

Introduction to Computation

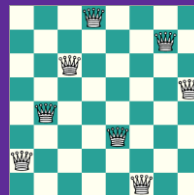
Autumn, 2023

Prof. Fan Cheng

Shanghai Jiao Tong University

chengfan85@gmail.com

<https://github.com/ichengfan/itc>





Outline

- Recursion

Recursion



Function calls

- In python, a function can call another function defined before it
- In a program, we have a chain of function calls: $A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow \dots \rightarrow A_n$
 - Call order: $A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow \dots \rightarrow A_n$
 - Return order: $A_n \rightarrow A_{n-1} \rightarrow A_{n-2} \rightarrow \dots \rightarrow A_1$
 - Example: the boss of a company plans to check the progress of a new staff

```
def f1():  
    print("f1: begin")  
  
def f2():  
    print("f2: begin")  
    f1()  
    print("f1: finish")  
  
def f3():  
    print("f3: begin")  
    f2()  
    print("f2: finish")  
    print("f3: finish")  
  
f3()
```



```
f3: begin  
f2: begin  
f1: begin  
f1: finish  
f2: finish  
f3: finish
```



一个函数可以调用它自己吗?

Experiment

- Assume that we would like to implement a function $f(x)$, where $f(x)$ will call itself inside

```
def f(x):  
    do_sth()  
  
    f(y)  
  
    return
```

- The order of function call $x \rightarrow y \rightarrow z \rightarrow w \rightarrow u \dots$
- If there are infinite function calls in the program, an error will occur
- In `do_sth()`, there must be a condition where the function execution is returned and the self call will not be invoked

```
13 def f(x):  
14     if x == 0:  
15         return 3  
16  
17     return 1 + f(x-1)  
18  
19 print(f(100))
```

Recursion (递归)

- It is legal for a function to call another function
- In mathematics, **recursive functions** allow a series to define itself by the other items

$$f_0 = f_1 = 1, f_{n+2} = f_{n+1} + f_n$$

- Is it legal to define a function by invoke itself in python? Looks weird!
- Recall: A function call is determined by both

(function_name, parameters)

For example, (print, 3) \neq (print, [3])

Thus, in logic it does make sense to invoke itself inside the function definition.

- The challenge is that recursion may be never ended (Logic error)

$$f(n) \rightarrow f(n-1) \rightarrow f(0) \rightarrow f(-1), \dots, f(-\infty)$$

There should be an end!

- Idea: To define a function $f(n)$
 - For the simple cases like $n=0$ or 1 , we directly return its values
 - For the general cases, we try to solve $f(n)$ by reduce it to its previous solutions $f(0), f(1), \dots, f(n-1)$
 - By mathematical induction (数学归纳法), the solutions above always terminate in finite steps

Recursive function

- The formal definition: a method exhibits recursive behavior when it can be defined by **two properties**:
 1. A simple base case (or cases) (基本状态), and
 2. A set of rules which reduce (化简) all other cases toward the base case
- The first property will make sure the function will terminate finally. 简单的情况直接计算 (不递归)
- The second property will call the function itself to reach the base case. 复杂的情况转化为简单的情况 (递归)

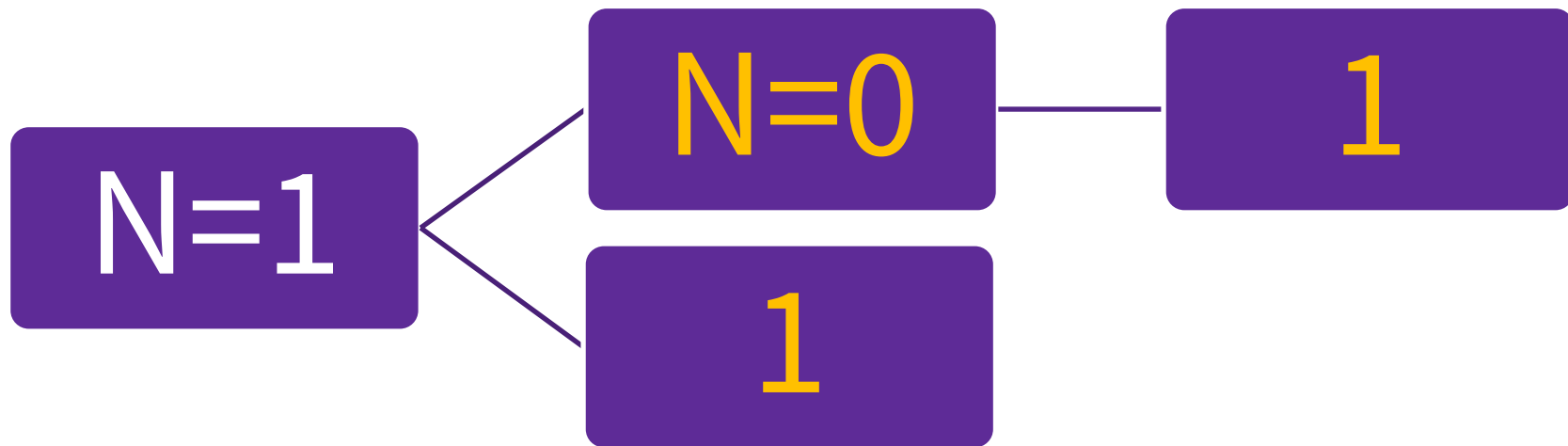
Example: compute the factorial(n): $n!$

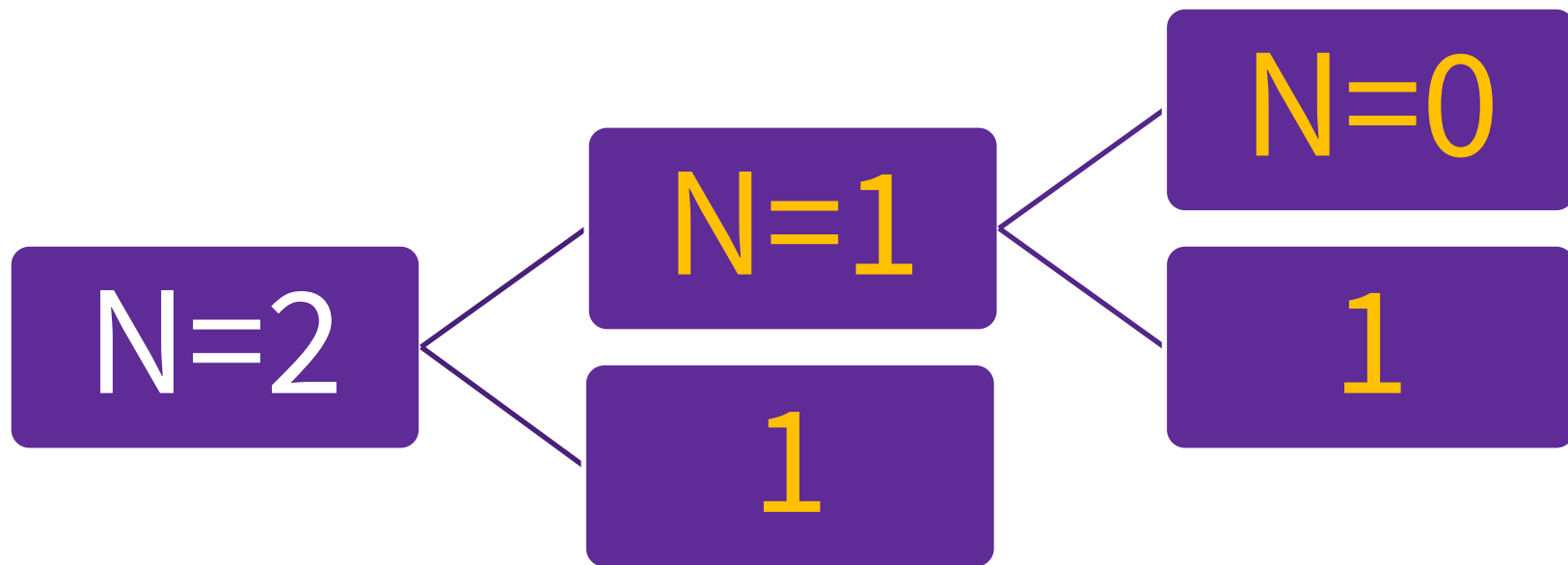
```
1 def factorial(n):
2     print(f"{n} begins")
3     if n == 0:
4         return 1
5     else:
6         f = factorial(n - 1)
7         print(f"{n-1} done")
8         return n * f
9
10
11 print(factorial(7))
```

```
7 begins
6 begins
5 begins
4 begins
3 begins
2 begins
1 begins
0 begins
0 done
1 done
2 done
3 done
4 done
5 done
6 done
5040
```

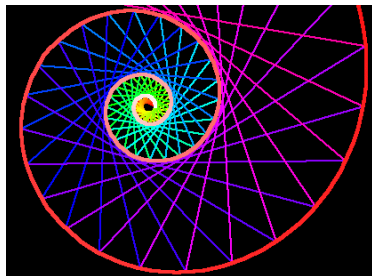
$N=0$

1





Recursion (递归)



递归函数的特点

1. 问题具有递归的结构，可以状态转移(从一个函数值到另一个)
2. 初始状态容易处理

递归函数编写的注意事项，必须严格按照以下步骤

1. 首先判断是否达到基本状态
2. 再决定是否用递归关系处理
3. 否则，会陷入死循环

一个函数调用由 函数名+具体的参数值 决定：f(1)和f(2)是不同的函数调用
递归调用：表面上是自己调用自己，实际上参数不同，是不同的函数调用

Recursive Function: Examples

A simple base case (or cases) (基本状态)

A set of rules which reduce (化简) all other cases toward the base case

```
def gcd(a, b):  
    if a < b:  
        return gcd(b, a)  
  
    if b==0:  
        return a  
    return gcd(b, a%b)  
  
print(gcd(12,3), gcd(3,3))
```

3 3

最大公约数

```
def fibonacci(n):  
    if n == 0:  
        return 0  
  
    if n == 1:  
        return 1  
  
    return fibonacci(n-1) + fibonacci(n-2)  
  
print(fibonacci(10))
```

3 2 1 1

Fibonacci数列

```
def sum_of_bits(n):  
    if n == 0 or n == 1:  
        return n  
  
    return sum_of_bits(n//2) + n%2  
  
print(sum_of_bits(7), sum_of_bits(5), sum_of_bits(4), sum_of_bits(1))
```

55

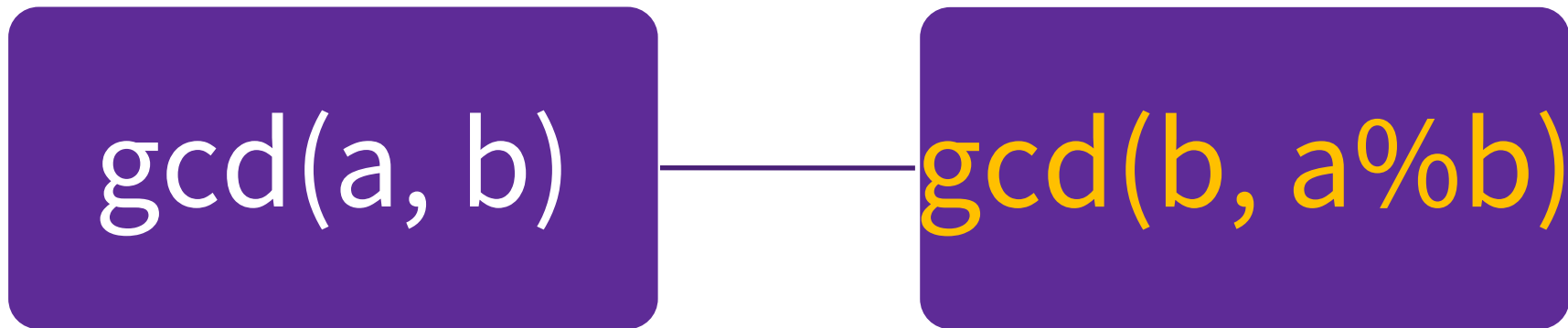
整数的二进制和

$$(9,6) = (6,9\%6) = (6,3) = (3,6\%3) = (3,0) = 3$$

$\text{gcd}(a, b)$

0 ($b=0$)

loga步结束



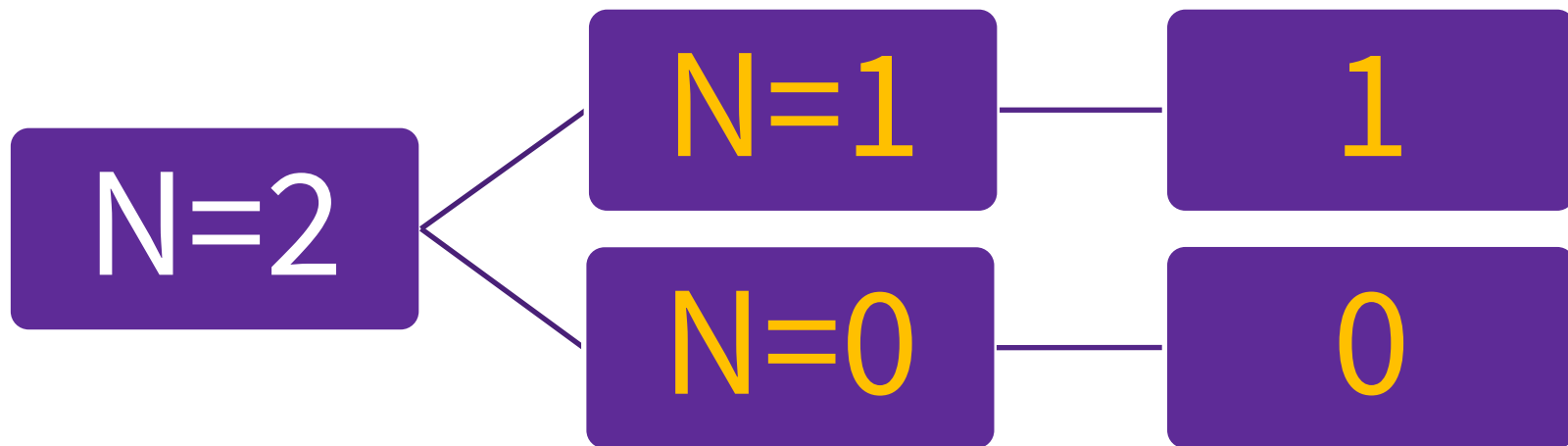
由于 $a > b, a \% b \leq b - 1 < b$, 所以有限步后
 $b=0$

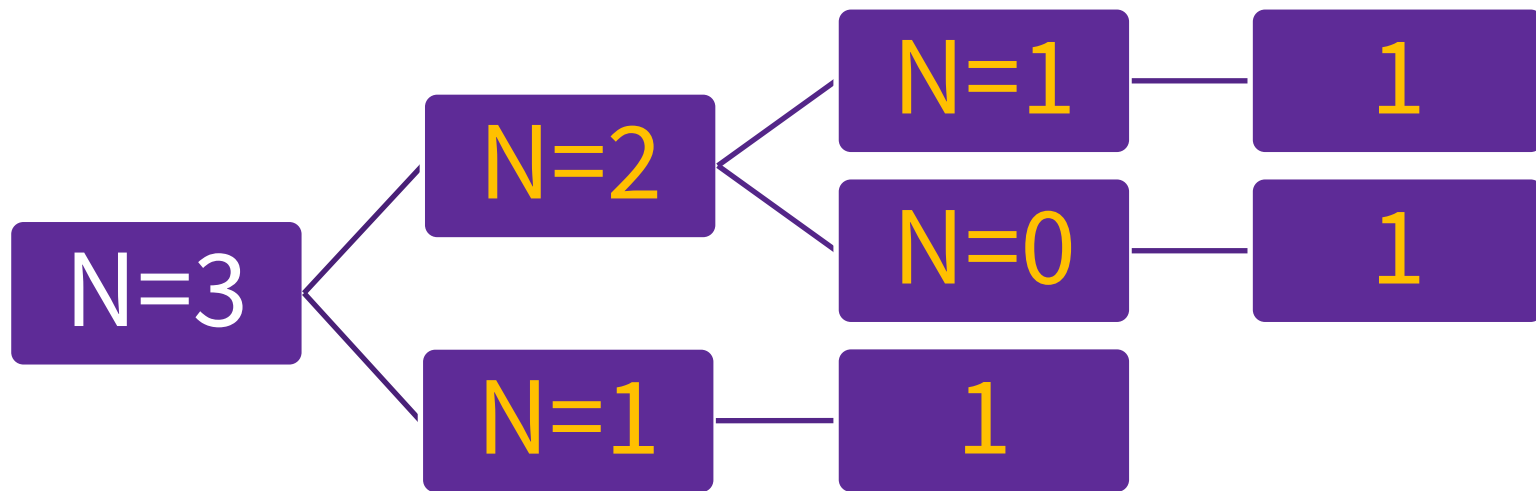
$N=0$

0

$N=1$

1





2^n 步结束

Infinite recursive function

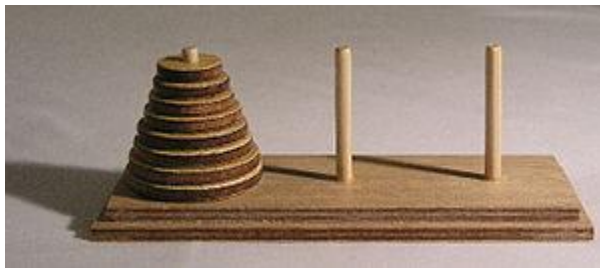
- Recursive functions without base case will lead to errors

```
94 def print_test(str1):  
95     print_test(str1)  
96  
97 print_test("Hello world")
```

```
Traceback (most recent call last):  
  File "C:\Users\fcheng\OneDrive\CS124计算导论\2018\lecture notes\2.py", line 5, in <module>  
    print_test("Hello world")  
  File "C:\Users\fcheng\OneDrive\CS124计算导论\2018\lecture notes\2.py", line 3, in print_test  
    print_test(str)  
  File "C:\Users\fcheng\OneDrive\CS124计算导论\2018\lecture notes\2.py", line 3, in print_test  
    print_test(str)  
  File "C:\Users\fcheng\OneDrive\CS124计算导论\2018\lecture notes\2.py", line 3, in print_test  
    print_test(str)  
  [Previous line repeated 995 more times]  
RecursionError: maximum recursion depth exceeded
```

RecursionError: maximum recursion depth exceeded

Hanoi



- The Tower of Hanoi (汉诺塔) is a mathematical game or puzzle.
- It consists of three rods and a number of disks of different sizes, which can slide onto any rod.
- The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.
- The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
 - A. Only one disk can be moved at a time.
 - B. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
 - C. No disk may be placed on top of a smaller disk

Give a solution to this problem.

假定，任务是从圆柱1，将圆盘经过圆柱2，全部移动到圆柱3。输出移动的过程

```

1  def Hanoi(n, x, y, z):
2      if n == 1:
3          print(f"{n}: {x}->{z}")
4          return
5
6      Hanoi(n - 1, x, z, y)
7      print(f"{n}: {x}->{z}")
8      Hanoi(n - 1, y, x, z)
9
10
11 print("Hanoi: n = 1")
12 Hanoi(1, 1, 2, 3)
13
14 print("Hanoi: n = 2")
15 Hanoi(2, 1, 2, 3)
16
17 print("Hanoi: n = 3")
18 Hanoi(3, 1, 2, 3)

```

```

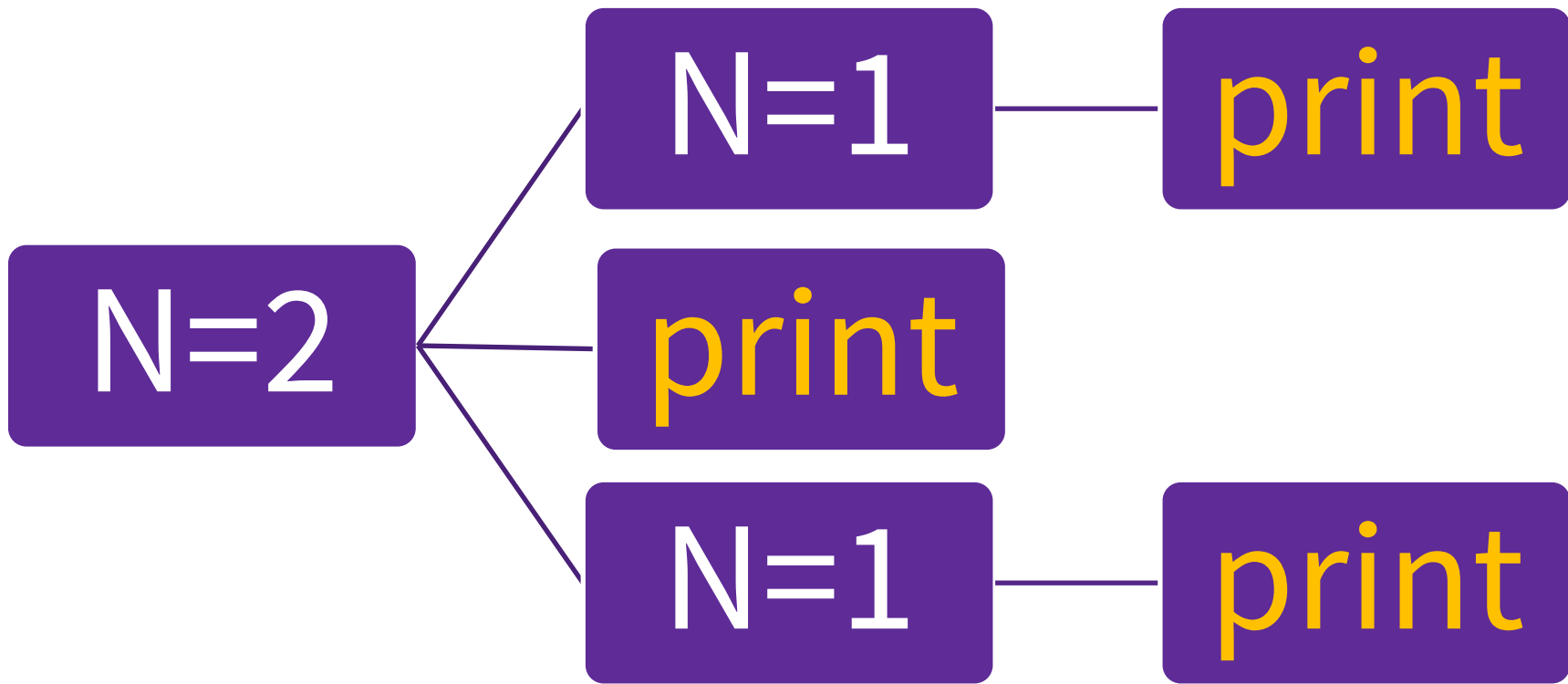
Hanoi: n = 1
1: 1->3
Hanoi: n = 2
1: 1->2
2: 1->3
1: 2->3
Hanoi: n = 3
1: 1->3
2: 1->2
1: 3->2
3: 1->3
1: 2->1
2: 2->3
1: 1->3

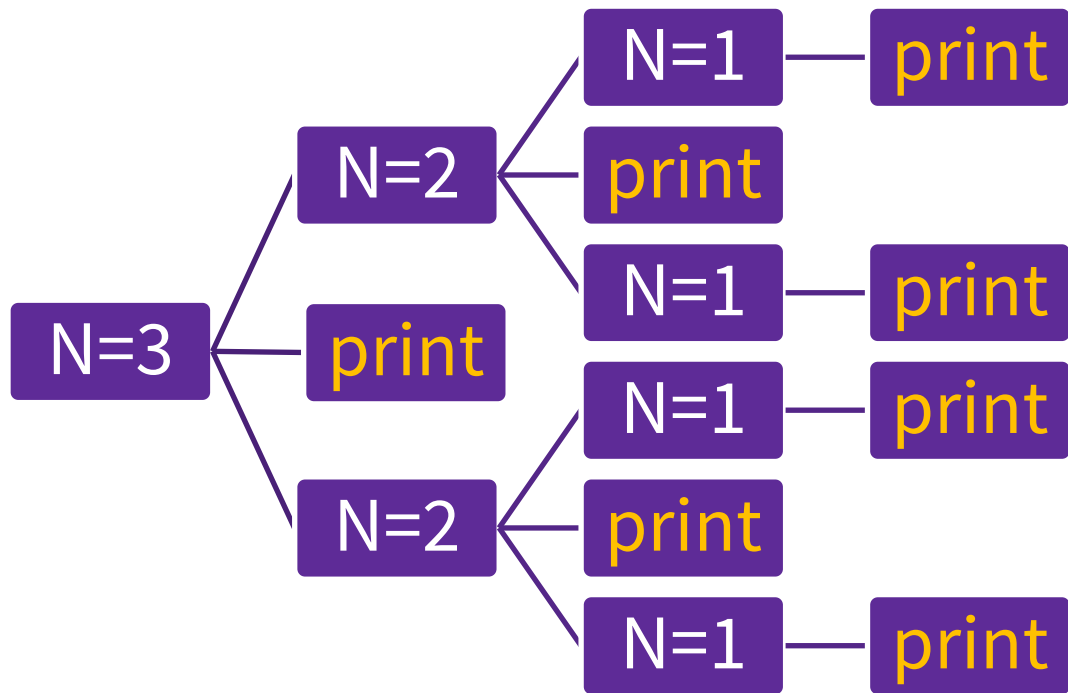
```

函数可以递归定义。但是函数调用时，即使函数名相同，参数不同也可以认为是不同的函数调用

$N=1$

print





递归函数

- 函数：一段具有某种**功能**的代码。
 - 函数执行结束，表示着已经完成了函数**所定义的功能**。
- 递归函数：
 - 1. 自己调用自己：从一个参数状态到另一个参数状态的转移 (状态转移)
 - 2. 初始状态结束递归 (保证不会死循环)
 - 3. 一个函数调用由 函数名+参数 决定，参数不同、函数名相同也是不同的调用
- 功能可以是：
 - 函数值，譬如gcd (简单，好理解)
 - 抽象的一组事件。譬如Hanoi (抽象，要仔细分析)
 - 有问题？？ 从定义出发思考

函数：一段具有某种功能的代码。函数执行结束，表示着已经完成了函数所定义的功能

以函数Hanoi(n, x, y, z)为例：

定义功能：输出将n个盘子从x途经y搬到z的过程 (学习体会！)

由于Hanoi中每个小盘子只能放到大盘子上

首先要把盘子n，从x搬到z。必须先将1-(n-1)号盘子先搬到中间点y。

Hanoi(n-1, x, z, y) # 调用函数 n-1, x, z, y。函数结束运行后，表明已经完成了定义的功能（即输出了所有的步骤！细细体会！！）

然后将n号盘子从x搬到z。

最后一步，我们需要将1-(n-1)号盘子从y搬到z，递归调用即可

Hanoi(n-1, y, x, z)#调用函数 n-1, x, z, y。函数结束运行后，表明已经完成了定义的功能（即输出了所有的步骤！细细体会！！）

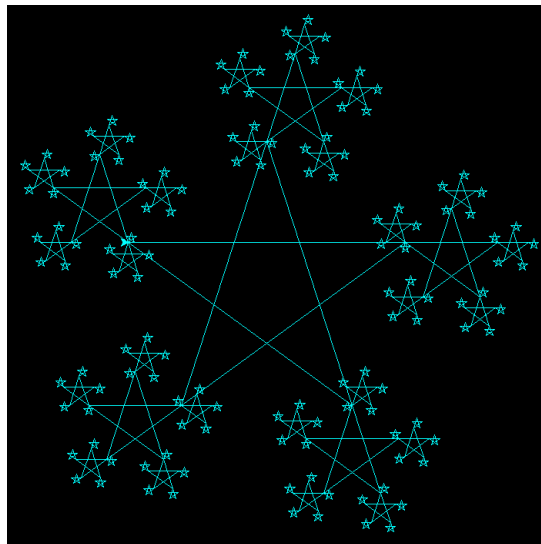
```
def Hanoi(n, x, y, z):  
    if n == 1: # 确保函数会结束递归  
        print(f"{n}: {x}->{z}")  
        return
```

下面三个函数调用完整地实现了Hanoi既定的功能

```
Hanoi(n-1, x, z, y)  
print(f"{n}: {x}->{z}")  
Hanoi(n-1, y, x, z)
```

Fractal

```
1  import turtle
2
3  tur = turtle.Turtle()
4  tur.speed(6)
5  tur.getscreen().bgcolor("black")
6  tur.color("cyan")
7  tur.penup()
8  tur.goto((-200, 50))
9  tur.pendown()
10
11
12  def star(turtle, size):
13      if size <= 10:
14          return
15      else:
16          for i in range(5):
17              turtle.forward(size)
18              star(turtle, size / 3)
19              turtle.left(216)
20
21
22  star(tur, 360)
23  turtle.done()
```



Stack (栈) diagram



- Suppose we have a collection of books. When we pile them on a desk, we will first put a book on surface of the desktop and put the second one on the top of the first one, and so on. This process is called **push (推)**
- When we want to pick up the first book, we need to move the books from the last one to the second one. This process is called **pop (弹)**
- The whole process can be modeled by **stack**.
 - FIFO: First in last out (push→pop)
- In a program, we have a chain of function calls: $A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow \dots \rightarrow A_n$
 - Stack diagram: when a function calls another function, the processed can also be modeled by a stack.
 - The functions A_1, A_2, \dots, A_n will be **pushed** into the stack
 - When A_n is executed, it will be **pop** up from the stack. The system will repeat popping up A_{n-1}, \dots, A_2, A_1