递归

4 递归

4.1 目标

本章目标如下：

- 了解：某些难以处理的问题也许用递归可以轻松解决。
- 掌握：设计递归式程序。
- 了解并应用：递归三大法则。
- 实现：问题的递归式描述。
- 了解：计算机系统中递归的实现

4.2 何为递归

递归是一种解决问题的方法，将一个问题分解为越来越小的子问题，直到问题的规模小到可以被轻松解决。通常来说，递归中有函数对自己本身的调用。从表面上来看似乎并没有什么特别指出，然而递归算法对于某些问题具有一针见血的气消。

4.3 对数字列表求和

首先来看一个并不需要递归也可以解决的问题。假设相对一个数字列表求和，比如[1,3,5,7,9]。迭代式算法如可执行代码1所示。函数使用求和变量（theSum）来计算列表中数字之和。

** 可执行代码1： 迭代求和 **

```
def listsum(numList):
    theSum = 0
    for i in numList:
        theSum = theSum + i
    return theSum

print(listsum([1,3,5,7,9]))
```

那么，现在假设不可以使用while或者for循环，那又该如何做？数学家可能会想到加法本身是一种接受两个参数的函数，即一对数字。对于该列表求和问题便可转化为多对数字求和问题。先写为全括号化的表达式，比如：

$$((((1+3)+5)+7)+9)$$

也可以反方向来：

$$(1+(3+(5+(7+9))))$$

注意到最里面的括号组（7+9）是不需要迭代或者其它特殊结构便可解决的问题。实际上，可以将求和问题写为以下简化形式：

$$total=(1+(3+(5+(7+9))))total=(1+(3+(5+16)))total=(1+(3+21))total=(1+24)total=25$$

如何将这种思想转化为Python程序？首先，将求和问题按Python列表的方式来考虑。可能读者会认为对列表的求和即是列表中第一个元素numList[0]与其它所有元素numList[1:]之和，以函数形式写出即为：

$$listSum(numList)=first(numList)+listSum(rest(numList))$$

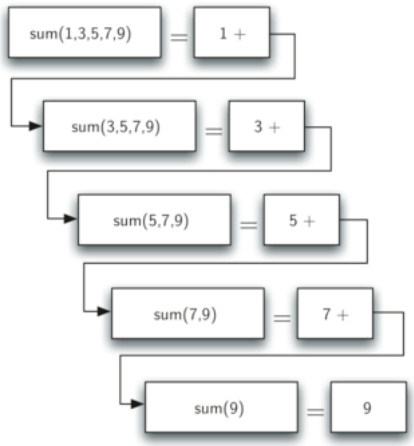
在以上表达式中，first(numList)返回列表中第一个元素，而rest(numList)返回的是去除了第一个元素的原列表（也是一个列表）。用Python表达式可以写为如可执行代码2：

** 可执行代码2：递归求和（第一次递归） **

```
def listsum(numList):
   if len(numList) == 1:
        return numList[0]
   else:
        return numList[0] + listsum(numList[1:])

print(listsum([1,3,5,7,9]))
```

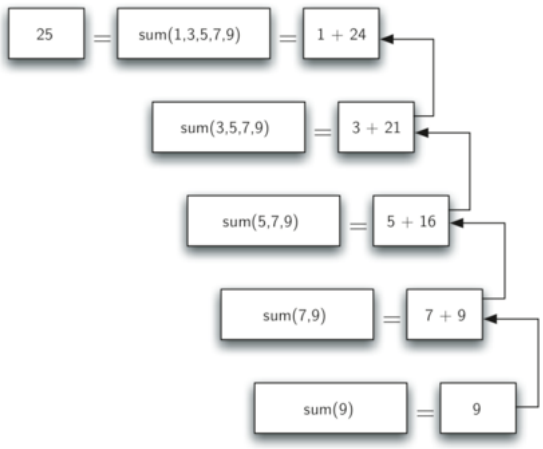
在这段代码中，有些地方需要注意。首先，在行2检查了列表是否仅有1个元素。这一判定很关键，它是控制函数结束的必要语句。长度为1的列表求和显然便是其本身。其次，在行5该函数调用了它自己本身。这便是将listsum归为递归算法的原因。递归函数是一种调用自身的函数。

图1演示了对列表[1,3,5,7,9]求和的递归调用。读者可以将这些有序调用认为是一次次的问题简化过程。每调用一次，便转化为解决一个更小更容易的问题，知道问题规模不能再缩小。



image

当问题已经最简化时，便开始将每个问题的答案拼装起来直到原始问题被解决。如图2所示，listsum沿着调用顺序返回。当listsum回到了最顶部的问题时，原始问题便得到了解决。



image

4.4 递归三定律

跟阿西莫夫（Asimov）机器人三定律一样，递归算法也必须遵守三条重要定律：

1. 递归算法必须要有个**约束条件**作为出口。
2. 递归算法必须不断改变自身状态并向约束条件演进。
3. 递归算法必须递归地调用自身。

现在来仔细研究一下这三条定律以及在listsum中是如何得到体现的。首先，约束条件是终止算法递归的条件。一般来说，约束条件都是问题规模已经小到足以直接解决的时候。在listsum算法中，约束条件是长度为1的列表。

为满足第二条定律，必须设法改变当前状态并使得算法向约束条件靠近。状态的改变意味着，算法使用的一些数据将被改变。通常来说，即为问题规模以某种方法缩小。在listsum算法中，基本数据结构是列表，因此需要在列表上想办法实现状态的改变。因为约束条件是列表长度为1，向约束条件演进的一种很自然的实现方式就是缩小列表，这便是可执行代码在行5所做的事：将列表长度缩小，然后将其作为参数来调用listsum本身。

最后一条定律是算法必须调用自己本身，这是递归根本的定义。递归对于许多初学者来说是个有些难懂的概念。读者应当已经发现，递归函数是一种很优秀的方法，借助它可以将大规模的问题拆分为小问题，解决这些小问题只需要写一个简单的函数来对它们处理即可。表面看起来，递归似乎是陷入了一种循环。但实际上它在逻辑上并不是循环的，递归的逻辑是用优美简洁的表达式将问题拆分为更小更简单的。

在本章的剩余部分将进一步研究递归，每个例子中都着重于用如何利用三定律来设计算法。

4.5 将整数转化为以任何进制表示的字符串

假设需要将整数转化为二进制与十六进制之间的某进制表示的字符串形式。比如，整数10的十进制字符串为"10"，而二进制字符串为"1010"。有很多算法可以解决该问题，包括前文说过的以栈来解决，但是用递归思想来解决该问题还是最简洁的。

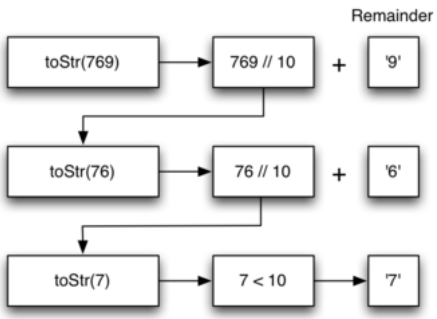
接下来以十进制的整数768为例演示一个具体的问题。假设用一组字符串来对应前10个数字，比如说convString = "0123456789" 。通过在字符串中检索，很容易便可将10以内的数字进行转化，例如整数9，那么字符串便是convString[9]，即"9"。将769拆分为三个单独的数字7，6，9，将其转化为字符串也很简单了。小于10的整数看起来是不错的约束条件。

确定基本进制后，整个算法将包括3部分：

1. 将原数字转化为一系列单数字。
2. 利用检索将单数字转为字符串。
3. 将各单数字字符串连接起来，形成最终的结果。

下一步是实现状态的改变，使得程序向约束条件靠近。因为研究对象是数字，不妨回想一想什么数学运算可以减少数字，最有可能的是减法和除法。虽然说减法也许是可行的，但有些令人困扰的是在这里如何确定减法操作的两个对象。而除法则清晰很多。

将整数769除以10，可以得到76余9，这两个数字很有用。首先，余数是一个小于进制（10）的数，因此可以直接使用检索转化为字符串。其次，得到了一个小于初始数字的数，这便向约束条件靠近了一步。现在需要做得便是将76转化为它的字符表示。同样，利用除法可以得到7余6。最后，问题退化为转化7。流程如图3所示。注意，需要被保存的数被放置在示意图右侧。



image

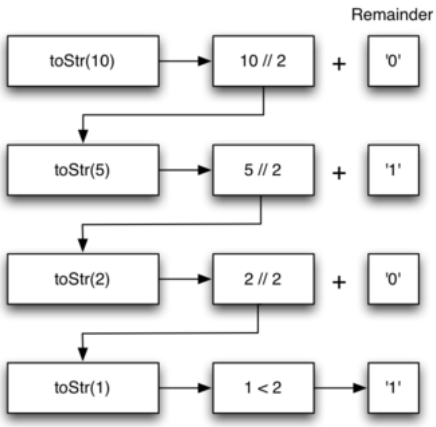
可执行代码1实现了上述的算法。

** 可执行代码1：迭代算法——整数转化为字符串 **

```
def toStr(n,base):
   convertString = "0123456789ABCDEF"
   if n < base:
      return convertString[n]
   else:
      return toStr(n//base,base) + convertString[n%base]

print(toStr(1453,16))
```

注意在第3行，检测了约束条件，即n小于目标进制。一旦检测到约束条件，便停止递归，然后根据convertString序列返回结果。在行6，通过递归调用和除法降低问题规模来同时满足了第2和第3定律。

再来追踪一次该算法，这次将整数10转化为2进制形式字符串（"1010"）



image

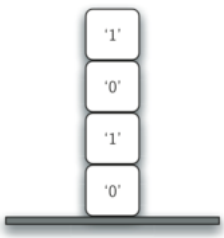
如图4所示，得到了所需结果，似乎数字的顺序错了。但是实际上，算法运行是正常的，因为是在行6中是先进行的递归调用，然后再将余数的字符串标识加上。

4.6 栈桢：实现递归

设想，如果不按上文那样将递归调用的单字符结果依次连接起来得到结果，而是将算法修改为在递归调用之前将字符串压入栈中。调整后的算法如可执行代码1所示。
** 可执行代码1：利用栈实现整数转化为字符串 **

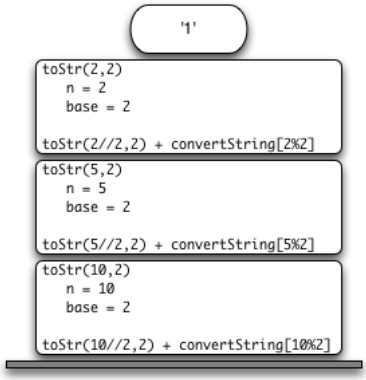
```
from pythonds.basic.stack import Stack

rStack = Stack()

def toStr(n,base):
    convertString = "0123456789ABCDEF"
    while n > 0:
        if n < base:
            rStack.push(convertString[n])
        else:
            rStack.push(convertString[n % base])
        n = n // base
    res = ""
    while not rStack.isEmpty():
        res = res + str(rStack.pop())
    return res

print(toStr(1453,16))
```

每一次调用toStr，都将一个字符放入栈中。就前一个例子来说，可以发现在第4次调用后，得到的栈如图5所示。注意，现在可以直接将字符从栈中推出，再将他们链接为最后的结果"1010"即可。



../_images/recstack.png

此例说明了Python是如何实现递归函数调用的。在Python中，调用一个函数时，系统会分配一个栈桢用来处理该函数中的局部变量。当函数执行完毕，返回值被放入栈顶等待被调用的函数处理。如图6所示。



../_images/newcallstack.png

注意对toStr(2//2, 2)的调用在栈中返回了值“1”。该返回值在表达式“1” + convertString[2%2]用来取代函数调用toStr(1,2)，执行后得到的结果“10”又被放在栈顶。这样一来，Python的调用栈取代了代码4中所使用的栈。在该例子中，读者可以认为栈顶的返回值替代了累加变量的作用。

栈桢也为对应函数使用的变量规定了作用域。虽然多次调用同一个函数，每一次调用都为函数的局部变量创建一个新的作用域。

4.7 图示递归

在上一节中介绍了一些使用递归算法可以轻松解决的问题，然而很难给出递归函数运行的可视化方法或者思维导图，这就使得递归算法不是那么容易掌握的。本节将研究一些递归的例子并画出示意图。通过研究这些示意图的形成过程，便会对递归过程有更加深刻的理解。

这里使用Python的海龟图像模块turtle进行可视化。turtle模块是所有Python版本的标配并且很容易上手，其符号也很简单。创建一只海龟，该海龟可以前进/后退/左转/右转等，并且可以控制尾巴的抬起和放下。当海龟的尾巴放下且在移动时，便会绘制出线。为了提高观赏性，可以改变海龟尾巴的宽度以及画笔的颜色。

下面简单说明turtle作图的基础，这里会利用turtle绘制一个螺旋线。如可执行代码所示。引入turtle模块后，创建一个turtle实例。当turtle生成后，为其创建一个窗口用来绘图。接下来定义drawSpiral函数。该函数的约束条件是待画线的长度（以len参数表示）减少到小于或等于0。若长度仍大于0，则命令海龟前进len单位然后右转90度。通过再次调用drawSpiral并且传入减小后的长度实现递归。在可执行代码1的最后，调用了函数myWin.exitonclick()，这是窗口的一个很便利的方法，它将海龟置于待机状态直到在窗口内发生点击操作，然后页面才清空并退出。

可执行代码1：使用海龟绘制递归螺旋线

```
import turtle

myTurtle = turtle.Turtle()
myWin = turtle.Screen()

def drawSpiral(myTurtle, lineLen):
    if lineLen > 0:
        myTurtle.forward(lineLen)
        myTurtle.right(90)
        drawSpiral(myTurtle,lineLen-5)

drawSpiral(myTurtle,100)
myWin.exitonclick()
```

这就是海龟模块中所需要了解的一些基本知识，据此便可以绘制漂亮的团了。下一个程序将绘制一个分形树。分形是数学的一个分支，它与递归有许多相似点。对于某个图形，无论放大多少倍，其基本形状都是一致的，此时便称之为**分形**。很多自然现象具有分形特点，比如海岸线，雪花，山脉，树以及灌木等，这使得CG（Computer Generated）电影具有了可行性。接下来的例子中将绘制一颗分形树。

为了画出这棵树，考虑如何用分形思想来描述一棵树是很有用的。回想一下，分形就是无论放大多少倍都具有相似性的东西。将这种思想用来研究树或者灌木，可以发现即使是小树枝也具有与整棵树一致的形状和特点。据此可以将树定义为一个左右分叉，且左右各有一个更小的“树”（分叉）。引入递归的概念，那么树便是这些递归的小树构成的。

现在将上述思路转换为Python代码。代码1利用海龟来绘制了一颗分形树。可以发现，行5和行7进行了递归调用。行5在右转20度后进行了递归调用，这便是前文提到的右分叉；在行7，海龟在左转40度后进行了另一次递归调用，这即为左分叉。此外，注意到每次调用tree后，都对branchlen参数进行了减法运算，这保证了递归树是越来越小的。读者应该也注意到了行2的**if**语句即是约束条件，在branchLen过小时触发。

代码1

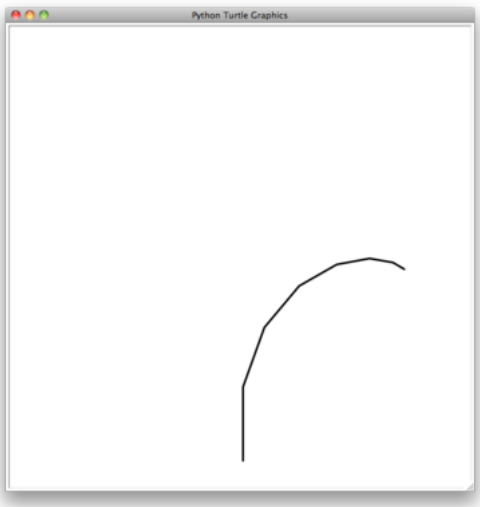
```
def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15,t)
        t.left(40)
        tree(branchLen-10,t)
        t.right(20)
        t.backward(branchLen)
```

完整的代码如可执行代码2所示。在运行代码前，思考一下是分叉树是如何绘制的。仔细观察这些调用，想想分叉树是如何展开的。它是同时对称地绘制左右分叉还是先右再左?

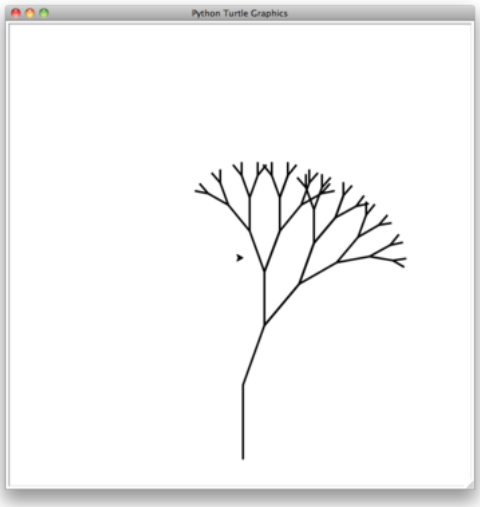
可执行代码2:递归绘制分叉树

```
def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15,t)
        t.left(40)
        tree(branchLen-15,t)
        t.right(20)
        t.backward(branchLen)

def main():
    t = turtle.Turtle()
    myWin = turtle.Screen()
    t.left(90)
    t.up()
    t.backward(100)
    t.down()
    t.color("green")
    tree(75,t)
    myWin.exitonclick()

main()
```

注意分叉树上的每一个分叉点是如何相应递归调用，以及分叉树是如何一路朝右绘制到最短的那个分支的。如图2所示。现在，仔细观察，程序是在整个树的右侧绘制完成后再沿着分叉返回，分叉树的右半侧如图1所示。接下来将绘制树的左侧，然而并不是直接画出最大的左侧，而是在每次到达最短枝前，先画出左分支的整个右半部分，然后再返回画左半部分。



../_images/tree1.png

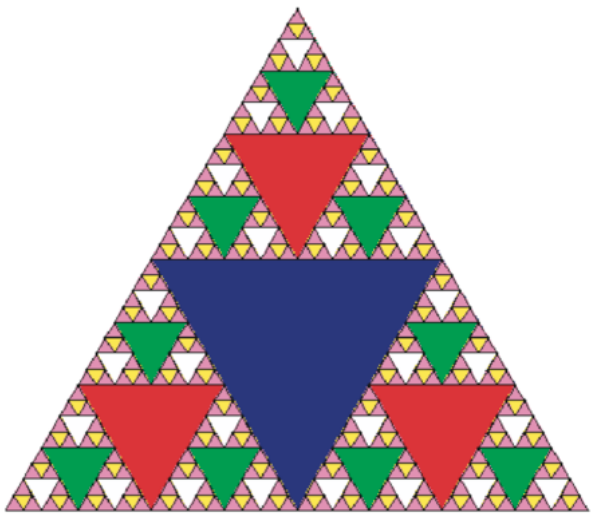


../_images/tree2.png

分形树只是一个起点，此外，读者也应该注意到这棵树看起来并不是那么真实，因为自然界中并不会真的像计算机程序那样具有高度对称性。本章的最后将会介绍一些有趣的选项，可以使得分叉树看起来更真实。

4.8 谢尔宾斯基三角形

谢尔宾斯基三角形也是一种具有自相似性的分形图形，如图三所示。谢尔宾斯基三角形是一种三向递归算法。手绘谢尔宾斯基三角形的方法很简单。从某个大的三角形开始，将这个大三角形分为4个小的三角形：连接每条边的中点；剔除掉新的4个三角形中间的那个，对剩余的三个采取同样的步骤；以此类推，不断递归循环。如果笔足够细的话，是可以无穷地画下去的。



../_images/sierpinski.png

既然这个算法可以无限运行，那就必须得找出约束条件。这里被设置为了任意的划分次数，有时这被称为相似性维数。每次进行递归调用，就将维数减1直到为0。代码如可执行代码1所示。

可执行代码1:绘制谢尔宾斯基三角形

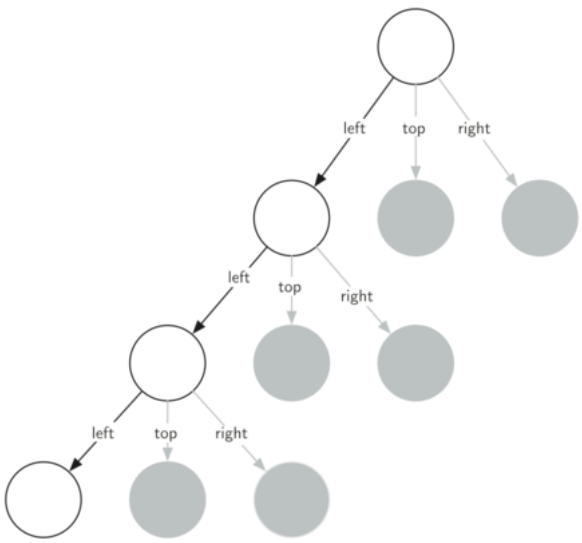
```
import turtle

def drawTriangle(points,color,myTurtle):
    myTurtle.fillcolor(color)
    myTurtle.up()
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.down()
    myTurtle.begin_fill()
    myTurtle.goto(points[1][0],points[1][1])
    myTurtle.goto(points[2][0],points[2][1])
```

```
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.end_fill()

def getMid(p1,p2):
    return ( (p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2)

def sierpinski(points,degree,myTurtle):
    colormap = ['blue','red','green','white','yellow',
                'violet','orange']
    drawTriangle(points,colormap[degree],myTurtle)
    if degree > 0:
        sierpinski([points[0],
                        getMid(points[0], points[1]),
                        getMid(points[0], points[2])],
                   degree-1, myTurtle)
        sierpinski([points[1],
                        getMid(points[0], points[1]),
                        getMid(points[1], points[2])],
                   degree-1, myTurtle)
        sierpinski([points[2],
                        getMid(points[2], points[1]),
                        getMid(points[0], points[2])],
                   degree-1, myTurtle)

def main():
   myTurtle = turtle.Turtle()
   myWin = turtle.Screen()
   myPoints = [[-100,-50],[0,100],[100,-50]]
   sierpinski(myPoints,3,myTurtle)
   myWin.exitonclick()

main()
```

代码1即是按前文所述思想写出来的。sierpinski做的第一件事是绘制最外部的三角形。接下来，进行了三次递归调用，每一次都是用来通过连接中点来获得新的三角形。这里又用了标准的海龟库。可以通过在Python提示符里输入help('turtle')来细究turtle库中可用的方法。

仔细研究代码，想想这些三角形的绘制顺序。确切地来说，绘制顺序取决于初始的设定，这里假设顺序是左下，上，右下。考虑到sierpinski函数调用其本身的方式，sierpinski先是沿最短路线直接到达可以绘制的最小的左下角的三角形，再返过来绘制其它的剩余部分：先是按类似的方式直接向最小的、最顶部的三角形前进；最后再绘制右下角的最小的三角形。

有时候根据函数调用的示意图来理解递归算法是很有用的。如图4所示，本例中的递归算法总是先向左边前进。黑线表示正在运行的，灰色的表示没有运行的，越靠近图4的底部，三角形越小。函数每次执行都消耗一个相似维度。到达最底部左边的三角形后，将继续绘制中间的三角形，并以此类推。



../_images/stCallTree.png

sierpinski函数很大程度上是基于getMid函数的。getMid函数接收两个参数作为端点，返回两者的中点。此外，可执行代码1中使用了begin_fill和end_fill方法来绘制带填充色的三角形。

4.9 复杂递归问题

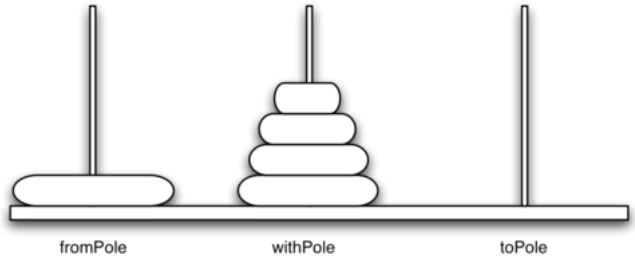
在前几节中，本书讨论了一些相对简单并且图形上很有趣的问题来帮助读者理解递归。在本节中，将研究一些使用迭代法难以解决然而却可以用递归法优雅而简洁地解决的问题。在最后，将讲解一个看起来似乎能用递归法解决但实际上并不能的问题。

4.10 河内（Hanoi）塔问题

河内塔问题由法国数学家Edouard于1883年受到印度教中的一个故事启发而提出，在故事中，年轻僧侣将被予以考题。在最开始，僧侣们会得到3根杆以及64个金圆盘，每个盘都比其下面的盘小一些。他们的任务是将这64个盘从一个盘移动到另外一个盘去，并且有两个限制条件：每次只允许移动一个盘；禁止将大盘置于小盘之上。僧侣们夜以继日地努力，每秒移动一个盘子。传说中，当他们完成此任务时，世界将会毁灭。

传说很有趣，但现在也不用担心世纪末日的问题。正确地完成该任务需要$2^{64}-1=18446744073709551615$次移动。1秒移动1次，也需要584942417355年。显然，实际所需时间比这更长。

图1演示了从第一根杆移动到第三根杆时，中间杆上的盘子的情况。注意到，按照规定，每根杆上的盘子都是从上到下依次变大的。如果你之前没有接触过该问题，那不妨现在来尝试一下。不必真的想象盘子和杆什么的，一堆书或者一堆纸即可。



image

那么用递归该如何解决这个问题？约束条件又是什么？现在来从头考虑下这个问题。假设有5个盘子，并且开始都放在杆1上。如果已经知道了如何将4个盘子移动到杆2上，便可以轻松地将底部的盘子移到杆3上，然后把那4个盘子从杆2移动到杆3上。但是不知道怎么移动4个盘子又该如何是好？那么考虑下，假如知道如何移动3个盘子的杆...以此类推。显然将1个盘子移动到杆3是很容易的。看起来似乎这就是约束条件了。

下面是对盘子从起始杆通过中间杆移动到目标杆的高度概括（设盘数为Height）：

1. 利用目标杆，将高度为 Height - 1 的塔从起始杆移动到中间杆上。
2. 将剩下的那个盘子放到目标杆上。
3. 利用中间杆，将高度为 Height - 1 的塔从中间杆移动到目标杆上。

维持大盘在下的规则便可以递归使用上面的3步，便可以处理任何更大规模的问题。上述步骤中唯一缺少的就是对约束条件的确定。最简的河内塔问题是仅有一个盘子的塔。在这种情况下，只需要将该盘子直接移动到目标杆即可。高度为1的河内塔即是约束条件。此外，上述步骤通过在步骤1和3中减少塔的高度来实现了向约束条件收敛。代码1给出了解决河内塔问题的Python代码。

代码1

```
def moveTower(height,fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height-1,fromPole,withPole,toPole)
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole)
```

该代码看起来似乎就像英文描述性文字一样。该算法的简便性关键在于进行了两次不同的递归调用。在行3，将除了最底部（最大的）盘子全部移到中间杆上，行4仅将底部的盘子移动到目标盘。行5则是将中间杆上的圆盘移动到最大盘的上部（目标杆）即可。约束条件即是塔高度为0。注意，处理约束条件在这种情况下是调用moveDisk函数的前提条件。

moveDisk函数，如代码2所示，也很简单。它仅仅是打印出将盘子从某个杆上移动到另一个杆的过程。读者可以试着运行一下，便可发现该算法可以很高效地解决该问题。

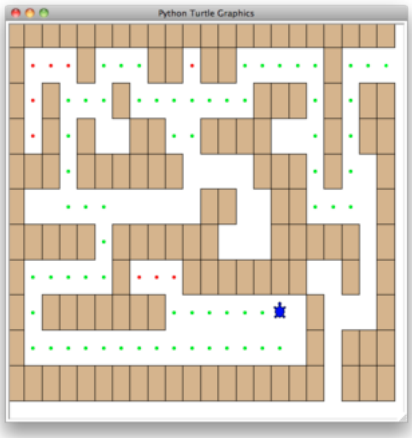
代码2

```
def moveDisk(fp,tp):
    print("moving disk from",fp,"to",tp)
```

看过moveTower和moveDisk的代码后，读者可能会想为何不直接使用栈来追踪任意盘子在哪个杆里。提示：如果要显式地追踪盘子，应该需要3个栈，每个对应一个杆。答案是，Python本身就隐式地提供了我们所需要的调用栈。

4.6 探索迷宫

本节将研究的问题跟拓宽机器人活动范围有关：如何走出迷宫？如果寝室内有一个Roomba真空打扫机器人，读者可能会想利用本节所学只是对其进行重新编程。本节的问题是帮助海龟走出虚拟迷宫。在该问题中，假设海龟掉入中央并开始虚招出炉。观察图2，思考如何解决该问题。



../_images/maze.png

为了简化问题，假设迷宫以矩形为单位划分。每个矩形要么是开放的要么是填充以墙。海龟只能通过开放的部分，如果遭遇了墙则必须调整方向。海龟需要系统性的步骤以走出该迷宫。下面是具体的步骤：

- 在初始位置尝试向北走一步，以此开始递归程序。
- 若第一步的向北移动失败，则向南移动，然后开始递归。
- 若第一步的向南移动也失败，则向西移动，然后开始递归。
- 若第一步的向北、向南、向西移动均失败，则向东移动。
- 若四个方向均不可行，则此路径无法走出迷宫，失败。

听起来很容易，然而有些细节需要说明一下。假设递归的第一步是向北移动的，按照制定的程序来说，下一步也将向北移动，此时如果不幸地被墙挡住，便会向南移动，于是便会到了起始状态。此时如果按照这种递归程序来做，只会向后退一步，然后陷入无限迭代中。因此，需要一种方法来记录走过的地方。在本例中，可以假设手里有一袋面包屑可以沿着走过的路径撒下。如果准备向某个方向前进一步时发现该方块已经有面包屑了，此时立刻退回来并且尝试另外一个方向。在该算法的代码中可以看出，往回走一步同递归函数调用结果返回一样。

跟其它递归算法一样，这里也需要研究约束条件。一部分读者可能根据前文已经推测出，在本算法中，有4个约束条件需要考虑：

1. 海龟碰到墙壁。由于方格被墙壁填充而无法通行。
2. 海龟到达已访问过的方格。为避免陷入无限迭代，不能在此位置继续前进。
3. 到达没有墙的边界之外，换言之已经走出迷宫。
4. 海龟在四个方向上都无法前进。

要让程序运行起来，需要一种方法来模拟迷宫。为了使之看起来更有趣，这里使用了海龟模块来绘制和探索迷宫，这样便可以观察该算法的动态效果了。迷宫对象应该要提供以下方法以便编写搜索算法。

- __init__ 读取表示迷宫的数据，初始化迷宫的内部表示，并且找到海龟的起始位置。
- drawMaze 在一个窗口中绘制迷宫。
- updatePosition 更新迷宫内部状态并更改海龟位置。
- isExit 判断当前位置是否为迷宫的一个出口。

Maze类也重载了操作符[]，这样算法程序便可以很容易地获取任意方格的状态。

下面看一下叫做searchFrom的搜索函数的代码，如代码3所示。注意到该函数接收3个参数：一个maze对象，起始行，起始列。从逻辑上来说，每次调用递归函数都将重新开始搜索。

代码3

```
def searchFrom(maze, startRow, startColumn):
    maze.updatePosition(startRow, startColumn)
  #  Check for base cases:
```

```
    #  1\. We have run into an obstacle, return false
     if maze[startRow][startColumn] == OBSTACLE :
         return False
     #  2\. We have found a square that has already been explored
     if maze[startRow][startColumn] == TRIED:
         return False
     # 3\. Success, an outside edge not occupied by an obstacle
     if maze.isExit(startRow,startColumn):
         maze.updatePosition(startRow, startColumn, PART_OF_PATH)
         return True
     maze.updatePosition(startRow, startColumn, TRIED)

     # Otherwise, use logical short circuiting to try each
     # direction in turn (if needed)
     found = searchFrom(maze, startRow-1, startColumn) or \
             searchFrom(maze, startRow+1, startColumn) or \
             searchFrom(maze, startRow, startColumn-1) or \
             searchFrom(maze, startRow, startColumn+1)
     if found:
         maze.updatePosition(startRow, startColumn, PART_OF_PATH)
     else:
         maze.updatePosition(startRow, startColumn, DEAD_END)
     return found
```

可以看出，在此算法中，第一步是调用了updatePosition函数。这仅仅是为了实现算法的可视化便于观察海龟是如何在迷宫中进行移动的。接下来该算法检测了4个约束条件的前3个：海龟是否碰到了墙壁？海龟是否回到了曾经到过的地方？海龟是否找到了出口？如果这些情况无一满足，则继续进行递归搜索。

读者也应当注意到了，在递归过程中使用了4次对searchFrom递归调用。很难预测到底进行了多少次递归调用，因为它们之间是用**or**连接的。如果第1个对searchFrom的调用返回结果为True，那么剩余的3次调用都不会执行。可以认为，移动至(row-1, column)是走出迷宫的一步。如果向北走并没有走出迷宫的方法，那么第2个调用将会执行，这个调用是移至南方。如果向南移动的也失败了，则向西，以此类推，最后是向东。如果4个递归函数返回的结果都是**False**，那么说明没走出去。读者应当下载或者手动输入以上程序，并逐一尝试改变调用顺序。

Maze类的如代码4、代码5及代码6所示。__init__方法接收1个文件名作为唯一的参数。该文件为文本文件，以"+"代表墙壁，" "代表开放性方块，"S"代表起始位置。图3是maze数据文件的一个例子。maze的内部表示时间上是由列表组成的列表（list of lists）。mazelist实例变量的每一行都是一个列表。二级列表的由上述的字符构成。对于图3所示的数据文件，在Python中的内部表示为：

```
[ ['+','+','+','+',...,'+','+','+','+','+','+','+','+'],
['+',' ',' ',' ',...,' ',' ',' ','+',' ',' ',' '],
['+',' ','+',' ',...,'+','+',' ','+',' ','+','+'],
['+',' ','+',' ',...,' ',' ',' ','+',' ','+','+'],
['+','+','+',' ',...,'+','+',' ','+',' ',' ','+'],
['+',' ',' ',' ',...,'+','+',' ',' ',' ',' ','+'],
['+','+','+','+',...,'+','+','+','+','+',' ','+'],
['+',' ',' ',' ',...,'+','+',' ',' ','+',' ','+'],
['+',' ','+','+',...,' ',' ','+',' ',' ',' ','+'],
['+',' ',' ',' ',...,' ',' ','+',' ','+','+','+'],
['+','+','+','+',...,'+','+','+',' ','+','+','+']]
```

drawMaze方法使用上述内部表示在屏幕上对迷宫进行初始绘制。

```
++++++++++++++++++++++
```

- ◦ ++ ++ +

- ◦ ▪ +++ + ++

- ◦ ▪ ++ ++++ + ++
  +++ ++++++ +++ + +

- ++ ++ + +

```
+++++ ++++++ +++++ +
```

- ```
  +   +++++++  + +
  ```
- +++++++ S + +
- ```
              + +++
  ```

++++++++++++++++++ +++

图3:Maze数据文件示例

updatePosition方法，如代码5所示使用了相同的内部表示来确定海龟是否撞墙。它也用来更新该内部表示，即使用"."或者"-"来分别表示已访问过的方块以及死胡同。此外，updatePosition方法使用了两个辅助方法，moveTurtle和dropBreadCrumb来在屏幕上更新视图。

最后，isExit方法使用海龟当前的位置来测试是否为出口。出口极为0行或者0列或者最右侧的列或者最底部的行。

代码4

```
class Maze:
    def __init__(self,mazeFileName):
        rowsInMaze = 0
        columnsInMaze = 0
        self.mazelist = []
        mazeFile = open(mazeFileName,'r')
        rowsInMaze = 0
        for line in mazeFile:
            rowList = []
            col = 0
            for ch in line[:-1]:
                rowList.append(ch)
                if ch == 'S':
                    self.startRow = rowsInMaze
                    self.startCol = col
                col = col + 1
            rowsInMaze = rowsInMaze + 1
            self.mazelist.append(rowList)
            columnsInMaze = len(rowList)

        self.rowsInMaze = rowsInMaze
        self.columnsInMaze = columnsInMaze
        self.xTranslate = -columnsInMaze/2
        self.yTranslate = rowsInMaze/2
        self.t = Turtle(shape='turtle')
        setup(width=600,height=600)
        setworldcoordinates(-(columnsInMaze-1)/2-.5,
                            -(rowsInMaze-1)/2-.5,
                            (columnsInMaze-1)/2+.5,
                            (rowsInMaze-1)/2+.5)
```

代码5

```
def drawMaze(self):
    for y in range(self.rowsInMaze):
        for x in range(self.columnsInMaze):
            if self.mazelist[y][x] == OBSTACLE:
                self.drawCenteredBox(x+self.xTranslate,
                                     -y+self.yTranslate,
                                     'tan')
    self.t.color('black','blue')

def drawCenteredBox(self,x,y,color):
    tracer(0)
    self.t.up()
```

```
    self.t.goto(x-.5,y-.5)
    self.t.color('black',color)
    self.t.setheading(90)
    self.t.down()
    self.t.begin_fill()
    for i in range(4):
        self.t.forward(1)
        self.t.right(90)
    self.t.end_fill()
    update()
    tracer(1)

def moveTurtle(self,x,y):
    self.t.up()
    self.t.setheading(self.t.towards(x+self.xTranslate,
                                     -y+self.yTranslate))
    self.t.goto(x+self.xTranslate,-y+self.yTranslate)

def dropBreadcrumb(self,color):
    self.t.dot(color)

def updatePosition(self,row,col,val=None):
    if val:
        self.mazelist[row][col] = val
    self.moveTurtle(col,row)

    if val == PART_OF_PATH:
        color = 'green'
    elif val == OBSTACLE:
        color = 'red'
    elif val == TRIED:
        color = 'black'
    elif val == DEAD_END:
        color = 'red'
    else:
        color = None

    if color:
        self.dropBreadcrumb(color)
```

代码6

```
def isExit(self,row,col):
     return (row == 0 or
             row == self.rowsInMaze-1 or
             col == 0 or
             col == self.columnsInMaze-1 )

def __getitem__(self,idx):
     return self.mazelist[idx]
```

完整代码如可执行代码1所示。该程序使用了如下所示的maze2.txt数据文件。注意到由于出口和海龟起始位置很近，因此本例要简单得多。

可执行代码1

```
import turtle

PART_OF_PATH = 'O'
TRIED = '.'
OBSTACLE = '+'
DEAD_END = '-'
```

```
class Maze:
    def __init__(self,mazeFileName):
        rowsInMaze = 0
        columnsInMaze = 0
        self.mazelist = []
        mazeFile = open(mazeFileName,'r')
        rowsInMaze = 0
        for line in mazeFile:
            rowList = []
            col = 0
            for ch in line[:-1]:
                rowList.append(ch)
                if ch == 'S':
                    self.startRow = rowsInMaze
                    self.startCol = col
                col = col + 1
            rowsInMaze = rowsInMaze + 1
            self.mazelist.append(rowList)
            columnsInMaze = len(rowList)

        self.rowsInMaze = rowsInMaze
        self.columnsInMaze = columnsInMaze
        self.xTranslate = -columnsInMaze/2
        self.yTranslate = rowsInMaze/2
        self.t = turtle.Turtle()
        self.t.shape('turtle')
        self.wn = turtle.Screen()
        self.wn.setworldcoordinates(-(columnsInMaze-1)/2-.5,-(rowsInMaze-1)/2-.5,(columnsInMaze-1)/2+.5,
(rowsInMaze-1)/2+.5)

    def drawMaze(self):
        self.t.speed(10)
        self.wn.tracer(0)
        for y in range(self.rowsInMaze):
            for x in range(self.columnsInMaze):
                if self.mazelist[y][x] == OBSTACLE:
                    self.drawCenteredBox(x+self.xTranslate,-y+self.yTranslate,'orange')
        self.t.color('black')
        self.t.fillcolor('blue')
        self.wn.update()
        self.wn.tracer(1)

    def drawCenteredBox(self,x,y,color):
        self.t.up()
        self.t.goto(x-.5,y-.5)
        self.t.color(color)
        self.t.fillcolor(color)
        self.t.setheading(90)
        self.t.down()
        self.t.begin_fill()
        for i in range(4):
            self.t.forward(1)
            self.t.right(90)
        self.t.end_fill()

    def moveTurtle(self,x,y):
        self.t.up()
        self.t.setheading(self.t.towards(x+self.xTranslate,-y+self.yTranslate))
        self.t.goto(x+self.xTranslate,-y+self.yTranslate)

    def dropBreadcrumb(self,color):
        self.t.dot(10,color)

    def updatePosition(self,row,col,val=None):
        if val:
```

```
            self.mazelist[row][col] = val
        self.moveTurtle(col,row)

        if val == PART_OF_PATH:
            color = 'green'
        elif val == OBSTACLE:
            color = 'red'
        elif val == TRIED:
            color = 'black'
        elif val == DEAD_END:
            color = 'red'
        else:
            color = None

        if color:
            self.dropBreadcrumb(color)

    def isExit(self,row,col):
        return (row == 0 or
                row == self.rowsInMaze-1 or
                col == 0 or
                col == self.columnsInMaze-1 )

    def __getitem__(self,idx):
        return self.mazelist[idx]


def searchFrom(maze, startRow, startColumn):
    # try each of four directions from this point until we find a way out.
    # base Case return values:
    #  1. We have run into an obstacle, return false
    maze.updatePosition(startRow, startColumn)
    if maze[startRow][startColumn] == OBSTACLE :
        return False
    #  2. We have found a square that has already been explored
    if maze[startRow][startColumn] == TRIED or maze[startRow][startColumn] == DEAD_END:
        return False
    # 3. We have found an outside edge not occupied by an obstacle
    if maze.isExit(startRow,startColumn):
        maze.updatePosition(startRow, startColumn, PART_OF_PATH)
        return True
    maze.updatePosition(startRow, startColumn, TRIED)
    # Otherwise, use logical short circuiting to try each direction
    # in turn (if needed)
    found = searchFrom(maze, startRow-1, startColumn) or \
            searchFrom(maze, startRow+1, startColumn) or \
            searchFrom(maze, startRow, startColumn-1) or \
            searchFrom(maze, startRow, startColumn+1)
    if found:
        maze.updatePosition(startRow, startColumn, PART_OF_PATH)
    else:
        maze.updatePosition(startRow, startColumn, DEAD_END)
    return found


myMaze = Maze('maze2.txt')
myMaze.drawMaze()
myMaze.updatePosition(myMaze.startRow,myMaze.startCol)

searchFrom(myMaze, myMaze.startRow, myMaze.startCol)
```

4.12 动态规划

计算机科学中的很多程序都是为了实现优化。比如说，求出两点之间的最短路径，给出一组数据点的最佳拟合曲线，求出满足一定条件的对象的最小集合。计算机科学中有很多策略用来解决这些问题。本书的目的之一便是向读者介绍一些不同的求解策略。**动态规划**是用于解决优化问题的策略之一。

优化问题的一个经典例子就是使用最少的硬币完成找零。假设你是自动贩卖机制造商的程序员，你司试图在每次交易中找零最少的硬币以简化工作流程。比如说，某个客户投入了1美元购买37美分的商品，那么找零用的硬币最少个数为多少？答案为6：2个25美分，1个10美分，3个1美分。如何得到这个答案？首先要尽量使用库存中面额最大的硬币（25美分），接着使用下一个可用的面值最大的硬币，同样尽量多地使用。第一种方法为**贪心法**，因为这里试图将此问题尽可能地分为比较大的块来解决。

使用美国硬币时，贪心法效果还不错，但是假设你司决定在某共部署其自动贩卖机，除了一般的1，5，10及25分之外，还有21分的。在这种情况下，贪心法便不能给出找零63分的最优解了。即使提供了21分，贪心法给出的解仍是6。显然，最优解是3个21分。

再来看一个可以保证能找到最优解的算法。因为本节跟递归有关，读者有可能猜到了会与递归有关。首先想一想约束条件。如果要找的零钱于某个硬币面额一致，那么答案便是只需要一个该硬币。

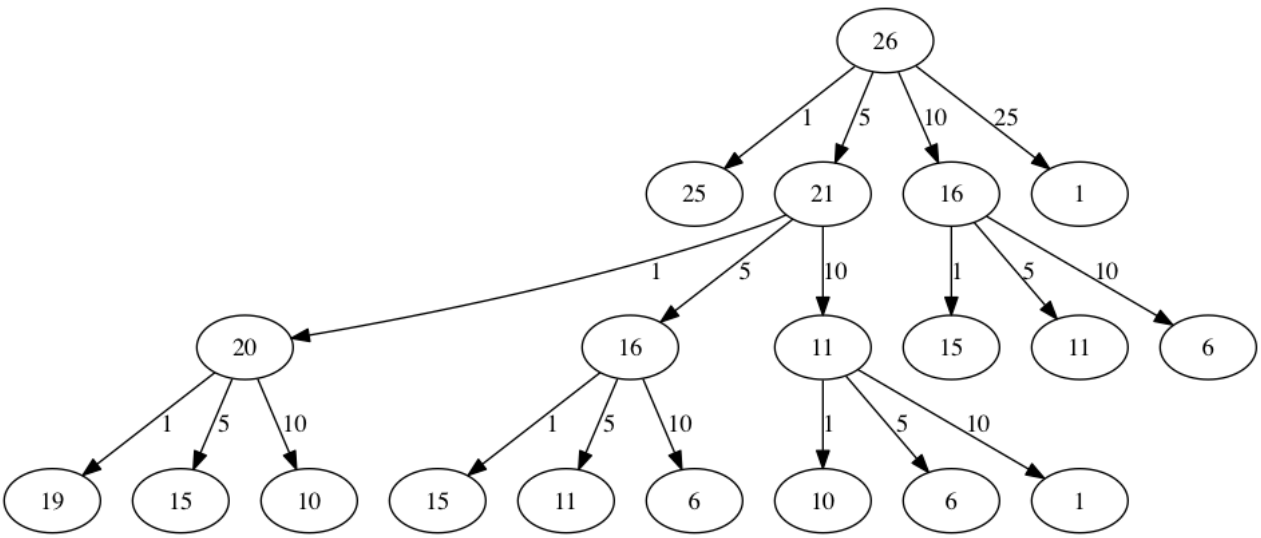
如果面额并不匹配，那么便有了几种选择。可以是原始所需找零减去1个美分后的值再求最小值，也可以是减去5美分后的值求最小值...等等。因此，原始所需找零的最小硬币数可以用下面的算法来计算：
$numCoins = min(1 + numCoins(originalamount - 1), 1 + numCoins(originalamount - 5)1 + numCoins(originalamount - 10), 1$
实现代码如代码7所示。在第三行中检测了约束条件，即试图用一个硬币来实现找零。如果没有找到恰好对应找零的硬币，则进行上述递归调用。行6使用列表表达式来过滤出那些比当前所需找零值面额更小的硬币。该递归调用也对所需找零值减去了选用的硬币面额。行7进行了递归调用。质疑到，在同一行对硬币总数进行了+1，因为使用了一个硬币。在递归调用满足约束条件时，也要加1。

代码7

```
def recMC(coinValueList,change):
   minCoins = change
   if change in coinValueList:
     return 1
   else:
      for i in [c for c in coinValueList if c <= change]:
         numCoins = 1 + recMC(coinValueList,change-i)
         if numCoins < minCoins:
            minCoins = numCoins
   return minCoins

print(recMC([1,5,10,25],63))
```

代码7的问题过于低效。实际上，它需要67，716，925次递归调用来找出4中硬币时找零63美分的最优解。为了理解该算法的致命缺陷，请看图5，它演示了找零26美分所需的377次函数调用的一小部分。

图像中的每一个节点对应一个recMC的调用。节点的标签表示当前正计算的找零数，箭头上的数字表示所使用的硬币。沿着图像走，可以找到实现图像中任意一点所需要的列表组合。最重要的问题是，进行了太多重复计算。比如说，该算法为15美分至少要重新计算3次。每次计算15美分的组合都需要52次函数调用。显然这里浪费了太多时间和资源来计算已有的结果。



image

减少工作量的关键在于储存一些已经计算过的结果。一个简单的解决方法是将硬币找零最小数存储在一个表哥中。当进行一次新的最小值计算之前，先在表格中检测一下是否是已知的结果。如果是表中已经有的，那么直接只用表中的结果而不是重新计算。可执行代码1给出了结合表格查询机制后的改进算法。

可执行代码1

```
def recDC(coinValueList,change,knownResults):
   minCoins = change
   if change in coinValueList:
      knownResults[change] = 1
      return 1
   elif knownResults[change] > 0:
      return knownResults[change]
   else:
       for i in [c for c in coinValueList if c <= change]:
         numCoins = 1 + recDC(coinValueList, change-i,
                              knownResults)
         if numCoins < minCoins:
            minCoins = numCoins
            knownResults[change] = minCoins
   return minCoins

print(recDC([1,5,10,25],63,[0]*64))
```

注意在行6检测了在表格中是否已经有计算好的结果。如果没有，便调用递归算法来计算该最小值并将其存储在表格中。使用这种改进的方法，可以将4种硬币，找零63美分的情况递归调用次数降低至221次。

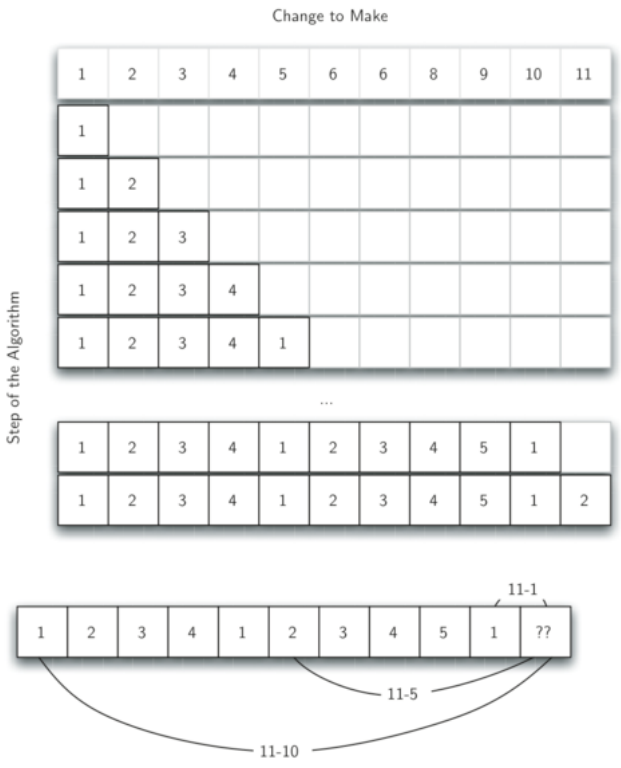
尽管可执行代码1是正确的，但它看起来有点奇怪，并且可以发现knownResults有很多地方还是空的。实际上，刚才的操作并不算是动态规划，仅仅是通过“缓存”技术提升了程序的性能。

真正的动态规划会使用更加系统的方法来解决该问题。动态规划解法应是从1美分开始，然后逐步系统性地推导出所需找零。这可以保证，在算法中的每一步都已经知道比当前找零数更小的所需硬币数。

考虑如何为找零11美分填制最小硬币数表，如图4所示。从1美分开始，唯一解是1个1美分的硬币。下一行给出了1美分和2美分的最小值。同样的，唯一解是2个1美分。第五行开始变得有趣起来了。现在有两种方法了，5个1美分或者1个5美分。如何选择最优的？通过查表发现为4美分进行找零所需硬币数是4，需要加1才能变为5美分，因此需要5个1美分。或者根据0美分加1个5美分的思路来得到5美分，那么答案是1。因为1和5的中取最小值为1，那么在表格中存储1即可。接着快速向表格末端前进，该考虑11美分了。如图5所示，有3个选项可供考虑。

1. 1个1美分加上 11 - 1 = 10 美分的最小值（1）；
2. 1个5美分加上 11 - 5 = 6 美分的最小值（2）；
3. 1个10美分加上 11 - 10 = 1 美分的最小值（1）；

1和3都会给出11美分找零所需的最小硬币数，即2。



代码8是解决找零问题的一种动态规划算法。dpMakeChange接受3个参数：可用硬币面额列表，找零值，每个找零值所需硬币数的列表。当函数运行完毕时，minCoins中会包含从0到change的所有找零所需要的硬币数。

代码8

```
def dpMakeChange(coinValueList,change,minCoins):
   for cents in range(change+1):
      coinCount = cents
      for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
               coinCount = minCoins[cents-j]+1
      minCoins[cents] = coinCount
   return minCoins[change]
```

注意，dpMakeChange并不是一个递归函数，虽然在一开始给出了本问题的递归解。这是非常重要的一点：虽然可以写出递归解法，但该解法并不一定就是最优或者最有效率的解。该函数的大部分工作内容是通过行4开始的迭代完成的。在该迭代中，考虑了由cents制定的找零数的所有可能的硬币面值。就像前例中11美分的例子一样，将最优解存储在列表minCoins中。

虽说以上程序在找出最小硬币数上完成得很好，然而它并不能真的用来对换硬币，因为它没有跟踪使用了的硬币。通过存储每次为minCoins表格添加数据时所使用的最后一个硬币，便可以实现dpMakeChange对所使用的硬币的跟踪。如果知道最后一个加入的硬币，便可以根据找零值减去该硬币的币值所得到值在表中进行查询，从而进行找零。可以一直沿着表格往回追溯直到起点。

可执行代码2给出了改进的dpMakechange算法，它实现了追踪所使用的硬币，此外还有一个函数printCoins回溯表格来打印出使用的硬币的值。该算法也可以用于某共的朋友。main的前两行设置了要转换的量并且创建了可用硬币的列表。接下来两行创建了用来存储结果的列表。coinsUsed即是找零所用的列表，coinCount是表中相应位置处找零所用的最少硬币数。

注意打印出来的硬币是直接来源于coinUsed数组的。第一次调用时，从数组的63位置开始，并打印出21。然后得到63-21=42，在列表中查询第42个元素。同样地，在那里找到的硬币是21。最后，数组的第21号元素也是21，那么最终结果就是3个21美分的硬币了。

```
def dpMakeChange(coinValueList,change,minCoins,coinsUsed):
   for cents in range(change+1):
      coinCount = cents
      newCoin = 1
      for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
               coinCount = minCoins[cents-j]+1
               newCoin = j
      minCoins[cents] = coinCount
      coinsUsed[cents] = newCoin
   return minCoins[change]

def printCoins(coinsUsed,change):
   coin = change
   while coin > 0:
      thisCoin = coinsUsed[coin]
      print(thisCoin)
      coin = coin - thisCoin

def main():
    amnt = 63
    clist = [1,5,10,21,25]
    coinsUsed = [0]*(amnt+1)
    coinCount = [0]*(amnt+1)

    print("Making change for",amnt,"requires")
    print(dpMakeChange(clist,amnt,coinCount,coinsUsed),"coins")
    print("They are:")
    printCoins(coinsUsed,amnt)
    print("The used list is as follows:")
    print(coinsUsed)

main()
```

可执行代码2:找零问题的解法

4.13 总结

本章研究了几种递归算法。这些算法用于向读者展示几种递归算法可以有效解决的问题。本章的重点为：

- 递归算法必须有约束条件。
- 递归算法必须改变当前状态并向约束条件靠近。
- 递归算法必须递归地调用其本身。
- 递归算法有时可以替代迭代算法。
- 递归算法通常与所求问题的公式表达自然地契合。
- 递归算法并不是万能的。有时递归算法的计算资源开销会比其它算法高。