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

....

递归替代循环

递归替代循环

```
def doSomething():
    print("ok")
n = 8
for i in range(n):
    doSomething()
```

递归替代循环

```
def loop(n,f):
    if n == 0:
        return
    f()
    loop(n-1,f)
def doSomething():
    print("ok")
loop(8,doSomething)
```

递归求列表最大值

循环解法：

```
def maxValue(a):
    ans = a[0]
    for x in a:
        if ans < x:
            ans = x
    return ans
```

递归求列表最大值

$O(n^2)$的递归解法：

```
def maxValue(a):
    if len(a) == 1:
        return a[0]
    b = maxValue(a[1:]) #递归求a[1:]的最大值
    if a[0] < b:
        return b
    else:
        return a[0]
```

递归求列表最大值

O(n)的递归解法：

```
def maxValue(a):
    def maxV(start): #求a[start:]中的最大值
        if start == len(a)-1: #a[start:]中只有一个元素
            return a[start]
        b = maxV(start+1) #递归求a[start+1:]的最大值
        if a[start] < b:
            return b
        else:
            return a[start]
    return maxV(0)
```

递归求列表的反转(颠倒)

$O(n^2)$的递归解法：

```
def reverse(a): #返回前后颠倒的a(a是列表)
    if len(a) == 0:
        return []
    else:
        return [a[-1]] + reverse(a[:-1])

print(reverse([1,2,3,4])) #>>[4, 3, 2, 1]
print(reverse([])) #>>[]
```

递归求列表的反转(颠倒)

O(n)的递归解法：

```
def reverse(a): #返回前后颠倒的a(a是列表)
    result = []
    def rev(length): #返回 a[0:length]的颠倒
        if length == 0:
            return
        result.append(a[length-1])
        rev(length-1)
    rev(len(a))
    return result
```

递归替代多重循环
例题: N皇后

宁夏中卫沙坡头

N皇后问题

属于一类典型的问题：有若干个位置，每个位置可以放不同东西。求有哪些摆法（或多少种摆法，或至少一种摆法）能够满足指定的条件。

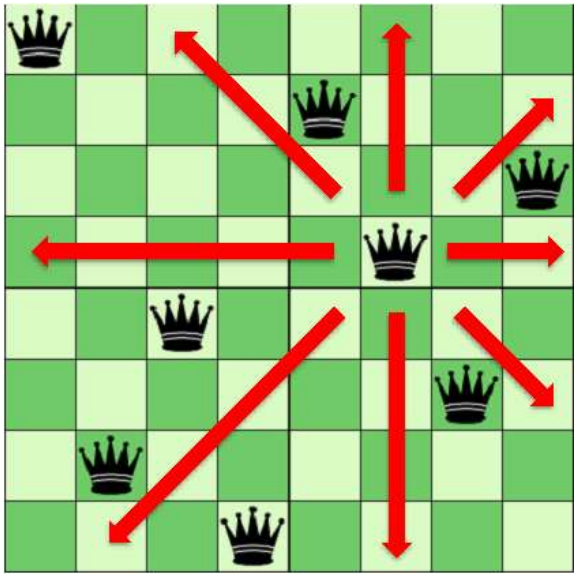
本质是有若干变量，每个变量可以有若干不同取值，求能满足指定条件的变量的取值的组合（求一种，或求全部）

解法：用递归枚举每个位置(变量)可能的摆法(取值)

dfs(i)表示从第0个到第i-1个位置已经摆好的情况下，再往下有哪些成功的摆法

八皇后问题

- 国际象棋棋盘是由8×8共64个方格构成。要求在棋盘上摆8个皇后，使得他们互相之间都吃不着，即没有两个皇后处于同一行、同一列、或正方形的对角线(斜线)上。



八皇后问题

- 用八重循环枚举所有皇后可能的摆法，每行的皇后有8种摆法，共8行，所以总的摆法是8^8种，对每种摆法验证是否符合要求即可找到所有合法的摆放方案(92种）。

- 4皇后问题输出结果：

1 3 0 2

2 0 3 1

4皇后解题程序

```
result = [0] * 4          #等价于 result = [0,0,0,0]，存放摆放方案
#result[i]表示第i行的皇后已经放在result[i]这个位置
def isOk(n,pos):          #判断第n行的皇后放在位置pos是否可行
#此时第0行到第n-1行的皇后的摆放位置已经存放在result[0]至result[n-1]中
    for i in range(n):#检查位置pos是否会和前面0~n-1行已经摆好的皇后冲突
        if result[i] == pos or abs(i-n) == abs(result[i] - pos):
            return False
    return True
```

4皇后解题程序

```
def main():
    for p0 in range(4):          #枚举第0行所有可能位置
        result[0] = p0           #第0行的皇后放在第p0列
        for p1 in range(4):              #枚举第1行所有可能位置
            if isOk(1,p1):
                result[1] = p1   #第1行的皇后放在第p1列
                for p2 in range(4): #枚举第2行所有可能位置
                    if isOk(2,p2):
                        result[2] = p2
                        for p3 in range(4): #枚举第3行所有可能位置
                            if isOk(3,p3):
                                result[3] = p3
                                for x in result: #找到成功摆法，输出之
                                    print(x,end = " ")
                                print("")
main()
```

N皇后问题

样例输入

```
4
```

样例输出

```
1 3 0 2
2 0 3 1
```

N皇后解题程序

```
result = [0] * 12          #本程序最多能解决12皇后问题
#result[i]表示第i行的皇后已经放在了result[i]列
def isOk(n,pos):           #判断第n行的皇后放在第pos列是否和已经摆好的皇后冲突
    for i in range(n): #检查位置pos是否会和前0~n-1行已经摆好的皇后冲突
        if result[i] == pos or abs(i-n) == abs(result[i] - pos):
            return False
    return True
```

N皇后解题程序

```
def queen(N,m):#解决N皇后问题，现在第0行到第m-1行的m个的皇后已经摆放好了
                    #要摆放第m行的皇后，
    if m == N: #已经摆好了N个皇后，说明问题已经解决，输出结果即可
        for k in range(N):
            print(result[k], end=" ")
        print("")
        return True
    succeed = False
    for i in range(N):      #枚举所有位置
        if isOk(m,i):        #看可否将第m行皇后摆在第i列
            result[m] = i   #可以摆在第i列，就摆上
            succeed = queen(N,m+1) or succeed    #接着去摆放第i+1行的皇后
    return succeed
N = int(input())
if not queen(N,0):
    print("NO ANSWER")
```

N皇后解题程序

#如果只要找一组解

```
def queen(N,m):#解决N皇后问题，现在第0行到第m-1行的m个的皇后已经摆放好了
                    #要摆放第m行的皇后，
    if m == N: #已经摆好了N个皇后，说明问题已经解决，输出结果即可
        for k in range(N):
            print(result[k], end=" ")
        print("")
        return True
    succeed = False
    for i in range(N):      #枚举所有位置
        if isOk(m,i):        #看可否将第m行皇后摆在第i列
            result[m] = i   #可以摆在第i列，就摆上
            if queen(N,m+1): return True    #接着去摆放第i+1行的皇后
    return succeed
```

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
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

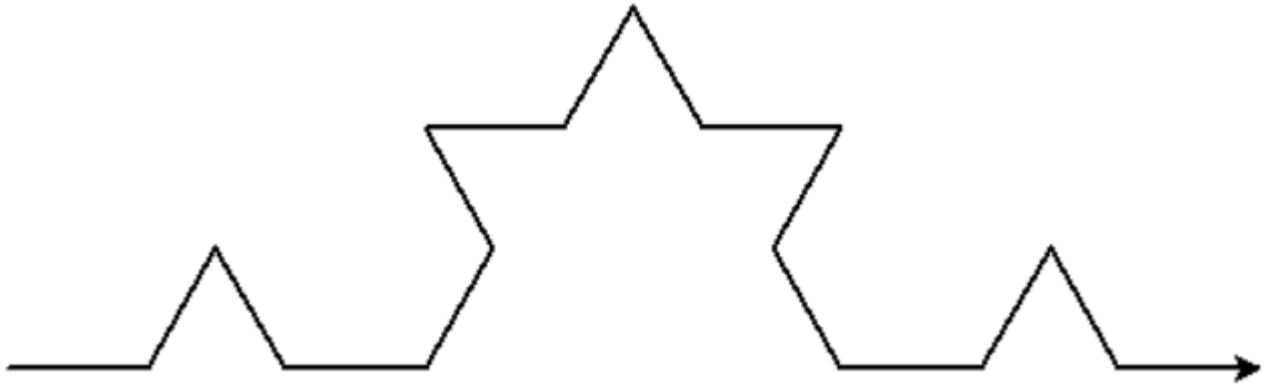
递归绘制雪花曲线（科赫曲线）

0阶0度雪花曲线

1阶0度雪花曲线

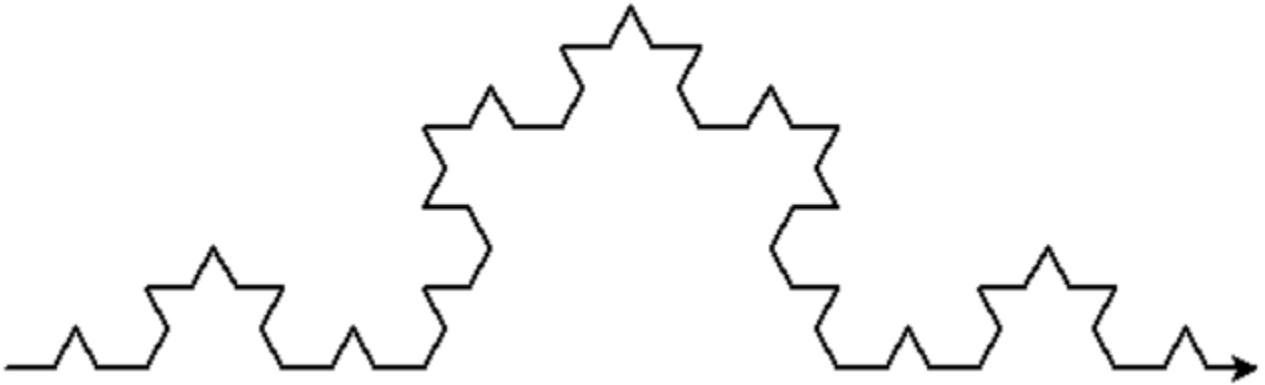
递归绘制雪花曲线（科赫曲线）

2阶0度雪花曲线

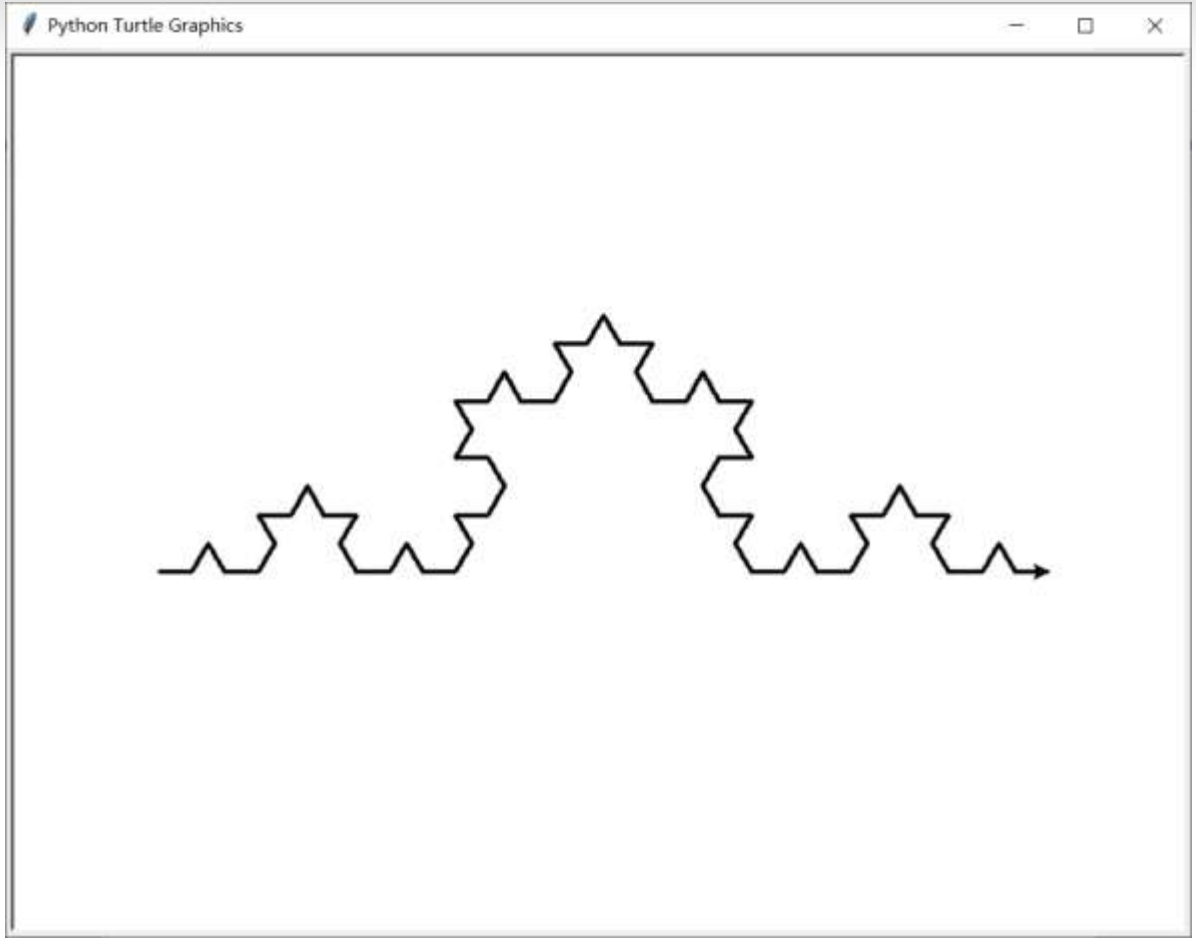


递归绘制雪花曲线（科赫曲线）

3阶0度雪花曲线



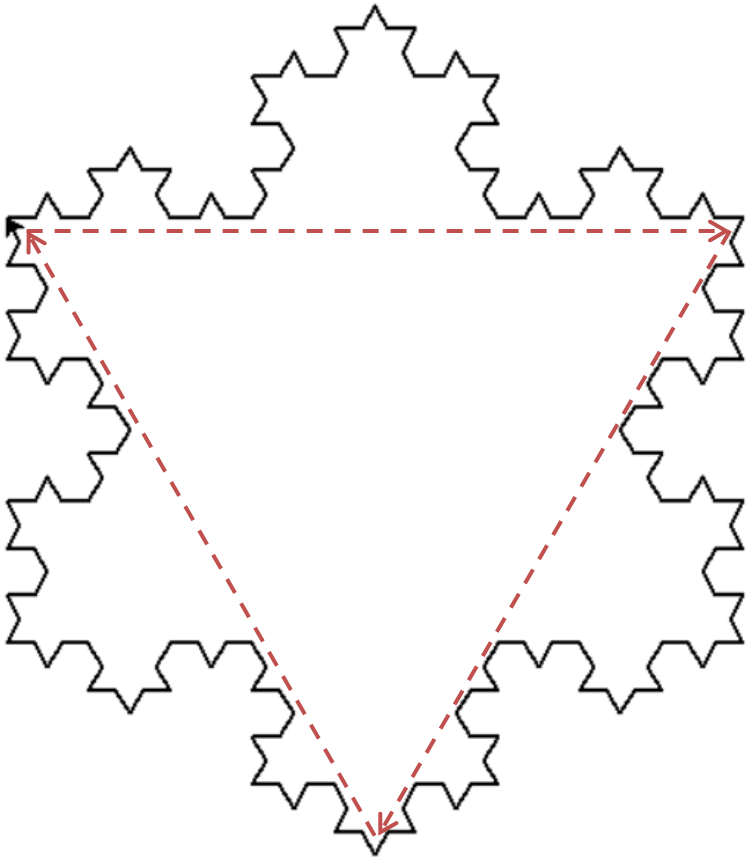
```
import turtle     #画图要用这个turtle包
def snow(n,size): #n是阶数目，size是长度 从当前起点出发，在当前方向画一个长度为size，阶为n的雪花曲线
    if n == 0:
        turtle.fd(size)   #笔沿着当前方向前进size
    else:
        for angle in [0,60,-120,60]: #对列表中的每个元素angle:
            turtle.left(angle)   #笔左转angle度 , turtle.lt(angle)也可
            snow(n-1,size/3)

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)
```

栈和递归的关系

日本箱根芦之湖

用栈实现递归

- 编译器生成的代码自动维护一个栈，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
- 函数调用返回时，就会退一层栈

用栈实现递归

- 求斐波那契数列第n项的函数的返回地址: a,b,c三处

```
1.  def fib(n):
2.      if n == 1 or n == 2:
3.          return 1
4.      else:
5.          t = fib(n-1)
6.          return t + fib(n-2)
7.  print(fib(4))
```

a

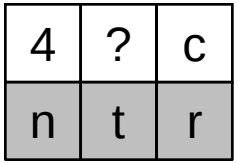
b

c

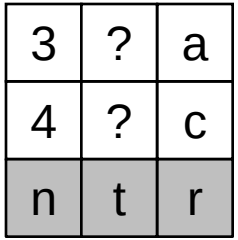
写 return fib(n-1)+fib(n-2)
本质上也需要临时变量t存放fib(n-1)的返回值

用栈实现递归

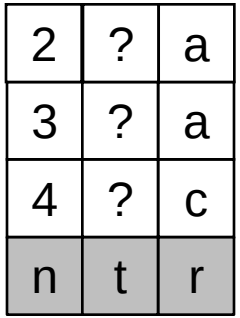
- 执行fib(4)时栈的变化



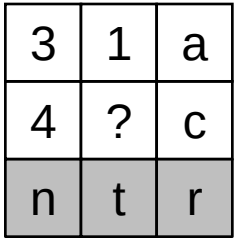
(1)



(2)

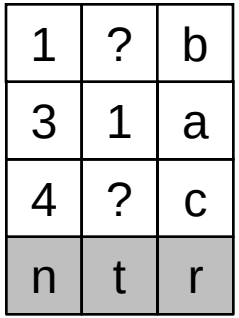


(3)

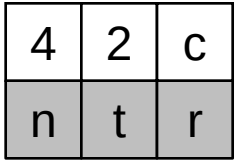


(4)

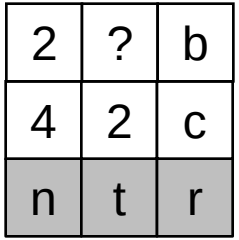
r是返回地址



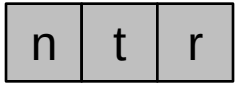
(5)



(6)



(7)



(8)

用栈模拟递归求斐波那契数列第n项的过程

```
def fib(n):
    class Status:  #放入栈中的元素
        def __init__(self,n,t,r):
            self.n,self.t,self.r = n,t,r
    stack = [Status(n,None,'c')]
    retVal = retAdr = None  #retVal是返回值，retAdr是返回地址
    while stack != []:
        status = stack[-1]
        n = status.n
        if n == 2 or n == 1:
            retVal = 1
            retAdr = status.r
            stack.pop()
```

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

用栈模拟递归求斐波那契数列第n项的过程

```
    else:
        if retAdr == None:
            stack.append(Status(n-1, None, 'a'))
        elif retAdr == 'a':
            stack[-1].t = retVal
            stack.append(Status(n-2,None,'b'))
            retAdr = None
        elif retAdr == 'b':
            retVal = status.t + retVal
            retAdr = status.r
            stack.pop()
        else:
            stack.pop()
return retVal
```

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