第 7 讲 · 讲义

递归

课件的每一页，配上讲这一页时说的话。幻灯片是网页原生渲染的，文字可以选中、可以搜索，也可以直接打印。



第 1 页 · Recursion 递归

> 上一讲学会用容器组织记录。本讲把注意力放到层级和分支：资料目录里还有子目录，应该怎样处理全部文件?
>
> 今天重点理解每次调用怎样保存状态、怎样完成较小的同类任务，以及何时应放弃一个无法完成的搜索分支。

How Large Is the Course Archive?

A folder may contain files **and more folders**.



> 子文件夹还能继续嵌套，怎样统计所有文件的大小?

先给学生一个常见任务：整理课程资料，统计整个目录的文件大小。文件夹本身不能当作一个文件大小直接相加，还要继续查看里面。

图是任务场景，不承担精确目录结构。后面会给出固定的五个文件和嵌套列表表示，让程序、图与结果逐项对应。

Predict: Open and Close



扫码作答

https://taohuang.info/cs1602/zh/quiz/7/

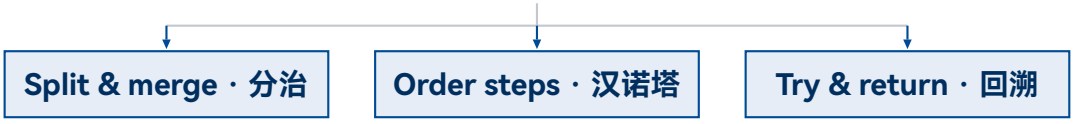
第 3 页 · Predict: Open and Close

> 第一个问题把进入和退出文件夹对应到 open 与 close。请学生先独立预测，不马上公布答案。
>
> 题目中的函数会再次调用自身，暂时不要求学生掌握递归语法。讲清楚调用者怎样等待返回之后，在专门的回顾页重新解释同一份代码。

How This Lecture Fits Together

One idea: solve **smaller tasks of the same kind**.

1. **A folder contains smaller folders.**
   统计子目录，仍然是在做“统计目录大小”这件事。
2. **A call solves a smaller task, then returns.**
   先算出子目录的小计，再回到父目录继续相加。
3. **Design smaller tasks for different goals.**



先理解为什么能递归，再看调用怎样完成，最后学习怎样设计小任务。

第 4 页 · How This Lecture Fits Together

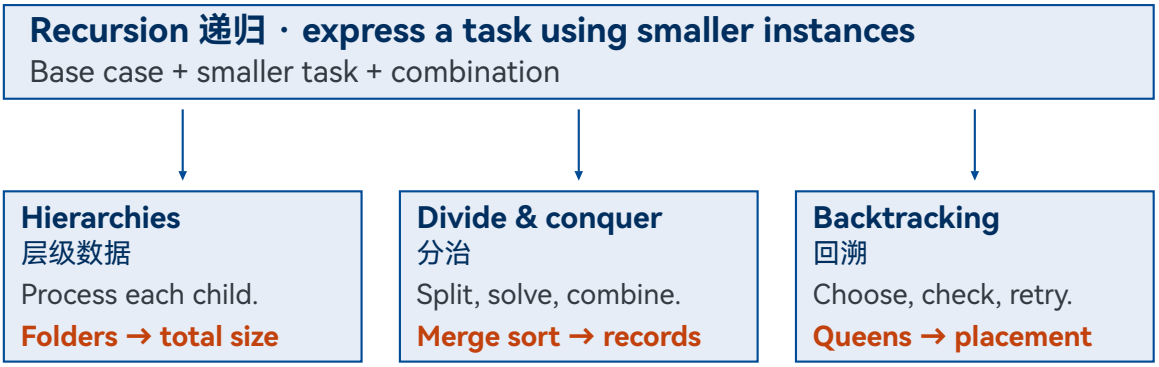
先接住开场的目录例子：统计一个目录时，遇到子目录，仍然要做统计目录大小这件事，只是范围变小了。这解释为什么要考虑递归，而不是先背函数调用自己的定义。

接着问程序怎样完成它：先算完子目录的小计，返回父目录继续相加。为此要看懂调用、等待、返回与每次调用保留的状态；阶乘动画把这个过程完整展开，目录程序再完成开场任务。正确之后还要检查有没有重复计算。

最后才进入设计：如果任务没有现成的目录层级，怎样主动设计小任务？排序是拆开后合并，汉诺塔是按依赖安排先后，皇后是尝试选择、遇到冲突就返回换路。底部三条分支是不同目标下的设计思路，不是依次执行的三个步骤，也不是三类互斥算法。下一页再说明递归这种表达方式与设计策略的关系。

How Recursion Connects the Strategies

Recursion expresses tasks; **the strategy chooses the subproblems**.



Hanoi: arrange smaller tasks in a required order.

递归是表达方式；分治与回溯是设计策略，也能用循环和显式栈实现。

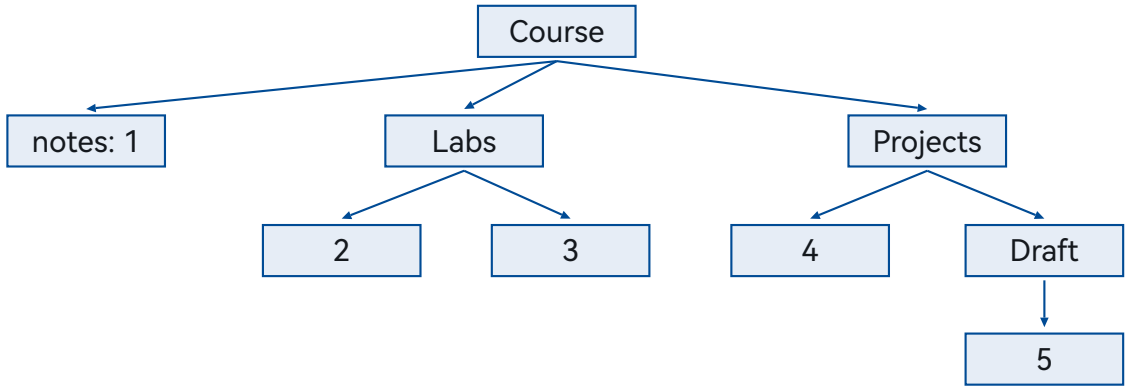
第 5 页 · How Recursion Connects the Strategies

先把三个词放在不同层面。递归是一种程序表达方式：用更小的同类任务完成当前任务，并保留基准和返回后的工作。分治是设计子问题与组合结果的策略，回溯是扩展部分解并在冲突时返回换路的搜索策略。它们不是三个互斥的选项，也不是必须先分治再回溯的流水线。

箭头表示本讲用递归来表达这些任务，不表示分治和回溯必须递归实现。左侧目录统计利用数据自身的层级，中间归并排序显式拆分并合并，右侧皇后根据冲突选择继续或返回。汉诺塔则强调较小任务的先后次序。此时不要求学生掌握实现，只建立稍后能回来的地图。

One Loop Does Not Reach Every File

Same archive: **five files**, sizes measured in MB.



> 看到文件就处理；看到文件夹，就继续处理它的内容。

> 固定输入是 notes 文件 1 MB，Labs 中两个文件为 2、3 MB，Projects 中有 4 MB 文件和 Draft 子目录中的 5 MB 文件。
>
> 外层只有三个项目，其中两个是目录。再写一层循环也不能预先覆盖任意层数。每次进入子目录，仍然要完成统计这一层全部文件大小的相同任务。

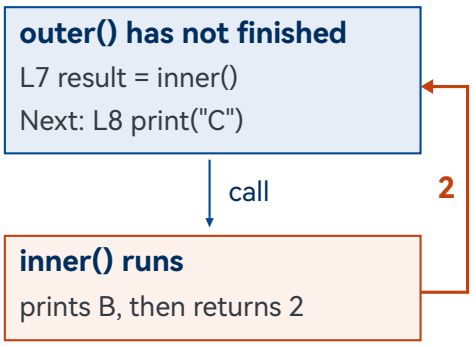
I · Where Does outer Continue?

Calling inner() leaves **unfinished work** in outer().

I · Calls 调用 · pause at L7

```
 1 def inner() -> int:
 2     print("B")
 3     return 2
 4
 5 def outer() -> int:
 6     print("A")
→7     result = inner()
 8     print("C")
 9     return result + 1
10
11 print(outer())
```

Call, then continue 调用后继续



调用 inner 时，outer 要记住自己停在哪一行。

7

第 7 页 · I · Where Does outer Continue?

前面的目录任务需要处理完子目录后回到父目录。先用最普通的 outer/inner 观察进入一个函数后怎样回到调用者，暂不引入调用栈，也不要求记 push 或 pop。两个函数都已定义，主程序调用 outer，outer 打印 A 后走到第 7 行。请学生先找出调用 inner 的位置，再预测 inner 返回后会打印什么。

沿蓝色箭头进入 inner：先打印 B，再返回 2。沿唯一终点带箭头的橙色折线回到第 7 行，先完成 result 的赋值，再执行第 8 行打印 C；outer 不会重新从第 6 行开始。为了做到这一点，程序必须保留 outer 尚未完成的操作。右图概括这一段执行，不是同时运行两个函数。

Why Keep a Call Stack 调用栈?

Each call keeps its **local values and place to continue** in a frame.

Frame 调用帧 · one call's record

Push 压入

Call inner(): add its frame.

Wait 等待

outer stays at L7 until inner returns.

Pop 弹出

Return 2: remove inner's frame; resume outer at L7.

Most recent call runs ↑

inner() · running
L2 print("B")

outer() · waiting
L7 result = inner()
result is not assigned yet

Program · waiting
L11 print(outer())

这些尚未完成的调用记录组成调用栈；最后进入的调用先返回。

8

第 8 页 · Why Keep a Call Stack 调用栈?

承接上一页需要记住暂停位置的问题，把每次调用的局部值和继续执行的位置称为调用记录，也叫调用帧。多个尚未完成的调用记录组成调用栈。右图仍是上一页同一个例子：主程序等 outer，outer 在第 7 行等 inner，只有最上面的 inner 在运行。

调用 inner 时新增一条记录，是压入 push；outer 保留未完成的位置而暂停，是等待状态，不是另一个栈操作。inner 返回 2 时移除其记录，是弹出 pop，随后 outer 从第 7 行完成赋值。result 在赋值前还不存在，因此右图明确写出尚未赋值。图只表示逻辑调用关系，不是 CPython 物理内存布局。下一页从主程序开始逐步播放完整过程。

Follow the Calls: outer → inner

Begin at the top-level call.

Code 程序 · 01 / 10

```
 1 def inner() -> int:
 2     print("B")
 3     return 2
 4
 5 def outer() -> int:
 6     print("A")
 7     result = inner()
 8     print("C")
 9     return result + 1
10
11 print(outer())
```

Call stack 调用栈 · top ↑

Program 主程序
Running L11

Printed 已输出 –

箭头：当前语句；橙框：运行；蓝框：等待。

> 函数定义已执行完，从主程序第 11 行开始。箭头表示即将执行或恢复执行的语句。
>
> 栈图是调用与等待的逻辑模型，不是 CPython 物理内存布局。函数定义过程已略去。逐页点击，一页推进一个执行状态。

Follow the Calls: outer → inner

Push outer(): the caller waits.

Code 程序 · 02 / 10

```
 1 def inner() -> int:
 2     print("B")
 3     return 2
 4
 5 def outer() -> int:
→6     print("A")
 7     result = inner()
 8     print("C")
 9     return result + 1
10
11 print(outer())
```

Call stack 调用栈 · top ↑

outer()
Running L6

Program 主程序
Wait at L11

Printed 已输出 –

箭头：当前语句；橙框：运行；蓝框：等待。

> 新调用 outer() 压入栈顶，开始执行第 6 行。下面的调用者停在自己的调用行，保留参数和尚未完成的操作。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Calls: outer → inner

Call inner; keep the assignment unfinished.

Code 程序 · 03 / 10

```
1  def inner() -> int:
2      print("B")
3      return 2
4
5  def outer() -> int:
6      print("A")
7      result = inner()
8      print("C")
9      return result + 1
10
11 print(outer())
```

Call stack 调用栈 · top ↑

outer()
Running L7

Program 主程序
Wait at L11

Printed 已输出 A

箭头：当前语句；橙框：运行；蓝框：等待。

> outer 在第 7 行等待 inner 返回。result 此时尚未绑定，不能提前画成 2。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Calls: outer → inner

Push inner(): the caller waits.

Code 程序 · 04 / 10

```
 1 def inner() -> int:
→2     print("B")
 3     return 2
 4
 5 def outer() -> int:
 6     print("A")
 7     result = inner()
 8     print("C")
 9     return result + 1
10
11 print(outer())
```

Call stack 调用栈 · top ↑

inner()
Running L2

outer()
Wait at L7

Program 主程序
Wait at L11

Printed 已输出 A

箭头：当前语句；橙框：运行；蓝框：等待。

> 新调用 inner() 压入栈顶，开始执行第 2 行。下面的调用者停在自己的调用行，保留参数和尚未完成的操作。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Calls: outer → inner

Return from the active frame.

Code 程序 · 05 / 10

```
 1 def inner() -> int:
 2     print("B")
→3     return 2
 4
 5 def outer() -> int:
 6     print("A")
 7     result = inner()
 8     print("C")
 9     return result + 1
10
11 print(outer())
```

Call stack 调用栈 · top ↑

inner()
Running L3

outer()
Wait at L7

Program 主程序
Wait at L11

Printed 已输出 A · B

箭头：当前语句；橙框：运行；蓝框：等待。

> 即将执行 inner() 的 return，使用这一层自己的局部变量。下一页才展示它返回后弹出的状态。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Calls: outer → inner

inner() → 2; resume outer().

Code 程序 · 06 / 10

```
 1 def inner() -> int:
 2     print("B")
 3     return 2
 4
 5 def outer() -> int:
 6     print("A")
→7     result = inner()
 8     print("C")
 9     return result + 1
10
11 print(outer())
```

Call stack 调用栈 · top ↑

outer()
Received 2 · resume L7

Program 主程序
Wait at L11

Printed 已输出 A · B

返回值先交给调用者，再完成原先暂停的语句。

> inner() 返回 2，该调用帧已弹出。箭头回到调用者第 7 行：先使用刚收到的值完成这一行，再继续下一行。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Calls: outer → inner

Execute this print, then continue.

Code 程序 · 07 / 10

```
 1 def inner() -> int:
 2     print("B")
 3     return 2
 4
 5 def outer() -> int:
 6     print("A")
 7     result = inner()
→8     print("C")
 9     return result + 1
10
11 print(outer())
```

Call stack 调用栈 · top ↑

outer() result = 2
Running L8

Program 主程序
Wait at L11

Printed 已输出 `A · B`

箭头：当前语句；橙框：运行；蓝框：等待。

> 箭头指向即将执行的第 8 行 print。右下方只显示已经真正打印的内容，下一页才出现本行的新输出。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Calls: outer → inner

Return from the active frame.

Code 程序 · 08 / 10

```
 1 def inner() -> int:
 2     print("B")
 3     return 2
 4
 5 def outer() -> int:
 6     print("A")
 7     result = inner()
 8     print("C")
→9     return result + 1
10
11 print(outer())
```

Call stack 调用栈 · top ↑

outer() result = 2
Running L9

Program 主程序
Wait at L11

Printed 已输出 `A · B · C`

箭头：当前语句；橙框：运行；蓝框：等待。

16

第 16 页 · Follow the Calls: outer → inner

> 即将执行 outer() 的 return，使用这一层自己的局部变量。下一页才展示它返回后弹出的状态。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Calls: outer → inner

outer() → 3; resume Program 主程序.

Code 程序 · 09 / 10

```
 1 def inner() -> int:
 2     print("B")
 3     return 2
 4
 5 def outer() -> int:
 6     print("A")
 7     result = inner()
 8     print("C")
 9     return result + 1
10
11 print(outer())
```

Call stack 调用栈 · top ↑

Program 主程序
Received 3 · resume L11

Printed 已输出 `A · B · C`

返回值先交给调用者，再完成原先暂停的语句。

> outer() 返回 3，该调用帧已弹出。箭头回到调用者第 11 行：先使用刚收到的值完成这一行，再继续下一行。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Calls: outer → inner

Finished: the stack is empty.

Code 程序 · 10 / 10

```python
def inner() -> int:
    print("B")
    return 2

def outer() -> int:
    print("A")
    result = inner()
    print("C")
    return result + 1

print(outer())
```

Call stack 调用栈 · top ↑

Printed 已输出 A · B · C · 3

最后压入的调用先返回；每个调用保留自己的状态。

> 程序结束，所有调用均已返回，输出完整保留。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Recursive Factorial

For n ≥ 1: **n! = n × (n − 1)!**; 0! = 1.

code

```python
def factorial(n: int) -> int:
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(3))
```

Output

```
6
```

函数调用自身就是递归；基准直接返回，其余情况等待子任务再组合。

函数在执行中再次调用自身，称为递归。代码第4行把较小的阶乘交给同一个函数。阶乘把一到 n 连乘；三阶乘是三乘二乘一。为了算 3!，先求 2!，再乘三。输入约定为非负整数，0! 直接定义为一。

n 为零时返回一，其他调用先得到 factorial(n - 1) 的值，再乘当前 n。每次减一最终到零，类型提示不会自动排除负数。下一组动画会逐层展开同一份代码。

Follow the Stack: factorial(3)

Begin at the top-level call.

Code 程序 · 01 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
→6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

Program 主程序
Running L6

Printed 已输出 –

箭头：当前语句；橙框：运行；蓝框：等待。

> 函数定义已执行完，从主程序第 6 行开始。箭头表示即将执行或恢复执行的语句。
>
> 栈图是调用与等待的逻辑模型，不是 CPython 物理内存布局。函数定义过程已略去。逐页点击，一页推进一个执行状态。

Follow the Stack: factorial(3)

Push factorial(3): the caller waits.

Code 程序 · 02 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

factorial(3) n = 3
Running L2

Program 主程序
Wait at L6

Printed 已输出 –

箭头：当前语句；橙框：运行；蓝框：等待。

> 新调用 factorial(3) 压入栈顶，开始执行第 2 行。下面的调用者停在自己的调用行，保留参数和尚未完成的操作。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Stack: factorial(3)

This multiplication needs a smaller result.

Code 程序 · 03 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

factorial(3) n = 3
Running L4

Program 主程序
Wait at L6

Printed 已输出 –

箭头：当前语句；橙框：运行；蓝框：等待。

> factorial(3) 即将调用更小一层；当前 n 为 3，乘法要等待递归调用返回。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Stack: factorial(3)

Push factorial(2): the caller waits.

Code 程序 · 04 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

factorial(2) Running L2	n = 2
factorial(3) Wait at L4 · 3 × ?	n = 3
Program 主程序 Wait at L6	

Printed 已输出 –

箭头：当前语句；橙框：运行；蓝框：等待。

新调用 factorial(2) 压入栈顶，开始执行第 2 行。下面的调用者停在自己的调用行，保留参数和尚未完成的操作。

保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Stack: factorial(3)

This multiplication needs a smaller result.

Code 程序 · 05 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

factorial(2) n = 2
Running L4

factorial(3) n = 3
Wait at L4 · 3 × ?

Program 主程序
Wait at L6

Printed 已输出 –

箭头：当前语句；橙框：运行；蓝框：等待。

> factorial(2) 即将调用更小一层；当前 n 为 2，乘法要等待递归调用返回。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Stack: factorial(3)

Push factorial(1): the caller waits.

Code 程序 · 06 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

factorial(1) Running L2	n = 1
factorial(2) Wait at L4 · 2 × ?	n = 2
factorial(3) Wait at L4 · 3 × ?	n = 3
Program 主程序 Wait at L6	

Printed 已输出 –

箭头：当前语句；橙框：运行；蓝框：等待。

25

第 25 页 · Follow the Stack: factorial(3)

> 新调用 factorial(1) 压入栈顶，开始执行第 2 行。下面的调用者停在自己的调用行，保留参数和尚未完成的操作。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Stack: factorial(3)

This multiplication needs a smaller result.

Code 程序 · 07 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

factorial(1) Running L4	n = 1
factorial(2) Wait at L4 · 2 × ?	n = 2
factorial(3) Wait at L4 · 3 × ?	n = 3
Program 主程序 Wait at L6	

Printed 已输出 –

箭头：当前语句；橙框：运行；蓝框：等待。

> factorial(1) 即将调用更小一层；当前 n 为 1，乘法要等待递归调用返回。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Stack: factorial(3)

Push factorial(0): the caller waits.

Code 程序 · 08 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

factorial(0) Running L2	n = 0
factorial(1) Wait at L4 · 1 × ?	n = 1
factorial(2) Wait at L4 · 2 × ?	n = 2
factorial(3) Wait at L4 · 3 × ?	n = 3
Program 主程序 Wait at L6	

Printed 已输出 –

箭头：当前语句；橙框：运行；蓝框：等待。

新调用 factorial(0) 压入栈顶，开始执行第 2 行。下面的调用者停在自己的调用行，保留参数和尚未完成的操作。

保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Stack: factorial(3)

Return from the active frame.

Code 程序 · 09 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

factorial(0) Running L3	n = 0
factorial(1) Wait at L4 · 1 × ?	n = 1
factorial(2) Wait at L4 · 2 × ?	n = 2
factorial(3) Wait at L4 · 3 × ?	n = 3
Program 主程序 Wait at L6	

Printed 已输出 –

箭头：当前语句；橙框：运行；蓝框：等待。

> 即将执行 factorial(0) 的 return，使用这一层自己的局部变量。下一页才展示它返回后弹出的状态。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Stack: factorial(3)

factorial(0) → 1; resume factorial(1).

Code 程序 · 10 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

factorial(1) Next: 1 × 1 = 1	n = 1
factorial(2) Wait at L4 · 2 × ?	n = 2
factorial(3) Wait at L4 · 3 × ?	n = 3
Program 主程序 Wait at L6	

Printed 已输出 –

返回值先交给调用者，再完成原先暂停的语句。

29

第 29 页 · Follow the Stack: factorial(3)

> factorial(0) 返回 1，该调用帧已弹出。箭头回到调用者第 4 行：先使用刚收到的值完成这一行，再继续下一行。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Stack: factorial(3)

factorial(1) → 1; resume factorial(2).

Code 程序 · 11 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

factorial(2) n = 2
Next: 2 × 1 = 2

factorial(3) n = 3
Wait at L4 · 3 × ?

Program 主程序
Wait at L6

Printed 已输出 –

返回值先交给调用者，再完成原先暂停的语句。

> factorial(1) 返回 1，该调用帧已弹出。箭头回到调用者第 4 行：先使用刚收到的值完成这一行，再继续下一行。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Stack: factorial(3)

factorial(2) → 2; resume factorial(3).

Code 程序 · 12 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

factorial(3) n = 3
Next: 3 × 2 = 6

Program 主程序
Wait at L6

Printed 已输出 –

返回值先交给调用者，再完成原先暂停的语句。

> factorial(2) 返回 2，该调用帧已弹出。箭头回到调用者第 4 行：先使用刚收到的值完成这一行，再继续下一行。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Stack: factorial(3)

factorial(3) → 6; resume Program 主程序.

Code 程序 · 13 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
→6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

Program 主程序
Received 6 · resume L6

Printed 已输出 –

返回值先交给调用者，再完成原先暂停的语句。

> factorial(3) 返回 6，该调用帧已弹出。箭头回到调用者第 6 行：先使用刚收到的值完成这一行，再继续下一行。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Stack: factorial(3)

Execute this print, then continue.

Code 程序 · 14 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 answer = factorial(3)
→7 print(answer)
```

Call stack 调用栈 · top ↑

Program 主程序 answer = 6
Running L7

Printed 已输出 –

箭头：当前语句；橙框：运行；蓝框：等待。

33

第 33 页 · Follow the Stack: factorial(3)

> 箭头指向即将执行的第 7 行 print。右下方只显示已经真正打印的内容，下一页才出现本行的新输出。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Follow the Stack: factorial(3)

Finished: the stack is empty.

Code 程序 · 15 / 15

```
1 def factorial(n: int) -> int:
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
5
6 answer = factorial(3)
7 print(answer)
```

Call stack 调用栈 · top ↑

Printed 已输出 6

最后压入的调用先返回；每个调用保留自己的状态。

> 程序结束，所有调用均已返回，输出完整保留。
>
> 保持代码和调用帧位置不动，请学生先预测下一页栈顶、执行行或输出的变化。

Opening Quiz Review: Close the Inner Folder First



The paused calls still have their own **level**.

code

```python
def inspect(level: int) -> None:
    print("open", level)
    if level > 0:
        inspect(level - 1)
    print("close", level)

inspect(2)
```

Output

```
open 2
open 1
open 0
close 0
close 1
close 2
```

开场答案：open 2、1、0，然后 close 0、1、2。

回到开场同一份代码。先连续进入 2、1、0；零层不再调用，先打印 close 0。返回后上一层从递归调用后面继续，因此接着输出 close 1，最后才是 close 2。

这与文件管理器进入子目录再回到父目录的顺序对应。三个 level 属于不同调用，不会被最内层覆盖。

A Base Case Must Be Correct and Reachable

Check 检查	Factorial example 阶乘	What to ask 问题
Task 任务	Return n! for n ≥ 0	What must this call accomplish?
Base 基准	factorial(0) → 1	Is this direct result correct?
Progress 进展	3 → 2 → 1 → 0	Does a bounded measure decrease?
Combination 组合	n × factorial(n−1)	Does this complete the current task?

> 写出 if 还不够，要把基准与参数更新一起检查。

> 递归设计包括任务、基准、子任务、组合和进展。把 factorial(n - 1) 作为已完成的子任务，乘 n 能完成当前任务；还要分别核对零的答案正确、非负参数必定到达零。
>
> 并不是所有参数都必须减少，稍后的皇后问题中已放数量增加，剩余行数减少。

Predict: Will the Base Case Be Reached?



扫码作答

https://taohuang.info/cs1602/zh/quiz/7/

第 37 页 · Predict: Will the Base Case Be Reached?

> 第二题从 down(3) 开始，每次减 2。先让学生判断会返回哪个数或出现什么错误，不在投影页提前展示参数路径。
>
> 公布之后强调函数的输入范围、更新规则和基准必须配套。

Quiz Review: Smaller Can Still Miss Zero

There is a base case, but this input **never reaches it**.

down(n − 2): start at 3



Base: n == 0

3 → 1 skips 0; every later value is odd.

数值变小，还要检查能否到达基准。

本例实际异常：RecursionError。先检查可达性，再讨论深度。

38

第 38 页 · Quiz Review: Smaller Can Still Miss Zero

参数是 3、1、负 1、负 3，始终是奇数。程序不会返回负 1，也不会到达零，最终触发实际运行记录中的 RecursionError。

另一种错误是没有基准，或再次传入完全相同的参数。改成 n 小于等于零是否合理，还要根据函数任务判断，不能机械修改。

Termination, Depth and Saved Work

Check **termination**, **maximum depth**, and saved state.

Version 写法	State kept 状态	Practical choice 使用判断
Recursive factorial	One waiting frame per call	A long valid chain can be too deep
Loop factorial	Running product and loop index	Direct for sequential accumulation
Nested traversal	Unfinished levels or branches	Use calls or an explicit stack

能终止，也可能调用过深；循环适合顺次累计，层级任务仍要保存状态。

阶乘用循环和递归都会做相近数量的乘法，递归另需保留等待状态。前一题 down(3) 到不了基准；另一类程序虽然会终止，也可能在到达基准前超过当前解释器的调用深度限制。这两个判断要分开。

真实运行中的 countdown(5000) 每次减一能到零，但会触发 RecursionError；深度限制不是恰好第一千次用户调用。顺次累计通常用循环直接，层级与分支则需要调用栈或显式状态。参考区用同一目录演示显式栈。

II · Data and Repeated Subproblems

Use recursion on data; then ask **how much work repeats**.

Carry forward 已掌握

Each call keeps its state and returns a result.

Next decision 接着思考

Follow nested data; check whether the same task repeats.

Use it for 用途: Archive size; fewer repeated calculations.

> 路线第二站：刚才已经知道调用怎样保存状态、怎样终止。现在把这个机制放回开场资料目录，处理未知层数的嵌套数据。随后用一个 Fibonacci 反例说明返回结果正确不代表没有重复工作。用途是统计层级资料，并避免重复求解同一子问题。

Total Size of a Nested Archive

A loop handles **this level**; recursion handles **the next**.

code

```python
def deep_sum(items: list) -> int:
    total = 0
    for x in items:
        if isinstance(x, list):
            total += deep_sum(x)
        else:
            total += x
    return total

archive = [1, [2, 3], [4, [5]]]
print(deep_sum(archive))
```

Output

```
15
```

isinstance(x, list) 判断是否为列表；子目录先求小计，再加入本层。

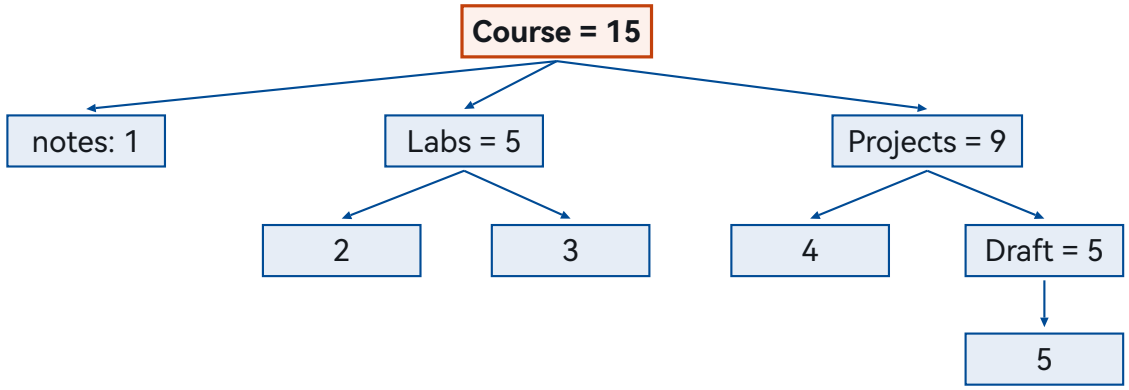
41

整数代表文件大小，列表代表文件夹内容。isinstance(x, list) 判断当前项是不是列表，空列表也满足。输入是有限、没有循环引用的嵌套列表，叶子均为整数；这是教学数据表示，不是磁盘 API。

每层有独立 total。整数直接相加，子列表先递归得到小计，再加入本层。外层输入与开场五个文件对应，总大小十五 MB。

Archive Review: Every File Is Counted Once

The opening task is complete: **15 MB**.



空文件夹返回 0；外层与内层各自累计。

> Labs 返回 5，Draft 返回 5，Projects 返回 9，最外层得到 1 加 5 加 9 等于 15。
>
> 空文件夹没有项目，循环零次，total 保持零返回，不必专门写一个 if。请学生解释空文件夹贡献零的原因，并指出各层 total 分别属于谁。

Fibonacci: Two Calls for One Result

Standard indexing: **F(0) = 0, F(1) = 1**.

code

```python
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)

print([fib(n) for n in range(8)])
```

Output

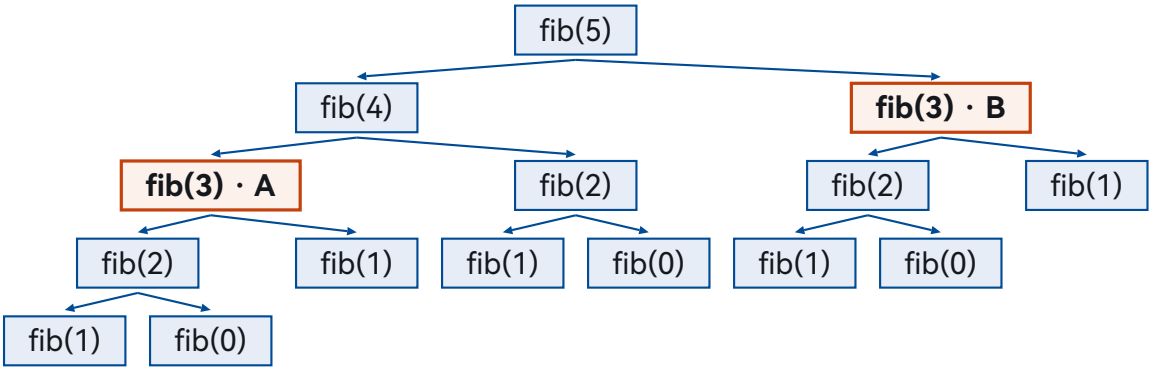
```
[0, 1, 1, 2, 3, 5, 8, 13]
```

返回值正确，还要检查有没有重复做相同任务。

> 代码保留标准 Fibonacci 初值，非基准项需要先得到前两项，再相加。实际前八项是 0、1、1、2、3、5、8、13。
>
> 这里要区别返回值与计算过程。同一个参数是否会沿不同路径重新计算，要看实际调用树。

The Same Subproblem Is Solved Again

Find repeated **fib(3)**: 15 calls for fib(5).



15 calls; fib(3) · A and fib(3) · B are separate calls.

保存结果或循环递推可以避免重算；返回值正确仍要检查重复工作。

fib(5) 的两条路径都需要 fib(3)，每个节点是一次实际调用，含基准共有十五次，最终返回五。请学生指出哪两棵子树在重复完成同一个任务。

函数名和参数相同不会自动复用结果。保存已经得到的值，或改成从底向上的循环递推，可以避免重算；仅把代码换个写法不保证更快。这里只讨论重复工作，不再引入台阶编号或另一份完整实现。后续选择递归或循环时，还要分别检查存储与深度。

III · Divide and Conquer 分治

Design the pieces; **combine what the task needs**.

Carry forward 已掌握

A smaller result can be saved and reused.

Next decision 接着思考

Choose one half, or solve both halves and combine.

Use it for 用途: Binary search finds an item; merge sort orders records.

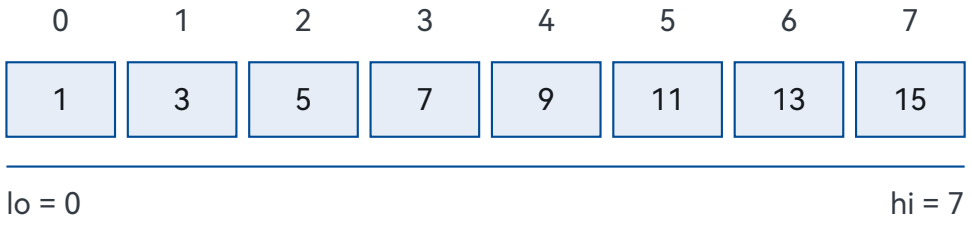
> 路线第三站：前面已经会调用更小的任务，也看见重复求解的代价。现在主动设计子问题。二分利用有序性只保留一个范围，归并求两个有序子结果后组合。递归表达调用，分治决定怎样拆分和使用结果；实际用途是查找已排序数据和整理记录。

One Chosen Half, or Both Halves?

Ordered search keeps **one half**; merge sort needs **both halves**.

target = 15

Sorted input; inclusive range [lo, hi]



Indices 0–7; values increase from left to right.

有序查找排除一半候选；归并排序分别求解两半，再组合结果。

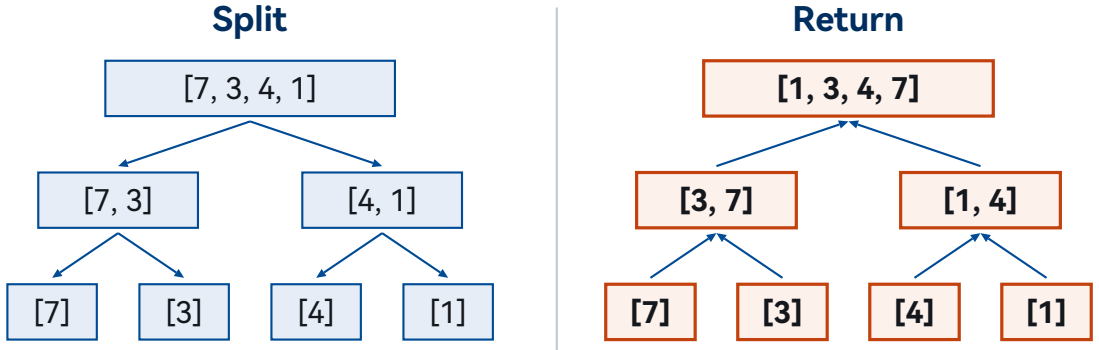
第 46 页 · One Chosen Half, or Both Halves?

目标十五位于有序的八个数中，中点数值七小于十五，所以七及其左侧均可排除，只保留右边。这依赖输入已排序；比较一次只需继续一个子问题。

只用这一张图建立对照，不进入区间边界代码。下一页排序必须处理两半，再合并；同样使用递归，子问题的数量和返回后的工作仍不同。二分查找的命中、缺失和空区间实现保存在讲义。

Sort Smaller Piles, Then Combine

Splitting alone is not enough: **merge the sorted results**.



Base cases: one-element lists are already sorted.

> 把四张号码卡 7、3、4、1 分成两堆，继续分到单张，再分别得到有序的 3、7 和 1、4。
>
> 当前任务还没有完成，必须把两个有序结果合并。图中的拆分与返回是同一份输入的两个阶段。

Only Compare the First Unused Cards

Both piles are sorted; choose the **smaller front item**.

a = [3, 7] b = [1, 4]

Before	Compare	Append	Merged prefix
i=0, j=0	3 vs 1	1 (b[j])	[1]
i=0, j=1	3 vs 4	3 (a[i])	[1, 3]
i=1, j=1	7 vs 4	4 (b[j])	[1, 3, 4]
i=1, j=2	b exhausted	7 (a[i])	**[1, 3, 4, 7]**

i and j point to the first unused item in each input.

> 两个有序列表中，各自第一个未使用元素就是该侧剩余最小值。比较两侧候选，将较小者加入结果，然后移动对应下标。
>
> 依次加入 1、3、4，右侧耗尽后接上剩余的 7。这里保留重复值，不修改原有两堆。

Recursive Merge Sort

Two smaller results need **one combination step**.

Excerpt · 核心片段

```python
def merge_sort(
    nums: list[int]
) -> list[int]:
    if len(nums) <= 1:
        return nums.copy()
    mid = len(nums) // 2
    left = merge_sort(nums[:mid])
    right = merge_sort(nums[mid:])
    return merge(left, right)
```

Complete runnable example 完整程序

Actual output 实跑输出

```
[1, 3, 4, 7]
```

- Base: return a copy of an empty or single-item list.
- Sort both halves, then merge.
- Slices and new results also use storage.

这里是核心递归片段，完整可运行程序链接包含 merge 与 merge_sort。实际运行得到 1、3、4、7 的列表。

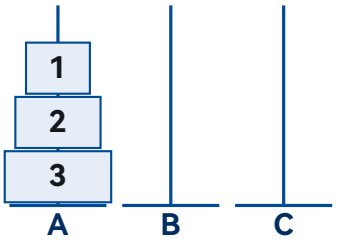
每次都处理更短的列表，长度零或一直接返回副本。课堂理解两次排序和一次合并即可，长比较循环在讲义中查阅。

Hanoi Rules

Plan **legal moves in a required order**, from A to C.



A physical puzzle 实物



One disk per move

Top disk only

No larger disk on a smaller disk

Goal: A → C

Disk number increases with disk size.

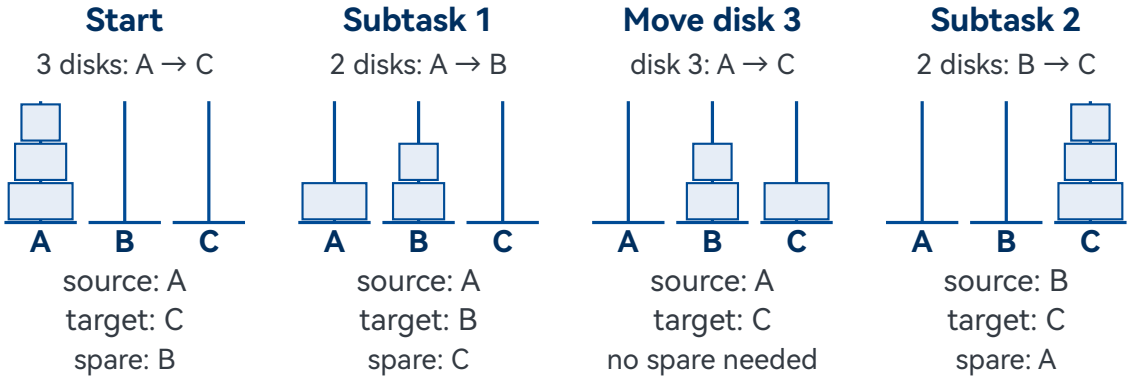
第 50 页 · Hanoi Rules

前面组合子结果得到有序数据，这里把递归用于安排有先后约束的动作。实物与右侧三盘和固定 A、B、C 柱名说明本讲规则：一次只移动顶盘，大盘不能放到小盘上。

目标是全部移到 C。盘号越大盘越大，照片中的盘数不决定本讲示例盘数。先让学生手势说明最大盘为什么暂时不能动。

Free the Largest Disk First

Two smaller transfers surround **one required disk move**.



Each subtask transfers two disks; the middle step moves one disk.

T(0)=0；T(n)=2T(n−1)+1=2^n−1。每个小任务也必须合法完成。

51

第 51 页 · Free the Largest Disk First

要把最大盘从 A 移到 C，先把上面 n-1 个盘完整移到 B，再移动最大盘，最后把 B 的小盘移到 C。每次小盘组转移由合法单盘移动组成。

这些前后转移都是必要的：最大盘直接到 C 前，小盘必须全在 B；最大盘到 C 后，小盘还要移回。让最大盘经过备用柱会增加小盘转移，不能减少下界。三阶段构造达到两次最优小任务加一次移动，T(0)=0，T(n)=2T(n-1)+1，得到 2 的 n 次方减一。详细最少步数说明在精选参考中。

Pillar Names Stay; Their Roles Change

Name the **subtask** before arranging its parameters.

Current task 当前任务	source 起点	target 终点	spare 备用
hanoi(3, A, C, B)	A	C	B
First subtask: move 2 disks A → B	A	B	C
Second subtask: move 2 disks B → C	B	C	A

> 先说清从哪到哪、借哪根柱子，再写参数。

> A、B、C 是固定柱名，起点、终点和备用柱却随当前任务改变。先用一句话说完整的子任务，再对应参数位置。
>
> 第一子任务的目标是 B，所以 C 才是备用；第二子任务从 B 起步，使用 A 备用。

Three Stages Become Three Lines

Smaller stack → largest disk → smaller stack.

code

```python
def hanoi(n: int, source: str,
          target: str, spare: str) -> None:
    if n == 0:
        return
    hanoi(n - 1, source, spare, target)
    print(f"disk {n}: {source} -> {target}")
    hanoi(n - 1, spare, target, source)

hanoi(2, "A", "C", "B")
```

Output

```
disk 1: A -> B
disk 2: A -> C
disk 1: B -> C
```

两个子调用都会完成；中间只移动当前最大盘。

53

第 53 页 · Three Stages Become Three Lines

函数任务是输出合法移动指令，n 为非负整数、柱名互不相同。零盘直接返回，其他情况按三阶段执行。

移动最大盘的 print 位于两个子调用之间，所以输出顺序正好反映任务约束。此函数不维护柱子列表。

Predict: The First Hanoi Subtask



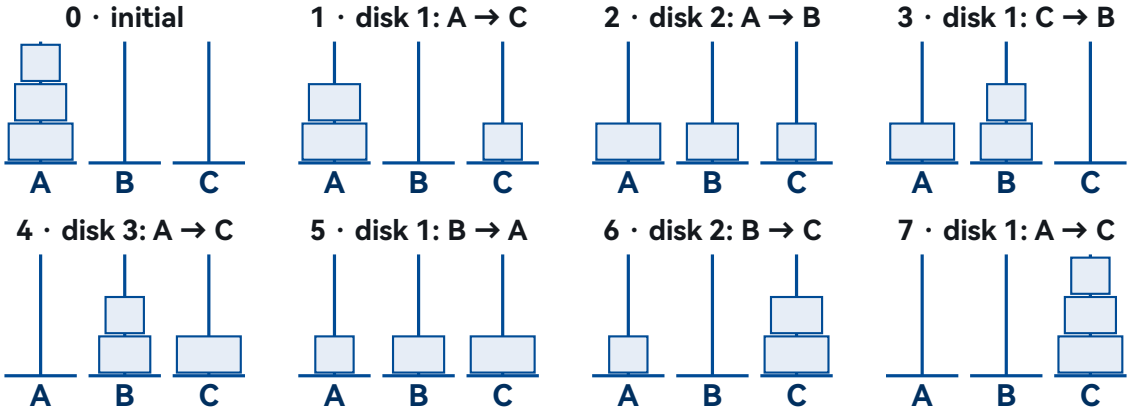
扫码作答

https://taohuang.info/cs1602/zh/quiz/7/

> 最后一道小测检查三盘任务中，移动盘 3 之前要完成哪个子调用。要求学生说清起点、终点和备用柱，避免凭参数排列猜答案。
>
> 公布后用下一页前3次移动回顾这个子任务，再看第4步最大盘怎样移动。

Seven Moves, Eight States

Quiz review: **hanoi(2, A, B, C)** comes first.



前3步完成小盘组 A → B，第4步才移动最大盘；小测答案见上方子调用。

回顾第三题：三盘 A 到 C 之前，先完成两盘 A 到 B、C 作备用的子任务，因此第一子调用是 hanoi(2,A,B,C)。图中前面三次移动正好完成这个子任务，第四次才移动最大盘。

图含初态和七次合法移动后的状态，先读第一行再读第二行。每步只取顶盘，大盘不能压小盘。父调用的 A 到 C 任务不会被子调用的参数覆盖，返回后仍按它自己原先的顺序继续。

Moves, Return Values and State

Printing an instruction is **not returning a move list**.

Excerpt · 核心片段

```
result = hanoi(1, "A", "C", "B")
print(result)
```

Actual output 实跑输出

```
disk 1: A -> C
None
```

- Lab: return a list of moves.
- Check legal moves and the final state.
- A legal sequence need not be minimal.

Complete runnable example 完整程序

当前课堂函数先打印盘一从 A 到 C，再打印返回值 None。个人作业要求返回移动列表，仅打印指令不能满足接口。需要区分生成指令、返回数据和执行这些动作后的真实状态。

核验必须检查顶盘、大小规则和终态；相邻版还要检查移动相邻性。合法不等于最优，一份多走几步但合法完成的序列仍应通过合法性核验，最少步数由生成任务另行要求。不再比较七次移动与十五次函数调用。

V · Choices and Backtracking 回溯

When the next choice is unknown, **search and reject conflicts**.

Carry forward 已掌握

Hanoi gives a required order of smaller tasks.

Next decision 接着思考

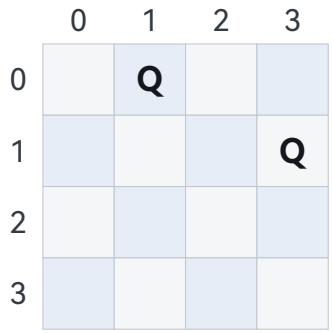
Try partial choices; stop when a conflict appears.

Use it for 用途: Find feasible seating plans, assignments and lineups.

> 路线第五站：汉诺塔的下一项子任务可以直接推导，皇后的下一列则需要尝试多个候选。递归表达做完当前选择后安排剩余部分；回溯决定哪些分支继续、哪些分支因冲突返回，返回后父层再试下一候选。用途是寻找满足约束的布局、分配或排列，不把下一种选择当成一个已知公式。

Place Queens One Row at a Time

Place four queens: **no shared row, column or diagonal**.



queens = [1, 3]

Next row = len(queens) = 2

Try columns 0, 1, 2, 3.

Keep only safe choices.

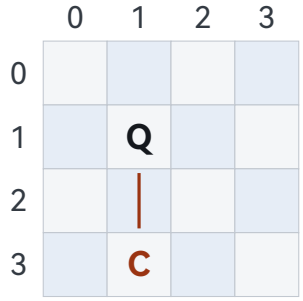
部分解只保存已经放好的前几行。

> 目标是在四乘四棋盘放四个皇后，任意两个不能同行、同列或同对角线。先约定每行放一个。这里只放好前两行，queens=[1,3] 表示第零行在第一列，第一行在第三列。下一行等于当前列表长度二。
>
> 逐行放置已经避免同行冲突，还需排除同列和同对角线。不要把完整解当成部分解，先让学生判断下一行有哪些候选。

See the Conflict Before Writing the Formula

A diagonal has **equal absolute row and column differences**.

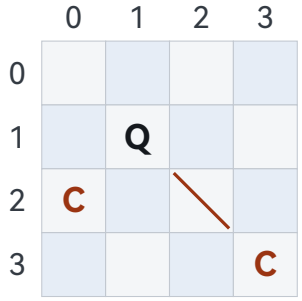
Same column



Q(1,1), C(3,1)
col = c = 1

Q: placed
C: candidate

Both diagonal directions

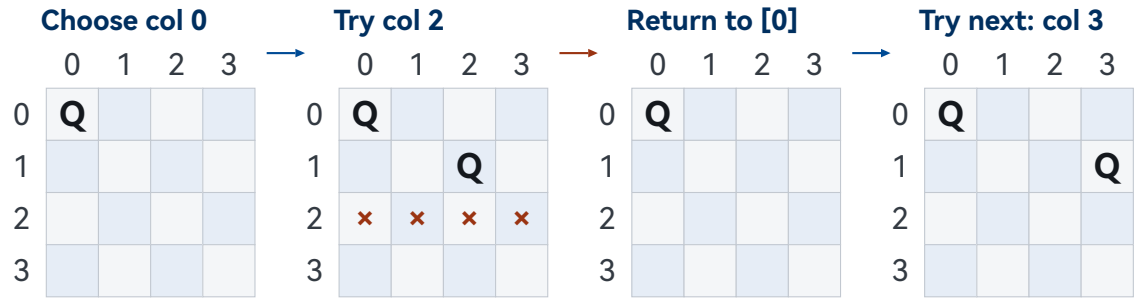


|3 − 1| = |3 − 1| = 2
|2 − 1| = |0 − 1| = 1

> Q 是已放皇后，C 是候选。先在棋盘上识别同列和两种方向的对角线，再对应列号相等与行差、列差绝对值相等。
>
> 实际安全检查要与所有已放皇后比较。行号与列号从零开始，几何关系不依赖棋盘颜色。

No Legal Choice? Return and Try the Next

A failed child does not end the **parent's search**.



No legal square in row 2 → return, then try another column.

每个分支使用新列表；返回后，父层的 [0] 仍然保留。

> 先放第零列，再尝试下一行第二列，得到 [0,2]。第三行四个位置全部冲突，子任务无法完成，返回父层 [0]。
>
> 父层继续下一候选第三列，得到 [0,3]。这里每次用新列表，父层原列表保留，不需要原地撤销；如果改成共享列表 append，就必须相应恢复。

Count Only the Safe Branches

Check conflicts **before** making the next recursive call.

Excerpt · 核心片段

```
def count_queens(
    n: int, queens: list[int]
) -> int:
    if len(queens) == n:
        return 1
    total = 0
    for col in range(n):
        if is_safe(queens, col):
            total += count_queens(
                n, queens + [col])
    return total
```

Complete runnable example 完整程序

Actual output 实跑输出

```
2
92
```

- A complete placement contributes 1.
- An impossible branch contributes 0.
- New lists keep the parent state intact.

> 安全分支增加一个皇后，剩余行数减少。完成 n 行返回一个解，否则把安全分支的解数相加。没有任何安全选择时 total 保持零。
>
> 完整程序含 is_safe，实跑得到四皇后两个解、八皇后九十二个解。课堂先理解剪枝与返回换路，完整检查实现可从讲义独立运行。

Return to the Route: Choose a Design

Match the **task structure** to the strategy and saved state.

Task 任务	Design choice 设计选择	Why it helps 用途
Nested archive	Recursion over a hierarchy	Follow folders of unknown depth
Sorted data lookup	Keep one smaller range	Discard impossible candidates
Sort records	Divide, sort, then merge	Combine ordered smaller results
Hanoi instructions	Order smaller recursive tasks	Respect move dependencies
Queen placement	Backtrack over partial choices	Stop conflicting branches early

明确任务，检查基准、进展与组合；L08 用这些依据审查和验证程序。

回到开头路线图，逐项把实际任务与设计选择对应起来。目录利用已有层级，二分排除不可能范围，归并拆分后合并，汉诺塔安排依赖顺序，皇后在冲突时提前返回。递归表达子任务，不自动决定哪种分解正确，也不自动消除重复计算。

循环与递归是实现表达的比较；分治与回溯是任务设计和搜索策略。顺次累计通常用循环直接，层级或分支可以用递归，也可显式保存待处理状态。仍须核对基准、进展、重复工作和输入规模。下一讲把这些依据落实到代码审查与验证。

Lab 7: Design, Implement and Check

For each task: state the contract and test **small boundaries**.

- **A · Recursion**: digits, reversal, palindrome, flatten and coins.
- **B · Josephus**: compare simulation with recursive reasoning.
- **C · Tools**: search files, count and summarize with shell commands.

> 先解释为何正确，再用实际输入核对。

> 实验的核心是独立完成函数，并给零、空输入、一般输入和较大输入记录验证。数字递归还要与循环比较实际深度边界。
>
> 具体函数签名、分值、工具命令和操作记录入口以双语实验页为准。不要把理论课看懂代码当成已经完成实验。

Solo Assignment 1: Change One Movement Rule

The same objects can define a **different recursive task**.

Ordinary 普通	Adjacent only 相邻限制
Any pair of pillars	A ↔ B ↔ C; no direct A ↔ C
$2^n - 1$ moves	$3^n - 1$ end-to-end moves
Return and validate a move list	Derive the new task decomposition

> 提交移动过程和核验 · 5% · due 10 Nov, 23:59 GMT+8。

64

第 64 页 · Solo Assignment 1: Change One Movement Rule

> 个人作业包含普通汉诺塔与只允许相邻柱移动的变体。两种规则都要给出移动列表并验证合法性与终态。
>
> 变体不能直接沿用普通最大盘一步跨越端点的移动。参考区给出分解与步数关系，完整序列构造和核验仍由学生独立完成。

AFTER CLASS

Selected References 精选参考

> 参考区仅保留六项直接帮助实验和主线理解的材料。三阶递推、二维递推、Ackermann、显示工具和其他扩展仍在理论讲义，不必继续堆入课件。
>
> 这些页面提供可独立理解的说明，标为片段的内容明确给出完整程序链接。

Use an Explicit Stack for the Same Archive

pending explicitly stores **unfinished work**.

Complete code

```
def sum_stack(items: list) -> int:
    pending = items.copy()
    total = 0
    while pending:
        x = pending.pop()
        if isinstance(x, list):
            pending.extend(x)
        else:
            total += x
    return total
archive = [1, [2, 3], [4, [5]]]
print(sum_stack(archive))
```

pop 取出	Action / total
[4, [5]]	extend; total = 0
[5]	extend; total = 0
5	add → 5
4	add → 9
[2, 3]	extend; total = 9
3	add → 12
2	add → 14
1	add → 15

pop 从末尾取出；列表展开，整数累加。原输入保持不变。

pending 初始复制外层列表。右侧按实际 pop 顺序展开：先取 Projects 列表，再取 Draft，随后依次累加五、四、三、二、一。

extend 把子列表内容加入待处理栈，不增加 total；整数才累加。这个版本从后往前访问，与递归次序不同，但总和仍为十五。如果任务要求按原次序压平，就不能忽略访问顺序。

Boundary Checks You Can Explain

Input 输入	Expected behavior 预期	Reason 原因
factorial(0)	1	The direct base result
deep_sum([])	0	The loop has no items
deep_sum([1, [], [2]])	3	Empty folders contribute zero
merge_sort([])	[] (a new list)	Nothing remains to sort
hanoi(0, …)	No printed moves	No disks to transfer
A very deep valid chain	May raise RecursionError	Finite does not mean shallow

从任务约定推导预期，再运行核对。

这些边界不是随意添加的测试，而是直接来自函数约定和基准。先解释空列表、零盘与零阶乘分别对应什么任务，再运行完整程序核对。

有限样例不能证明全部正确性；合法但过深的输入仍可能超过解释器限制。完整函数定义分别位于讲义相应章节。

Coin Combinations: Use or Skip a Denomination

The two branches are **disjoint and exhaustive**.

Branch 分支	Smaller task 子任务	What decreases 进展
Use a 2-value coin	4 → 2, keep {2, 1}	Remaining amount
Skip the 2-value coin	Keep 4, only {1} remains	Available denominations
Amount = 0	One empty completion	Direct answer
Positive amount, no coins	No completion	Direct answer

> 金额 4 的三种组合：2+2、2+1+1、1+1+1+1。顺序不算新方案。

以金额四、面额二和一说明两类不重叠方案：使用至少一个二元硬币，或完全不使用二元硬币。使用分支保留当前面额，跳过分支不再返回它。

固定小例子有 2+2、2+1+1、1+1+1+1 三种组合。顺序不算新方案；面额必须互异且为正。这里只给任务分解，完整计数函数由实验完成。

Why the Hanoi Count Is Minimal

A construction is optimal only after a **matching lower bound**.

Necessary stage 必需阶段	Cost 下界	Construction 构造
Clear the largest disk	At least T(n−1)	Transfer small stack to spare
Move the largest disk	At least 1	One source → target move
Finish the smaller stack	At least T(n−1)	Transfer small stack to target

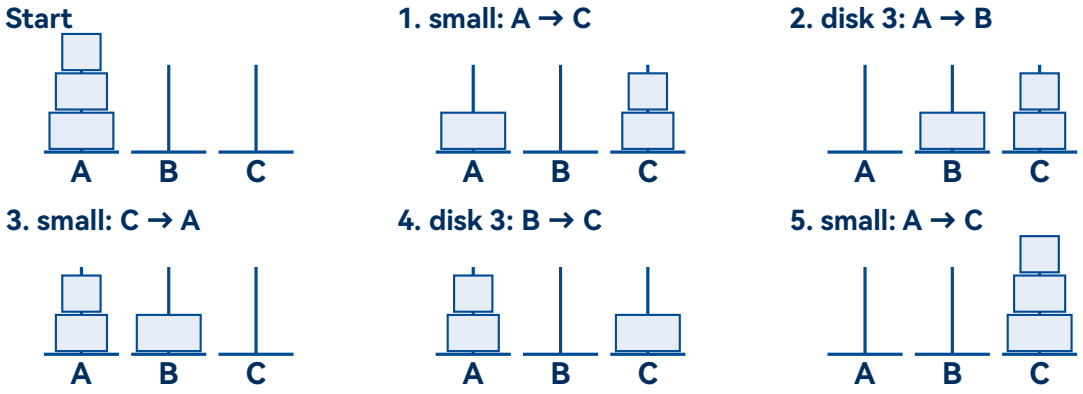
T(0)=0；T(n)=2T(n−1)+1=2^n−1。

最大盘直接移向目标之前，上面的小盘必须完整位于备用柱；移到目标后，小盘还要转移到它上面。最大盘若经过备用柱，需要额外转移小盘，不会减少下界。

递归三阶段恰好达到两次小盘转移加一次最大盘移动的下界，因此 T(0)=0，T(n)=2T(n-1)+1，得到 2 的 n 次方减一。

Adjacent-Only Hanoi: Five Necessary Stages

No direct A ↔ C move: transfer the small stack **three times**.



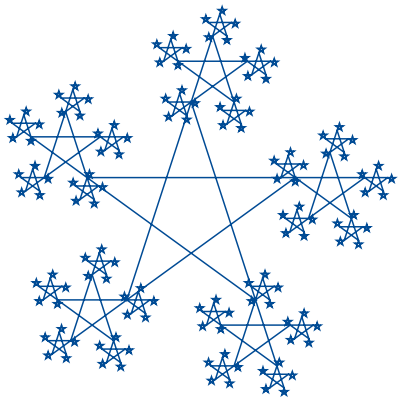
T(0)=0；T(n)=3T(n−1)+2=3^n−1。small 是子任务，不能整组搬盘。

六个状态分别是初态和五阶段后的状态。为了让最大盘 A 到 B，先把小盘移到 C；为清空 C，再把小盘移回 A；最大盘 B 到 C 后，小盘最后从 A 移到 C。

因此端点转移满足 T(0)=0、T(n)=3T(n-1)+2=3 的 n 次方减一。各小盘组转移必须由合法相邻移动实现，图中的整组箭头只概括子任务，不是一次搬一叠盘。

The Recursive Star: See the Actual Path

Each non-base call draws **five smaller recursive branches**.



360 → 120 → 40

→ 13⅓ → stop

780 segments

781 calls

小图画完，位置与方向恢复。

每个小图结束都恢复位置与方向，父层才能继续下一条边。

这张路径由讲义中同一份 star 函数实际调用 forward 和 left 生成。每次前进 size 后画小图，再左转 216 度；size 小于等于十停止。

360 开始得到 780 条线段、781 次调用，结束时位置和方向都恢复。图展示真实程序轨迹，不依赖本机 turtle 窗口。完整本机图形程序及环境要求仍在讲义。