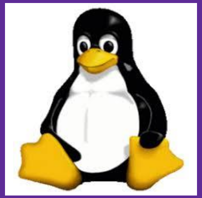
Introduction to Computation

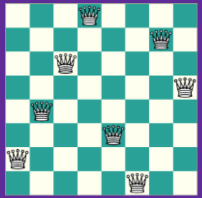
Autumn, 2023

Prof. Fan Cheng

Shanghai Jiao Tong University

chengfan85@gmail.com
https://github.com/ichengfan/itc





5

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: $A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow \cdots \rightarrow A_n$
  - Call order: $A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow \cdots \rightarrow A_n$
  - Return order: $A_n \rightarrow A_{n-1} \rightarrow A_{n-3} \rightarrow \cdots \rightarrow A_1$
  - Example: the boss of a company plans to check the progress of a new staff

```python
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()
```

调用顺序

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

返回顺序

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

```python
def f(x):
    do_sth()

    f(y)

    return
```

- The order of function call $x \to y \to z \to w \to u \dots \dots .$
- 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 will be returned, and the self call will not be invoked

```python
def f(x):
    if x == 0:
        return 3

    return 1 + f(x-1)

print(f(100))
```

```
103
```

一个函数可以调用它自己

Recursion (递归)

It is legal for a function to call itself

- In mathematics, recursive functions allow a series to define itself by the other items

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

- Recall: A function call is determined by both

  (function_name, parameters)

  For example, (print, 3) ≠ (print, [3])

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

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

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

  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 $f(0)$, $f(1)$, …, $f(n-1)$
  - By mathematical induction (数学归纳法), the solutions above always terminate in finite steps

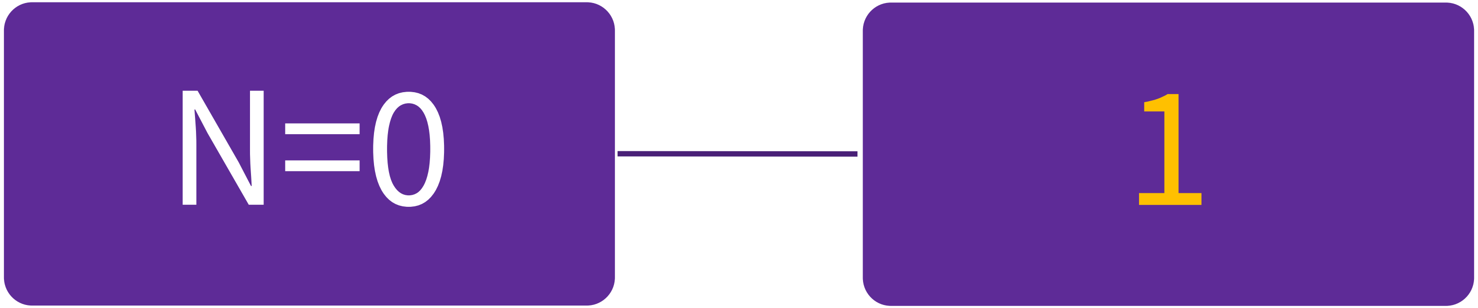
Recursive function

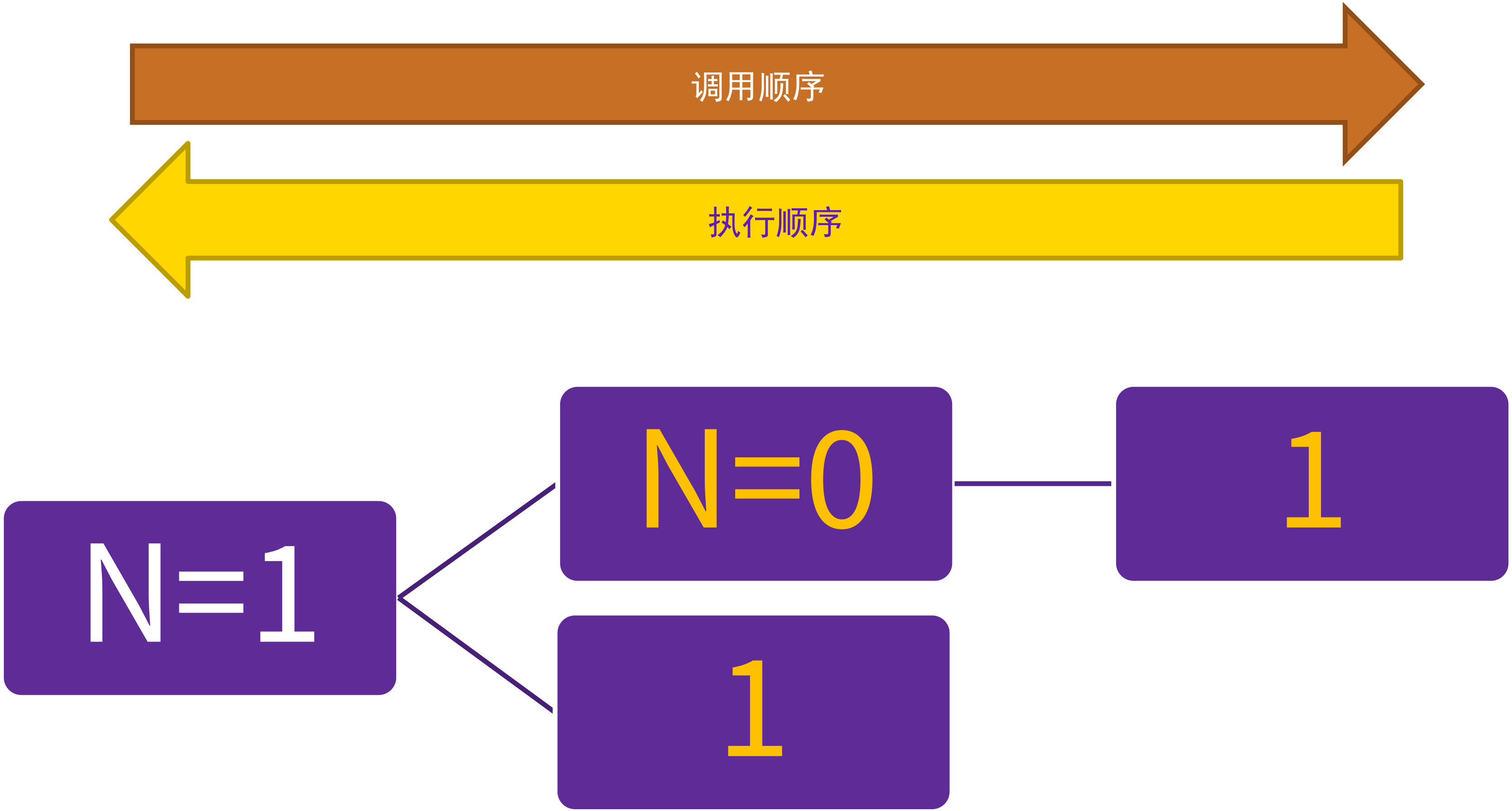
- The formal definition: a method exhibits recursive behavior when it can be defined by two properties:
  1. A simple base case (or cases) (基本状态), and
  2. A set of rules which reduce (化简) all other cases toward the base case
- The first property will make sure the function will terminate finally. 简单的情况直接计算（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): $n!$

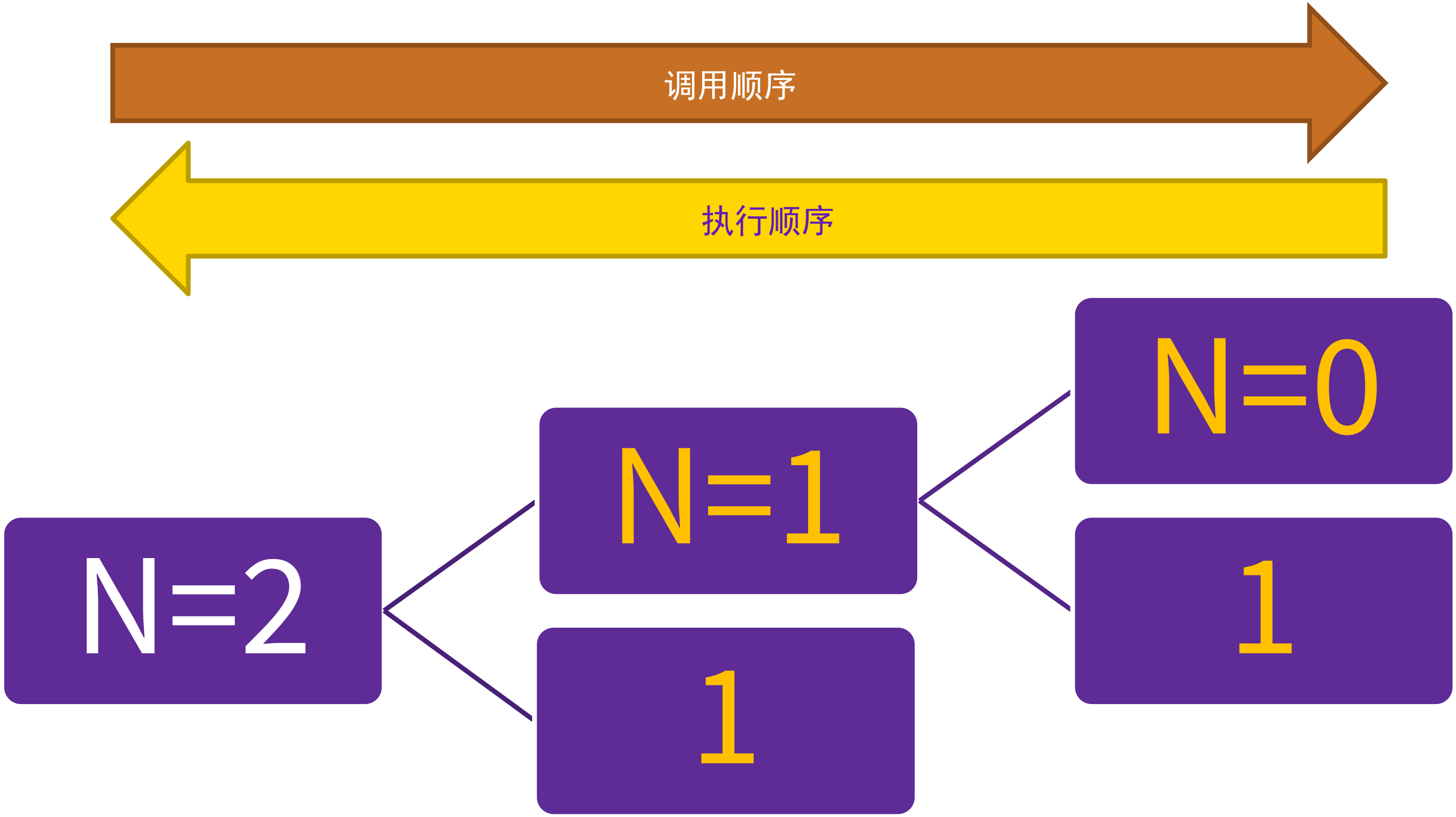
```python
def factorial(n):
    print(f"{n} begins")
    if n == 0:
        return 1
    else:
        f = factorial(n - 1)
        print(f"{n-1} done")
        return n * f


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
```

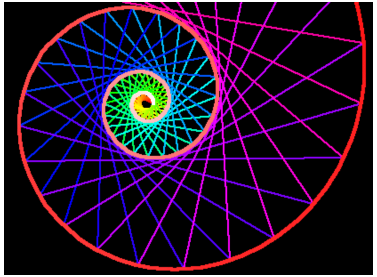

N=0
1



调用顺序
执行顺序
N=1
N=0
1
1


调用顺序
执行顺序
N=2
N=1
1
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

<u>**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**</u>

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

```python
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

```python
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
```

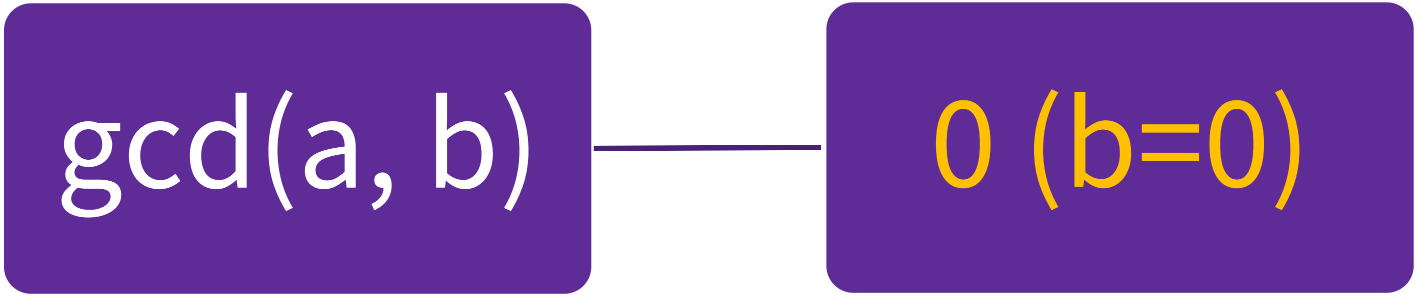
Fibonacci数列

```python
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))
```

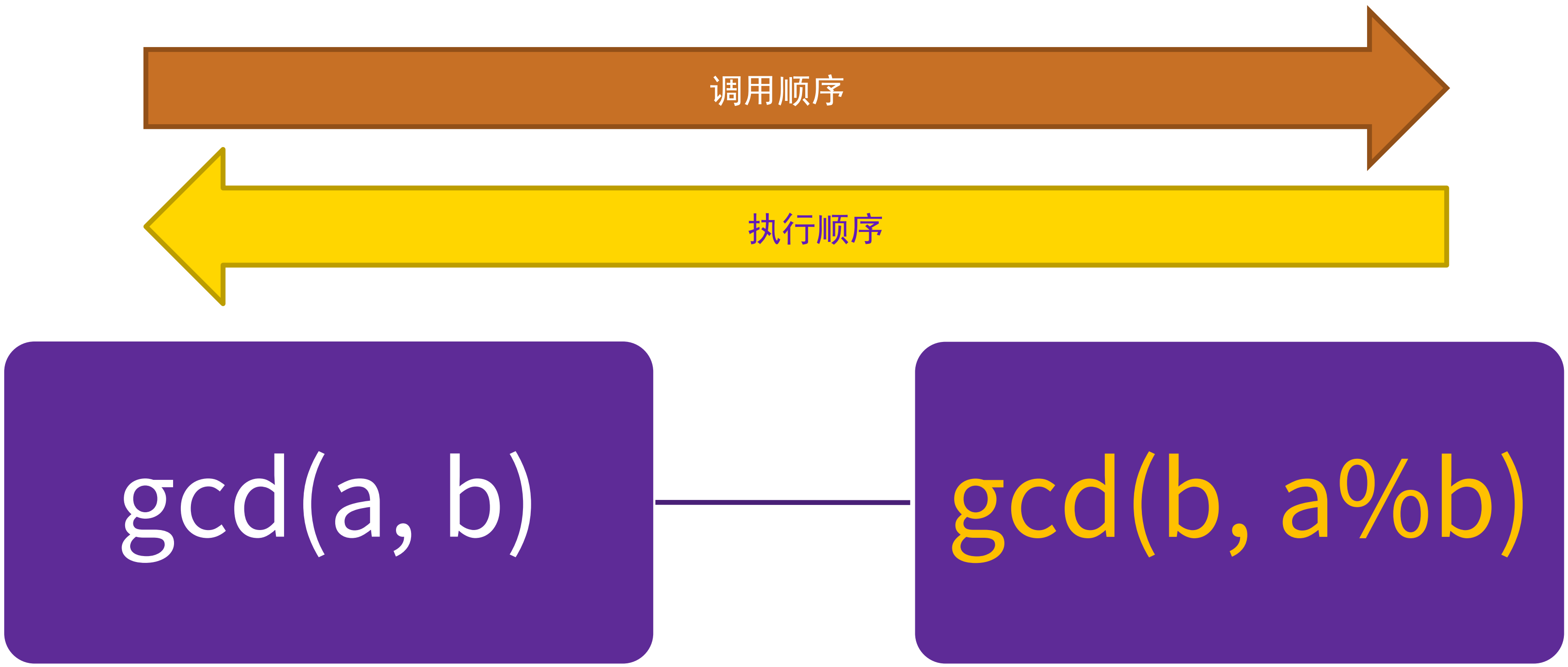
```
55
```

整数的二进制和

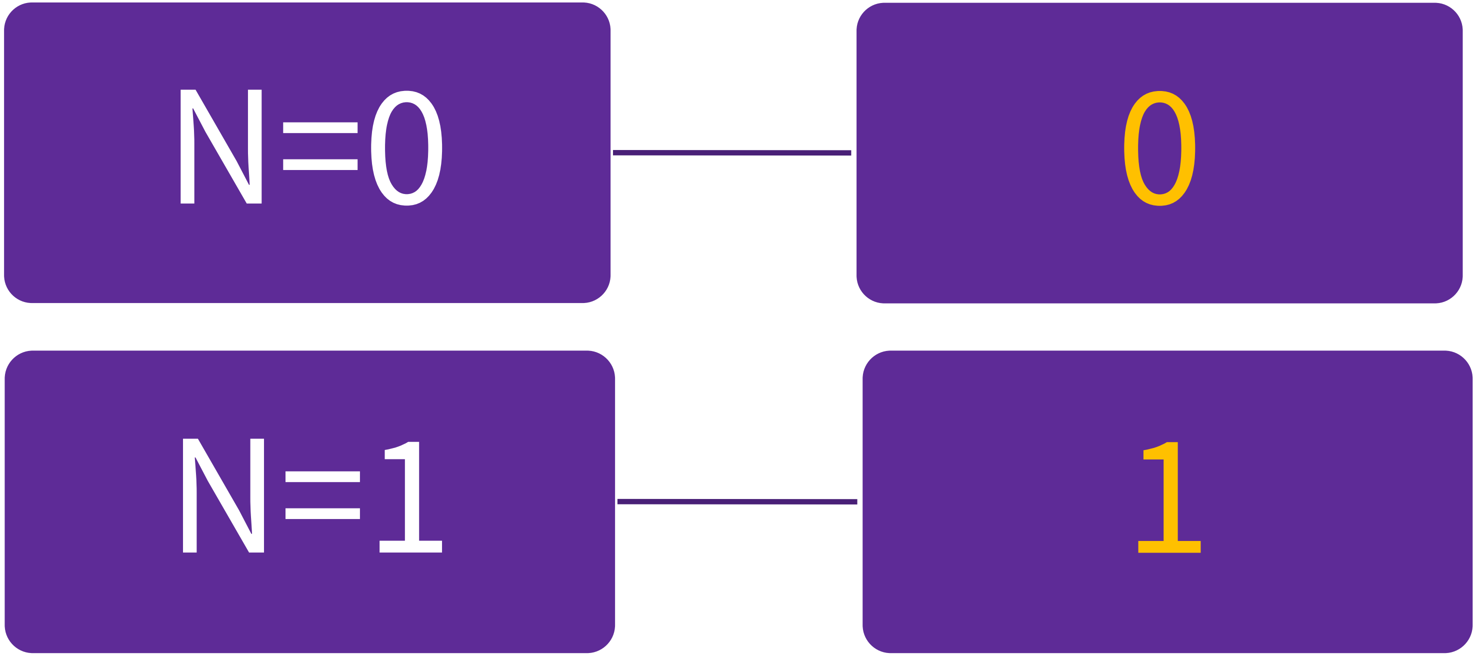
$$(9,6) \; = (6,9\%6) = (6,3) = (3,6\%3) = (3,0) = 3$$

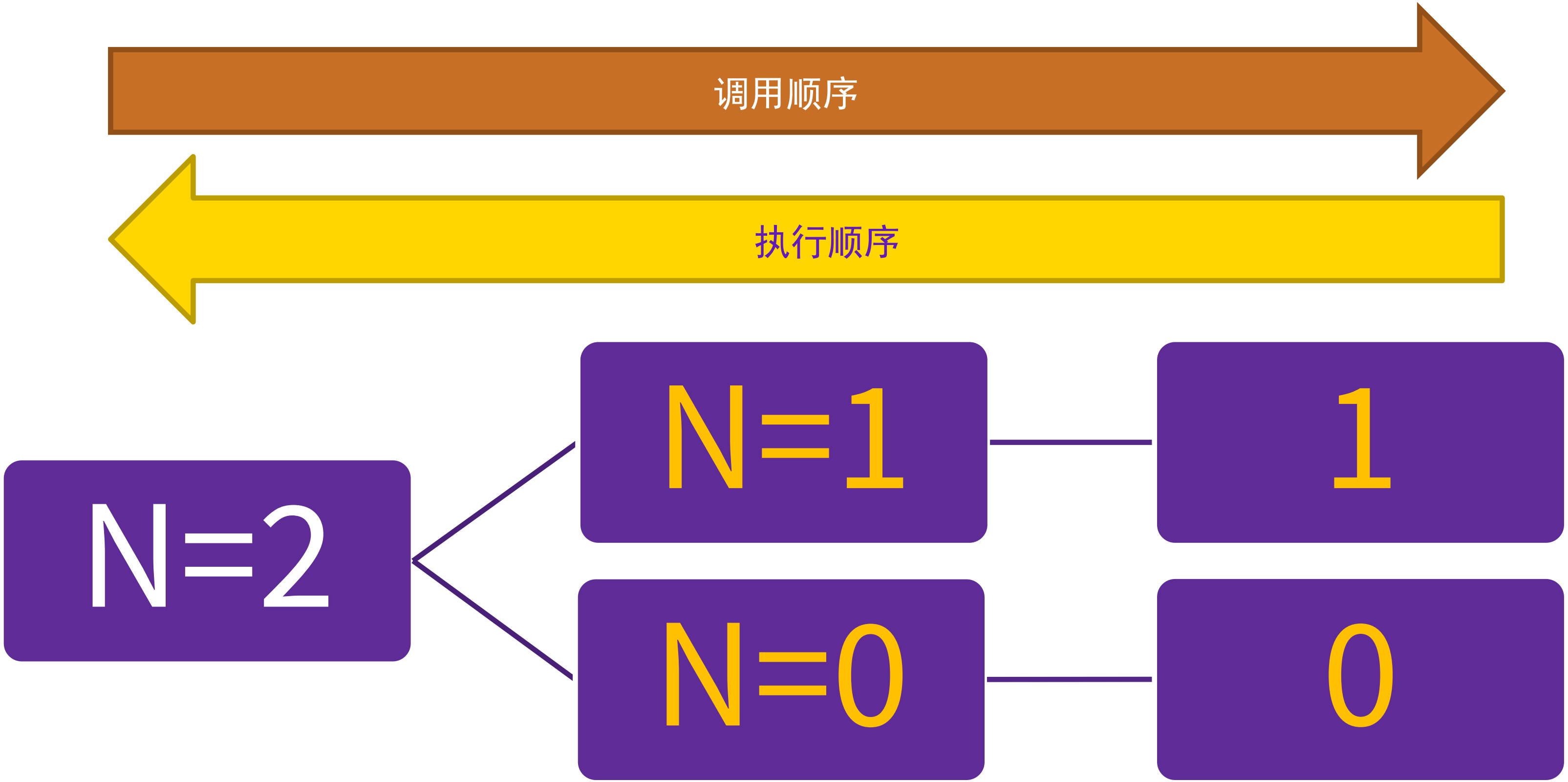


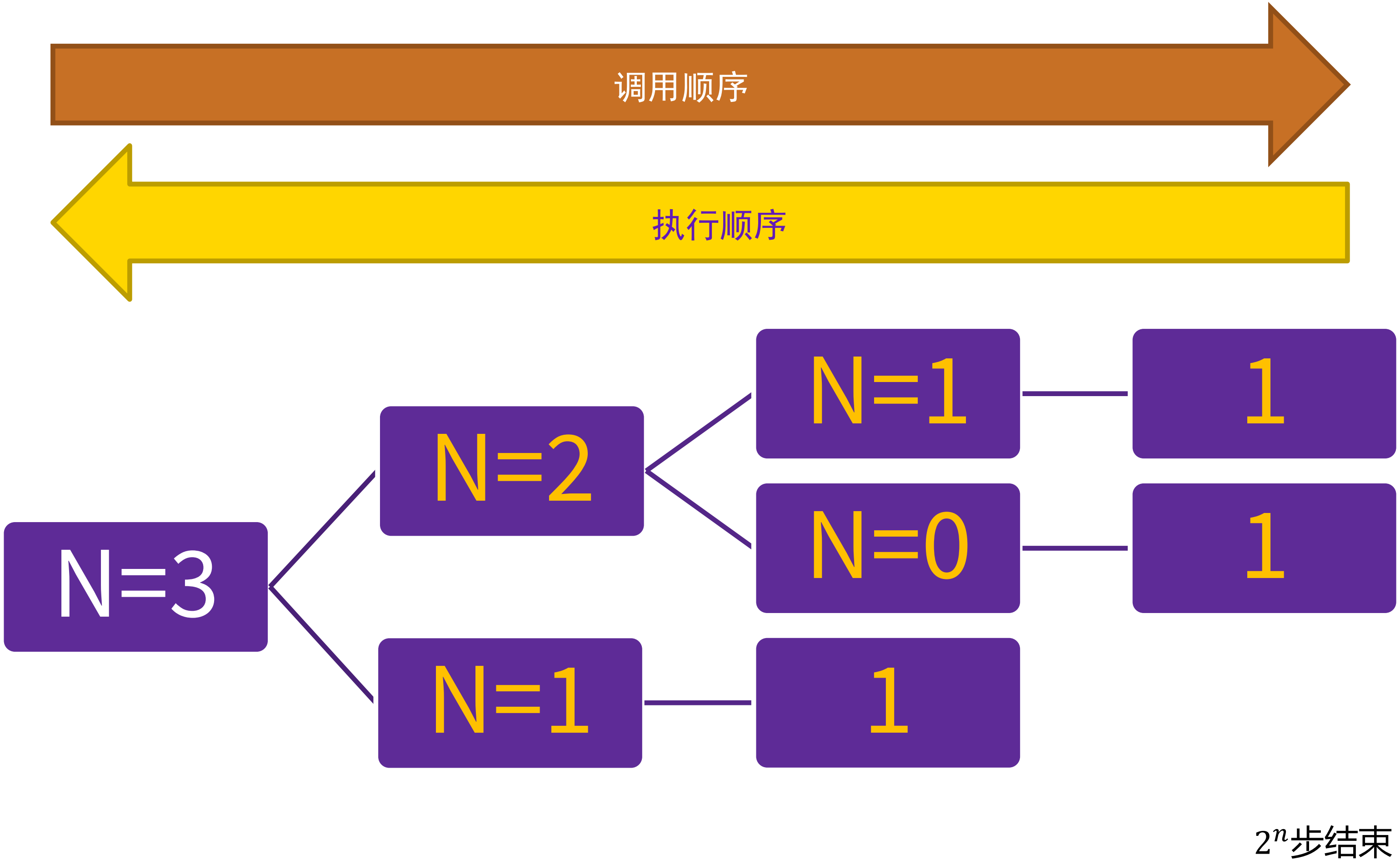
$\log a$步结束



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


N=0
0
N=1
1


调用顺序
执行顺序
N=2
N=1
N=0
1
0



调用顺序
执行顺序
N=3
N=2
N=1
N=1
N=0
1
1
1
2^n步结束

Infinite recursive function

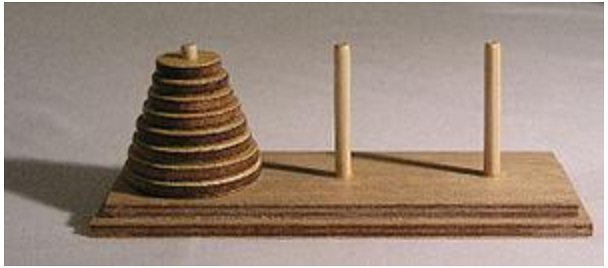
- Recursive functions without base case will lead to errors

```python
def print_test(str1):
    print_test(str1)

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

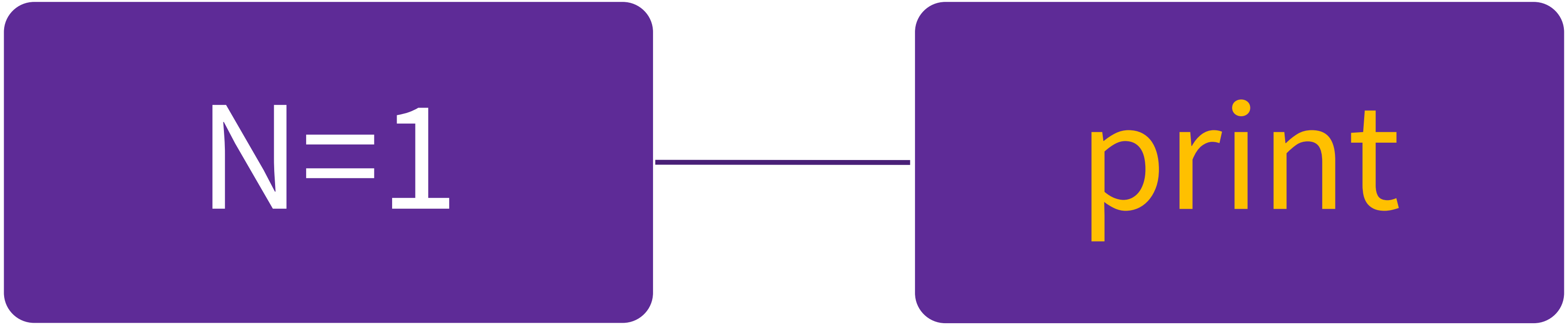
Give a solution to this problem.

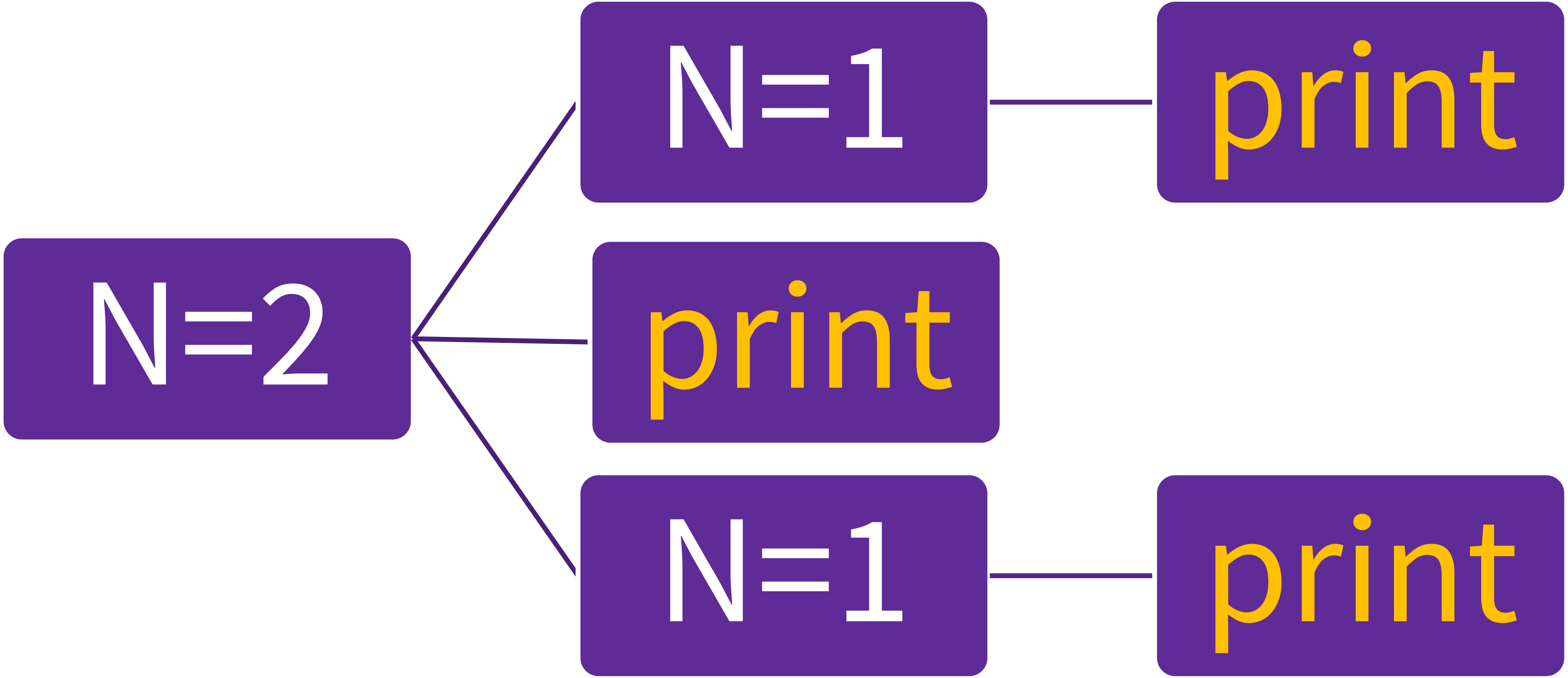
假定，任务是从圆柱1，将圆盘经过圆柱2，全部移动到圆柱3。输出移动的过程

```python
def Hanoi(n, x, y, z):
    if n == 1:
        print(f"{n}: {x}->{z}")
        return

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


print("Hanoi: n = 1")
Hanoi(1, 1, 2, 3)

print("Hanoi: n = 2")
Hanoi(2, 1, 2, 3)

print("Hanoi: n = 3")
Hanoi(3, 1, 2, 3)
```

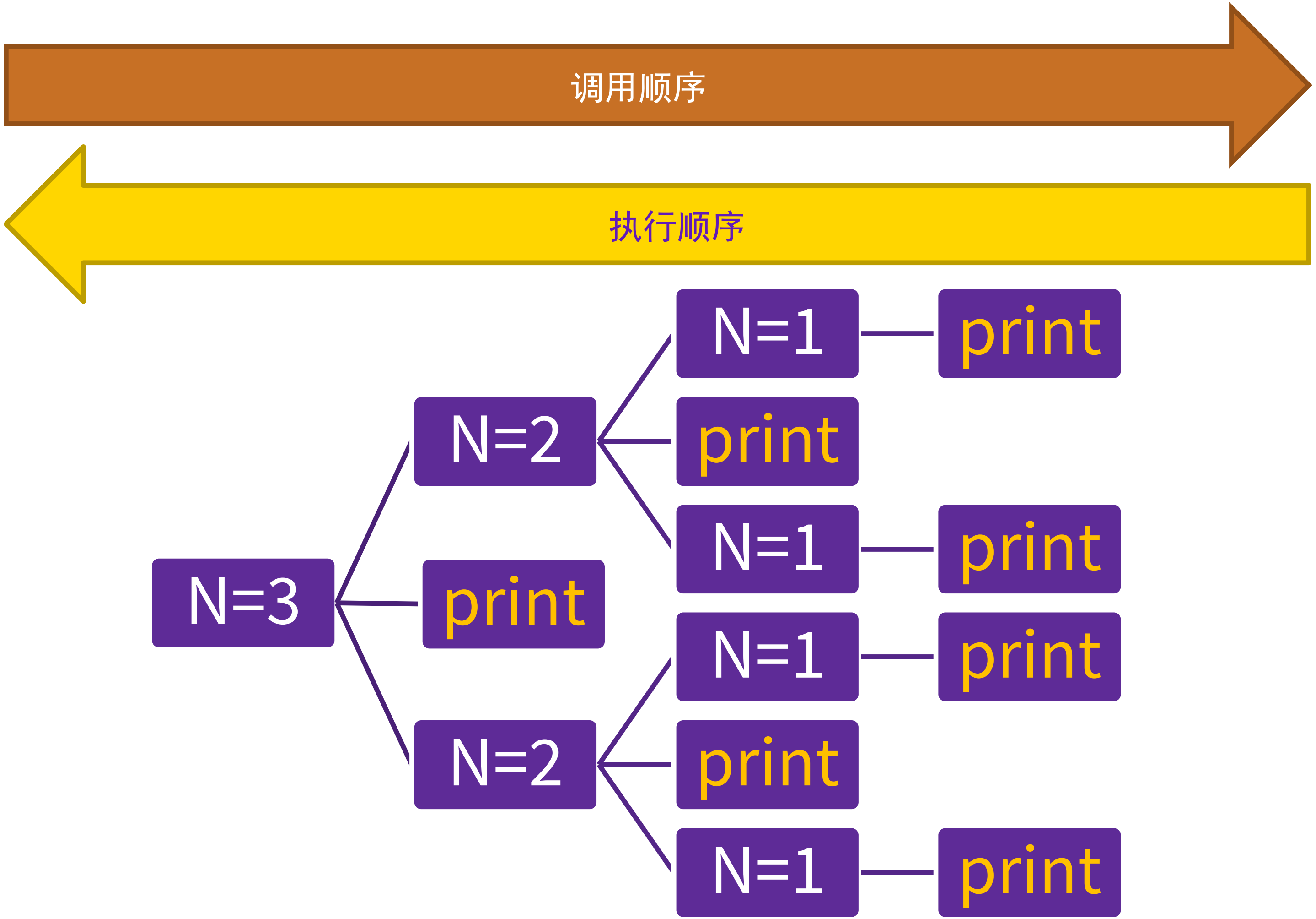
```
Hanoi: n = 1
1: 1->3
Hanoi: n = 2
1: 1->2
2: 1->3
1: 2->3
Hanoi: n = 3
1: 1->3
2: 1->2
1: 3->2
3: 1->3
1: 2->1
2: 2->3
1: 1->3
```

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



N=2
N=1
print
print
N=1
print


调用顺序
执行顺序
N=1
print
N=2
print
N=1
print
N=3
print
N=1
print
N=2
print
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
- 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
- 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

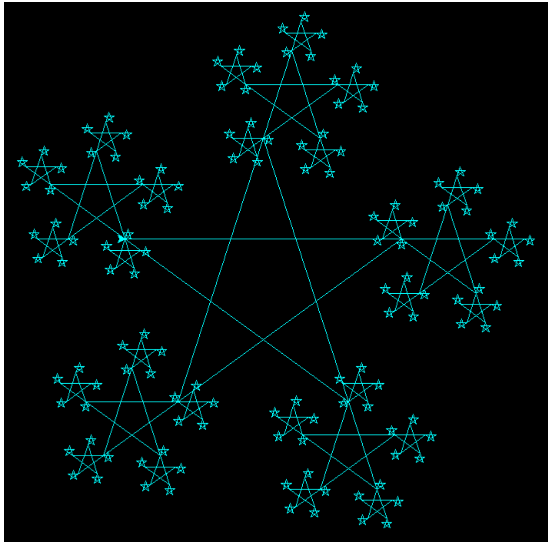
函数：一段具有某种功能的代码。函数执行结束，表示着已经完成了函数所定义的功能

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

```python
import turtle

tur = turtle.Turtle()
tur.speed(6)
tur.getscreen().bgcolor("black")
tur.color("cyan")
tur.penup()
tur.goto((-200, 50))
tur.pendown()


def star(turtle, size):
    if size <= 10:
        return
    else:
        for i in range(5):
            turtle.forward(size)
            star(turtle, size / 3)
            turtle.left(216)


star(tur, 360)
turtle.done()
```



Sort

- Given a list of integers: number= [7, 3, 4, 5, 1, 2, 3]. Rearrange the numbers in ascending order
- Divide the list into two parts in equal sizes: number1, number2
- Sort number1 and number2, respectively
- Merge the sorted number1 and number2 to recover number

```python
def merge(number, left, right):
    pass

def merge_sort(number, left, right):
    if left >= right: return

    merge_sort(number, left, (left+right)//2)
    merge_sort(number, (left+right)//2+1, right)

    merge(number, left, right)


number = [-x for x in range(10)]
merge_sort(number, 0, 9)
print(number)
```

Nested List

- Find the sum of the numbers in a nested list
- [[1, [1], [1]], [2], [3]]

```python
def sum_of_nested(number):
    if len(number) == 0:
        return 0

    if type(number[-1]) != type([]):
        return number[-1] + sum_of_nested(number[0:-1])

    return sum_of_nested(number[-1]) + sum_of_nested(number[0:-1])

number_lst = [[1, [1], [1]], [2], [3], 1, 1, 1]
print(sum_of_nested(number_lst))

number_lst = [[[[[[1]]]]]]
print(sum_of_nested(number_lst))

number_lst = [[[[[[]]]]]]
print(sum_of_nested(number_lst))
```

```
11
1
0
```

a**n

- How to implement a^n, where n is an integer
- $y = a^{n/2}$
- $a = y \times y$

```python
def my_pow(a, n):
    if n==0: return 1

    t = my_pow(a, n//2)

    return t*t if n%2==0 else t*t*a

for i in range(4):
    print(my_pow(2, i))
```

```
1
2
4
8
```

Binary_search

- Given a non-decreasing list of numbers: S=[-3, -4, 1, 3, 5], find whether $x \in S$
- Halving S equally, search the left or the right part according to the comparison of x and mid

```python
def binary_search(number, x, left, right):
    if left > right:
        return None

    mid = (left + right) // 2
    if number[mid] == x:
        return mid

    if number[mid] > x:
        return binary_search(number, x, left, mid - 1)

    return binary_search(number, x, mid + 1, right)


number = [-3, -4, 1, 3, 5]

print(binary_search(number, -6, 0, len(number) - 1))
print(binary_search(number, -3, 0, len(number) - 1))
print(binary_search(number, 1, 0, len(number) - 1))
print(binary_search(number, 0, 0, len(number) - 1))
print(binary_search(number, 5, 0, len(number) - 1))
print(binary_search(number, 7, 0, len(number) - 1))
```

```
None
0
2
None
4
None
```

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: $A_1 \to A_2 \to A_3 \to \cdots \to A_n$
  - Stack diagram: when a function calls another function, the processed can also be modeled by a stack.
  - The functions $A_1, A_2, \ldots, A_n$ 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 $A_{n-1}, \ldots, A_2, A_1$