递归

What is recursion
什么是递归

What is Recursion?

什么是递归Recursion？

Recursion is a solution to a problem, whose essence is

递归是一种解决问题的方法，其精髓在于

Break the problems down into smaller sizes of the same problem, consistently decomposed until the problem size is small enough to be solved in a very simple and straightforward way.

将问题分解为规模更小的相同问题，持续分解，直到问题规模小到可以用非常简单直接的方式来解决。

The decomposition method of recursive problem is very unique, and the obvious feature of the algorithm is: calling itself in the algorithm process.

递归的问题分解方式非常独特，其算法方面的明显特征就是：在算法流程中调用自身。

Recursion gives us an elegant solution to complex problems, and subtle recursive algorithms are often surprisingly simple and impressive.

递归为我们提供了一种对复杂问题的优雅解决方案，精妙的递归算法常会出奇简单，令人赞叹。

A first look at recursion: sequence summation
初识递归：数列求和

Question: Given a list, it returns the sum of all the numbers

问题：给定一个列表，返回所有数的和

The number of numbers in the list is variable, so a loop and an additive variable are required to sum iteratively

列表中数的个数不定，需要一个循环和一个累加变量来迭代求和

The program is very simple, but what if there is no circular sentence?

程序很简单，但假如没有循环语句？

Can you still sum up lists of uncertain lengths without using for or while?

既不能用for，也不能用while，还能对不确定长度的列表求和么？

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

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

A first look at Recursion: sequence summation
初识递归：数列求和

We realize that summation is actually ultimately achieved by addition again and again, and that addition has just 2 operands, which is certain.

我们认识到求和实际上最终是由一次次的加法实现的，而加法恰有2个操作数，这个是确定的。

How to break down the large scale problem of summing up the list into a smaller and fixed problem of summing 2-number (addition)?

看看怎么想办法，将问题规模较大的列表求和，分解为规模较小而且固定的2个数求和(加法)？

The same is a sum-up problem, but the scale has changed, in line with the characteristics of the recursive solution problem!

同样是求和问题，但规模发生了变化，符合递归解决问题的特征！

A first look at Recrecursion: sequence summation

初识递归：数列求和

Another ation: full bracket expression (1 + (3 + (5 + (7 + 9))))

换个方式来表达数列求和：全括号表达式(1+(3+(5+(7+9))))

The formula above, the innermost bracket (7 + 9), this can be calculated without circulation, in fact, the whole process of summing is like this:

上面这个式子，最内层的括号(7+9)，这是无需循环即可计算的，实际上整个求和的过程是这样：

$$
\begin{aligned}
total &= (1 + (3 + (5 + (7 + 9)))) \\
total &= (1 + (3 + (5 + 16))) \\
total &= (1 + (3 + 21)) \\
total &= (1 + 24) \\
total &= 25
\end{aligned}
$$

A First look at Recursion: sequence summation

初识递归：数列求和

Observe the repetition patterns included in the process above , and the sum-up problem can be summarized as follows:

观察上述过程中所包含的重复模式，可以把求和问题归纳成这样：

The sum of the sequence = the sum of the first number + the sum of remaining sequence

数列的和="首个数" + "余下数列" 的和

If the number contains as few as one, the sum is the number

如果数列包含的数少到只有1个的话，它的和就是这个数了

This is small enough to do the simplest process

这是规模小到可以做最简单的处理

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

question
问题

decompose
分解

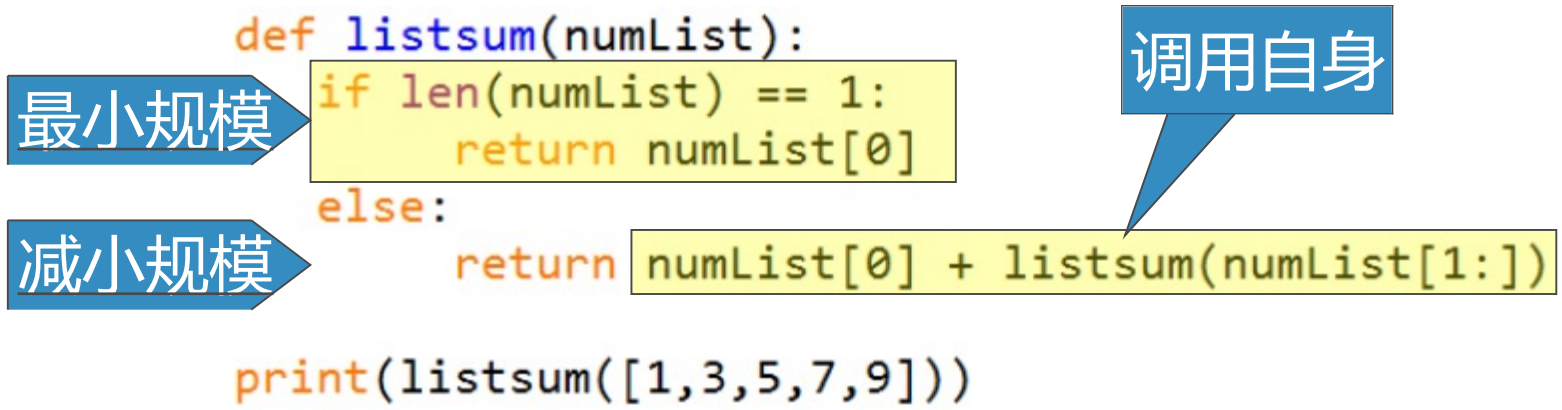
Same — problem, much smaller in scale
相同问题，规模更小

A First look at Recrecursion: sequence summation

初识递归：数列求和

Make the recursive algorithm above into a program

上面的递归算法变成程序



Key points of the procedure above:

上面程序的要点：

1. The problem breaks down into the same problem on a smaller scale and appears as "calling itself"

1，问题分解为更小规模的相同问题，并表现为“调用自身”

2. Solution to the "minimal scale" problem: simple and straightforward

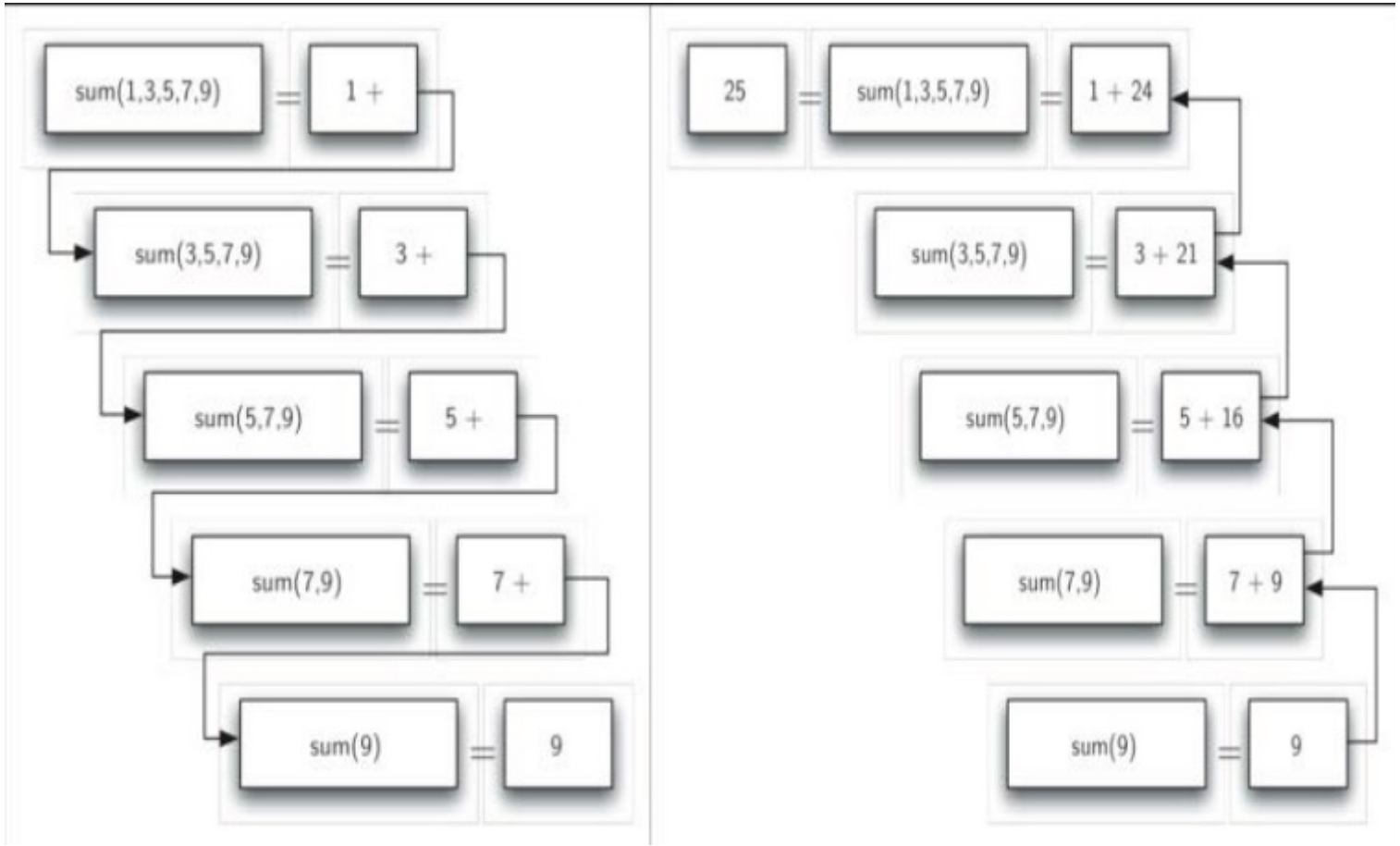
2，对“最小规模”问题的解决：简单直接

How is the recursive program executed?

递归程序如何被执行？

The chain of recursive function calls and return procedures

递归函数调用和返回过程的链条



"three laws" of recursion

递归“三定律”

To tribute to Asimov's three laws of robotics, recursive algorithms also summed up the three laws:

为了向阿西莫夫的“机器人三定律”致敬，递归算法也总结出“三定律”：

1, The recursive algorithm must have a basic end condition (direct solution of the minimum size problem)

1，递归算法必须有一个基本结束条件(最小规模问题的直接解决)

2, the recursive algorithm must be able to change the state to the basic end condition (reduce the size of the problem)

2，递归算法必须能改变状态向基本结束条件演进(减小问题规模)

3, and the recursive algorithm must call itself (solve the same problem with the reduced size)

3，递归算法必须调用自身(解决减小了规模的相同问题)

"three laws" of recursion: sequence summation
递归“三定律”：数列求和问题

The sequence summation problem first has the basic end condition: when the list length is 1, directly output the unique number contained

数列求和问题首先具备了基本结束条件：当列表长度为1的时候，直接输出所包含的唯一数

The summed data object is a list, and the basic end condition is a list of length 1, so the recursive algorithm changes the list and evolves into a state of length 1

数列求和处理的数据对象是一个列表，而基本结束条件是长度为1的列表，那递归算法就要改变列表并向长度为1的状态演进

We see its specific approach is to reduce the list length by 1.

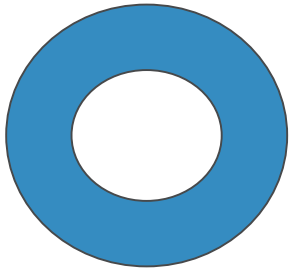
我们看到其具体做法是将列表长度减少1。

Calling itself is the most difficult part to understand in the recursive algorithm, which we actually understand as "the same problem breaks down into smaller size."

调用自身是递归算法中最难理解的部分，实际上我们理解为“问题分解成了规模更小的相同问题”就可以了

In the sequence summation algorithm is the "the summing problem of shorter sequence"

在数列求和算法中就是“更短数列的求和问题”



Recursive application: arbitrary adic conversion

递归的应用：任意进制转换

Integers are converted to an arbitrary decimal system

整数转换为任意进制

This algorithm, as discussed in the data structure *stack*, is back!

这个在数据结构栈里讨论过的算法，又回来了！

Recursion and stack, there must be something related

递归和栈，一定有关联

If you were overwhelmed by the "in" and "out" last time, this recursive algorithm will make you fresh

如果上次你被“入栈”“出栈”搞得挺晕乎的话，这次递归算法一定会让你感到清新

And this time we have to solve anything from binary to hexadecimal conversion of number systems

而且这次我们要解决从二进制到十六进制的任意进制转换

Integers are converted to an arbitrary decimal system

整数转换为任意进制

We analyze this problem under the most familiar decimal system

我们用最熟悉的十进制分析下这个问题

Decimal has ten different symbols: convString = "0123456789" is an integer smaller than ten, converted into a decimal, directly check the table can be: convString [n]

十进制有十个不同符号：convString="0123456789"比十小的整数，转换成十进制，直接查表就可以了：convString[n]

Find a way to split integers which are more than ten into a series of integers which is smaller ten, one by one. Such as 769, dismantled into seven, six, nine, check the table to get 769

想办法把比十大的整数，拆成一系列比十小的整数，逐个查表比如七百六十九，拆成七、六、九，查表得到769就可以了

Integers are converted to an arbitrary decimal system

整数转换为任意进制

So, in three laws of recursion, we find the "basic end condition", which is an integer less than ten

所以，在递归三定律里，我们找到了“基本结束条件”，就是小于十的整数

- The process of disassembling integers is the evolution to the "basic end condition"
  - 拆解整数的过程就是向“基本结束条件”演进的过程

We use the integer division and the remainder to separate the integer step by step

我们用整数除，和求余数两个计算来将整数一步步拆开

- Divide by the base base (/ / base)
  - 除以“进制基base”(//base)
- Surder (%base)
  - 对“进制基”求余数(%base)

Integers are converted to an arbitrary decimal system

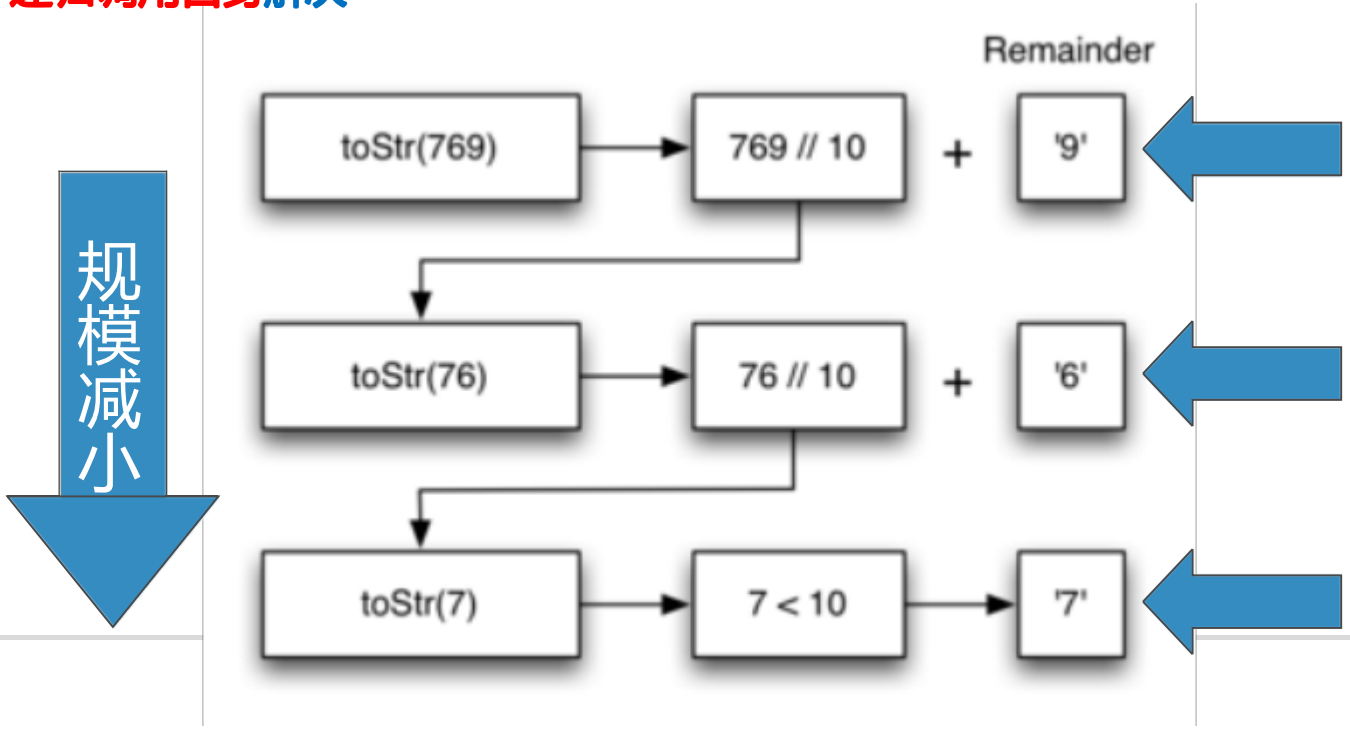
整数转换为任意进制

The problem breaks down into:

问题就分解为：

The remainder is always less than the "decimal base", which is the "basic end condition". It can directly check the table and convert the integer quotient to a "smaller scale" problem, which is solved by recursive calling itself

余数总小于“进制基base”，是“基本结束条件”，可直接进行查表转换整数商成为“更小规模”问题，通过递归调用自身解决



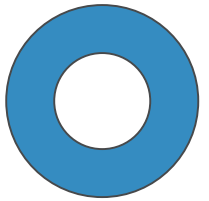
Integer conversion to arbitrary precession: code
整数转换为任意进制：代码

Here is the code for the recursive algorithm
下面就是递归算法的代码

```
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))
```

Minimum scale
最小规模

Reduce the scale and call itself
减小规模，调用自身



Implementation of the recursive calls
递归调用的实现

Implementation of the recursive calls

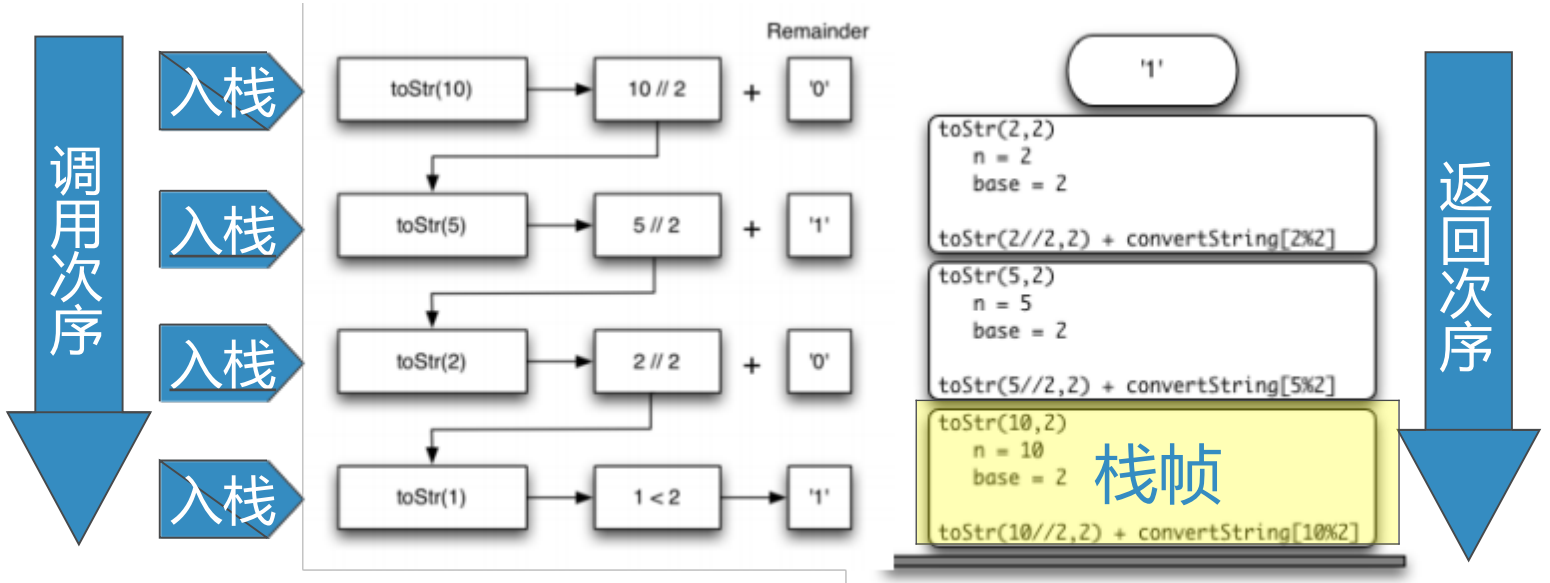
递归调用的实现

When a function is called, the system pushes the field data of that call into the system calling stack

当一个函数被调用的时候，系统会把调用时的现场数据压入到系统调用栈

Each call, the field data pressed into the stack is called the stack frame. When the function returns, you need to get the return address from the top of the call stack, restore the scene, pop up the stack frame, and return by the address.

每次调用，压入栈的现场数据称为栈帧，当函数返回时，要从调用栈的栈顶取得返回地址，恢复现场，弹出栈帧，按地址返回。



Recursive depth limit in the Python

Python中的递归深度限制

When debugging recursive algorithm programs, you often encounter such an error: RecursionError

在调试递归算法程序的时候经常会碰到这样的错误：RecursionError

There are too many recursive layers, as the capacity of system calling stack is limited

递归的层数太多，系统调用栈容量有限

```
>>>
 RESTART: /Users/chenbin/Documents/教学项
) /素材/tell_story.py
给你讲个故事:
“从前有座山，山上有座庙，庙里有个老和尚，他在讲:
“从前有座山，山上有座庙，庙里有个老和尚，他在讲:
“从前有座山，山上有座庙，庙里有个老和尚，他在讲:
/素材/tell_story.py", line 2, in tell_story
    print("“从前有座山，山上有座庙，庙里有个老和尚，他在讲: ")
RecursionError: maximum recursion depth exceeded while
```

Recursive depth limit in the Python

Python中的递归深度限制

At this time to check whether the program forgot to set the basic end conditions, resulting in infinite recursion

这时候要检查程序中是否忘记设置基本结束条件，导致无限递归

Or the base end condition evolves too slowly, causing too many recursive layers and the call stack to overflow

或者向基本结束条件演进太慢，导致递归层数太多，调用栈溢出

```
1  def tell_story():
2      print("“从前有座山，山上有座庙，庙里有个老和尚，他在讲：")
3      tell_story()
4
5  print("给你讲个故事：")
6  tell_story()
```

Recursive depth limit in the Python

Python中的递归深度限制

The sys module built in Python can acquire and adjust the maximum recursive depth

在Python内置的sys模块可以获取和调整最大递归深度

```
>>> import sys
>>> sys.getrecursionlimit()
1000
>>> sys.setrecursionlimit(3000)
>>> sys.getrecursionlimit()
3000
```

The story of recursion
递归的故事

Predestination .2014

前目的地.Predestination.2014

It produces its own closed-loop brain-burning recursion

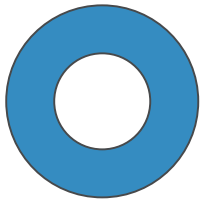
自身产生自身的闭环烧脑递归

Triangle.2009

恐怖游轮.Triangle.2009

The big mix of calling stack and stack frame, how to end everything and return the main function?

调用栈栈帧大混合，如何才能终结一切，返回主函数？



Recursive visualization: Fractal tree

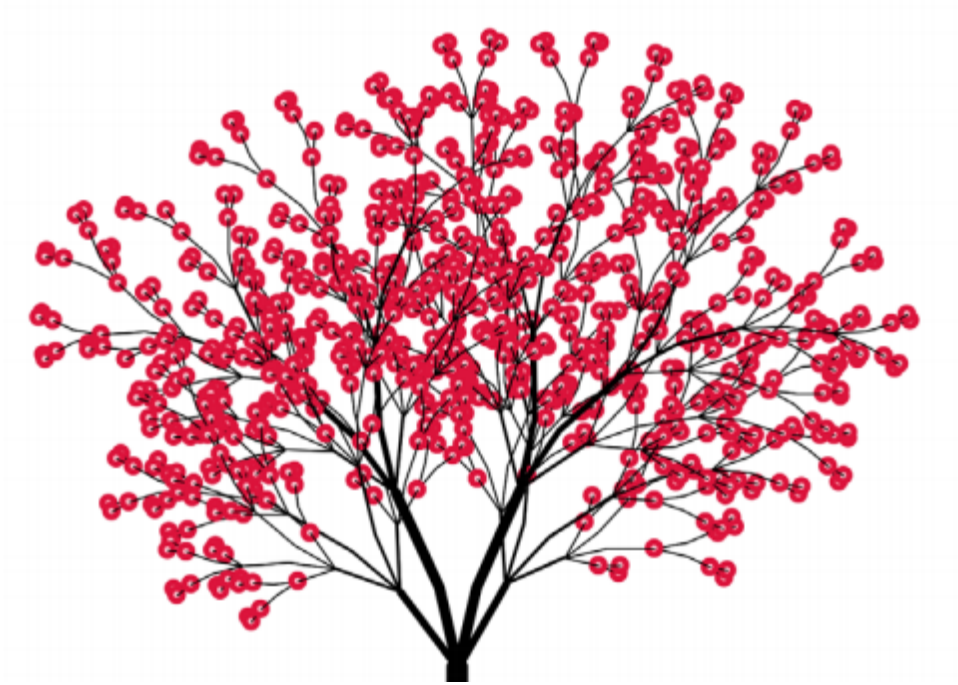
递归可视化：分形树

Recursive visualization: Figure

递归可视化 : 图示

The previous recursive algorithms show a simple and powerful side, but still not an intuitive concept. Let's use the recursive mapping to show the visual image of the recursive callings

前面的种种递归算法展现了其简单而强大 的一面 ，但还是难有个直观的概念，下面我们通过递归作图来展现递归调用的视觉影像



Recursive visualization: Figure

递归可视化：图示

The Python's sea turtle mapping system, turtle module

Python的海龟作图系统turtle module

The built-in, readily available, and is based on the creative LOGO language, whose imagery mimics the footprints of a sea turtle crawling on the beach

Python内置，随时可用，以LOGO语言的创意为基础，其意象为模拟海龟在沙滩上爬行而留下的足迹

Crawl: forward (n); backward (n)

爬行： `forward(n);backward(n)`

Steering: left (a); right (a)

转向： `left(a); right(a)`

Pen: penup(); pendow n()

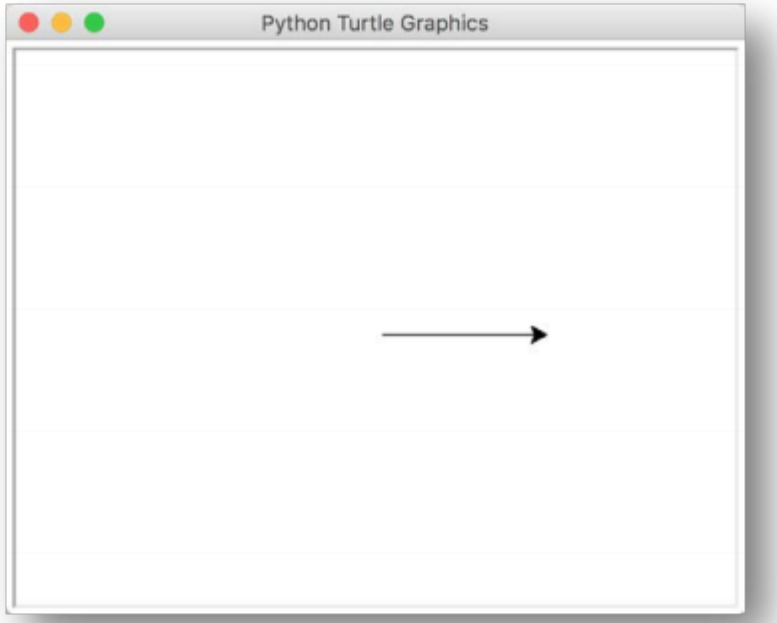
抬笔放笔： `penup(); pendown()`

Pen Properties: pensize(s); pencolor (c)

笔属性： `pensize(s); pencolor(c)`

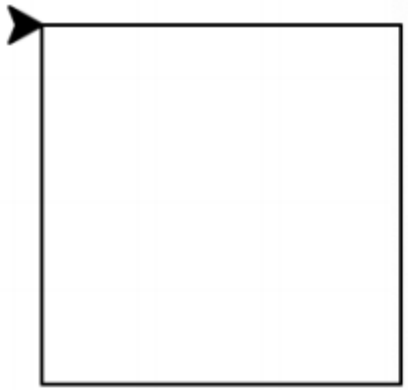
Recursive visualization: Figure
递归可视化 : 图示

```
import turtle
t= turtle.Turtle()

# 作图开始

t.forward(100) #指挥海龟作图

# 作图结束

turtle.done()
```



Turtles

海龟作图



t1.py

```
1  import turtle
2
3  t = turtle.Turtle()
4
5  for i in range(4):
6      t.forward(100)
7      t.right(90)
8
9  turtle.done()
10
```

Turtles

海龟作图

t1.py t2.py

```
1  import turtle
2
3  t = turtle.Turtle()
4
5  t.pencolor('red')
6  t.pensize(3)
7  for i in range(5):
8      t.forward(100)
9      t.right(144)
10 t.hideturtle()
11
12 turtle.done()
13
```

An example of a recursive mapping: helix

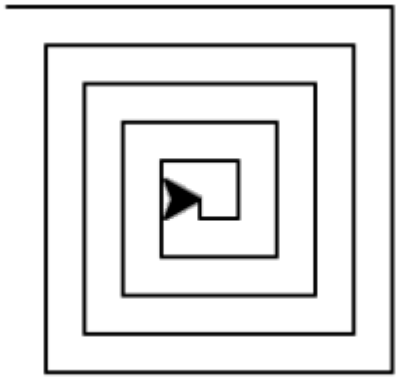
一个递归作图的例子：螺旋

```
1  import turtle
2
3  t = turtle.Turtle()
4
5  def drawSpiral(t, lineLen):
6      if lineLen > 0:
7          t.forward(lineLen)
8          t.right(90)
9          drawSpiral(t, lineLen - 5)
10
11 drawSpiral(t, 100)
12
13 turtle.done()
```

最小规模，0直接退出

减小规模，边长减5

调用自身



Fractal tree: Self-similar recursive graph

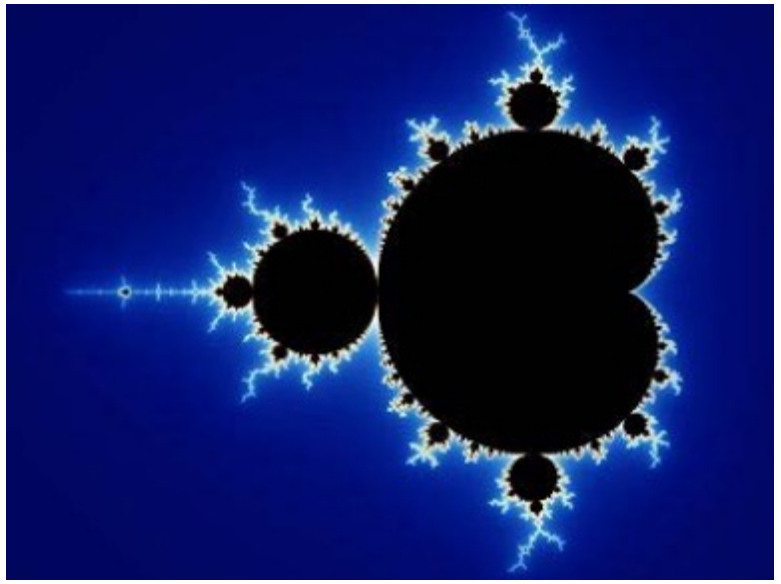
分形树：自相似递归图形

Fractal Fractal, is a new discipline initiated by Mandelbrot in 1975

分形Fractal ，是1975年由Mandelbrot 开创的新学科

"A rough or fragmentary geometry, which can be divided into several parts, and each part is (at least approximately) the overall reduced shape", meaning it has self-similar attribute.

“一个粗糙或零碎的几何形状，可以分成数个部分，且每一部分都(至少近似地)是整体缩小后的形状”，即具有自相似的性质。



Fractal tree: Self-similar recursive graph

分形树：自相似递归图形

Many objects with fractal attribute can be found in nature

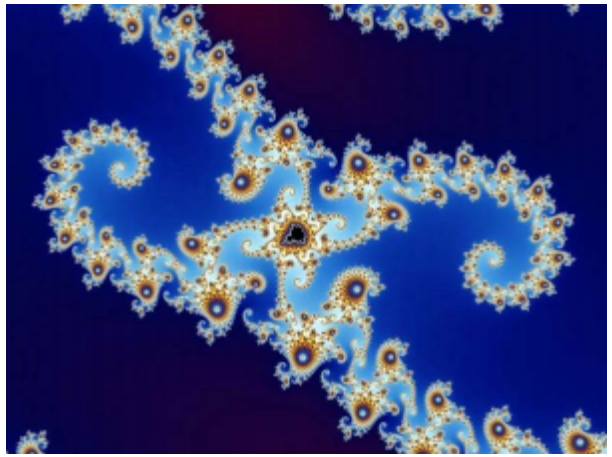
自然界中能找到众多具有分形性质的物体

Shorelines, mountains, lightning, clouds, snowflakes, trees

海岸线、山脉、闪电、云朵、雪花、树

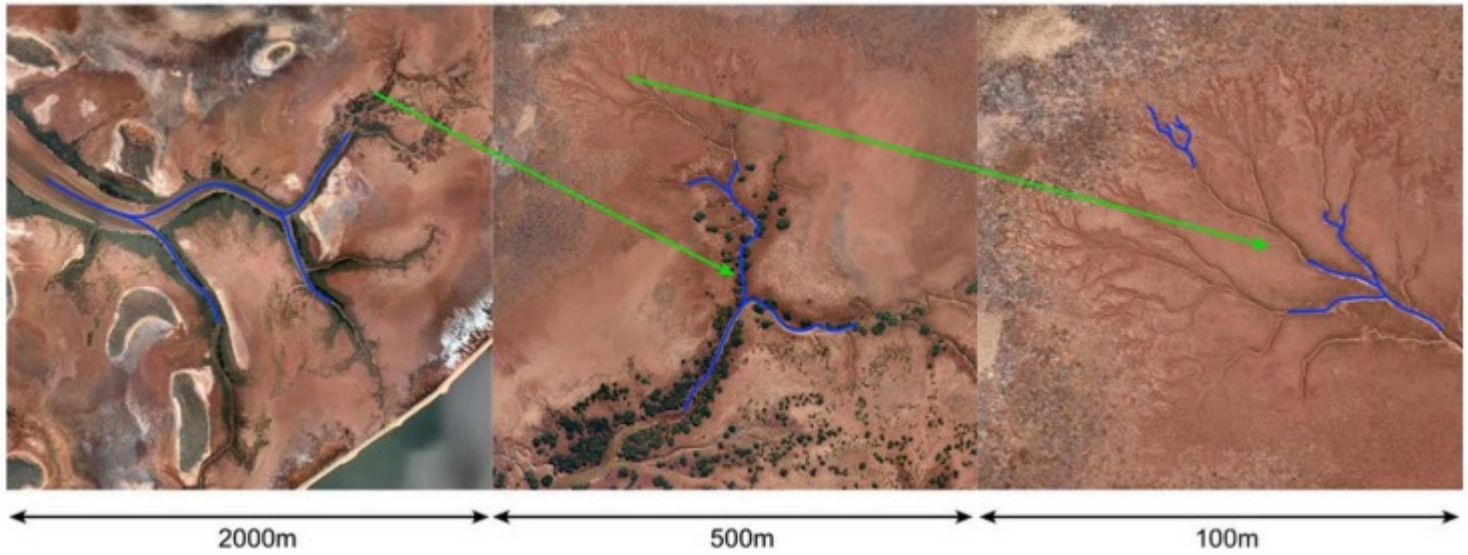
http://paulbourke.net/fractals/googlee arth/

http://recursivedrawing.com/



Nature is not smooth

自然界不是平滑的



Fractal tree: Self-similar recursive graph

分形树：自相似递归图形

The fractal characteristics in natural phenomena make the computer can generate extremely vivid scenes of natural through the fractal algorithm. Fracals are similar on different scales

自然现象中所具备的分形特性，使得计算机可以通过分形算法生成非常逼真的自然场景，分形是在不同尺度上都具有相似性的事物

We can see that every fork and every branch of a tree actually has the shape characteristics of the whole tree (but also gradually bifurcated)

我们能看出一棵树的每个分叉和每条树枝，实际上都具有整棵树的外形特征(也是逐步分叉的)

Fractal tree: Self-similar recursive graph

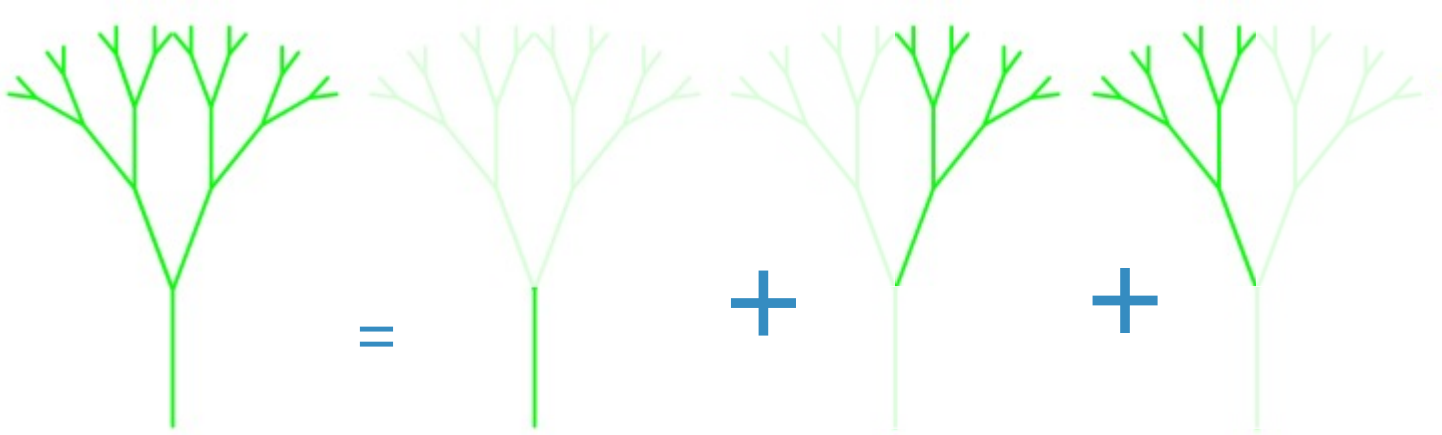
分形树：自相似递归图形

In this way, we can break down the tree into three parts: the trunk, the tree on the left, the tree on the right

这样 ，我们可以把树分解为三个部分：树 干、左边的小树、右边的小树

After decomposition, it fits the definition of recursion: a call to itself

分解后，正好符合递归的定义： 对自身的调用



二叉树树干 倾斜的右小树倾斜的左小树

Fractal tree: Code

分形树 : 代码

```
import turtle


def tree(branch_len):
    if branch_len > 5:  # 树干太短不画，即递归结束条件
        t.forward(branch_len)  # 画树干
        t.right(20)  # 右倾斜20度
        tree(branch_len - 15)  # 递归调用，画右边的小树，树干减15
        t.left(40)  # 向左回40度，即左倾斜20度
        tree(branch_len - 15)  # 递归调用，画左边的小树，树干减15
        t.right(20)  # 向右回20度，即回正
        t.backward(branch_len)  # 海龟退回原位置
```

```
t = turtle.Turtle()
t.left(90)
t.penup()
t.backward(100)
t.pendown()
t.pencolor('green')
t.pensize(2)
tree(75)  # 画树干长度75的二叉树
t.hideturtle()
turtle.done()
```

Recursive visualization: Fractal tree

递归可视化：谢尔宾斯基三角形

Shelbinsky Sierpinski Triangle

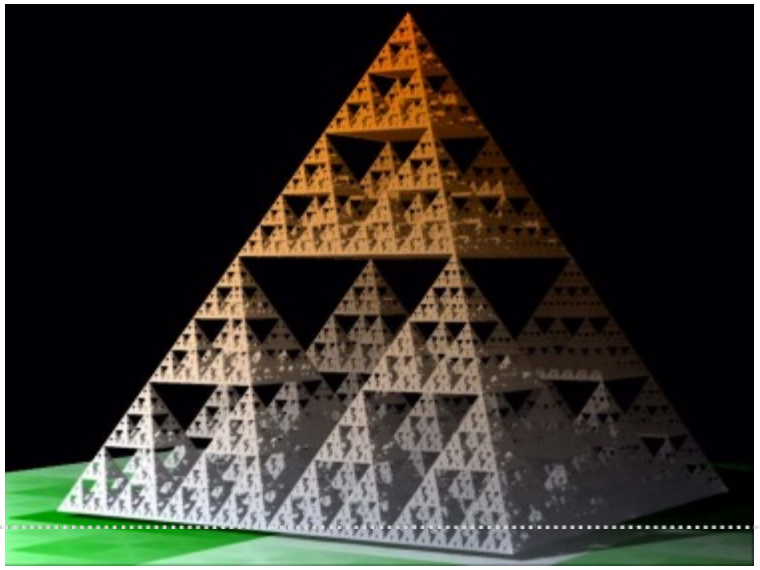
谢尔宾斯基Sierpinski三角形

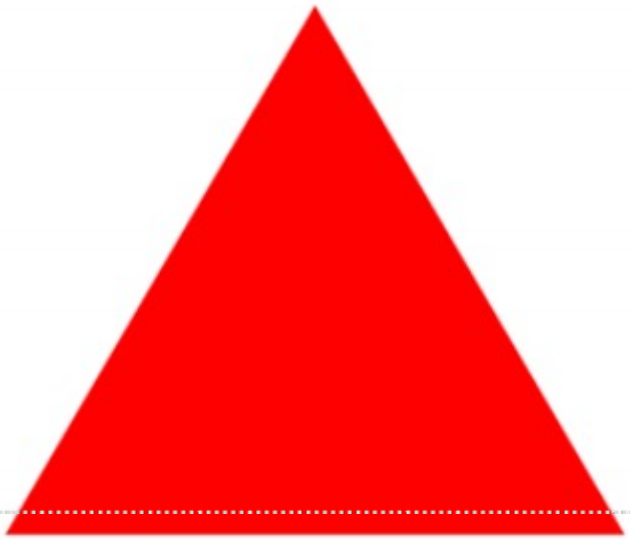
Fractal construction, the plane called Sherbsky triangle, stereo called Sherbinsky pyramid

分形构造，平面称谢尔宾斯基三角形，立体称谢尔宾斯基金字塔

In fact, the true Sherbinsky triangle is completely invisible, with an area of 0, but an infinite circumference, and is a fractional dimension (about 1.585 dimensions) constructed between one and two dimensions.

实际上，真正的谢尔宾斯基三角形是完全不可见的，其面积为0，但周长无穷，是介于一维和二维之间的分数维(约1.585维) 构造。





Shelbinsky Triangle: Drawing ideas
谢尔宾斯基三角形：作图思路

According to the self-similarity characteristics, the Sherbinsky triangle is made up of three size-halved Sherbinsky triangles folded according to the shape of “品”

根据自相似特性，谢尔宾斯基三角形是由3个尺寸减半的谢尔宾斯基三角形按照品字形拼叠而成

Because we cannot really make a Sherbinsky triangle (degree->), we can only make limited approximations of degree.

由于我们无法真正做出谢尔宾斯基三角形(degree->∞)，只能做degree有限的近似图形。

0 1 23

Shelbinsky Triangle: Drawing ideas

谢尔宾斯基三角形：作图思路

In the case of a finite degree, the triangle of degree =n, is made up of three degree =n-1 triangles stacked according to th shape of “品”

在degree有限的情况下 ，degree=n的三角形 ，是由3个degree=n- 1的三角形按照品字形拼叠而成

At the same time, the three degree =n-1 triangle edges are all half the length of the degree =n triangle (reduced in size).When degree =0, it is an equilateral triangle, which is the recursive elementary end condition

同时，这3个degree=n-1的三角形边长均为degree=n的三角形的一半(规模减小)。当degree=0，则就是一个等边三角形，这是递归基本结束条件

0 1 23

Sherbinsky Triangles: Code
谢谢尔宾斯基三角形：代码

equilateral triangle
等边三角形

Minimum size, 0 direct exit
最小规模，0直接退出

Reduce the scale: the getM id side length is halved
减小规模： getMid边长减半

Call itself, left and right order
调用自身，左上右次序

```
1  import turtle
2
3  def sierpinski(degree, points):
4      colormap = ['blue', 'red', 'green', 'white', 'yellow', 'orange']
5      drawTriangle(points, colormap[degree])
6      if degree > 0:
7          sierpinski(degree - 1,
8                     {'left':points['left'],
9                     'top':getMid(points['left'], points['top']),
10                    'right':getMid(points['left'], points['right'])})
11     sierpinski(degree - 1,
12                    {'left':getMid(points['left'], points['top']),
13                    'top':points['top'],
14                    'right':getMid(points['top'], points['right'])})
15     sierpinski(degree - 1,
16                    {'left':getMid(points['left'], points['right']),
17                    'top':getMid(points['top'], points['right']),
18                    'right':points['right']})
```

Sherbinsky Triangles: Code
谢谢尔宾斯基三角形：代码

```
20  def drawTriangle(points, color):
21      t.fillcolor(color)
22      t.penup()
23      t.goto(points['top'])
24      t.pendown()
25      t.begin_fill()
26      t.goto(points['left'])
27      t.goto(points['right'])
28      t.goto(points['top'])
29      t.end_fill()
30
31  def getMid(p1, p2):
32      return ( (p1[0] + p2[0]) / 2, (p1[1] + p2[1]) /2 )
33
34  t = turtle.Turtle()
35
36  points = {'left':(-200, -100),
37            'top':(0, 200),
38            'right':(200, -100)}
39  sierpinski(5, points)
40
41  turtle.done()
```

Draw the equilateral triangles绘制等边三角形

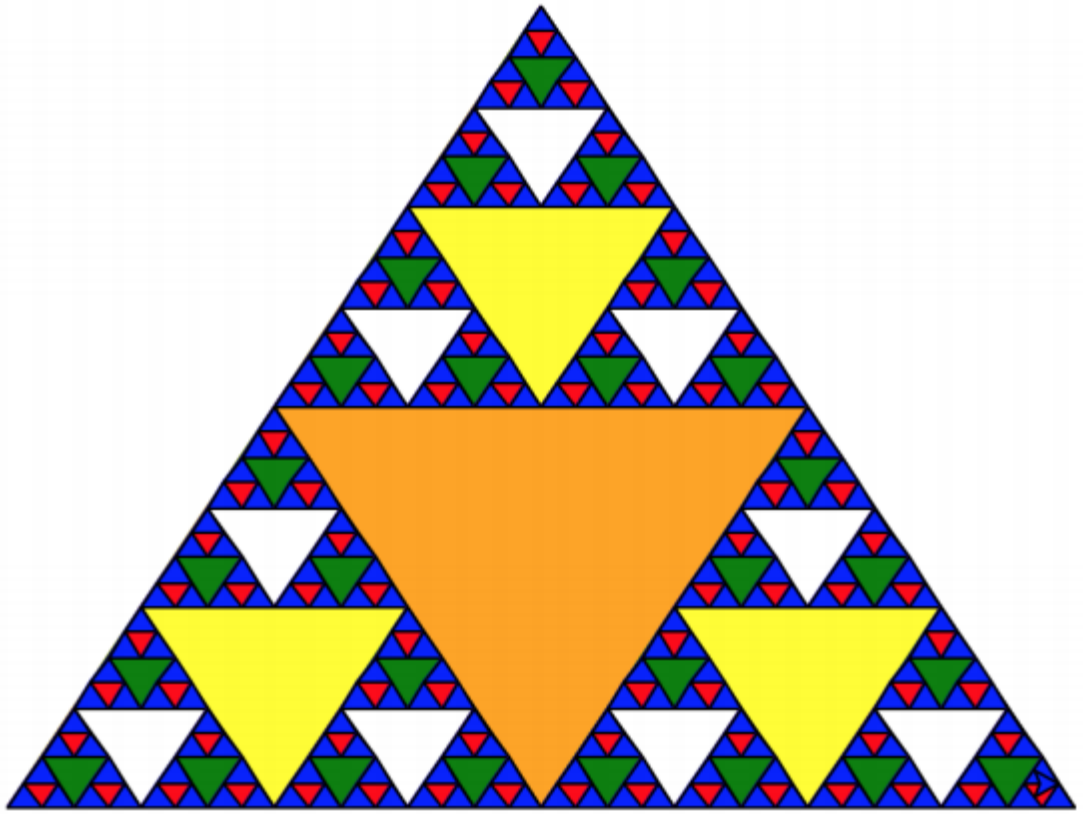
Take the midpoint of the two points取两个点的中点

Outside the outline of the three vertices绘制等边三角形

Draw a triangle of degree =5画degree=5的三角形

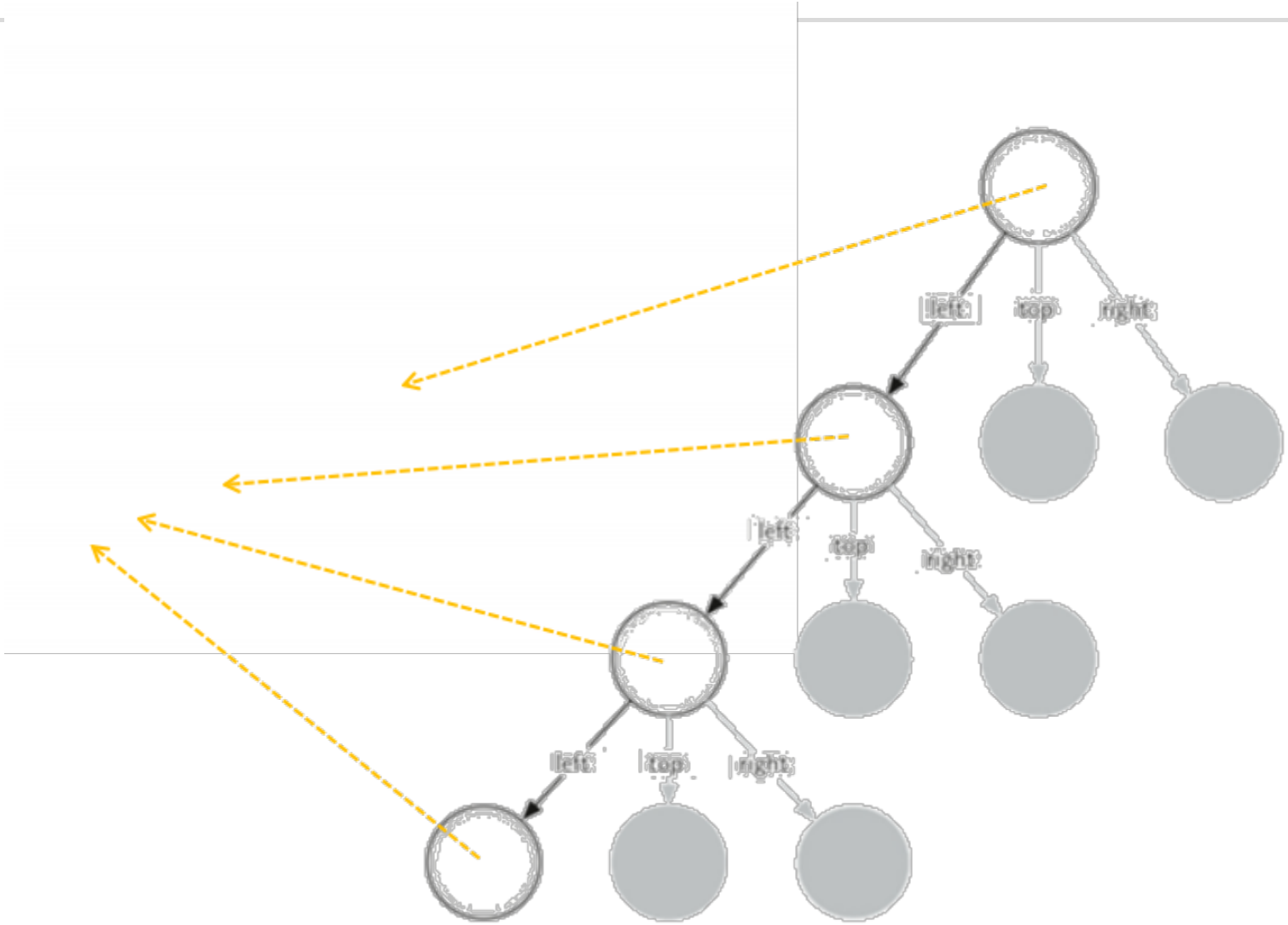
Triangles of a degree =5

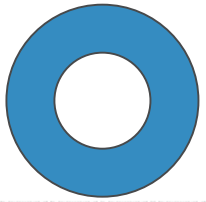
degree=5的三角形



The drawing procedure of degree =3

degree=3的绘制过程





Recursive application: Hanota

递归的应用：汉诺塔

Complex recursion problem: Hanota
汉诺塔问题

The Hanota problem was proposed by the French mathematician Edouard Lucas in 1883.

汉诺塔问题是法国数学家EdouardLucas于1883年，根据传说提出来的。

Legend has it that in a Hindu temple, there are three pillars, one of which contains 64 gold plates of different size. The monks are asked to move the stack of gold plates from one pillar to another, but there are two rules:

传说在一个印度教寺庙里，有3根柱子，其中一根套着64个由小到大的黄金盘片，僧侣们的任务就是要把这一叠黄金盘从一根柱子搬另一根，但有两个规则：

①Move only 1 plate at a time

一次只能搬1个盘子

②Large plates should not be folded on small plates

大盘子不能叠在小盘子上

God's will says that once these plates are finished moving

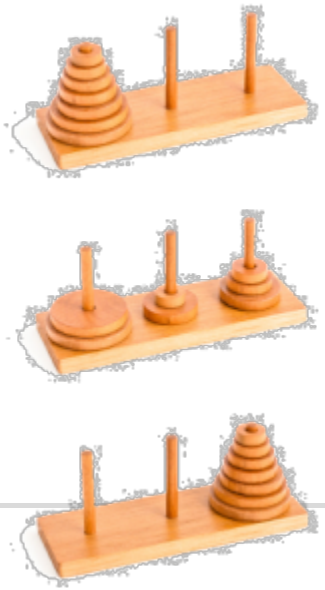
神的旨意说一旦这些盘子完成迁移

The temple will collapse and the world will destroy……

寺庙将会坍塌，世界将会毁灭……

And God's will is absolutely true!

神的旨意是千真万确的！



Hanoi tower problem
汉诺塔问题

Although these gold discs have mysterious links to the end of the world, but we don't have to worry too much, as calculated, to move these 64 discs:

虽然这些黄金盘片跟世界末日有着神秘的联系，但我们却不必太担心，据计算，要搬完这64个盘片：

The number of moves required is $2^{64}-1=18,446,744,073,709,551,615$ views

需要的移动次数为$2^{64}-1=18,446,744,073,709,551,615$次

If moved once per second, it takes 584,942,417,355 (500 billion) years!

如果每秒钟搬动一次，则需要584,942,417,355(五千亿)年！

We still analyze the Tower of Hanoi problem from the three-law of recursion

我们还是从递归三定律来分析汉诺塔问题

Basic end conditions (minimum scale problem), how to reduce the scale, call itself

基本结束条件(最小规模问题)，如何减小规模，调用自身

Hanot Tower problem: decomposition into a recursive form
汉诺塔问题：分解为递归形式

Suppose we have 5 plates in 1 # and need to move to 3 #

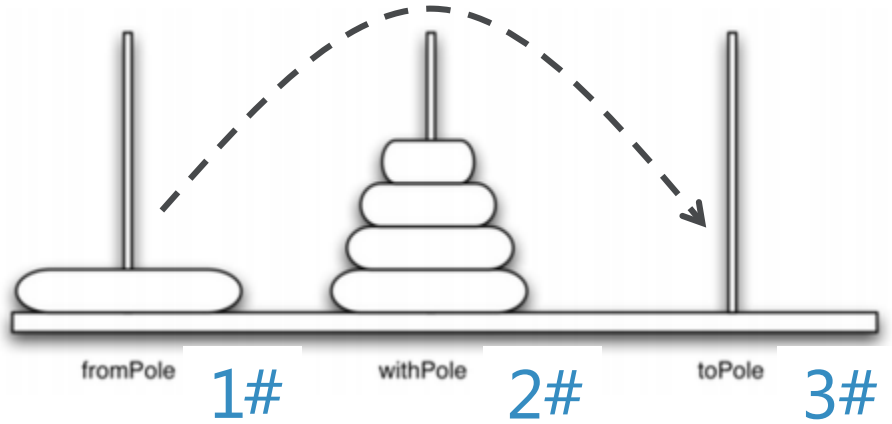
假设我们有5个盘子，穿在1#柱，需要挪到3#柱

If there was a way to move the top pile of four plates to column 2 #, the problem would be solved:

如果能有办法把最上面的一摞4个盘子统统挪到2#柱，那问题就好解决了：

Move the remaining largest plate directly from column 1 # to column 3 #, and then move the stack of four plates on the column 2 # column in the same way, and then the whole move is completed

把剩下的最大号盘子直接从1#柱挪到3#柱，再用同样的办法把2#柱上的那一摞4个盘子挪到3#柱，就完成了整个移动



Hanota problem: Analysis

汉诺塔问题：分析

The next question is solving how can the 4 plates be moved from 1 # to 2 #?

接下来问题就是解决4个盘子如何能从1#挪到2#？

The size of the problem has now been reduced!The same way is to move the top pile of three plates to column 3 #, move the remaining largest plate from column 1 # to column 2 #, and then use the same way to move the top pile of three plates from column 3 # to column 2 # in the same way

此时问题规模已经减小！同样是想办法把上面的一摞3个盘子挪到3#柱，把剩下最大号盘子从1#挪到2#柱，再用同样的办法把一摞3个盘子从3#挪到2#柱

The movement of a pile of three plates also followed this:

一摞3个盘子的挪动也照此：

Divided into a top pile of 2, and the largest plate below

分为上面一摞2个，和下面最大号盘子

So how to move the 2 plates?

那么2个盘子怎么移动？

If this cannot work out, then just break down into 1 plate of movement

不行，就再分解为1个盘子的移动

Hanota problem: Recursive thinking

汉诺塔问题：递归思路

Move the disc tower from the start column, through the middle column, to the target column:

将盘片塔从开始柱，经由中间柱，移动到目标柱：

First, the upper N-1 plate plate tower, from the start column, through the target column, moved to the middle column;

首先将上层N-1个盘片的盘片塔，从开始柱，经由目标柱，移动到中间柱；

Then move the N th (largest) disc from the start column to the target column;

然后将第N个(最大的)盘片，从开始柱，移动到目标柱；

Finally, the N-1 plate placed in the middle column is moved to the target column through the start column.

最后将放置在中间柱的N-1个盘片的盘片塔，经由开始柱，移动到目标柱。

The basic end condition, or the minimum-scale problem, is:

基本结束条件，也就是最小规模问题是：

Moving problem of 1 disc

1个盘片的移动问题

Hanota problem: Recursive thinking
汉诺塔问题：递归思路

The above idea is written in Python, almost the same as the language description:

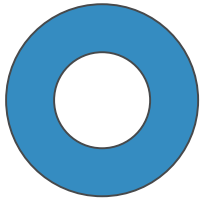
上面的思路用Python写出来，几乎跟语言描述一样：

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

Hanota problem: Code
汉诺塔问题：代码

```
1  def moveTower(height, fromPole, withPole, toPole):
2      if height >= 1:
3          moveTower(height - 1, fromPole, toPole, withPole)
4          moveDisk(height, fromPole, toPole)
5          moveTower(height - 1, withPole, fromPole, toPole)
6
7  def moveDisk(disk, fromPole, toPole):
8      print(f"Moving disk[{disk}] from {fromPole} to {toPole}")
9
10 moveTower(3, "#1", "#2", "#3")
```

```
>>> %Run hanoi.py
  Moving disk[1] from #1 to #3
  Moving disk[2] from #1 to #2
  Moving disk[1] from #3 to #2
  Moving disk[3] from #1 to #3
  Moving disk[1] from #2 to #1
  Moving disk[2] from #2 to #3
  Moving disk[1] from #1 to #3
```



Recursive application: Explore the maze

递归的应用：探索迷宫

Explore the maze: a maze of ancient Greece
探索迷宫：古希腊的迷宫

King Minos of Crete in ancient Greece
古希腊克里特岛米诺斯王

牛头人身怪物米诺陶洛斯

sacrifice of boys and girls , Prince Theseus of Athens

童男童女献祭，雅典王子忒修斯

Princess, sharp sword, line ball

公主，利剑，线团

The old king who jumps into the sea……Aegean Sea

老国王投海……爱琴海



Explore the maze: the yellow flower array in Summer Palace

探索迷宫：圆明园的黄花阵

Located in western building scenic spot of Summer Palace

位于圆明园西洋楼景区

Explore the maze
探索迷宫

How to find the turtle in the middle of the maze

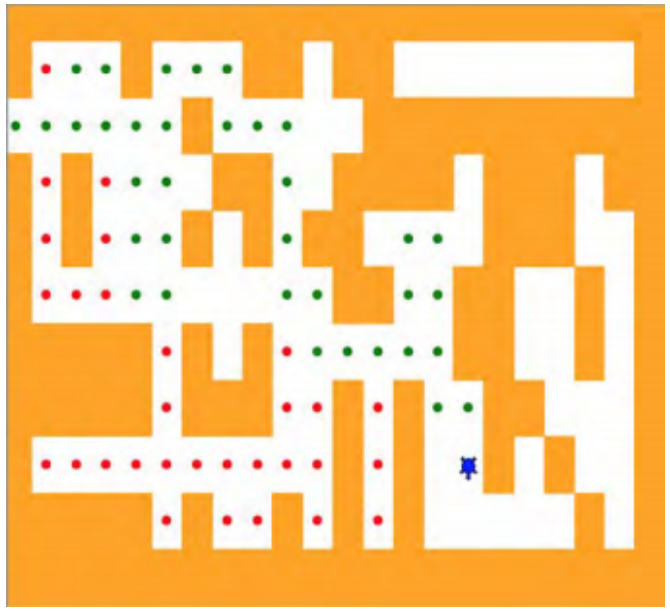
将海龟放在迷宫中间，如何能找到出口

First, we divide the space (rectangle) of the maze into rows of squares, distinguishing walls and channels.

首先，我们将整个迷宫的空间(矩形)分为行列整齐的方格，区分出墙壁和通道。

Give each square a ranks position, and give "wall", "channel" attributes

给每个方格具有行列位置，并赋予“墙壁”、“通道”的属性



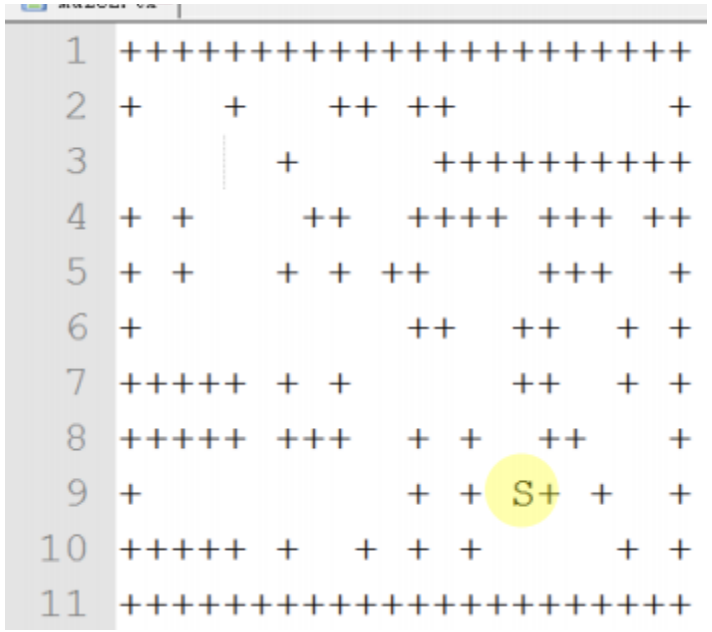
Data structure of the maze
迷宫的数据结构

Consider using a matrix approach to implement the maze data structure

考虑用矩阵方式来实现迷宫数据结构

The two-level list method of "Data item being char list" is used to save the square content, and different chars are used to represent "wall +", "channel" and "turtle dropping point S" ,as well as read the maze data from a text file line by line

采用“数据项为字符列表的列表”这种两级列表的方式来保存方格内容，采用不同字符来分别代表“墙壁+”、“通道”、“海龟投放点s”从一个文本文件逐行读入迷宫数据



Data structure of the maze: MazeClass

迷宫的数据结构：MazeClass

```
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)
```

Save the matrix
保存矩阵

Data structure of the maze: MazeClass

迷宫的数据结构：MazeClass

After successfully reading the data file

读入数据文件成功后

The mazelist is shown in the figure below

mazelist如下图示意

`mazelist[row][col]=='+'`

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

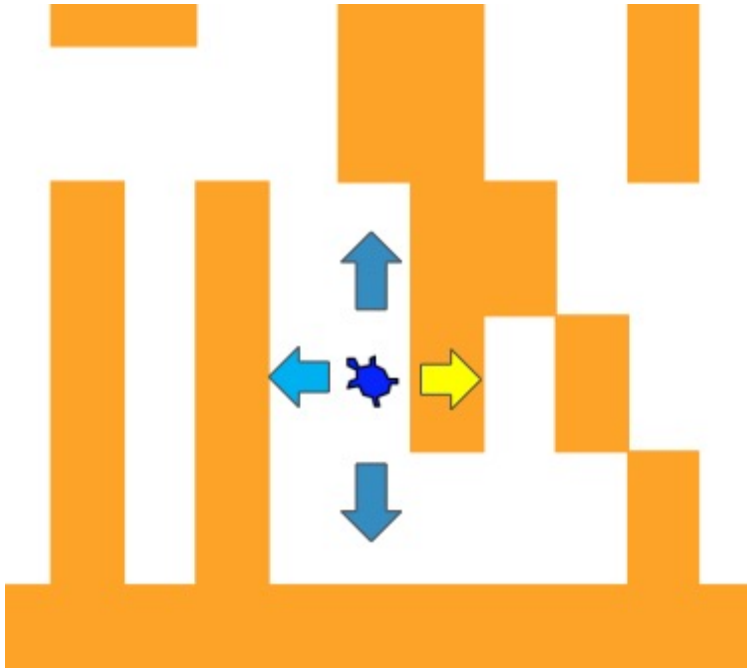
Explore the maze: algorithm ideas
探索迷宫：算法思路

With the maze data structure determined, we know that for a turtle, it is in a certain square

确定了迷宫数据结构之后，我们知道，对于海龟来说，其身处某个方格之中

The direction that it can move, it must be in the direction of the channel, and if one direction is a wall square, it must move in another direction

它所能移动的方向，必须是向着通道的方向，如果某个方向是墙壁方格，就要换一个方向移动



Explore the maze: algorithm ideas
探索迷宫：算法思路

In this way, the recursive algorithm for exploring the maze is as follows: 这样，探索迷宫的递归算法思路如下：

①Move the turtle one step north from its original location, and recursively call the exploration maze to a new location to find the exit;
将海龟从原位置向北移动一步，以新位置递归调用探索迷宫寻找出口；

②If the step above does not find the exit, then move the turtle south from the original position to explore the maze in the new position;
如果上面的步骤找不到出口，那么将海龟从原位置向南移动一步，以新位置递归调用探索迷宫；

③If the exit is not found in the south, then move the turtle west from the original position to recursively explore the maze with the new position;
如果向南还找不到出口，那么将海龟从原位置向西移动一步，以新位置递归调用探索迷宫；

④If no exit can be found to the west, then move the turtle east from its original position, and explore the maze recursively in a new position;
如果向西还找不到出口，那么将海龟从原位置向东移动一步，以新位置递归调用探索迷宫；

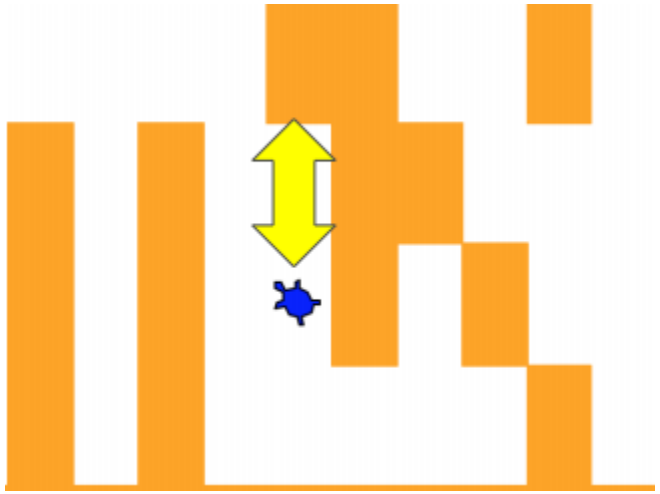
⑤If none of the exits can be found in the top four directions, then the maze has no exits!
如果上面四个方向都找不到出口，那么这个迷宫没有出口！

Explore the maze: algorithm ideas
探索迷宫：算法思路

The idea looks perfect, but some details are crucial
思路看起来很完美，但有些细节至关重要

If we move the turtle in a certain direction (such as north) , if the north of new position is a wall, then the recursive call at the new position will make the turtle try to the south, but one square to the south of the new position is exactly the original position before the recursive call, so it falls into an infinite loop of infinite recursive.

如果我们向某个方向(如北)移动了海龟，那么如果新位置的北正好是一堵墙壁，那么在新位置上的递归调用就会让海龟向南尝试，可是新位置的南边一格，正好就是递归调用之前的原位置，这样就陷入了无限递归的死循环之中



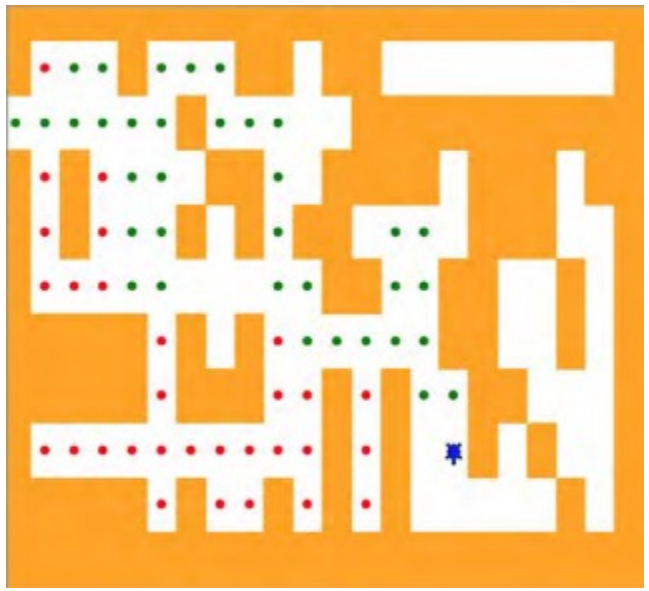
Explore the maze: algorithm ideas
探索迷宫：算法思路

So we need a mechanism to record the path that turtle went through

所以需要有个机制记录海龟所走过的路径

Sprinkle "bread crumbs" along the way, once you find "breadcrumbs" in the forward direction, you can't step on it, but you must try another direction. For recursive calls, it means that "breadcrumbs" are found on a square in a certain direction, and immediately return to the previous level from the recursive call.

沿途洒“面包屑”，一旦前进方向发现“面包屑”，就不能再踩上去，而必须换下一个方向尝试。对于递归调用来说，就是某方向的方格上发现“面包屑”，就立即从递归调用返回上一级。



Explore the maze: algorithm ideas
探索迷宫：算法思路

The "basic end condition" for the recursion call is summarized as follows:

递归调用的“基本结束条件”归纳如下：

①If the turtle meets the "wall" square, the recursive call ends and returns failure;

海龟碰到“墙壁”方格，递归调用结束，返回失败；

②If the turtle touches the "breadcrumb" square, indicating that the square has been visited, the recursion call ends, and returns failure;

海龟碰到“面包屑”方格，表示此方格已访问过，递归调用结束，返回失败；

③If the turtle touches the "exit" square, which is the "channel at the edge" square, then the recursive call ends and returns successfully!

海龟碰到“出口”方格，即“位于边缘的通道”方格，递归调用结束，返回成功！

④The turtle failed to explore in all four directions, with recursive calls ending and return failure

海龟在四个方向上探索都失败，递归调用结束，返回失败

Exploring the maze: an auxiliary animation process
探索迷宫：辅助的动画过程

To allow the turtle to run in the maze map, we added some members and methods to the maze data structure MazeClass

为了让海龟在迷宫图里跑起来，我们给迷宫数据结构MazeClass添加一些成员和方法

t: A mapping turtle with shape to the turtle (default is an arrow)

t：一个作图的海龟，设置其shape为海龟的样子(缺省是一个箭头)

drawMaze( ): Draw the drawing of the maze, draw the wall with a solid square

updatePosition

drawMaze()：绘制出迷宫的图形，墙壁用实心方格绘制

updatePosition(row, col, val): update the position of the turtle, and mark

updatePosition(row,col,val)：更新海龟的位置，并做标注

isExit (row, col): determine whether is the "exit"

isExit(row,col)：判断是否“出口”

Explore the maze: Recursive algorithm code
探索迷宫：递归算法代码

```
96  def searchFrom(maze, startRow, startColumn):
97      #  1. 碰到墙壁，返回失败
98      maze.updatePosition(startRow, startColumn)
99      if maze[startRow][startColumn] == OBSTACLE :
100         return False
101
102     #  2. 碰到面包屑，或者死胡同，返回失败
103     if maze[startRow][startColumn] == TRIED or \
104        maze[startRow][startColumn] == DEAD_END:
105         return False
106
107     #  3. 碰到了出口，返回成功!
108     if maze.isExit(startRow,startColumn):
109         maze.updatePosition(startRow, startColumn, PART_OF_PATH)
110         return True
111
112     #  4. 洒一下面包屑，继续探索
113     maze.updatePosition(startRow, startColumn, TRIED)
114
115     # 向北南西东4个方向依次探索，or操作符具有短路效应
116     found = searchFrom(maze, startRow-1, startColumn) or \
117             searchFrom(maze, startRow+1, startColumn) or \
118             searchFrom(maze, startRow, startColumn-1) or \
119             searchFrom(maze, startRow, startColumn+1)
120
121     # 如果探索成功，标记当前点，失败则标记为“死胡同”
122     if found:
123         maze.updatePosition(startRow, startColumn, PART_OF_PATH)
124     else:
125         maze.updatePosition(startRow, startColumn, DEAD_END)
126     return found
```

Discuss

1）During the execution of recursive algorithms, the data structure that the computer system must use is ( ).

在递归算法执行过程中，计算机系统必定会用到的数据结构是（ ）。

A.queue 队列　　B.linked list 链表　　C.stack 栈　　D.binary tree 二叉树

栈的特点是“先进后出，后进先出”，在程序执行过程中，主程序先进栈，被调用的程序后进栈；当被调用程序结束后，先出栈，最后主程序运行结束了，主程序才出栈。在递归调用函数时，系统会在栈上分配内存，因此一定会用到栈。

2）A recursive algorithm is set up as follows, and the final printed result is ( ).

设有递归算法如下，最终打印结果是（ ）。

```
int foo(int a ,int b)
{
    if (b == 0) return 0;
    if (b % 2 == 0) return foo(a+a,b/2);
    return foo(a+a,b/2)+a;
}
int main()
{
  printf("%d", foo(1,3));
  return 0;
}
```

对于foo(1,3),a=1,b=3,返回值等于foo(2,1)+1；
对于foo(2,1),a=2,b=1,返回值等于foo(4,0)+2；
对于foo(4,0),a=4,b=0,返回值等于0；
因此foo(1,3)=1+2+0=3，故A正确，B、C、D错误

A.3　　B.4　　C.5　　D.6

Discuss

3）The Hanoi Tower of the four disks, with a total number of moves (　).

4个圆盘的Hanoi塔,总的移动次数为（　）。

A.7　　B.8　　C.15　　D.16

答案为C。设f(n)为n个圆盘的hanoi塔总的移动次数，其递推方程为f(n)=f(n-1)+1+f(n-1)=2*f(n-1)+1。理解就是先把上面n-1个圆盘移到第二个柱子上(共f(n-1)步），再把最后一个圆盘移到第三个柱子(共1步），再把第二柱子上的圆盘移动到第三个柱子上（共f(n-1)步）。而f(1)=1；于是f(2)=3,f(3)=7,f(4)=15。故答案为C。进一步，根据递推方程其实可以得出f(n) = 2^n - 1。

4）The factorial of a positive integer is multiplied from 1 to itself. The following recursive function calculates the factorial. Please analyze its time complexity to be (　).

一个正整数的阶乘是从1连乘到它本身，下面递归函数实现了阶乘的计算，请分析出它的时间复杂度为（　）。

```
int factorial(int n)
{
    if(i <= 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

A.O(1)　　B.O(logN)　　C.O(N)　　D.O(N*logN)

由代码可知展开f(n) = n * f(n-1) = n * (n-1) * f(n-2) = … = n * (n-1) * (n-2) *…*1，每进入一次递归函数factorial，n减小1，且每次只需要O(1)时间完成乘法和减法操作，因此总的时间复杂度为O(N)。

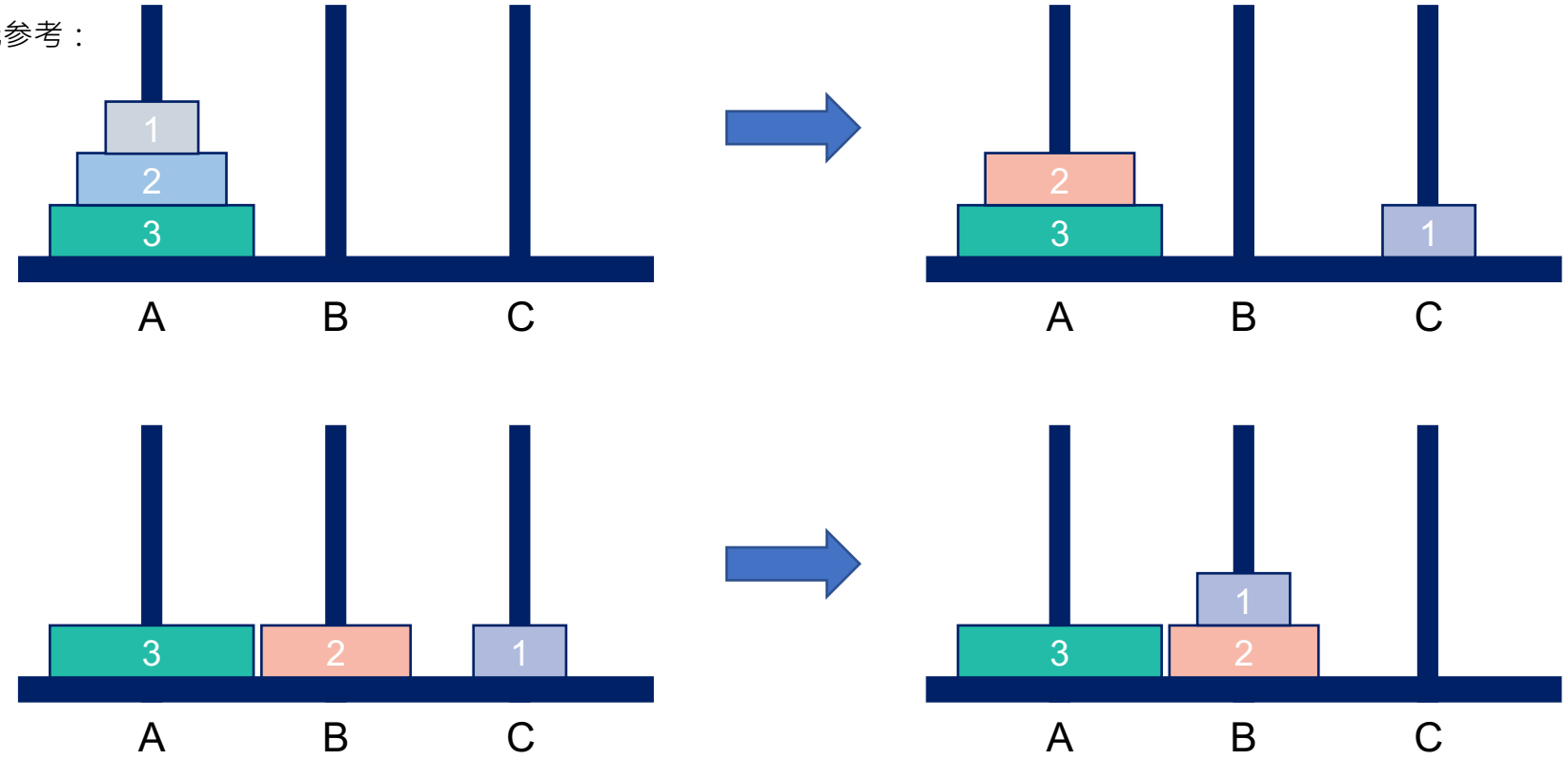
Discuss

make some noise

1）Draw the call stack for the Hannotta problem (assuming there are three plates in the initial stack).

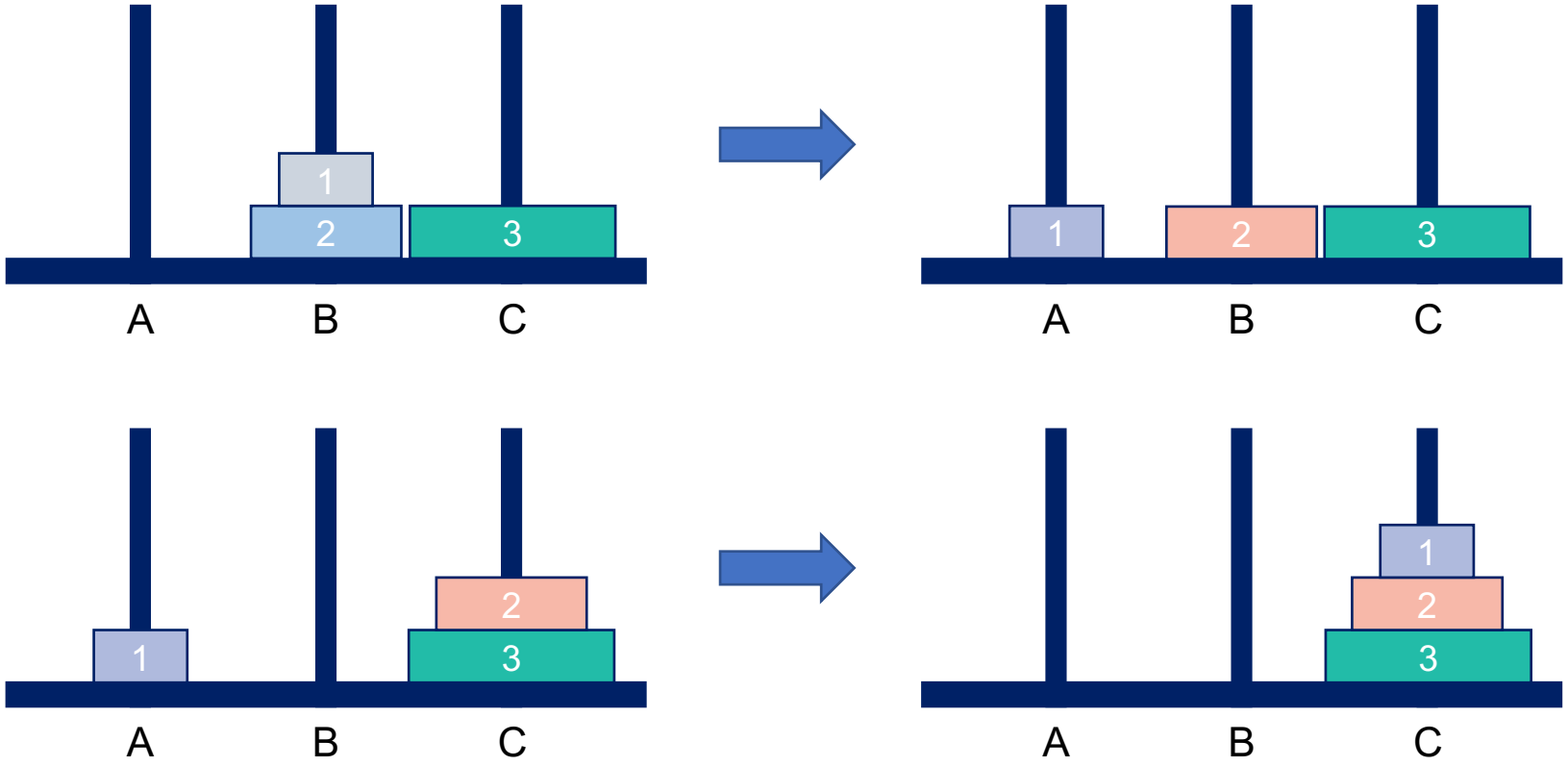
画出汉诺塔问题的调用栈（假设起初栈中有3个盘子）。

图示化参考：



Discuss

make some noise



Discuss

make some noise

2）Write a recursive function to invert the list.

写一个递归函数来反转列表。

```
rl = []
def reverseList(l, idx):
    global rl
    if idx >= len(l):
        return
    else:
        reverseList(l, idx + 1)
    rl.append(l[idx])

def reverseListtest():
    mrl = [1, 2, 3, 4, 5]
    reverseList(mrl, 0)
    print(rl)

if __name__ == '__main__':
    reverseListtest()
```

Discuss

make some noise

3）There are ***n*** steps on the stairs. A frog can go up 1,2 or 3 steps at a time to achieve a method of calculating how many ways the frog can go up the stairs.

楼梯有n个台阶,一个青蛙一次可以上1,2或3阶,实现一个方法,计算该青蛙有多少种上完楼梯的方法。

假如青蛙上10阶，那么其实相当于要么站在第9阶向上走1步，要么站在第8阶向上走两步，要么在第7阶向上走3步，每个大于3阶楼梯的问题都可以看成几个子问题的堆叠。

If a frog goes up 10 steps, it is equivalent to either standing up 1 step on the ninth step, or two steps on the eighth step, or three steps on the seventh step. Each problem larger than the third step can be viewed as a stack of several sub-problems.

变化：令f(n)为青蛙上n阶的方法，则f(n) = f(n -1) + f(n - 2) + f(n -3)，当n>=3

Change: If ***f(n)*** is the method of frog's upper n-order, ***f(n) = f(n-1) + f(n-2) + f(n-3)***, when ***n>=3***

递归出口：当n=0时，青蛙不动，f(0)=0；n=1时，有1种方法，n=2时，有2种方法

Recursive exit: frog does not move when ***n=0***, ***f(0)=0***; When ***n=1,*** there is one method, and when ***n=2***, there are two methods

Discuss

所以能得到如下递归写法：So you get the following recursive writings:

```
def f(n):
    if n == 0 :
        return 1
    if n == 1:
        return 1
    if n == 2:
        return 2
    return f(n - 1) + f(n - 2) + f(n - 3)
```

#站着不动也得返回1的, 因为实际上0种方法的是没意义

Standing still also has to return 1, because in fact the **0** method is meaningless

显然这样的递归方法不是很直观，不如直接列出前几阶楼梯的走法看看规律的变化：

Obviously, this recursive method is not very intuitive. It's better to list the walking methods of the first few stairs to see the changes of the law:

阶数 steps	走法 means				
1	1	0->1			
2	2	0->1->2	0->2		
3	4	0->1->2->3	0->1->3	0->2->3	0->3
4	7	...			
...	...				

Discuss

Let's take a look at the walking method when the order is 4:

再看看阶数为4时的走法：

0->1->4	0->1->2->4	0->1->2->3->4
	0->2->4	0->1->3->4
		0->2->3->4
		0->3->4

If you don't look at the red part, the three columns represent the methods of the first, second and third order respectively. Now take a look with the red -> 4:

如果不看红色部分的话，三列分别代表了上第1，2，3阶的方法。现在带着红色的->4一起看：

第一列：相当于先上到第1阶再一次上到4（因为最大可以跨3阶）The first column: it is equivalent to going up to the first level and then up to 4th

第二列：相当于先上到第2阶再一次上到4（相当于最后一次跨2阶）The second column: it is equivalent to going up to the second level and then up to 4th

第三列：相当于先上到第3阶再一次上到4（最后一次跨1阶即可）The third column: it is equivalent to going up to the third level and then up to the 4th

显然上到第4阶的方法刚好就是这三列的和了。Obviously, the method up to the fourth order is just the sum of these three columns.

不难得出，上到第5阶的走法如下：

第一列：相当于先上到第2阶再一次上到5（因为最大可以跨3阶）

第二列：相当于先上到第3阶再一次上到5（相当于最后一次跨2阶）

第三列：相当于先上到第4阶再一次上到5（最后一次跨1阶即可）

显然上到第5阶的方法刚好也是这三列的和。

Discuss

不难得出阶数为n时的走法，如下： It is not difficult to find out the walking method when the order is n, as follows:

第一列：相当于先上到第n-3阶再一次上到n The first column: it is equivalent to going up to the **nth-3rd** order and then up to **n**

第二列：相当于先上到第n-2阶再一次上到n The second column: it is equivalent to going up to the **nth-2ed** order first and then up to **n**

第三列：相当于先上到第n-1阶再一次上到n The third column: it is equivalent to going up to the **nth-1st** order first and then up to **n**

同样，上到第n阶的方法刚好就是这三列的和，所以也就有`f(n) = f(n - 1) +f(n - 2) + f(n - 3)`

Similarly, the method up to the nth order is just the sum of these three columns, so there is $f(n) = f(n-1) + f(n-2) + f(n-3)$

可以写出如下的递归算法：The following recursive algorithm can be written:

```
def go_stairs(n):
    if n <= 1:
        return 1
    if n == 2 :
        return  2
    if n == 3 :
        return  4
    return go_stairs(n - 1) + go_stairs(n - 2) + go_stairs(n - 3)
```

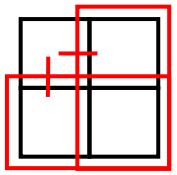
Discuss

4) There is a grid of ***x*y*** size. A robot can only walk on grid points and can only walk to the right or down. Now the robot wants to walk from the upper left corner to the lower right corner, please design an algorithm to calculate how many walking methods the robot has. Given two positive integers ***x*** and ***y***, return the number of walking methods of the robot

有一个x*y大小的方格, 一个机器人只能走格点且只能向右或者向下走, 现机器人要从左上角走到右下角，请设计一个算法, 计算机器人有多少种走法，给定两个正整数x和y, 返回机器人走法的数目。

显然，一个格子就只有一种走法，f(1,1) = 1

Obviously, there is only one way to walk in a grid, f(1,1) = 1



2*2时，此时机器人处在(1,1)的位置，第一步有两个选择：

向右，进入（1，2），下一步只能向下

向下，进入（2，1），下一步只能向右，可以得到：f(2,2) = f(2,1) + f(1,2) = 1 +1 = 2

事实上我们有：

f(n,1) = 1，因为向右只有1列；f(1,n) = 1，因为向下只有1行。

2*2, there are two choices in the first step:

Right, into (1, 2), next step can only be down;

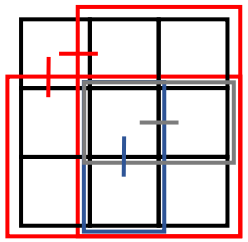
Down, into (2,1), the next step can only be right, you can get: **f(2,2) = f(2,1) + f(1,2) = 1 +1 = 2**

In fact, we have:

f(n,1) = 1, because there is only one column to the right;

f(1,n) = 1, because there is only one line down.

Discuss



3*3时，同样拆解为两种选择的和：
当机器人在(3，2)时，可以通过“向右”进入(3，3)，即f(3,2)；
当机器人在(2，3)时，可以通过“向下”进入(3，3)，即f(2,3)，
可以得到：f(3,3) = f(3,2) + f(2,3)

At 3*3, it is also split into the sum of two choices:
When the robot is in (3,2), it can enter (3,3), that is, f(3,2), by "going right"
When the robot is in (2,3), it can enter (3,3), that is, f(2,3), by "going down"
you can get: **f(3,3) = f(3,2) + f(2,3)**

那么f(3,2)呢？同样可以拆解为两种选择的和：
当机器人在(3，1)时，可以通过“向右”进入(3，2)，即f(3,1) = 1
当机器人在(2，2)时，可以通过“向下”进入(3，2)，即f(2,2) ，
之前已经算过了

What about f(3,2)? It can also be split into two alternative sums:
When the robot is in (3,1), it can enter (3,2) by "going right", that is, f(3,1) = 1
When the robot is in (2,2), it can enter (3,2), that is, f(2,2), by going "down", which has been previously calculated.

同样地，对于f(2,3)，也可以将其拆分为两种选择的和：
当机器人在(1，3)时，可以通过“向下”进入(2，3)，即f(1,3) = 1
当机器人在(2，2)时，可以通过“向右”进入(2，3)，即f(2,2) ，
之前已经算过了

Similarly, for f(2,3), it can be split into two alternative sums:
When the robot is in (1,3), it can enter (2,3) by "down", that is, f(1,3) = 1
When the robot is in (2,2), it can enter (2,3), that is, f(2,2), by "going right", which has been calculated before.

Discuss

从而我们可以得到如下的递归写法：

So we can get the following recursive writings:

```
def robot_go_grim(x, y):
    if (x == 1 or y == 1):
        return 1
    return  robot_go_grim(x - 1 , y ) +robot_go_grim(x , y - 1 )
```

Discuss

5）Given ***n*** pairs of parentheses, write a function to generate all the legal combinations of n pairs of parentheses.
For example, given n=3, the solution is:
给出n对括号，请编写一个函数来生成所有的由n对括号组成的合法组合。
例如，给出n=3，解集为："((()))", "(()())", "(())()", "()()()", "()(())"

该问题相当于一共n个左括号和n个右括号，可以给我们使用，我们需要依次组装这些括号。每当我们使用一个左括号之后，就剩下n-1个左括号和n个右括号给我们使用，结果拼在使用的左括号之后就行了，因此后者就是一个子问题，可以使用递归。

This problem is equivalent to a total of ***n*** left and ***n*** right parentheses, which can be used for us, and we need to assemble them in turn. Whenever we use a left bracket, ***n-1*** left brackets and ***n*** right brackets are left for us to use. The result is spelled after the used left bracket, so the latter is a subproblem and recursion can be used.

●终止条件：左右括号都使用了n个，将结果加入数组。
●返回值：每一级向上一级返回后续组装后的字符串，即子问题中搭配出来的括号序列。
●本级任务：每一级就是保证左括号还有剩余的情况下，使用一次左括号进入子问题，或者右括号还有剩余且右括号使用次数少于左括号的情况下使用一次右括号进入子问题。

☐Termination condition: n left and right parentheses are used to add the result to the array.
☐Return value: Each level goes up to the next level and returns the subsequently assembled string, a sequence of parentheses matched in the subproblem.
☐Task at this level: Each level is to enter a subproblem with one left parenthesis if there is still one left parenthesis, or with one left parenthesis and fewer right parentheses than left parentheses.

Discuss

但是这样递归不能保证括号一定合法，我们需要保证左括号出现的次数比右括号多时我们再使用右括号就一定能保证括号合法了，因此每次需要检查左括号和右括号的使用次数。

However, such recursion does not guarantee that brackets are legitimate. We need to make sure that when left brackets occur more often than right brackets, we can make sure that right brackets are legitimate, so we need to check the number of times left and right brackets are used each time.

具体做法：

step1：将空串与左右括号各自使用了0个送入递归。 *Empty strings and left and right parentheses are used in **0** feed-in recursion.*

step2：若是左右括号都使用了n个，此时就是一种结果。 *If **n** brackets are used, this is a result.*

step3：若是左括号数没有到达n个，可以考虑增加左括号，或者右括号数没有到达n个且左括号的使用次数多于右括号就可以增加右括号。 *If the number of left brackets does not reach **n**, consider increasing the number of left brackets, or if the number of right brackets does not reach **n** and left brackets are used more than right brackets.*

Discuss

python实现代码如下：

```
class Solution:
    def recursion(self, left:int, right:int, temp:str, res:List[str], n:int):
        #左右括号都用完了，就加入结果
        if left == n and right == n:
            res.append(temp)
            return
        #使用一次左括号
        if left < n:
            self.recursion(left + 1, right, temp + "(", res, n)
        #使用右括号个数必须少于左括号
        if right < n and left > right:
            self.recursion(left, right + 1, temp + ")", res, n)

    def generateParenthesis(self , n: int) -> List[str]:
        #记录结果
        res = list()
        #记录每次组装的字符串
        temp = str()
        #递归
        self.recursion(0, 0, temp, res, n)
        return res
```