result = 1
    elif n == 2:
        result = 17
    elif n > 2:
        result = 9 * b(n - 1) + a(n - 1) + c(n - 1)
    return result
#返回 n 位十进制中未出现 6 的数的个数
def c(n):
    result = 0
    if n == 1:
        result = 9
    elif n == 2:
        result = 72
    else:
        result = 9 * c(n - 1)
    return result
```

本例的 ch3_8.py 使用 ALG3_8.py 中的多重递归函数，输出 8 位十进制数中出现偶数个 6、奇数个 6 以及不含有 6 的数的个数等信息，运行效果如图 3.14 所示。

```
8位十进制数中出现偶数个数字6的个数是14076280.
8位十进制数中出现奇数个数字6的个数是37659968.
8位十进制数中未出现数字6的个数是38263752.
8位数一共有:90000000个.
```

图 3.14 输出数字的有关信息

ch3_8.py

```
from ALG3_8 import a,b,c
n = 8
sum = 0
count = a(n)
sum += count
print(f"{n}位十进制数中出现偶数个数字 6 的个数是{count}.")
count = b(n)
sum += count
print(f"{n}位十进制数中出现奇数个数字 6 的个数是{count}.")
count = c(n)
sum += count
print(f"{n}位十进制数中未出现数字 6 的个数是{count}.")
print(f"{n}位数一共有:{sum}个.")
```

3.6 经典递归

递归算法不仅能使得代码简练，容易理解解决问题的思路或发现数据的内部逻辑规律，而且具有很好的可读性。本节通过杨辉三角形、老鼠走迷宫和汉诺塔 3 个经典的递归进一步体会递归算法。特别是汉诺塔递归，通过其递归算法能洞悉数据规律，给出相应的迭代算法。

▶ 3.6.1 杨辉三角形

杨辉三角形形式如下：

```
1
1  1
1  2  1
```

```
1  3  3  1
1  4  6  4  1
1  5  10  10  5  1
…
```

杨辉三角形最早出现于中国南宋的数学家杨辉在1261年所著的《详解九章算法》中。法国数学家帕斯卡(Pascal)在1654年发现该三角形，因此又称帕斯卡三角形。

例 3-9 输出杨辉三角形。

按照编程语言的习惯，行、列的索引都是从0开始的。杨辉三角形的主要规律：杨辉三角形第0行有1个数，第1行有2个数，……，第 n 行有 $n+1$ 个数，第 n 行的第0列和最后一列的数都是1，即第0列和第 n 列的数都是1。用 $C(n,j)$ 表示第 n 行、第 j 列的数($j=0,\cdots,n$)，那么递归如下：

$$C(n,0)=1,\quad C(n,j)=C(n-1,j-1)+C(n-1,j)(0<j<n),\quad C(n,n)=1 \tag{3-1}$$

本例中ALG3_9.py中的 $C(n,j)$ 函数是依据公式(3-1)的递归算法，可以计算杨辉三角形第 n 行、第 j 列上的数，即该函数返回杨辉三角形第 n 行、第 j 列上的数。

在组合数学中，对于二项式系数有一个经典的公式，即对杨辉三角的第 n 行、第 j 列($j=0,\cdots n$)的数 $Y(n,j)$，有如下递归：

$$Y(n,0)=1,\quad Y(n,j)=Y(n,j-1)\times(n-j+1)/j(j>0,j<n),\quad Y(n,n)=1 \tag{3-2}$$

ALG3_9.py中的 $Y(n,j)$ 函数依据式(3-2)的递归算法，可以计算杨辉三角形第 n 行、第 j 列上的数，即该函数返回杨辉三角形第 n 行、第 j 列上的数。

ALG3_9.py

```
def C(n, j):
    result = 0
    if j == 0 or j == n: #每行的第0列和第n列上的数都是1
        result = 1
    else:
        result = C(n - 1, j - 1) + C(n - 1, j)
    return result
def Y(n, j):
    result = 0
    if j == 0 or j == n: #每行的第0列和第n列上的数都是1
        result = 1
    elif 0 < j < n:
        result = Y(n, j - 1) * (n - j + 1) // j
    return result
```

$C(n,j)$ 递归算法属于非线性递归，时间复杂度是 $O(n^2)$，空间复杂度是 $O(n)$。因为杨辉三角形的前 n 行里一共有 $n(n+1)/2$ 个数，递归过程中函数被调用的总次数是 $n(n+1)/2$，所以 $C(n,j)$ 的时间复杂度是 $O(n^2)$。递归过程中，根据

$$C(n,j)=C(n-1,j-1)+C(n-1,j)$$

可知，一个递归分支当栈的长度达到 n 时就会依次弹栈，返回到上一个递归分支，因此空间复杂度是 $O(n)$。

$Y(n,j)$ 递归属于线性递归，时间复杂度是 $O(n)$，空间复杂度是 $O(n)$。因为函数被调用的总次数是 n，所以 $Y(n,j)$ 的时间复杂度是 $O(n)$。递归过程可以看出压栈导致栈的长度最大是 n，因此空间复杂度是 $O(n)$。

本例的ch3_9.py输出了杨辉三角形的前9行，并比较了 $C(n,j)$ 递归函数和 $Y(n,j)$ 递归

函数计算第 n 行、第 j 列上的数的耗时。时间复杂度是 $O(n)$ 的耗时明显小于时间复杂度是 $O(n^2)$ 的耗时，运行效果如图 3.15 所示。

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5  10 10 5  1
1  6  15 20 15 6  1
1  7  21 35 35 21 7  1
1  8  28 56 70 56 28 8  1
线性递归求第28行,第14列40116600的耗时是0(毫秒)
非线性递归求第28行,第14列40116600的耗时是6825(毫秒)
```

图 3.15　输出杨辉三角形的前 9 行，比较了两个递归的耗时

ch3_9.py

```
from ALG3_9 import C,Y
import time
row = 8
for n in range(row + 1):                    #输出杨辉三角形的前 row+1 行
    for j in range(n + 1):
        result = Y(n, j)
        print(f"{result:5d}", end = "")
    print()
m = 28
j = m // 2
start = time.time()                         #开始执行的时间点
result = Y(m, j)
end = time.time()                           #结束的时间点
time_used = (end - start) * 1000            #转换为毫秒
print(f"线性递归求第{m}行,第{j}列{result}的耗时是{time_used:.0f}(毫秒)")
start = time.time()                         #开始执行的时间点
result = C(m, j)
end = time.time()                           #结束的时间点
time_used = (end - start) * 1000            #转换为毫秒
print(f"非线性递归求第{m}行,第{j}列{result}的耗时是{time_used:.0f}(毫秒)")
```

▶ 3.6.2　老鼠走迷宫

用列表模拟迷宫，列表元素值是 1 表示墙，0 表示路，2 表示出口。

假设老鼠走迷宫的递归函数是 move(a,rows,cols,i,j)，老鼠在迷宫某点 $p=(i,j)$ 的递归办法是，首先从 p 点向东调用 move()，如果找到出口，move()返回 1，结束递归；如果从 p 点向东无法找到出口，返回 0 结束递归，再从 p 点向南调用 move()，如果找到出口，move()返回 1，结束递归；如果从 p 点向南无法找到出口，返回 0 结束递归，再从 p 点向西调用 move()，如果找到出口，move()返回 1，结束递归；如果从 p 点向西无法找到出口，返回 0 结束递归，再从 p 点向北调用 move()，如果找到出口，move()返回 1，结束递归，否则返回 0 结束递归。

如果 move()最后返回的值是 1，说明老鼠找到出口，否则说明迷宫没有出口。

例 3-10　模拟老鼠走迷宫。

本例的 ALG3_10.py 中的 move(a,rows,cols,i,j)是老鼠走迷宫的函数，该函数是一个递归函数。

ALG3_10.py

```
def move(a, rows, cols,i, j):
    isOut = 0
    if a[i][j] == 2:                        # 出口
        isOut = 1
    elif a[i][j] == 0:
```

```
            a[i][j] = -1                          #用-1标记此点已经递归过,即老鼠走过了该点
            t = min(j + 1, cols - 1)              #东
            roadOrOut = a[i][t] == 0 or a[i][t] == 2          #是路或出口
            if roadOrOut and move(a, rows, cols, i, t):
                isOut = 1
                return isOut
            t = min(i + 1, rows - 1)              #南
            roadOrOut = a[t][j] == 0 or a[t][j] == 2
            if roadOrOut and move(a, rows, cols, t, j):
                isOut = 1
                return isOut
            t = max(j - 1, 0)                     #西
            roadOrOut = a[i][t] == 0 or a[i][t] == 2
            if roadOrOut and move(a, rows, cols, i, t):
                isOut = 1
                return isOut
            t = max(i - 1, 0)                     #北
            roadOrOut = a[t][j] == 0 or a[t][j] == 2
            if roadOrOut and move(a, rows, cols, t, j):
                isOut = 1
                return isOut
    return isOut
```

不难验证 move(a,rows,cols,i,j)算法的时间复杂度是 $O(n^2)$，空间复杂度也是 $O(n^2)$。

本例的 ch3_10.py 使用 ALG3_10.py 中的 move(a,rows,cols,i,j)函数走迷宫。老鼠走过迷宫后，输出老鼠走过的路时，用 m 表示老鼠走过的路，Y 表示老鼠到达的出口，运行效果如图 3.16 所示。

```
迷宫数据：0表示路，1表示墙，2表示出口.
0  0  0  1  1  1  1
1  0  0  0  0  1  1
1  1  0  1  0  0  1
1  0  0  0  1  1  1
1  0  0  0  0  2  1

老鼠走迷宫过程：m表示走过的路，Y是到达的出口.
  m  m  m  1  1  1  1
  1  0  m  m  m  1  1
  1  1  m  1  m  m  1
  1  0  m  m  1  1  1
  1  0  0  m  m  Y  1
老鼠成功走出迷宫
```

图 3.16　老鼠走迷宫

ch3_10.py

```
from ALG3_10 import move
a = [[0, 0, 0, 1, 1, 1, 1],
     [1, 0, 0, 0, 0, 1, 1],
     [1, 1, 0, 1, 0, 0, 1],
     [1, 0, 0, 0, 1, 1, 1],
     [1, 0, 0, 0, 0, 2, 1]]              #用列表模拟5行7列的迷宫
print("迷宫数据：0表示路，1表示墙，2表示出口.")
for row in a:
    for cell in row:
        print(cell, end = " ")
    print()
print()
rows = len(a)                            #获取行数
cols = len(a[0])                         #获取列数
isOut = move(a, rows, cols, 0, 0)
print("老鼠走迷宫过程：m表示走过的路，Y是到达的出口.")
for row in a:
    for cell in row:
```

```
            if cell == -1:                  # -1 表示老鼠走过的路
                print("%3c" % 'm', end = "")
            elif cell == 2: # 出口
                print("%3c" % 'Y', end = "")
            else:
                print("%3d" % cell, end = "")
        print()
    if isOut:
        print("老鼠成功走出迷宫")
```

▶ 3.6.3 汉诺塔

汉诺塔(Hanoi Tower)问题是来源于印度的一个古老问题。有名字为 A、B、C 的三个塔，A 塔上有从小到大的 64 个盘子，每次搬运一个盘子，最后要把 64 个盘子搬运到 C 塔。在搬运过程中，可以把盘子暂时放在 3 个塔中的任何一个上，但不允许大盘放在小盘上面。3 个盘子的汉诺塔如图 3.17 所示。

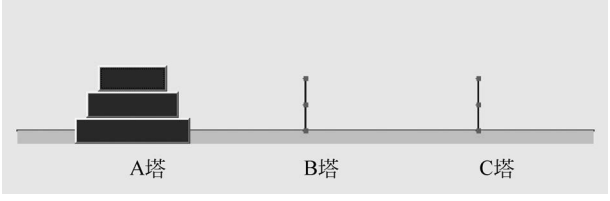


图 3.17 3 个盘子的汉诺塔

1. 汉诺塔的递归算法

递归算法如下：

(1) 如果 A 塔只有一个盘子，直接将盘子搬运到 C 塔。

(2) 如果盘子的数目 n 大于 1，首先将 $n-1$ 个盘子从 A 塔搬运到 B 塔，然后将第 n 个盘子从 A 塔搬运到 C 塔，最后将 $n-1$ 个盘子从 B 塔搬运到 C 塔。

3 个盘子的汉诺塔的搬运过程如图 3.18 所示。

例 3-11 汉诺塔的递归算法。

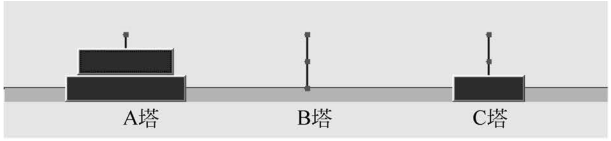
本例的 ALG3_11.py 中的 moveDish(n,A,B,C)是搬运盘子的递归函数。

ALG3_11.py

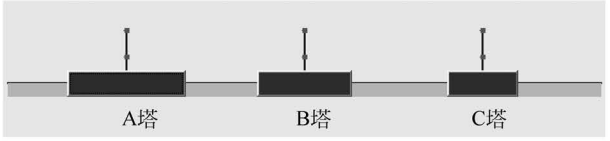
```
def moveDish(n, A, B, C):
    if n == 1:
        print("从%c塔搬运%d号盘到%c塔" % (A, n, C))
    else:
        moveDish(n-1, A, C, B)
        print("从%c塔搬运%d号盘到%c塔" % (A, n, C))
        moveDish(n-1, B, A, C)
```

如果汉诺塔有 n 个盘子，那么需要搬动 2^n-1 次，所以不难验证 moveDish(n,A,B,C)的时间复杂度是 $O(2^n)$，空间复杂度是 $O(n)$(验证方法见例 3-2)。

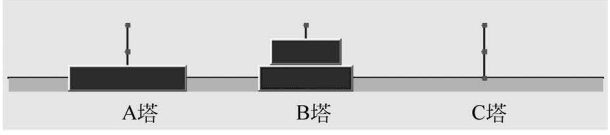
本例的 ch3_11.py 使用 ALG3_11.py 中的 moveDish(n,A,B,C)函数搬运 3 个盘子的汉诺塔和 4 个盘子的汉诺塔，运行效果如图 3.19 所示。



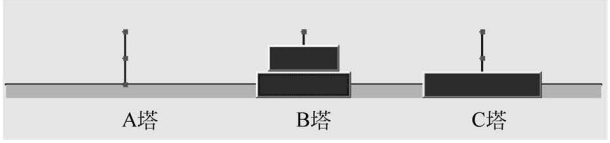
(a) 从A塔搬运1号盘到C塔



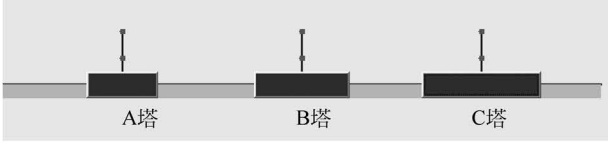
(b) 从A塔搬运2号盘到B塔



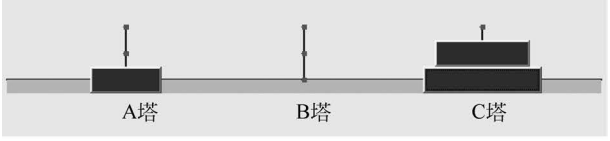
(c) 从C塔搬运1号盘到B塔



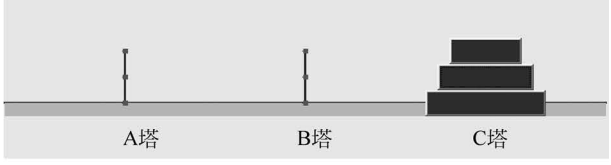
(d) 从A塔搬运3号盘到C塔



(e) 从B塔搬运1号盘到A塔



(f) 从B塔搬运2号盘到C塔



(g) 从A塔搬运1号盘到C塔

图 3.18　搬运 3 个盘子的汉诺塔

ch3_11. py

```
from ALG3_11 import moveDish
n = 3
print("汉诺塔有 %d 个盘子" % n)
moveDish(n, 'A', 'B', 'C')
n = 4
print("汉诺塔有 %d 个盘子" % n)
moveDish(n, 'A', 'B', 'C')
```

2. 汉诺塔的迭代算法

在 3.4 节讲过，递归的代码简练，容易理解解决问题的思路或发现数据的内部逻辑规律，具有很好的可读性。迭代的代码可能比较复杂，处理数据的过程也可能比较复杂，所以迭代的代码不如递归简练。

在给出汉诺塔的迭代算法之前，先总结一下汉诺塔问题中的一些规律。

(1) n 个盘子的汉诺塔需要搬运 2^n-1 次。

(2) 搬运的盘子的号码依次对应着 $1\sim2^{n+1}-1$ 中从小到大的偶数的二进制的尾部(低位)连续的 0 的个数。自然数的奇数的二进制的个位是 1，偶数是 0。例如盘子数目是 3，依次搬运的盘子的号码与二进制的尾部连续 0 的个数的对应关系如表 3.1 所示。

```
汉诺塔有3个盘子
从A塔搬运1号盘到C塔
从A塔搬运2号盘到B塔
从C塔搬运1号盘到B塔
从A塔搬运3号盘到C塔
从B塔搬运1号盘到A塔
从B塔搬运2号盘到C塔
从A塔搬运1号盘到C塔
汉诺塔有4个盘子
从A塔搬运1号盘到B塔
从A塔搬运2号盘到C塔
从B塔搬运1号盘到C塔
从A塔搬运3号盘到B塔
从C塔搬运1号盘到A塔
从C塔搬运2号盘到B塔
从A塔搬运1号盘到B塔
从A塔搬运4号盘到C塔
从B塔搬运1号盘到C塔
从B塔搬运2号盘到A塔
从C塔搬运1号盘到A塔
从B塔搬运3号盘到C塔
从A塔搬运1号盘到B塔
从A塔搬运2号盘到C塔
从B塔搬运1号盘到C塔
```

图 3.19 递归法搬运盘子

表 3.1 盘子的号码与二进制的尾部连续 0 的个数的对应表

依次搬运的盘子的号码	偶数的十进制	偶数的二进制	尾部连续 0 的个数
1	2	10	1
2	4	100	2
1	6	110	1
3	8	1000	3
1	10	1010	1
2	12	1100	2
1	14	1110	1

根据表 3.1，在搬运盘子的过程中，搬运的盘号依次是(如前面图 3.18 所示)：

1,2,1,3,1,2,1

(3) 二进制的尾部连续 0 的个数，每隔一次这个数目就是 1。也就是说，在搬运盘子的过程中，每隔一次就要搬动一次 1 号盘(盘号最小的盘)。

(4) 1 号盘的移动规律是：如果盘子的数目 n 是奇数，1 号盘找目标塔的规律是 CBA 的循环次序。如果 n 是偶数，1 号盘找目标塔的规律是 BCA 的循环次序。

(5) 当搬运大号盘时(盘号大于或等于 2)，上一次搬运的一定是 1 号盘(理由见(3))，所以搬运大号盘的目标塔，一定不是上一次搬运 1 号盘的目标塔(大盘不能放在小盘上)。

注意：实际上，这些规律都是人们通过研究汉诺塔的递归算法发现的，也就是通过递归可以发现数据的内部逻辑规律。

一个偶数通过不断地右位移可计算出尾部连续 0 的个数，例如 8 的二进制 1000 右位移 3 次，得到奇数，因此知道 8 的二进制尾部连续 0 的个数是 3。

例 3-12 汉诺塔的迭代算法。

根据迭代法的规律，本例给出汉诺塔的迭代算法。本例 ALG3_12.py 中的 moveDish(n) 函数是迭代算法。不难验证 moveDish(n)的时间复杂度是 $O(2^n)$，空间复杂度是 $O(1)$。

尽管本例的 moveDish(n)的时间复杂度和例 3-11 中的递归算法相同，空间复杂度低于递归算法，但简练性和可读性远远不如递归算法。在内存允许的范围内，还是递归算法更好。

ALG3_12. py

```
from collections import deque
import math
def getZeroCount(n):
    count = 0
    if n % 2 == 0:
        while n % 2 != 1:
            n = n >> 1
            count += 1
    return count
def moveDish(n):
    deque_tower_name = deque() #队列,存放目标塔的名字
    A = [] #列表,模拟A塔
    B = [] #列表,模拟B塔
    C = [] #列表,模拟C塔
    if n % 2 != 0:
        deque_tower_name.extend(['C', 'B', 'A'])
    else:
        deque_tower_name.extend(['B', 'C', 'A'])
    for i in range(n, 0, -1): #初始状态,盘子都在A塔上
        A.append(i)
    for i in range(1, int(math.pow(2, n))):
        dishNumber = getZeroCount(2 * i) # 盘号
        if dishNumber == 1:
            target = deque_tower_name.popleft()
            deque_tower_name.append(target)
            if dishNumber in A:
                print("从A塔搬运%d号盘到%c塔" % (dishNumber, target))
                if target == 'C':
                    C.append(A.pop())
                elif target == 'B':
                    B.append(A.pop())
            elif dishNumber in B:
                print("从B塔搬运%d号盘到%c塔" % (dishNumber, target))
                if target == 'A':
                    A.append(B.pop())
                elif target == 'C':
                    C.append(B.pop())
            elif dishNumber in C:
                print("从C塔搬运%d号盘到%c塔" % (dishNumber, target))
                if target == 'A':
                    A.append(C.pop())
                elif target == 'B':
                    B.append(C.pop())
        elif dishNumber >= 2:
            notTarget = deque_tower_name[-1]
            if dishNumber in A:
                if notTarget == 'C':
                    B.append(A.pop())
                    print("从A塔搬运%d号盘到%c塔" % (dishNumber, 'B'))
                elif notTarget == 'B':
                    C.append(A.pop())
                    print("从A塔搬运%d号盘到%c塔" % (dishNumber, 'C'))
            elif dishNumber in B:
                if notTarget == 'C':
                    A.append(B.pop())
                    print("从B塔搬运%d号盘到%c塔" % (dishNumber, 'A'))
                elif notTarget == 'A':
                    C.append(B.pop())
```

```
                print("从 B 塔搬运 %d 号盘到 %c 塔" % (dishNumber, 'C'))
            elif dishNumber in C:
                if notTarget == 'A':
                    B.append(C.pop())
                    print("从 C 塔搬运 %d 号盘到 %c 塔" % (dishNumber, 'B'))
                elif notTarget == 'B':
                    A.append(C.pop())
                    print("从 C 塔搬运 %d 号盘到 %c 塔" % (dishNumber, 'A'))
```

本例的 ch3_12.py 使用 ALG3_12.py 中的 moveDish(n)函数搬运 3 个盘子的汉诺塔和 4 个盘子的汉诺塔，运行效果如图 3.20 所示。

ch3_12.py

```
from ALG3_12 import moveDish
print("迭代法搬运盘子")
n = 3
print("汉诺塔有 %d 个盘子" % n)
moveDish(n)
n = 4
print("汉诺塔有 %d 个盘子" % n)
moveDish(n)
```

```
迭代法搬运盘子
汉诺塔有3个盘子
从A塔搬运1号盘到C塔
从A塔搬运2号盘到B塔
从C塔搬运1号盘到B塔
从A塔搬运3号盘到C塔
从B塔搬运1号盘到A塔
从B塔搬运2号盘到C塔
从A塔搬运1号盘到C塔
汉诺塔有4个盘子
从A塔搬运1号盘到B塔
从A塔搬运2号盘到C塔
从B塔搬运1号盘到C塔
从A塔搬运3号盘到B塔
从C塔搬运1号盘到A塔
从C塔搬运2号盘到B塔
从A塔搬运1号盘到B塔
从A塔搬运4号盘到C塔
从B塔搬运1号盘到C塔
从B塔搬运2号盘到A塔
从C塔搬运1号盘到A塔
```

图 3.20 迭代法搬运盘子

注意：本例用到了 collections 模块提供的队列 deque(这里的用法相对简单，容易理解)，其详细知识点见第 7 章。

3.7 优化递归

在 3.2 节讲解了非线性递归，即每次递归时函数调用自身两次或两次以上。非线性递归可以形成多个递归分支，即形成多个子递归过程。例如 3.2 节的例 3-2 的 ALG3_2.py 中的函数 $f(n)$(求 Fibonacci 序列的第 n 项)形成了两个递归分支：$f(n-1)$和 $f(n-2)$。

为了完成 $f(n)$的调用，递归过程中需要将 $f(n-1)$分支进行完毕再进行 $f(n-2)$分支。注意，在进行 $f(n-1)$分支递归时会完成 $f(n-2)$分支递归，那么再进行 $f(n-2)$分支就是一个重复的递归过程。

优化递归是在每次递归开始前，首先到某个对象中，例如字典中，查找本次递归是否已经实施完毕，即是否已经有了递归结果，如果字典中已经有了本次递归的结果，就直接使用这个结果，不再浪费时间进行本次递归，否则就执行本次递归，并将递归结果保存到字典。简而言之，优化递归就是避免重复子递归。

优化递归是典型的用空间换取时间的策略(需要额外地存储某些递归结果)。优化递归通常不会改变空间的复杂度，但一定可以降低时间复杂度，甚至可以将指数复杂度降低为线性或多项式复杂度。许多非线性递归的时间复杂度都是指数复杂度，例如例 3-2 中计算 Fibonacci 序列的递归算法，其时间复杂度是 $O(2^n)$。

例 3-13 优化 Fibonacci 序列的递归算法。

本例的 ALG3_13.py 给出了优化的 Fibonacci 序列的递归算法，使得其时间复杂度是 $O(n)$，而例 3-2 的 ALG3_2.py 中计算 Fibonacci 序列的递归函数 $f(n)$的时间复杂度是 $O(2^n)$。本例的递归函数避免了重复子递归，那么当 n 的值较大时，优化递归的耗时明显小于未优化的耗时，两者的空间复杂度都是 $O(n)$。

ALG3_13. py

```
import time
hash_map = {} #用字典优化递归
def f_optimize(n):
    result = -1
    if n == 1 or n == 2:
        result = 1
    elif n >= 3:
        if n in hash_map:
            result = hash_map[n]
        else:
            result = f_optimize(n-1) + f_optimize(n-2)
            hash_map[n] = result
    return result
def f(n): #未优化的递归
    if n == 1 or n == 2:
        return 1
    else:
        return f(n-1) + f(n-2)
```

本例 ch3_13. py 比较了例 3-2 的 ALG3_2. py 中的未优化的递归函数 $f(n)$和本例 ALG3_13. py 中优化后的递归函数 f_optimize(n)的运行耗时，运行效果如图 3. 21 所示。

```
优化求第35项9227465的用时是0.0000000000(秒)
未优化求第35项9227465的用时是1.3082036972(秒)
```

图 3. 21　Fibonacci 的优化和未优化递归的耗时

ch3_13. py

```
from ALG3_13 import f_optimize,f
import time
item = 35
start_time = time.time()
result = f_optimize(item)
end_time = time.time()
duration_optimized = end_time - start_time
print("优化求第%d项%d的用时是%1.10f(秒)" % (item, result, duration_optimized))
start_time = time.time()
result = f(item)
end_time = time.time()
duration_unoptimized = end_time - start_time
print("未优化求第%d项%d的用时是%1.10f(秒)" % (item, result, duration_unoptimized))
```

例 3-14　优化杨辉三角形的递归算法。

本例中的 ALG3_14. py 中的递归函数 C_optimize(n,j)是计算杨辉三角形的优化递归算法，其时间复杂度是 $O(n)$，而例 3-9 的 ALG3_9. py 中计算杨辉三角形的递归函数 $C(n,j)$是未优化递归算法，其时间复杂度是 $O(n^2)$；二者的空间复杂度都是 $O(n)$。

ALG3_14. py

```
table = {}                  #使用字典 table 来优化递归
def C_optimize(n, j):
    if (n, j) in table:
        return table[(n, j)]
    if j == 0 or j == n:
        result = 1
    else:
        result = C_optimize(n-1, j-1) + C_optimize(n-1, j)
```

```
        table[(n, j)] = result
        return result
    def C(n, j):                    #未优化
        result = 0
        if j == 0 or j == n:
            result = 1
        else:
            result = C(n - 1, j - 1) + C(n - 1, j)
        return result
```

本例的 ch3_14.py 比较了例 3-9 的递归函数 $C(n,j)$和本例的优化后的递归函数 C_optimize(n,j)的运行耗时,优化后的递归的运行耗时明显小于未优化的递归函数的运行耗时,运行效果如图 3.22 所示。

```
优化求第28行,第14项40116600的耗时是0.0（毫秒）
未优化求第28行,第14项40116600的耗时是6732.916355133057（毫秒）
```

图 3.22 杨辉三角形的优化和未优化递归的耗时

ch3_14.py

```
from ALG3_14 import C_optimize,C
import time
n = 28
j = n // 2
start_time = time.time()
result = C_optimize(n, j)
end_time = time.time()
used_time = (end_time - start_time) * 1000
print(f"优化求第{n}行,第{j}项{result}的耗时是{used_time}（毫秒）")
start_time = time.time()
result = C(n, j)
end_time = time.time()
used_time = (end_time - start_time) * 1000
print(f"未优化求第{n}行,第{j}项{result}的耗时是{used_time}（毫秒）")
```

注意:也可以使用函数缓存技术来优化递归,但函数缓存对函数的参数类型有较严格的限制,细节见第 12 章的 12.8 节。

习题 3

扫一扫

习题

扫一扫

自测题