

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Introduction to Computation (CS1602) · Lecture 7

Recursion 递归

Instructor: Tao Huang

Part III: Problem Solving with AI · Fall 2026

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?

Predict: Open and Close



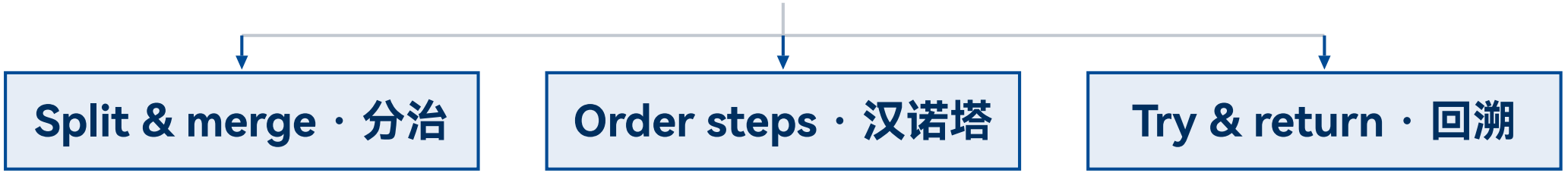
扫码作答

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

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。

How Recursion Connects the Strategies

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

Recursion 递归 · express a task using smaller instances

Base case + smaller task + combination

Hierarchies
层级数据

Process each child.

Folders → total size

Divide & conquer
分治

Split, solve, combine.

Merge sort → records

Backtracking
回溯

Choose, check, retry.

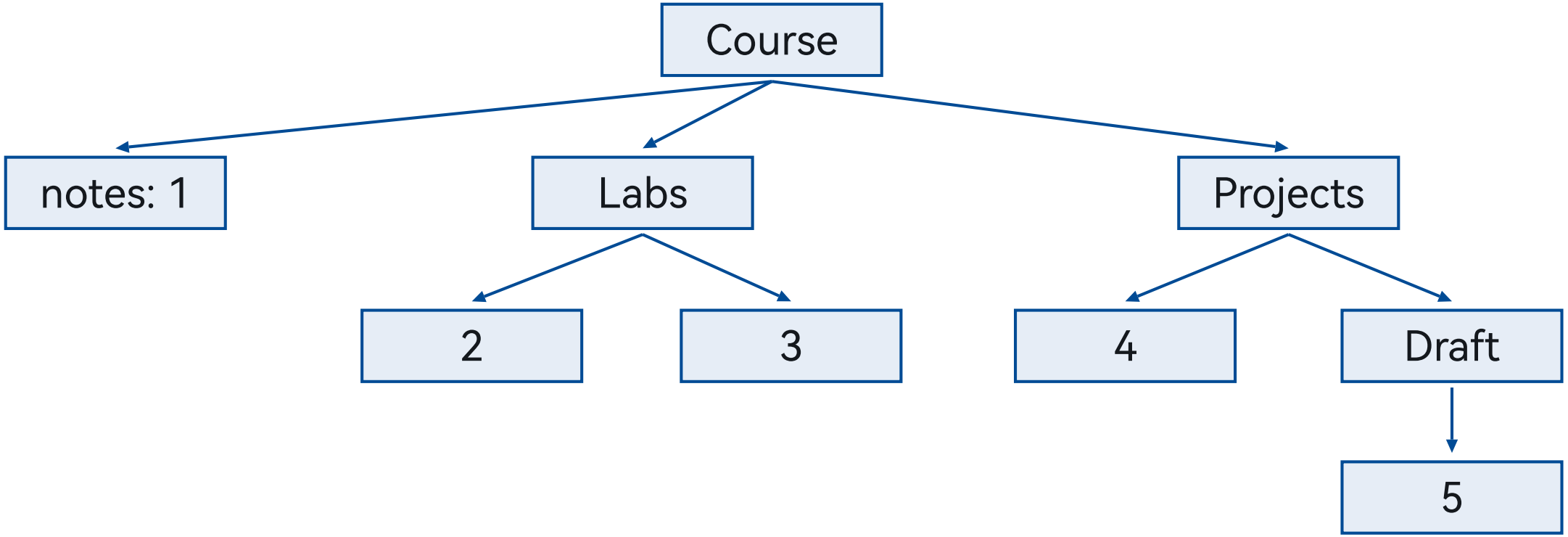
Queens → placement

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。

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。

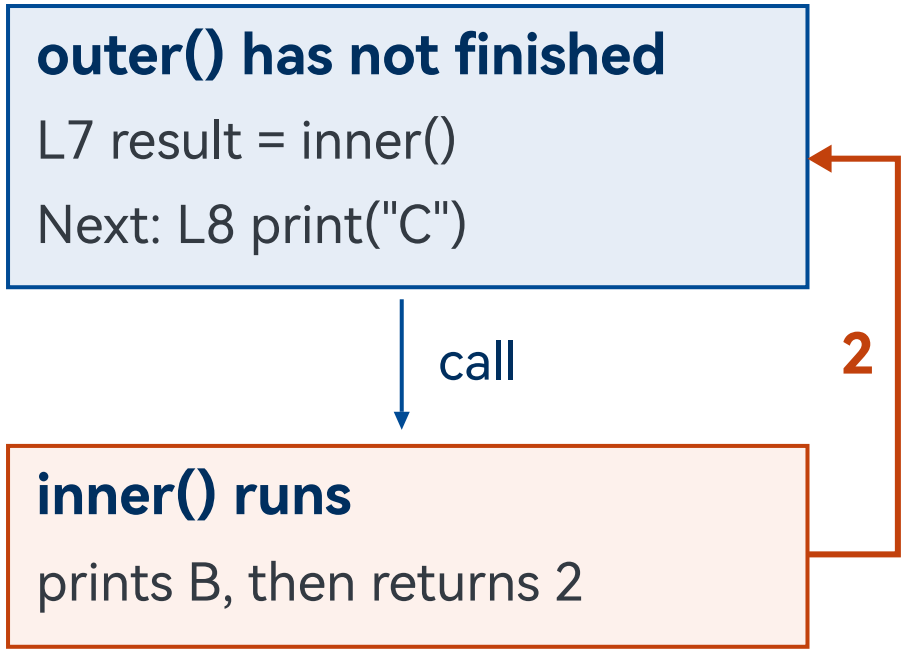
I · Where Does outer Continue?

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

Follow the Calls: outer → inner

Finished: the stack is empty.

Code 程序 · 10 / 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 ↑

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。

Recursive Factorial

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

code

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

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。

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。

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。

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。

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。

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。

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。

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。

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。

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) n = 1
Next: 1 × 1 = 1

factorial(2) n = 2
Wait at L4 · 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。

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。

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。

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。

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。

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。

Opening Quiz Review: Close the Inner Folder First

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

code

```
1  def inspect(level: int) -> None:
2      print("open", level)
3      if level > 0:
4          inspect(level - 1)
5      print("close", level)
6
7  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。

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。

Predict: Will the Base Case Be Reached?



扫码作答

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

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。

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。

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.

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))
```

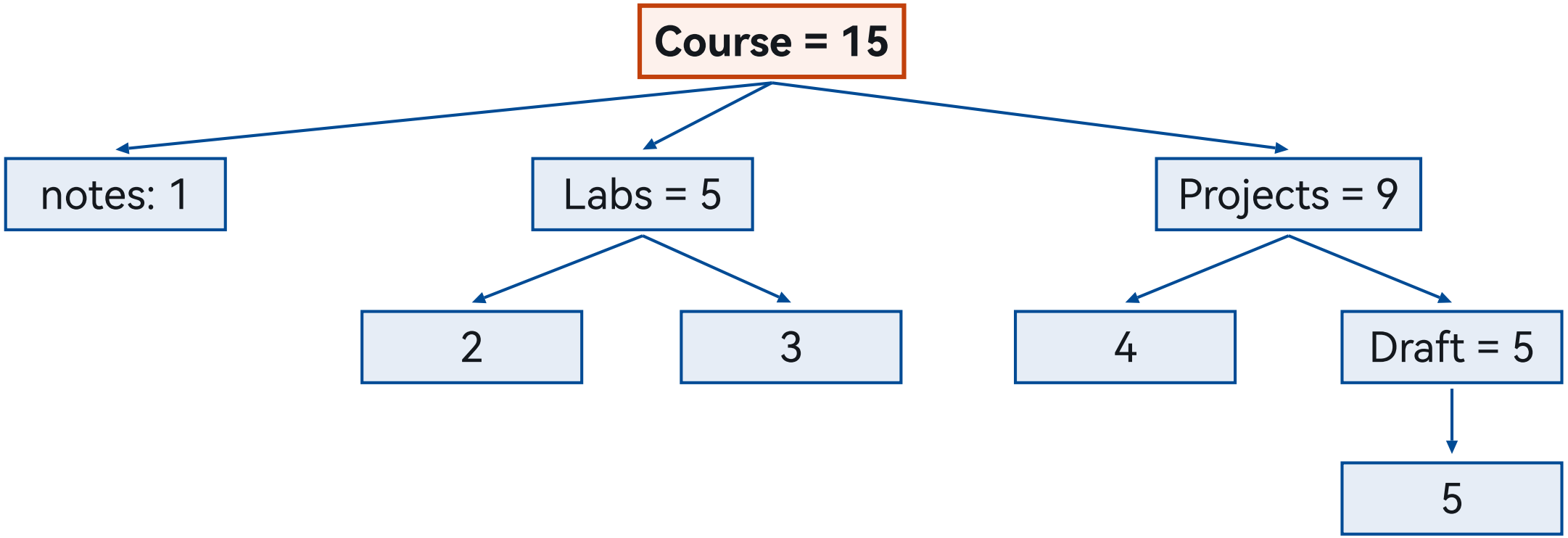
Output

```
15
```

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

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。

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)])
```

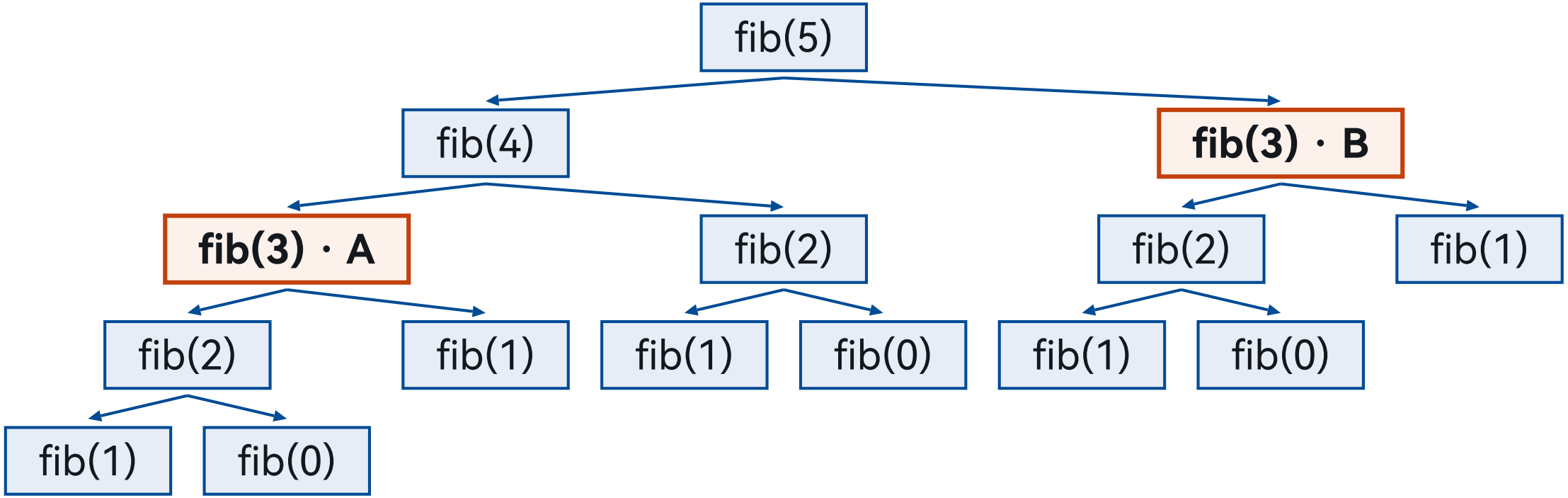
Output

```
[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。

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。

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.

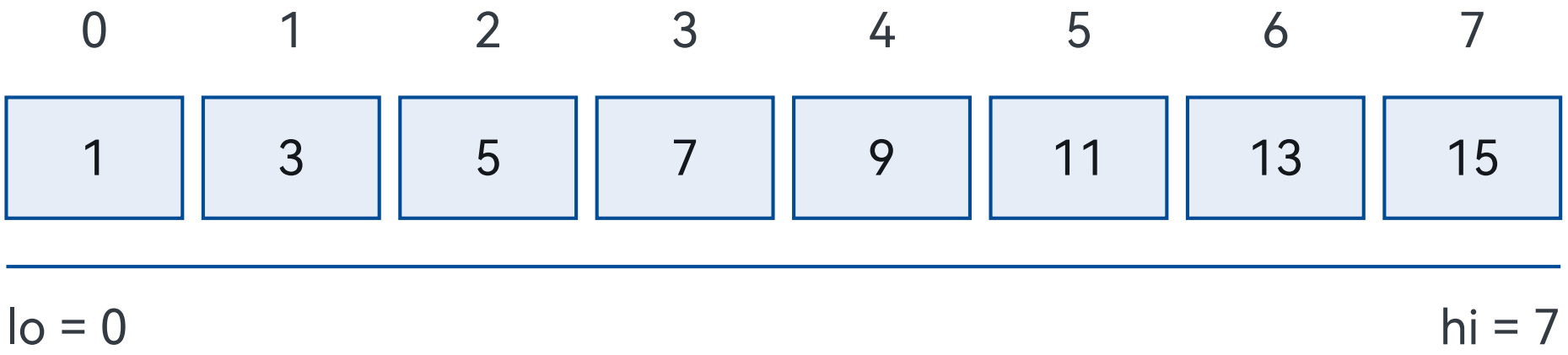
Use it for 用途: Binary search finds an item; merge sort orders records.

One Chosen Half, or Both Halves?

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

target = 15

Sorted input; inclusive range [lo, hi]

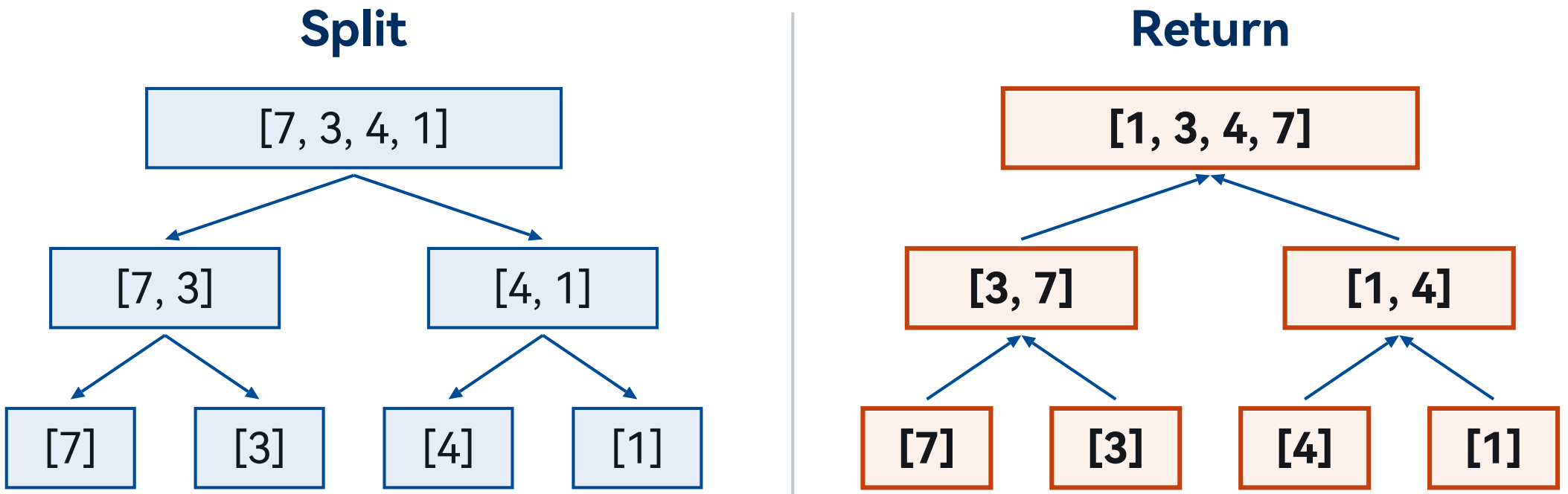


Indices 0–7; values increase from left to right.

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。

Sort Smaller Piles, Then Combine

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



Base cases: one-element lists are already sorted.

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.

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

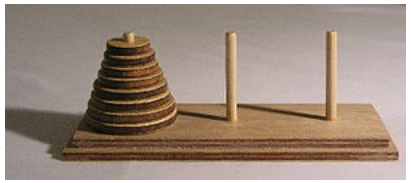
Actual output 实跑输出

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

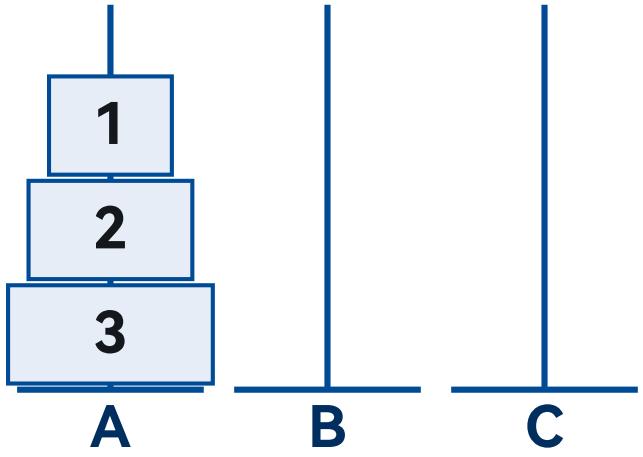
- Base: return a copy of an empty or single-item list.
- Sort both halves, then merge.
- Slices and new results also use storage.

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

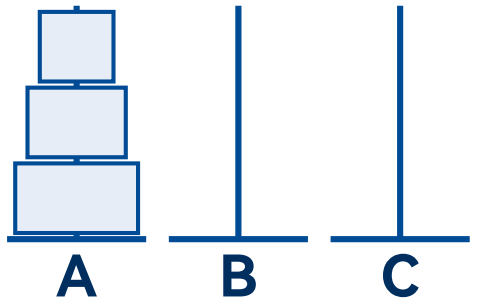
Disk number increases with disk size.

Free the Largest Disk First

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

Start

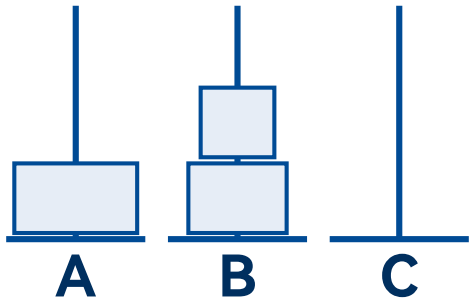
3 disks: A → C



source: A
target: C
spare: B

Subtask 1

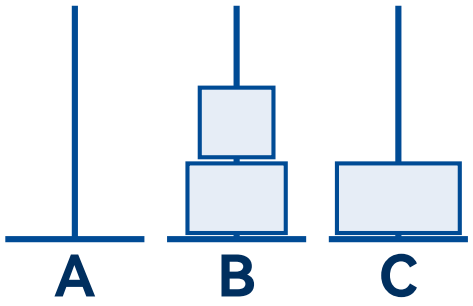
2 disks: A → B



source: A
target: B
spare: C

Move disk 3

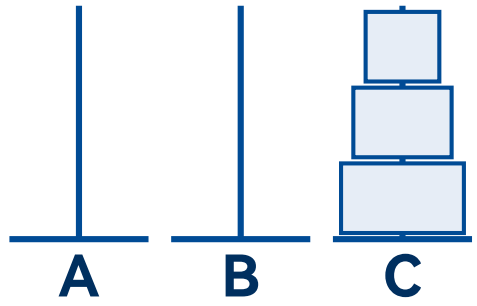
disk 3: A → C



source: A
target: C
no spare needed

Subtask 2

2 disks: B → C



source: B
target: C
spare: A

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。

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。

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。

Predict: The First Hanoi Subtask

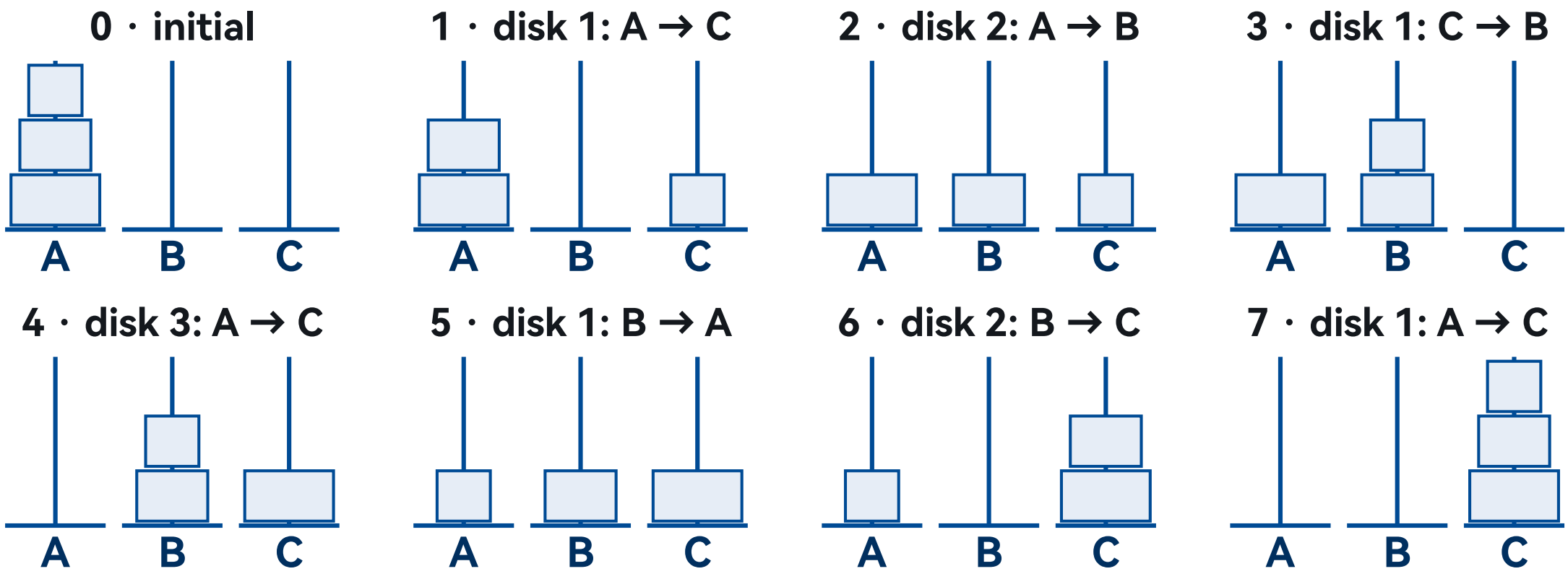


扫码作答

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

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。

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

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.

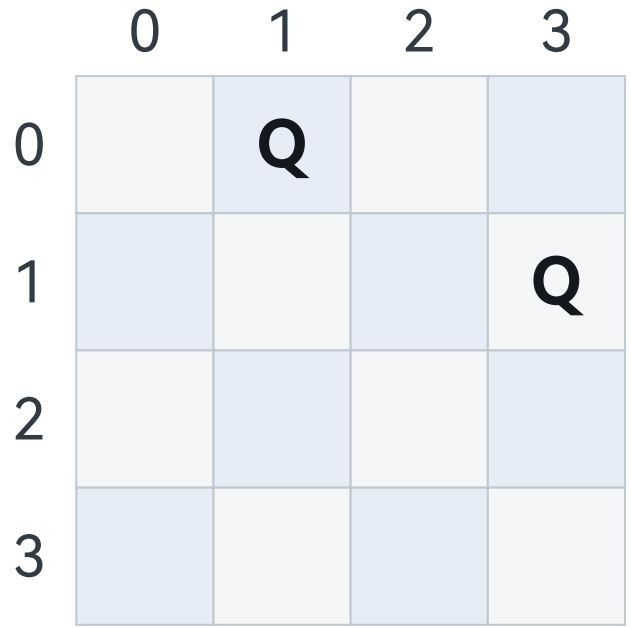
Next decision 接着思考

Try partial choices; stop when a conflict appears.

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

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.

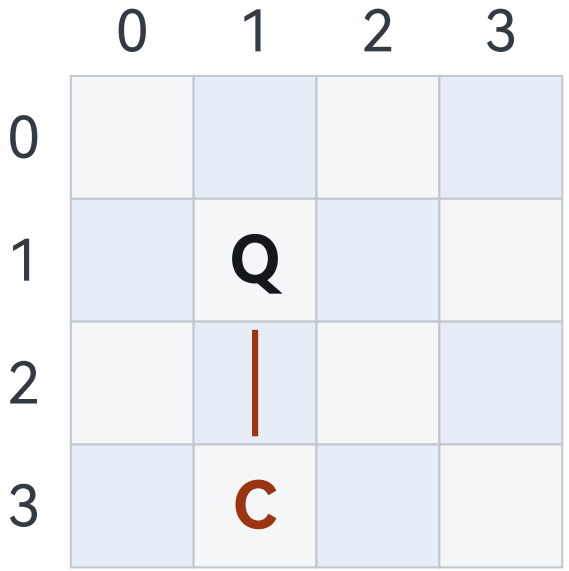
Keep only safe choices.

部分解只保存已经放好的前几行。

See the Conflict Before Writing the Formula

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

Same column



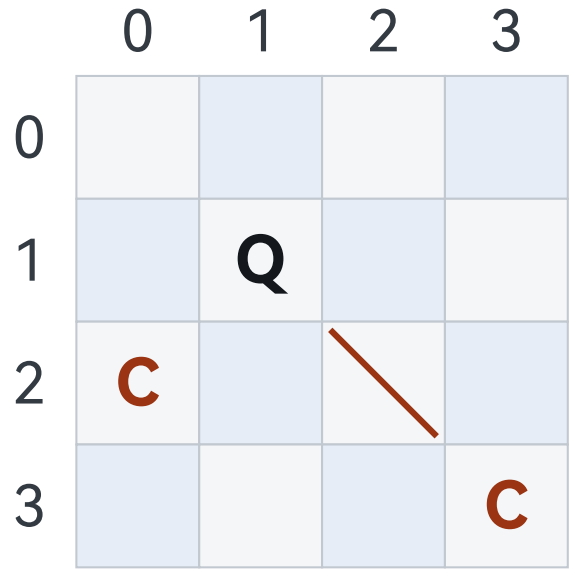
Q(1,1), C(3,1)

col = c = 1

Q: placed

C: candidate

Both diagonal directions

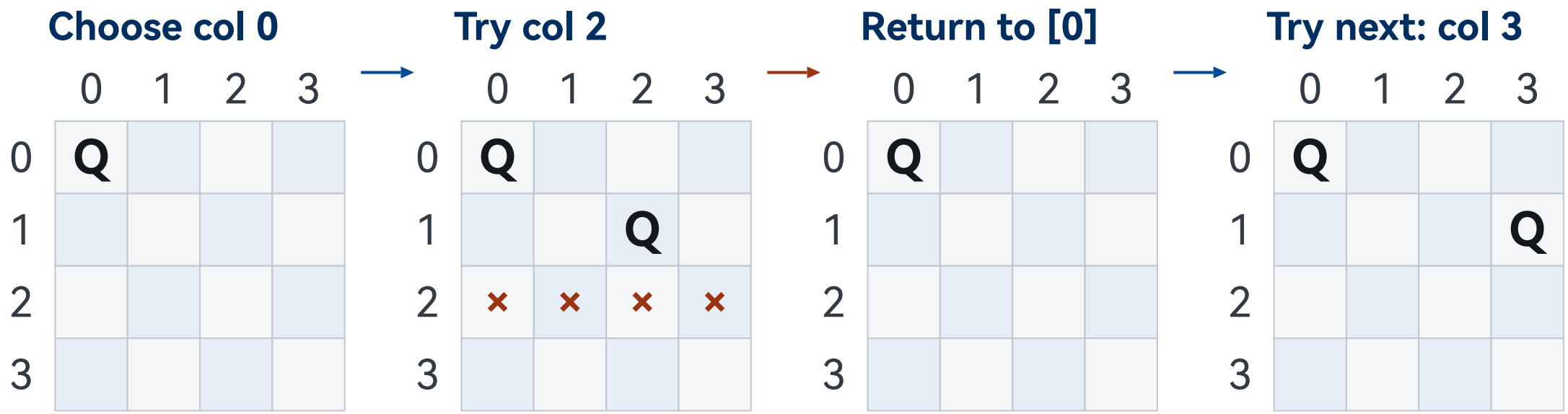


$|3 - 1| = |3 - 1| = 2$

$|2 - 1| = |0 - 1| = 1$

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。

Count Only the Safe Branches

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

Excerpt · 核心片段

```python
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
```

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.

Complete runnable example 完整程序

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。

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。

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。

AFTER CLASS

Selected References 精选参考

Six practical follow-ups.

Full programs and further recurrences remain in the lecture notes.

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。

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。

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。

Why the Hanoi Count Is Minimal

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

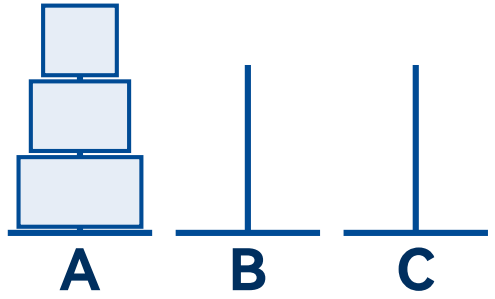
Necessary stage 必需阶段	Cost 下界	Construction 构造
Clear the largest disk	At least $T(n-1)$	Transfer small stack to spare
Move the largest disk	At least 1	One source → target move
Finish the smaller stack	At least $T(n-1)$	Transfer small stack to target

$T(0)=0$；$T(n)=2T(n-1)+1=2^n-1$。

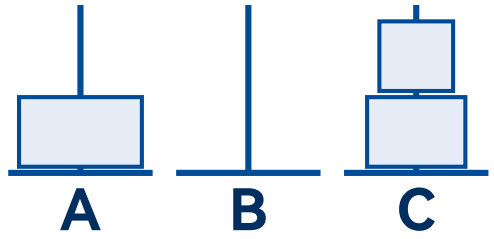
Adjacent-Only Hanoi: Five Necessary Stages

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

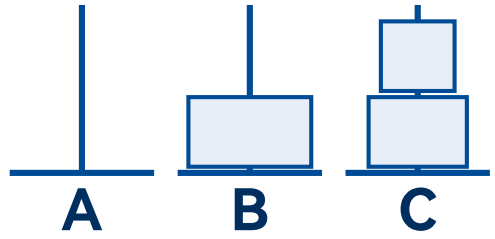
Start



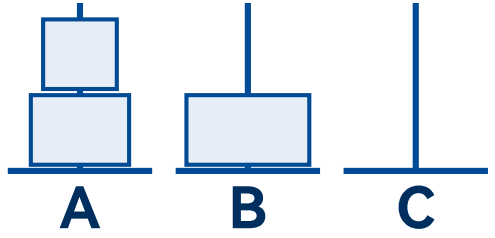
1. small: A → C



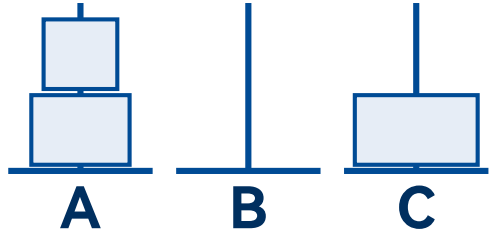
2. disk 3: A → B



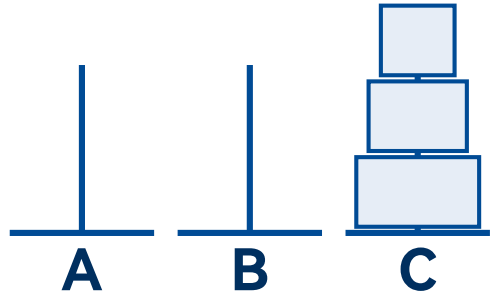
3. small: C → A



4. disk 3: B → C



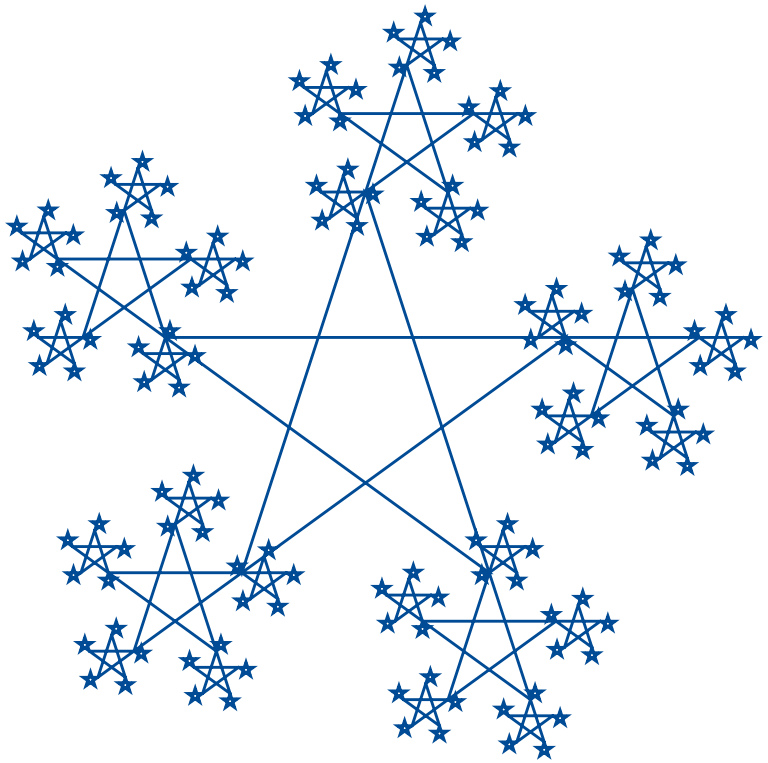
5. small: A → C



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

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。