数据结构和算法
（Python描述）

郭 炜

微信公众号

微博：http://weibo.com/guoweiofpku

学会程序和算法，走遍天下都不怕!

讲义照片均为郭炜拍摄

递归

递归的作用

递归的作用

1) 替代多重循环进行枚举
2) 解决本来就是用递归形式定义的问题
3) 将问题分解为规模更小的子问题进行求解

....

递归替代多重循环
例题:全排列

宁夏中卫沙坡头

全排列

- 解题思路

给定一个由不同的小写字母组成的字符串，输出这个字符串的所有全排列。 我们假设对于小写字母有'a' < 'b' < ... < 'y' < 'z'，而且给定的字符串中的字母已经按照从小到大的顺序排列。

样例输入

```
abc
```

样例输出

```
abc
acb
bac
bca
cab
cba
```

```
s = list(input())
s.sort()
N = len(s)
result = [0 for i in range(N)] #存最新找到的一个排列
used = [False for i in range(N)]  #used[i]表示字母s[i] 是否用过
def dfs(n): #摆放第n个位置及其右边的字母
    if n == N:
        print("".join(result))
    for i in range(N):
        if not used[i]: # s[i]这个字母没用过
            result[n] = s[i]
            used[i] = True
            dfs(n+1)
            used[i] = False
dfs(0)
```

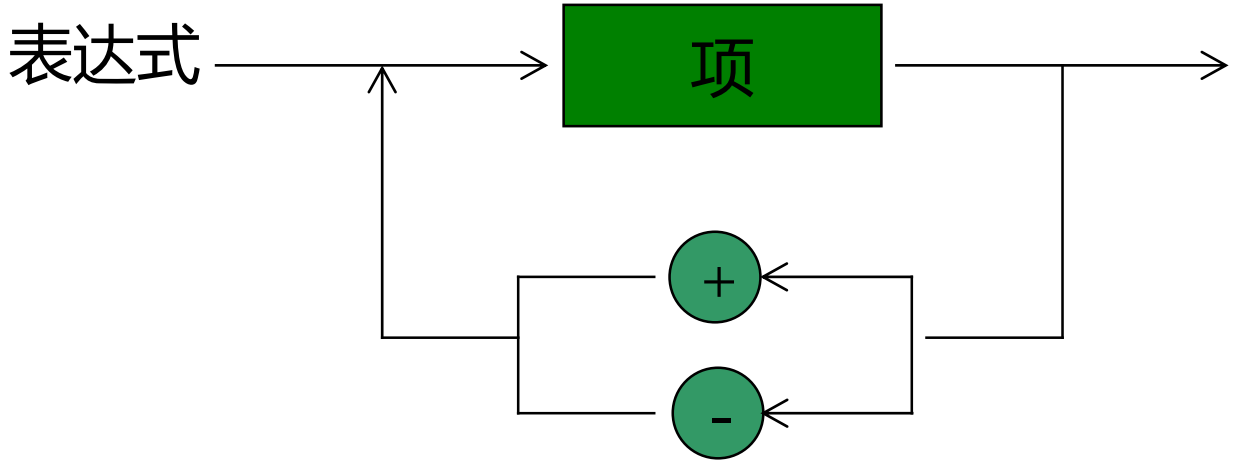
解决递归形式问题
四则运算表达式求值


甘肃张掖七彩丹霞（航拍）

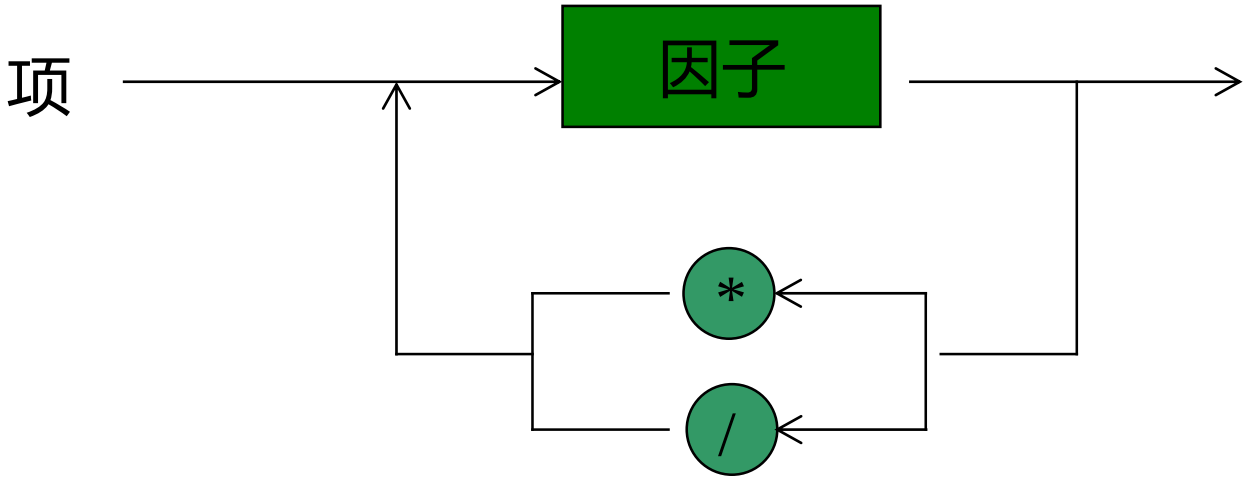
常规表达式计算

输入为普通四则运算表达式，仅由数字、+、 - 、*、/ 、(、) 组成，没有空格，要求求其值

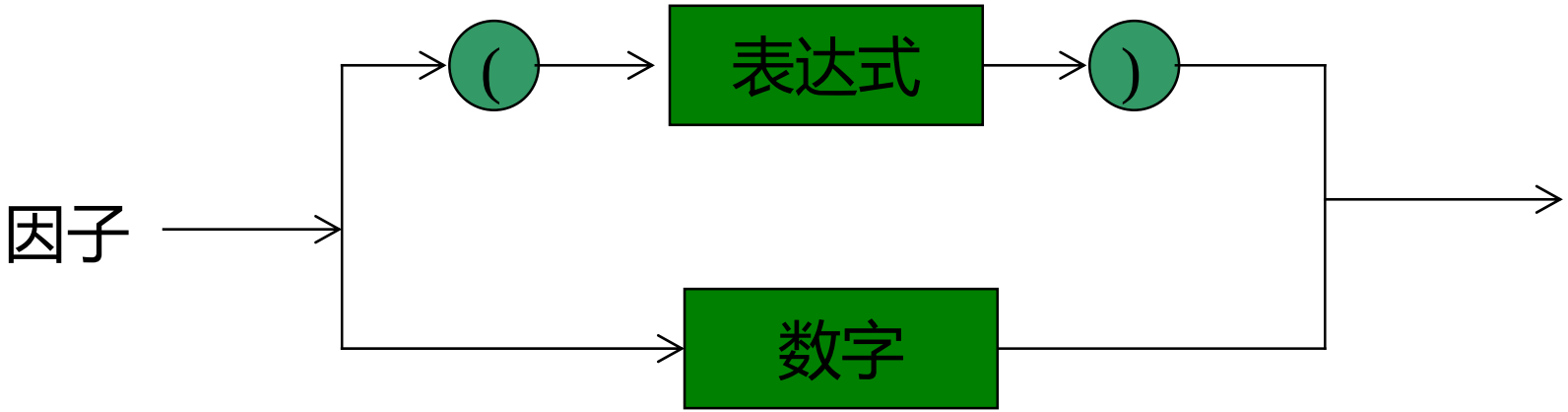
表达式的递归定义



表达式的递归定义



表达式的递归定义



表达式的递归计算

```
s = input()
ptr = 0                    #处理到s的第几个字符了

def expValue():            #读入并计算一个表达式的值
    global ptr
    v = itemValue()        #求第一项的值
    while ptr < len(s):
        if s[ptr] == '-':
            ptr += 1
            v -= itemValue()
        elif s[ptr] == '+':
            ptr += 1
            v += itemValue()
        else:
            break
    return v
```

表达式的递归计算

```
def itemValue(): #读入并计算一个项的值
        global ptr
        v = factorValue()
        while ptr < len(s):
                if s[ptr] == '*':
                        ptr += 1
                        v *= factorValue()
                elif s[ptr]  == '/':
                        ptr += 1
                        v /= factorValue()
                else:
                        break
        return v
```

表达式的递归计算

```
def factorValue():#读入并计算一个因子的值
        global ptr
        if s[ptr] == '(':
                ptr += 1
                v = expValue()
                ptr += 1
        else:
                start = ptr
                while ptr < len(s) and (s[ptr].isdigit() or
                                s[ptr] == "."):
                        ptr += 1
                v = float(s[start:ptr])
        return v

print("%.2f" % expValue())
```

解决递归形式问题
绘制雪花曲线

美国鹅颈湾

绘制雪花曲线（科赫曲线）

- 雪花曲线的递归定义

1）长为size，方向为x（x是角度）的0阶雪花曲线，是方向x上一根长为size的线段

2）长为size，方向为x的n阶雪花曲线，由以下四部分依次拼接组成：

1. 长为size/3，方向为x的n-1阶雪花曲线
2. 长为size/3，方向为x+60的n-1阶雪花曲线
3. 长为size/3，方向为x-60的n-1阶雪花曲线
4. 长为size/3，方向为x的n-1阶雪花曲线

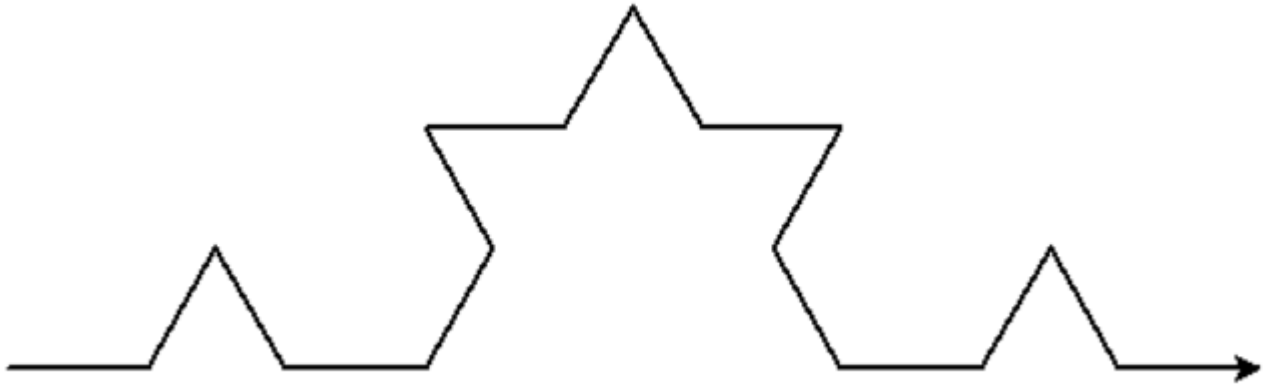
递归绘制雪花曲线（科赫曲线）

0阶0度雪花曲线

1阶0度雪花曲线

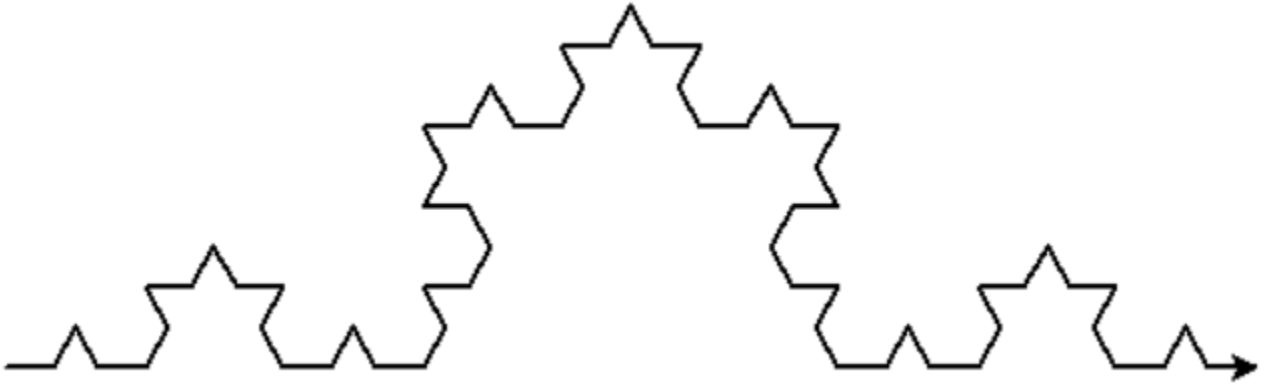
递归绘制雪花曲线（科赫曲线）

2阶0度雪花曲线

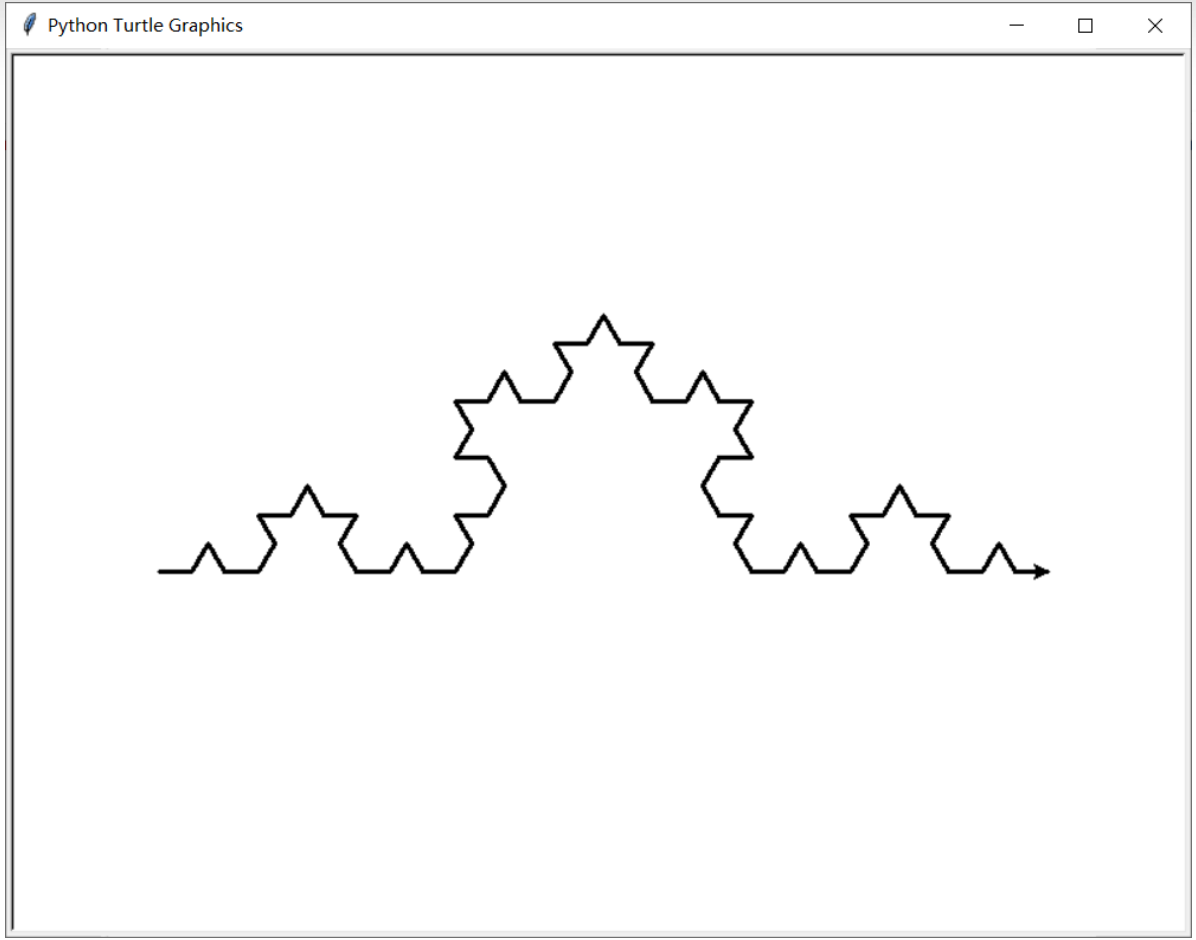


递归绘制雪花曲线（科赫曲线）

3阶0度雪花曲线



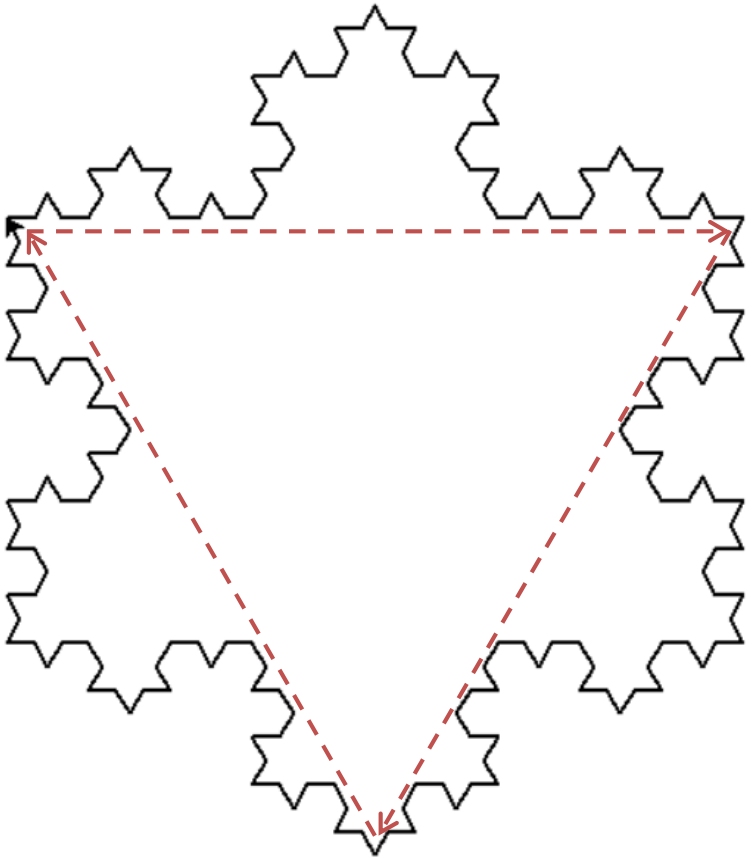
```
import turtle     #画图要用这个turtle包
def snow(n,size): #n是阶数目，size是长度 从当前起点出发，在当前方向画一个长度为size，阶为n的雪花曲线
    if n == 0:
        turtle.fd(size)   #笔沿着当前方向前进size
    else:
        for angle in [0,60,-120,60]: #对列表中的每个元素angle:
            turtle.left(angle)   #笔左转angle度 , turtle.lt(angle)也可
            snow(n-1,size/3)

turtle.setup(800,600)
#窗口缺省位于屏幕正中间，宽高800*600像素，窗口中央坐标(0,0)
#初始笔的前进方向是0度。正东方是0度，正北是90度
turtle.penup() #抬起笔
turtle.goto(-300,-50) #将笔移动到-300,-50位置
turtle.pendown() #放下笔
turtle.pensize(3) #笔的粗度是3
snow(3,600)      #绘制长度为600,阶为3的雪花曲线，方向水平
turtle.done()    #保持绘图窗口
```


Python Turtle Graphics

递归绘制雪花

- 由3段3阶雪花曲线组成

```
turtle.setup(800,800)
turtle.speed(1000)
turtle.penup()
turtle.goto(-300,100)
turtle.pendown()
turtle.pensize(2)
level =3
snow(level, 400)
turtle.right(120) #右拐 120 度
snow(level, 400)
turtle.right(120)
snow(level, 400)
turtle.done()
```



问题分解
例题:爬楼梯

美国加州1号公路17英里

用递归将问题分解为规模更小的子问题进行求解

例题: 爬楼梯

树老师爬楼梯，他可以每次走1级或者2级，输入楼梯的级数，求不同的走法数

例如：楼梯一共有3级，他可以每次都走一级，或者第一次走一级，第二次走两级，也可以第一次走两级，第二次走一级，一共3种方法。

输入

输入包含若干行，每行包含一个正整数N，代表楼梯级数，1 <= N <= 30输出不同的走法数，每一行输入对应一行

爬楼梯

输出

不同的走法数，每一行输入对应一行输出

样例输入

5
8
10

样例输出

8
34
89

爬楼梯

n级台阶的走法 =

先走一级后，n-1级台阶的走法 +
先走两级后，n-2级台阶的走法

f(n) = f(n-1)+f(n-2)

边界条件：

爬楼梯

n级台阶的走法 =

先走一级后，n-1级台阶的走法 +
先走两级后，n-2级台阶的走法

f(n) = f(n-1)+f(n-2)

边界条件： n < 0 0
n = 0 1

爬楼梯

n级台阶的走法 =

先走一级后，n-1级台阶的走法 +
先走两级后，n-2级台阶的走法

f(n) = f(n-1)+f(n-2)

边界条件： n < 0 0　　n = 0 1　　n = 1 1
n = 0 1　　n = 1 1　　n = 2 2

递归解法：

```
def stairs( n ):
    if n < 0:
        return 0
    if n == 0:
        return 1
    return stairs( n-1 ) + stairs( n-2 )
try:
    while True:
        N = int(input())
        print( stairs( N ))
except EOFError:
    pass
```

递推解法：

```
def stairs( n ):
    if n < 2:
        return 1
    a1,a2 = 1,1
    for i in range(2,n+1):
        a3 = a1 + a2
        a1,a2 = a2,a3
    return a3
try:
    while True:
        N = int(input())
        print( stairs( N ))
except EOFError:
    pass
```

问题分解
例题:出栈序列统计

河北草原天路

出栈序列统计

栈是常用的一种数据结构，有n个元素在栈顶端一侧等待进栈，栈顶端另一侧是出栈序列。你已经知道栈的操作有两种：push和pop，前者是将一个元素进栈，后者是将栈顶元素弹出。现在要使用这两种操作，由一个操作序列可以得到一系列的输出序列。请你编程求出对于给定的n，计算并输出由操作数序列1，2，...，n，经过一系列操作可能得到的输出序列总数。

输入

就一个数n(1≤n≤15)。

输出

一个数，即可能输出序列的总数目。

样例输入

3

样例输出

5

出栈序列统计

思路：

开始，有0个元素已经入过栈，栈里面有0个元素。问这种情况下有多少种出栈序列。

f(i,stackLen)表示已经有i个元素入过栈（其中有的可能已经出栈），栈里有stackLen个元素的情况下，会有多少种出栈序列。

整个问题就是要求f(0,0)。显然f(0,0) = f(1,1)，因第一步只能入栈一个元素

出栈序列统计

思路：

开始，有0个元素已经入过栈，栈里面有0个元素。问这种情况下有多少种出栈序列。

f(i,stackLen)表示已经有i个元素入过栈（其中有的可能已经出栈），栈里有stackLen个元素的情况下，会有多少种出栈序列。

整个问题就是要求f(0,0)。显然f(0,0) = f(1,1)，因第一步只能入栈一个元素

f(1,1) = f(1,0) + f(2,2)

因下一步有两种做法，即元素出栈，或者再压入一个新元素。所有的出栈序列，被分成两类。

出栈序列统计

思路：

推广至如何求`f(i,stackLen)`

先做一步。

若`stackLen > 0`

一步有两种做法：
1）将新元素入栈　　`f(i+1, stackLen+1)`
2）将栈顶元素弹出　　`f(i, stackLen-1)`

若`stackLen == 0`，则只有 `f(i+1,stackLen+1)`一种做法

出栈序列统计

思路：

边界条件：f(n,X) = 1， n为总元素个数，X为任何值。因此时的唯一的出栈序列就是把栈里的X个元素依次弹出。X = 0 则只有一个空序列。

出栈序列统计

```
#有重复计算，比较慢，复杂度指数级别。要用动态规划改进
def proc(i,stackLen):
        if i == n:
                return 1
        else:
                result = 0
                if stackLen > 0:
                        result += proc(i,stackLen-1)
                        result += proc(i+1,stackLen+1)
                else:
                        result = proc(i+1,stackLen+1)
        return result

n = int(input())
print(proc(0,0))
```

输出所有可能出栈序列

例如，输出"acdef"的所有可能出栈序列

```
total = 0    #出栈序列总数
result = []      #出栈序列
stack = []       #栈
s = ""           #比如是 "abcd"
def proc(i): #被调用时，已经有i个元素入过栈了
        global total
        global stack
        global result
        global s
        if i == len(s): #已经有i个元素都入过栈了
                while len(stack) > 0: #栈里所有元素弹出
                        result.append(stack.pop())
                total += 1
                r = "".join(result)
                print(r)
        else:
```

输出所有可能出栈序列

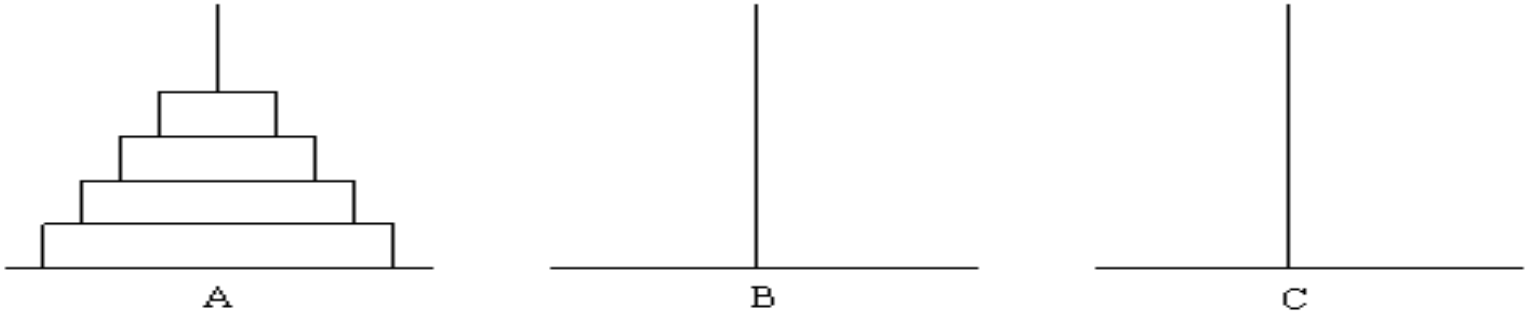
```
            if len(stack) > 0:
                    tmpStack = stack[:]     #备份stack
                    tmpResult = result[:] #备份当前出栈序列
                    result.append(stack.pop()) #处理元素出栈的做法
                    proc(i)
                    stack = tmpStack[:] #恢复stack
                    result = tmpResult[:]
                    stack.append(s[i])   #处理新元素入栈的做法
                    proc(i+1)
            else:
                    stack.append(s[i])
                    proc(i+1)
s = input()
proc(0) #0个元素入过栈
print(total)
```

用栈替代递归

日本箱根芦之湖

汉诺塔问题

古代有一个梵塔，塔内有三个座A、B、C，A座上有64个盘子，盘子大小不等，大的在下，小的在上（如图）。有一个和尚想把这64个盘子从A座移到B座，但每次只能允许移动一个盘子，并且在移动过程中，3个座上的盘子始终保持大盘在下，小盘在上。在移动过程中可以利用B座，要求输出移动的步骤。

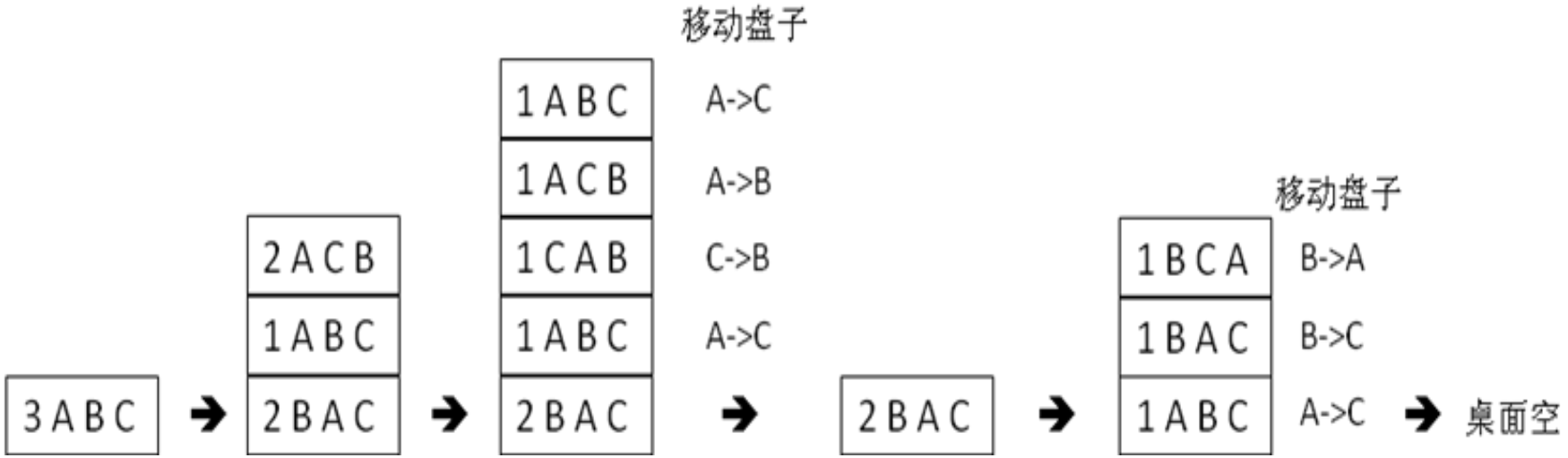


汉诺塔问题递归解法

```
def Hanoi(n, src,mid,dest):
    #将src座上的n个盘子，以mid座为中转，移动到dest座
    if( n == 1) : #只需移动一个盘子
        print(src +  "->" + dest) #直接将盘子从src移动到dest即可
        return
    Hanoi(n-1,src,dest,mid)   #先将n-1个盘子从src移动到mid
    print(src + "->" + dest) #再将一个盘子从src移动到dest
    Hanoi(n-1,mid,src,dest)   #最后将n-1个盘子从mid移动到dest

n = int(input())          #要移动n个盘子
Hanoi(n, 'A', 'B', 'C')
```

汉诺塔问题手工解法(三个盘子）

信封堆,每个信封放一个待解决的问题



汉诺塔问题非递归解法

```
class problem:
    def __init__(self,n,src,mid,dest):
        self.n,self.src,self.mid,self.dest = n,src, mid, dest
#一个Problem对象代表一个子问题，将src上的n个盘子，以mid为中介，移动到dest
stack = []
n = int(input())
stack.append(problem(n,'A','B','C')) #初始化了第一个信封
while len(stack) > 0: #只要还有信封，就继续处理
    curPrb = stack.pop() #取最上面的信封，即当前问题
    if curPrb.n == 1:
        print( curPrb.src + "->" +curPrb.dest)
    else:
        #先把分解得到的第3个子问题放入栈中
        stack.append(problem(curPrb.n -1,curPrb.mid,
                             curPrb.src, curPrb.dest))
        #再把第2个子问题放入栈中
        stack.append(problem(1,curPrb.src,curPrb.mid,curPrb.dest))
```

汉诺塔问题非递归解法

```
#最后放第1个子问题，后放入栈的子问题先被处理
stack.append(problem(curPrb.n -1,curPrb.src,
                     curPrb.dest,curPrb.mid))
```

求斐波那契数列第n项的递归解法

- 编译器生成的代码自动维护一个栈，相当于信封堆，每栈的每一层代表一个子问题
- 在进入下一层函数调用前，会将本层所有参数和局部变量，以及返回地址入栈中
- 返回地址表示了一个子问题解决后接下来应该做什么，下面的 (1),(2)就是返回地址
- 函数调用返回时，就会退一层栈

```
def fib(n):
    if n == 1 or n == 2:
        return 1
    else:
        tmp = fib(n-1)    #(1)   返回后要去算fib(n-2)
        return tmp + fib(n-2)#(2)  返回后要加上栈里面的tmp然后再返回
```

用栈模拟递归求斐波那契数列第n项的过程

```
class problem:
    def __init__(self,n_,tmp_,retAdr_):
        self.n,self.tmp,self.retAdr = n_,tmp_,retAdr_
#n 问题   tmp 局部变量   retAdr 返回地址

def fib(n):
    stack = []
    stack.append(problem(n,None,0))
    retVal = None                #刚弹出栈的那个子问题的答案
    while len(stack) > 0:
        curPrb = stack[-1]
        if curPrb.n == 1 or curPrb.n == 2:
            stack.pop()          #子问题解决
            retVal = 1
```

```
def fib(n):
    if n == 1 or n == 2:
        return 1
    else:
        tmp=fib(n-1)#(1)
        return tmp+fib(n-2)#(2)
```

用栈模拟递归求斐波那契数列第n项的过程

```
    else:
        if curPrb.retAdr == 0: #第一次碰到这个子问题
            curPrb.retAdr = 1 #改成第二次碰到
            newPrb = problem(curPrb.n-1,None,0)
            stack.append(newPrb)
        elif curPrb.retAdr == 1: #第二次碰到
            curPrb.tmp = retVal
            curPrb.retAdr = 2   #改成第三次碰到
            newPrb = problem(curPrb.n-2,None,0)
            stack.append(newPrb)
        else:   #第三次碰到，则子问题可以解决了
            retVal += curPrb.tmp
            stack.pop()
return retVal
```

```
def fib(n):
    if n == 1 or n == 2:
        return 1
    else:
        tmp=fib(n-1)#(1)
        return tmp+fib(n-2)#(2)
```