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。

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Introduction to Computation (CS1602) · Lecture 7

Recursion 递归

Instructor: Tao Huang

Part 3: Problem Solving with AI · Fall 2026

1

第 1 页 · Recursion 递归

> 上一讲介绍了元组、字典、字符串和集合，也用嵌套字典表示了更复杂的数据。容器的元素还可以是容器，处理数据时就可能遇到多层结构。
>
> 本讲研究递归：函数把当前任务分成规模更小的同类任务，通过调用自己完成这些子任务。先观察函数的调用与返回，再分析终止条件、重复计算、分治和回溯。

COURSE STRUCTURE

Part III · Problem Solving with AI

> 课程进入第三部分：问题求解与 AI 协作。第七讲先学习怎样分解问题，以及怎样判断递归程序能否得到正确结果。第八讲进一步讨论代码审查与验证。
>
> 本讲仍以学生独立完成实现为要求，课程政策见网站。这里不重复每讲都相同的政策页。

Containers and Nested Data 嵌套数据

- **List 列表**: an ordered sequence
- **Dictionary 字典**: keys map to values
- **Nesting 嵌套**: a container can contain another container

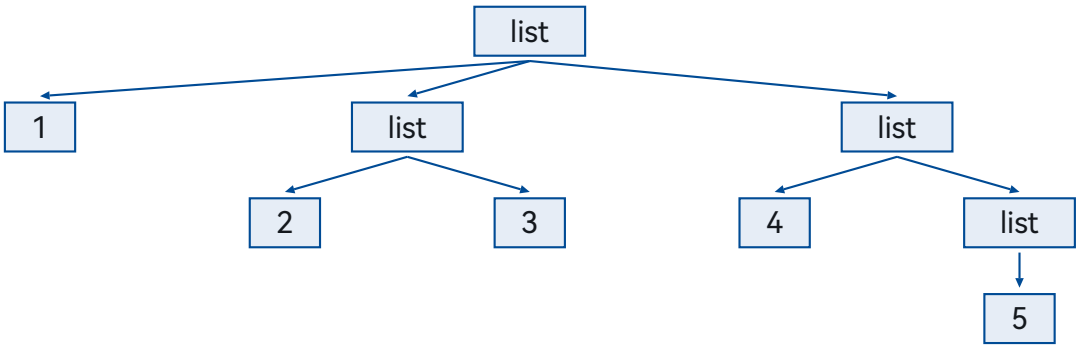
```
[1, [2, 3], [4, [5]]]
```

Three outer items, five integers. How can every integer be processed?

> 先回顾列表按位置组织元素，字典按键查值，再指出容器可以互相嵌套。下面的列表外层只有三个元素，其中包含五个整数。
>
> 只遍历外层时，第二项和第三项得到的是列表。要求全部整数之和，还需要继续处理这些内层列表。

Nested Records 嵌套记录

How can every integer be included when nesting depth varies?



[1, [2, 3], [4, [5]]]

第 4 页 · Nested Records 嵌套记录

> 沿图中的连接，可以找到整数 1、2、3、4、5。若输入永远只有两层，可以写两层循环；这里的输入允许不同深度，预先写固定层数的循环不能覆盖任意深度。
>
> 每个内层列表需要执行的任务与外层相同：逐项检查，整数直接处理，列表继续处理。这个结构稍后会成为递归的实际例子。图表示数据之间的包含关系，不表示程序的执行时间顺序。

Quiz 1 · Function Calls

扫码作答 · 一题

1 普通函数的调用与返回顺序

Predict the output order, then submit once.

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

> 开场题只使用已经讲过的函数调用、print 和 return。outer 调用 inner 时，outer 后面的语句什么时候执行？先根据代码预测打印顺序。
>
> 这一题只提交一次。下面用真实输出解释调用和返回，不要求提前知道递归。

Call and Return Order 调用与返回顺序

code

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

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

print(outer())
```

output

```
A
B
C
3
```


第 6 页 · Call and Return Order 调用与返回顺序

> outer 先打印 A，再调用 inner。inner 打印 B 并返回整数 2；得到返回值后，outer 把 2 绑定给 result，打印 C，再返回 3。最外层的 print 最后打印 3。
>
> 调用另一个函数不会自动结束调用者。调用者保留局部变量以及接下来要执行的位置，等被调用函数返回后再继续。递归使用的仍是这套调用规则。

Lecture Structure

1	Calls and termination	**调用与终止** · local state, return order and reachable base cases
2	Data and subproblems	**递归结构与重复计算** · nested lists, gcd, Fibonacci and power
3	Divide and conquer	**分治** · binary search and merge sort
4	Hanoi and backtracking	**汉诺塔、枚举与回溯** · instruction order and legal partial choices

> 先用阶乘分析每次调用的局部状态、调用顺序和返回顺序，再检查递归是否能够终止。之后回到嵌套数据，比较只有一个子问题和产生多个子问题的计算。
>
> 二分查找、归并排序和汉诺塔展示不同的任务分解方式。最后用排列与皇后问题说明枚举和回溯。较长的完整程序放在讲义与参考区，课堂优先讲清设计步骤。

I · Recursive Functions

> 普通函数可以调用别的函数，同一个函数也可以在当前调用尚未结束时再次被调用。
>
> 每次调用都有自己的参数和局部变量。递归程序是否正确，要分析每次调用承担的任务、哪些输入可以直接完成、下一次调用是否更接近这些输入。

Factorial Recurrence 阶乘递推关系

Nonnegative integer n · input domain 非负整数

Concrete example 具体例子	General rule 递推关系
3! = 3 × 2! = 3 × 2 × 1	**n! = n × (n − 1)!** for n ≥ 1
2! = 2 × 1!; 1! = 1 × 0!	**0! = 1** · direct answer 基准

先核对输入范围与零的阶乘，再分析子调用返回后如何乘以当前 n。

第 9 页 · Factorial Recurrence 阶乘递推关系

阶乘提供了一个熟悉的递推关系。为了计算 3!，可以先计算 2!，再乘 3；计算 2! 又可以先计算 1!。当输入是 0 时，直接得到 1。

函数约定为：输入非负整数 n，返回 n!。零的阶乘也是定义的一部分，不能遗漏。数学公式给出递归结构，但还需要检查代码中的参数变化和返回值。

Recursive Factorial 递归阶乘函数

What does factorial(3) return? 输入：非负整数

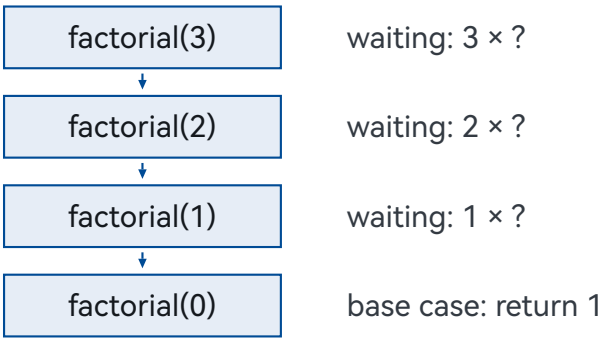
```python
def factorial(n: int) -> int:
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(3))
```


> 先检查基准情形：n 等于 0 时返回 1，不再调用。其他非负整数输入进入乘法表达式，其中 factorial(n - 1) 要先返回一个值。
>
> 调用 factorial(3) 时，3 乘以什么还没有确定；需要先执行 factorial(2)。每次调用参数减少 1，最终到达 0。先口头预测结果，再对照调用图。

The Call Chain 调用链

Call order



Each caller resumes after its callee returns.

> 从 factorial(3) 开始，依次调用 factorial(2)、factorial(1)、factorial(0)。外层调用都在等待内层结果，因此 3、2、1 对应的乘法尚未完成。
>
> factorial(0) 直接返回 1，调用链不再加深。图中的四个节点表示四次独立调用，同一段函数代码不表示只有一套局部变量。

Local Variables 局部变量

factorial(0)	local n = 0	current
factorial(1)	local n = 1	waiting: 1 × ?
factorial(2)	local n = 2	waiting: 2 × ?
factorial(3)	local n = 3	waiting: 3 × ?

Logical call frames; not a CPython memory layout.

Logical call frames 调用帧示意；not a CPython memory layout.

第 12 页 · Local Variables 局部变量

> 图中每个调用帧记录一次调用的局部状态。factorial(3) 的 n 是 3，factorial(2) 的 n 是 2；内层调用不会把外层的 n 改成 2。
>
> 当前执行的是 n 为 0 的调用，其余调用等待返回。调用按后进先出顺序结束，通常称为调用栈。这里画的是逻辑关系，不是 CPython 对象或物理内存的具体布局。

The Return Chain 返回顺序

Return order

f(3) = 6	3 × 2 = 6
f(2) = 2	2 × 1 = 2
f(1) = 1	1 × 1 = 1
f(0) = 1	base case: return 1

Each caller resumes after its callee returns.

第 13 页 · The Return Chain 返回顺序

> factorial(0) 返回 1。factorial(1) 得到 1 后完成 1 × 1，返回 1；factorial(2) 完成 2 × 1，返回 2；factorial(3) 完成 3 × 2，返回 6。
>
> 阶乘例子中的乘法在递归调用返回之后完成。一般递归也可以在调用前、调用后或两次调用之间进行计算，不能把这个例子的顺序当成所有递归的规则。

Tracing Recursive Calls 递归调用追踪

code

```python
def factorial(n: int, depth: int = 0) -> int:
    pad = "  " * depth
    print(f"{pad}enter {n}")
    if n == 0:
        result = 1
    else:
        result = n * factorial(n - 1, depth + 1)
    print(f"{pad}return {result}")
    return result

print(factorial(3))
```

output

```
enter 3
  enter 2
    enter 1
      enter 0
      return 1
    return 1
  return 2
return 6
6
```


> 进入每次调用时打印 enter，得到结果后打印 return。depth 只控制缩进，不参与阶乘计算；默认参数 0 表示最外层调用。
>
> 先连续出现 enter 3、2、1、0，然后按相反顺序出现返回值 1、1、2、6。最后一行 6 来自最外层 print。缩进帮助区分调用层次，不能仅凭输出中数字相同就把两次调用当成同一次。

Base Case and Progress 基准与终止度量

Check 检查	Meaning 含义	Factorial example 阶乘
Base case 基准	Complete the task directly; no recursive call	`factorial(0) → 1`
Progress 终止度量	A lower-bounded measure approaches the base case	**3 → 2 → 1 → 0**

- **Measure the remaining work**: a parameter, length or unfilled rows.
- **Not every parameter must decrease**: placed queens increase while remaining rows decrease.

> 基准情形是可以直接完成的输入，执行这条分支时不再递归。还要找出一个有下界的度量，证明每次调用都会使它接近基准。阶乘使用非负整数 n 作为度量。
>
> 并不是所有参数都必须减小。稍后的皇后问题中，已放皇后数量增加，但剩余行数减少。判断进展要根据任务，而不是只看某个数字是否变小。

Nonterminating Recursion 非终止递归

No base case 没有基准

```
def bad(n: int) -> int:
    return bad(n - 1)

bad(3)
```

- **No direct return** for any input

No progress 参数不变

```
def bad(n: int) -> int:
    if n == 0:
        return 0
    return bad(n)

bad(3)
```

- **n stays 3**; the base case is unreachable

Both independently checked examples raise **RecursionError**.

> 左侧每次都再次调用，没有任何输入可以直接返回。右侧虽然写了 n 等于 0 的分支，但从 3 开始始终再次传入 3，无法到达基准。
>
> 两段程序都在默认环境中触发 RecursionError。写出 if 并不自动保证终止；需要把基准分支和递归分支放在一起检查。本页的两个错误例子已分别独立运行。

An Unreachable Base Case 不可达基准

Will n reach 0? 参数依次为 3, 1, −1, −3, …

```python
def down(n: int) -> int:
    if n == 0:
        return 0
    return down(n - 2)

print(down(3))
```


第 17 页 · An Unreachable Base Case 不可达基准

> 从 3 开始，每次减 2，参数依次是 3、1、−1、−3，始终是奇数，因此不会等于 0。程序最终触发 RecursionError。
>
> 参数数值一直减少，仍不代表基准情形可达。如果任务约定允许全部非负整数，这段递推对奇数输入就不正确。也不能简单改成 n 小于等于 0 后返回，除非新的基准与函数的任务相符。

Reasoning About Recursion

1	**Task 任务**	State what the function must **return or do**.
2	**Base case 基准**	Check that the **direct result** is correct.
3	**Subproblem 子问题**	Assume its **specified result** is correct.
4	**Combination 组合**	Derive the **current result** from that result.
5	**Progress 终止**	Prove that a **base case can be reached**.

> 分析阶乘时，不必把所有调用无限展开。假设 factorial(n - 1) 能正确返回 (n - 1)!，乘以 n 就得到 n!。这是组合步骤的正确性。
>
> 这个假设需要另外两项支持：0 的结果正确，非负整数 n 经过不断减 1 确实能够到达 0。函数任务、基准、组合和终止条件缺少任何一项，都不能据此证明程序正确。

Recursion Depth 递归深度

Finite but too deep 有限但过深

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

print(countdown(5000))
```

Actual exception 实际异常

```
RecursionError:
maximum recursion depth exceeded
```

- **Finite**: 5000 → 4999 → … → 0
- **Too deep**: the interpreter stops before the base case.
- **Depth limit**: not a measurement of disk space or RAM.

Build: **Python 3.12.9** · `sys.getrecursionlimit()` → **1000**. Consider a loop or explicit state.

第 19 页 · Recursion Depth 递归深度

> 本页记录构建时实际解释器的递归上限。countdown(5000) 每次减 1，从数学上能够到达 0，但在当前环境中，调用链过深，提前触发 RecursionError。
>
> 递归上限限制解释器的调用深度，不保证恰好执行指定次数的学生函数才报错，也不证明物理内存已经耗尽。处理大输入应考虑循环或显式状态，不建议通过提高递归上限完成实验。

Recursion and Iteration 递归与循环

Recursion 递归

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

print(factorial(10))
```

output

```
3628800
```

Iteration 循环

```python
def factorial_loop(n: int) -> int:
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

print(factorial_loop(10))
```

output

```
3628800
```


第 20 页 · Recursion and Iteration 递归与循环

> 两段程序都返回 10!，真实输出均为 3628800。递归版本建立多层调用，循环版本在同一次调用中更新 result。阶乘适合用来解释递归机制，实际实现通常循环更直接。
>
> 循环也有计算和存储开销。这里比较的是同一个任务的两种结构，不能由此推出循环对所有任务都更清楚或更快。输入规模与问题结构也需要考虑。

II · Recursive Data and Subproblems

> 已经确认每次调用保留自己的局部状态。现在回到开场的嵌套列表：同样的逐项处理任务，可以由同一个函数负责每一层。
>
> 再比较最大公约数、Fibonacci 和快速幂。它们都使用递归，但产生的子问题数量和重复工作并不相同。

Type Checks 类型检查

code

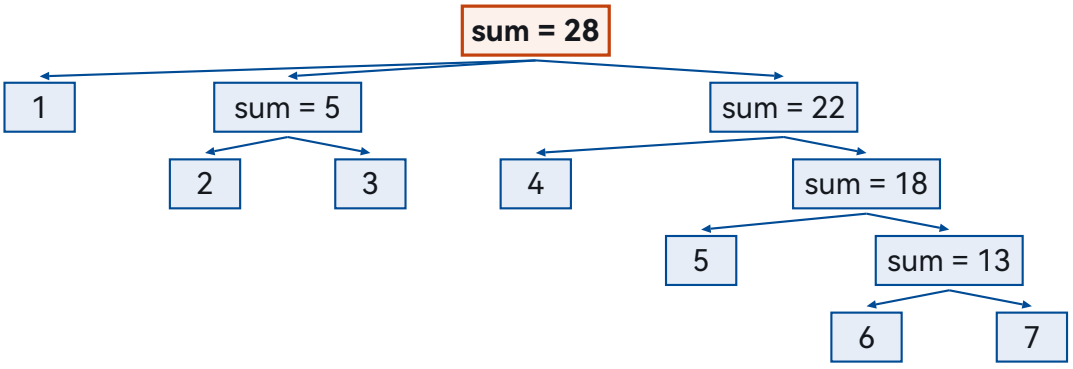
```
for x in [3, [1, 2], []]:
    print(isinstance(x, list))
```

output

```
False
True
True
```


> isinstance(x, list) 检查 x 是否是列表对象，结果为布尔值。整数 3 得到 False，普通列表和空列表都得到 True。
>
> 嵌套求和需要根据元素类型选择处理方式。本讲约定输入由整数和有限、没有循环引用的列表组成，不能把任意对象都默认当成整数累加。

Summing Nested Lists 嵌套列表求和



Integer leaves: 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28.

第 23 页 · Summing Nested Lists 嵌套列表求和

> 每个列表都逐项处理自己的元素。遇到整数就累加，遇到列表就递归求这个列表的和，再把返回值加入当前 total。图中所有整数叶子的总和是 28。
>
> 外层与各内层都有自己的 total，它们不会互相覆盖。结构图中的分支表示包含关系。程序实际按列表顺序处理各项，而不是同时处理全部分支。

Recursive Sum

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

print(deep_sum([1, [2, 3], [4, [5, [6, 7]]]]))
```

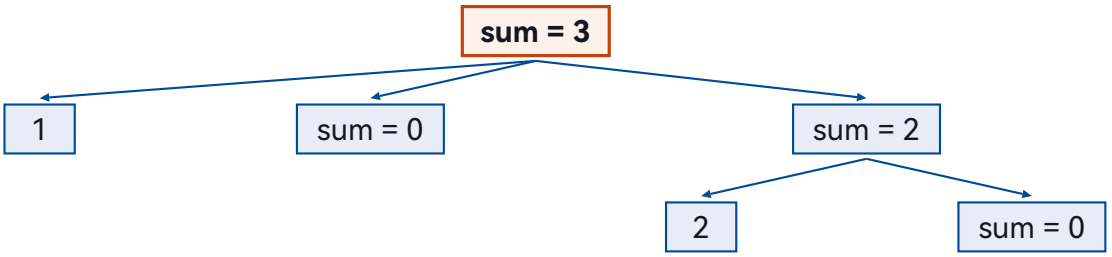
output

```
28
```


> deep_sum 用一个循环处理当前列表的元素，用递归处理内层列表。循环与递归可以出现在同一个函数中。当前 total 从 0 开始，结束时返回这一层全部元素的整数总和。
>
> 列表中的整数直接加入 total；内层列表先由 deep_sum 得到一个整数结果，再加入 total。代码实际运行得到 28。输入必须有限且没有循环引用。

Empty Lists and Nested Calls

Predict deep_sum([1, [], [2, []]]) before following the sums.



Empty lists return 0; total = 3.

> 空列表没有元素，for 循环执行零次，total 保持为 0 并返回。因此 deep_sum 不一定需要单独写空列表的 if。
>
> 外层整数 1，加空列表的 0，再加内层列表返回的 2，结果为 3。这里的基准由空列表的处理自然实现。包含自身的列表不符合输入约定，不能拿这个版本直接处理循环引用。

Euclid's Algorithm 欧几里得算法

gcd(a, b) = gcd(b, a % b) when b > 0

Current pair 当前参数	Next pair 下一参数	Remainder 余数
`(48, 18)`	`(18, 12)`	`48 % 18 = 12`
`(18, 12)`	`(12, 6)`	`18 % 12 = 6`
`(12, 6)`	**(6, 0) → 6**	`12 % 6 = 0`

> **Nonnegative integers**; `gcd(a, 0) = a`. The second argument strictly decreases, not necessarily the first.

第 26 页 · Euclid's Algorithm 欧几里得算法

> 欧几里得算法把求 a 与 b 的最大公约数，改为求 b 与 a 除以 b 的余数的最大公约数。b 为 0 时，按约定直接返回 a。
>
> 对非负整数且 b 大于 0，余数总在 0 到 b - 1 之间，因此第二个参数严格减少。第一个参数可能增加，例如从 (0, 5) 到 (5, 0)，仍然可以终止。不能只要求所有参数都变小。

Recursive gcd 最大公约数

code

```python
def gcd(a: int, b: int) -> int:
    if b == 0:
        return a
    return gcd(b, a % b)

print(gcd(48, 18))
print(gcd(1071, 462))
print(gcd(0, 5))
```

output

```
6
21
5
```


第 27 页 · Recursive gcd 最大公约数

> 代码的两个分支直接对应递推关系。b 等于 0 的分支先执行，避免对零取余。三个实跑结果分别是 6、21、5。
>
> 按本函数约定 gcd(0, 0) 返回 0；若使用不同数学约定，接口中需要明确。简短代码仍需要输入范围、递推正确性和终止说明，不能用行数代替这些条件。

Fibonacci Recurrence 斐波那契递推

code

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

for n in range(10):
    print(fib(n), end=" ")
```

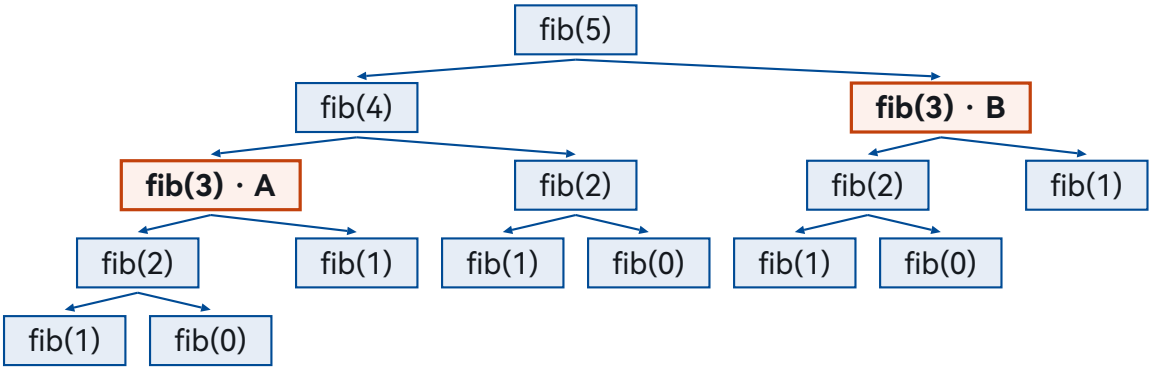
output

```
0 1 1 2 3 5 8 13 21 34
```


> 统一使用 F(0) 等于 0、F(1) 等于 1 的定义。n 大于等于 2 时，需要得到 F(n - 1) 与 F(n - 2)，再相加。输入约定为非负整数。
>
> 前十项实际为 0、1、1、2、3、5、8、13、21、34。返回值正确不代表计算过程没有浪费；这里第一次出现两个递归子调用，需要检查它们是否会重复处理同一个参数。

Repeated Subproblems 重复子问题

How many separate calls to fib(3) appear in fib(5)?



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

第 29 页 · Repeated Subproblems 重复子问题

> fib(5) 的完整调用树共有 15 个节点，每个节点都是一次实际调用。图中两个 fib(3) 分别属于不同调用路径，即使参数相同，也会各自重新计算。
>
> Python 不会因为函数名和参数相同就自动保存并复用返回值。判断重复工作，要统计实际调用，不能只数不同参数有几个。图中的线表示调用关系，执行时仍按代码先求左侧再求右侧。

Counting Function Calls 调用次数统计

code

```python
def fib_counted(n: int) -> tuple[int, int]:
    if n <= 1:
        return n, 1
    left, c1 = fib_counted(n - 1)
    right, c2 = fib_counted(n - 2)
    return left + right, 1 + c1 + c2

for n in [5, 10, 20, 25]:
    value, calls = fib_counted(n)
    print(n, value, calls)
```

output

```
5 5 15
10 55 177
20 6765 21891
25 75025 242785
```


第 30 页 · Counting Function Calls 调用次数统计

> 这个版本同时返回 Fibonacci 值与调用次数。基准调用返回 n 和 1；非基准调用把两个子调用的计数相加，再加当前这一次调用。
>
> 元组与解包已经在上一讲介绍。计数由参数和返回值传递，不需要使用尚未介绍的 global。计数中包含基准调用，不能与加法次数混为一谈。

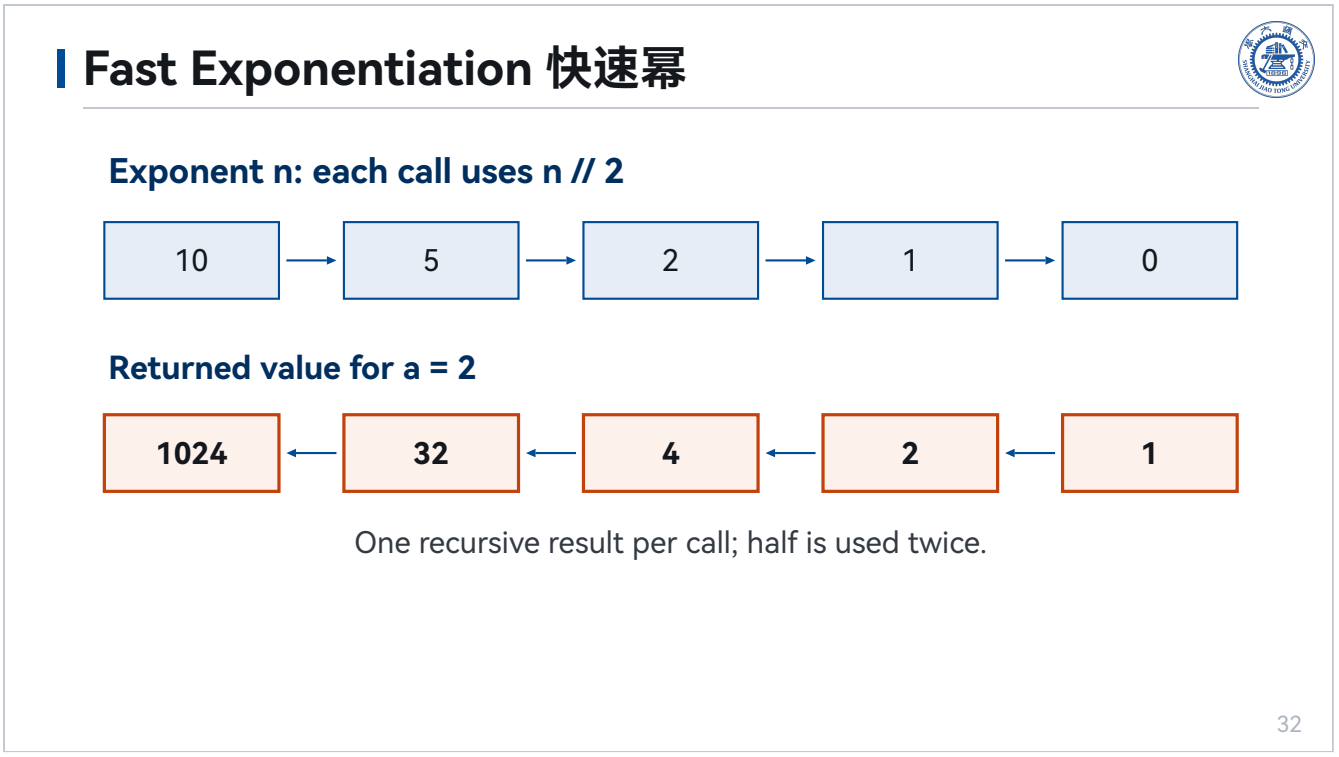
Fibonacci Call Counts

n	fib(n)	Function calls
5	5	15
10	55	177
20	6,765	21,891
25	75,025	**242,785**

Counts include the initial call and every base-case call.

Measured calls 实际调用次数；includes base-case calls.

> 输入从 5 增加到 25，调用次数从 15 增加到 242785。这里比较确定的调用次数，不比较容易受机器、后台任务和浏览器影响的秒数。
>
> C(0) 与 C(1) 都是 1，其他输入满足 C(n) = 1 + C(n - 1) + C(n - 2)。因此 2 的 n 次方不能写成这个程序的精确调用次数。保存重复子问题的结果可减少重复计算，后续课程再展开这些方法。



第 32 页 · Fast Exponentiation 快速幂

> 计算 a 的非负整数次幂，可以先求指数除以 2 后取整的子问题。偶数指数把子结果平方，奇数指数还要乘一次 a。指数 10 的调用链是 10、5、2、1、0。
>
> 与 Fibonacci 不同，这里每层只求一个子问题，然后使用得到的结果两次。基准 a 的 0 次幂返回 1；本程序也按这一约定返回 power(0, 0) 的结果。

Recursive Power

code

```python
def power(a: int, n: int) -> int:
    if n == 0:
        return 1
    half = power(a, n // 2)
    if n % 2 == 0:
        return half * half
    return half * half * a

print(power(2, 10))
print(power(3, 5))
```

output

```
1024
243
```


> half 保存 power(a, n // 2) 的返回值。偶数分支计算 half × half，奇数分支再乘 a。两个输出分别是 1024 和 243。
>
> 输入指数限定为非负整数。每次减半减少递归层数，但整数越来越大时，乘法实际时间和存储也会增加。不能把每次乘法都说成固定耗时。

Result Reuse 返回值复用

The even-exponent branch 偶数指数分支

One call 一次调用

```
half = power(a, n // 2)
return half * half
```

- **Compute once**
- **Use the returned value twice**

Two calls 两次调用

```
return (power(a, n // 2)
        * power(a, n // 2))
```

- **Both calls execute**
- **The same subtask is repeated**

> 比较计算量时，要区分复用返回值与再次调用；Python 不会自动缓存相同调用。

34

第 34 页 · Result Reuse 返回值复用

> 比较偶数分支。左侧先执行一次递归调用，把返回值绑定给 half，乘法中使用这个值两次。右侧写了两个函数调用，两个调用都会执行。
>
> 对于这个纯计算任务，结果相同，但右侧重复处理相同的子问题。奇数分支也需要保持相应的额外乘法。代码更短不代表计算更少，参数相同也不代表解释器自动缓存。

III · Divide and Conquer 分治

> 二分查找利用有序性，只保留一个仍可能含目标的子区间。归并排序先处理两个子列表，再把两个有序结果合并。
>
> 两者都分解问题，但子问题数量和组合步骤不同。嵌套列表的遍历也使用递归，不能仅凭递归语法就把不同算法的工作量说成相同。

Binary Search Preconditions 二分查找前提

Sorted input; inclusive range [lo, hi]. 查找目标：15

target = 15

Sorted input; inclusive range [lo, hi]

0	1	2	3	4	5	6	7
1	3	5	7	9	11	13	15

lo = 0 hi = 7

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

> 数组按非降序排列，下标从 0 到 7。二分查找维护闭区间 [lo, hi]，两端都包含在候选范围内。第一次比较中点下标 3，其值为 7。
>
> 目标 15 大于 7，有序性保证下标 0 到 3 的值都不可能等于 15，因此可以排除这一半。输入没有排序时，这个推理不成立，程序可能漏报实际存在的值。

Search Range 查找区间

target = 15

Blue: current interval; underline: mid

		0	1	2	3	4	5	6	7
[0, 7]	mid = 3	1	3	5	7	9	11	13	15
[4, 7]	mid = 5	1	3	5	7	9	11	13	15
[6, 7]	mid = 6	1	3	5	7	9	11	13	15
[7, 7]	mid = 7	1	3	5	7	9	11	13	**15**

Found: return index 7.

> 查找 15 的候选区间依次是 [0, 7]、[4, 7]、[6, 7]、[7, 7]，中点依次为 3、5、6、7。在最后一次比较中命中目标，返回下标 7。
>
> 已经比较且不等于目标的中点要排除。保留右半区间时用 mid + 1，保留左半时用 mid - 1。若仍把中点留在区间中，某些短区间就可能无法继续缩小。

Recursive Binary Search

One recursive call; −1 for an empty range. 闭区间

```python
def search_range(nums: list[int], target: int,
                 lo: int, hi: int) -> int:
    if lo > hi:
        return -1
    mid = (lo + hi) // 2
    if nums[mid] == target:
        return mid
    if nums[mid] < target:
        return search_range(nums, target, mid + 1, hi)
    return search_range(nums, target, lo, mid - 1)
```


第 38 页 · Recursive Binary Search

> 先检查 lo 是否大于 hi，空区间返回 -1。非空时比较中点，命中返回下标；目标较大只搜索右区间，目标较小只搜索左区间。每次只递归一个子区间。
>
> 这段辅助函数有显式的区间参数，不依赖跨页定义。学生通常使用讲义中的 binary_search 包装接口，它会设置初始区间。重复值可以返回任一命中下标，不承诺最左位置。

Unsuccessful Search 查找未命中

Target 8: why does the final range contain no elements?

target = 8

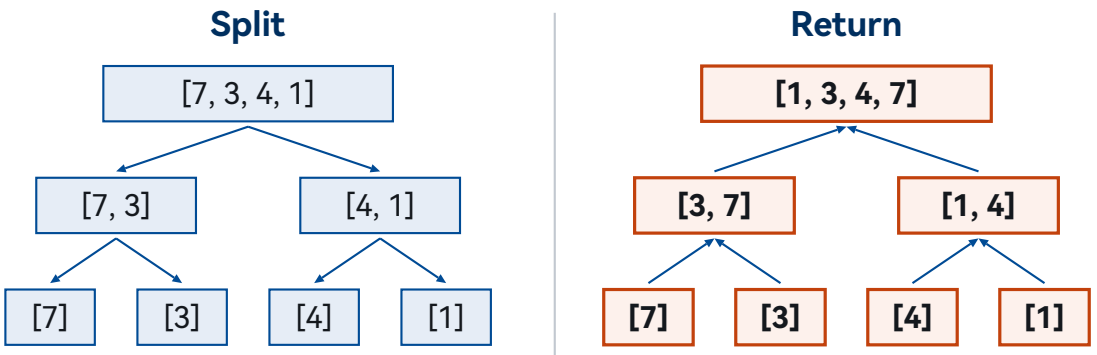
Blue: current interval; underline: mid

		0	1	2	3	4	5	6	7
[0, 7]	mid = 3	1	3	5	7	9	11	13	15
[4, 7]	mid = 5	1	3	5	7	9	11	13	15
[4, 4]	mid = 4	1	3	5	7	9	11	13	15
[4, 3]	empty	1	3	5	7	9	11	13	15

lo > hi: empty interval, return −1.

> 查找不存在的 8 时，区间依次是 [0, 7]、[4, 7]、[4, 4]、[4, 3]。最后 lo 大于 hi，表示候选元素已经全部排除，因此返回 -1。
>
> 空列表从一开始就是 [0, -1]，也立即返回 -1。空区间的分支必须在访问 nums[mid] 之前执行，否则可能越界或得到错误结果。

Merge Sort 归并排序



Base cases: one-element lists are already sorted.

> 把 [7, 3, 4, 1] 分成 [7, 3] 和 [4, 1]，再把它们分成单元素列表。长度 0 或 1 的列表已经有序。
>
> 返回时把 [7] 与 [3] 合并成 [3, 7]，把 [4] 与 [1] 合并成 [1, 4]，最后再合并这两个有序结果。递归排序两个子列表还不够，当前列表的有序结果需要通过归并得到。

Merging Sorted Lists 有序列表归并

Which unused element should be appended next? 合并有序列表

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.

> i 和 j 分别指向两个有序列表中尚未使用的第一项。比较 a[i] 与 b[j]，将较小值加入结果，再前移对应下标。
>
> 合并 [3, 7] 与 [1, 4] 时，依次加入 1、3、4，右侧耗尽后再接上左侧剩余的 7。每次只取一个元素，重复值全部保留。两边有序是这个方法的前提。

The Merge Function

Excerpt · compare candidates

```python
def merge(
    a: list[int], b: list[int]
) -> list[int]:
    result = []
    i = 0
    j = 0
    while (i < len(a)
           and j < len(b)):
        if a[i] <= b[j]:
            result.append(a[i])
            i += 1
        else:
            result.append(b[j])
            j += 1
```

Excerpt · finish the same function

```python
    result.extend(a[i:])
    result.extend(b[j:])
    return result

# a and b must be sorted
# Inputs are not modified
```

- **Both inputs sorted** before the loop
- **One side exhausted**: append the remainder
- **New result**: neither input is modified

第 42 页 · The Merge Function

> 左栏是比较循环，右栏接上尚未使用的两段并返回结果。循环条件要求两边都还有元素，保证访问 a[i] 与 b[j] 合法。
>
> 某一侧耗尽后，另一侧已经有序，可以直接 extend。函数返回新的 result，不修改输入 a 和 b。两栏是同一个函数的片段，讲义中有包含全部行的独立可运行程序。

Recursive Merge Sort

Excerpt · see complete program

```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, 2, 3, 3, 4, 5, 7]
```

- **Requires merge(a, b)** from the complete program.
- **Base case returns a copy**; neither input is changed.
- **Two halves** are sorted, then merged.

第 43 页 · Recursive Merge Sort

> 长度小于等于 1 时返回 nums.copy()，保证基准结果也是新列表。其他输入分成更短的左右列表，分别排序，再调用 merge 合并。完整程序实际输出 [1, 2, 3, 3, 4, 5, 7]。
>
> 本页明确是片段，merge 的完整定义在讲义中，不给不完整片段设置运行按钮。切片与合并也需要分配空间，不能把这种实现描述为只使用几个固定变量。

Recursive Algorithm Structure

Algorithm 算法	Subproblem 子问题	After the call 返回后
Binary search	**One** chosen subrange	Pass back the matching index
Merge sort	**Both** halves	Merge the sorted results
Power	**One** smaller exponent	Reuse the computed value

比较子问题的**数量、规模与组合步骤**，不能只看递归语法。

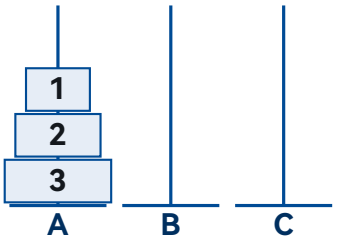
第 44 页 · Recursive Algorithm Structure

> 二分查找只保留一个候选区间，结果沿调用链返回。归并排序处理两个子列表，还要完成线性归并。快速幂处理一个减半后的指数，再复用结果进行乘法。
>
> 递归代码的外形相似，工作量却由子问题数量、规模和组合步骤共同决定。正式复杂度分析留到第十一讲，当前先能区分这些结构。

IV · Towers of Hanoi 汉诺塔

> 前面的函数主要返回数值或列表。汉诺塔改为组织一组有先后约束的移动指令：某个盘子移动之前，必须先完成其他盘子的转移。
>
> 先用三盘理解任务和顺序，再写递归函数。这个课堂版本打印指令；个人作业会保存移动列表，并验证每一步是否合法。

Hanoi Rules



One disk per move

Top disk only

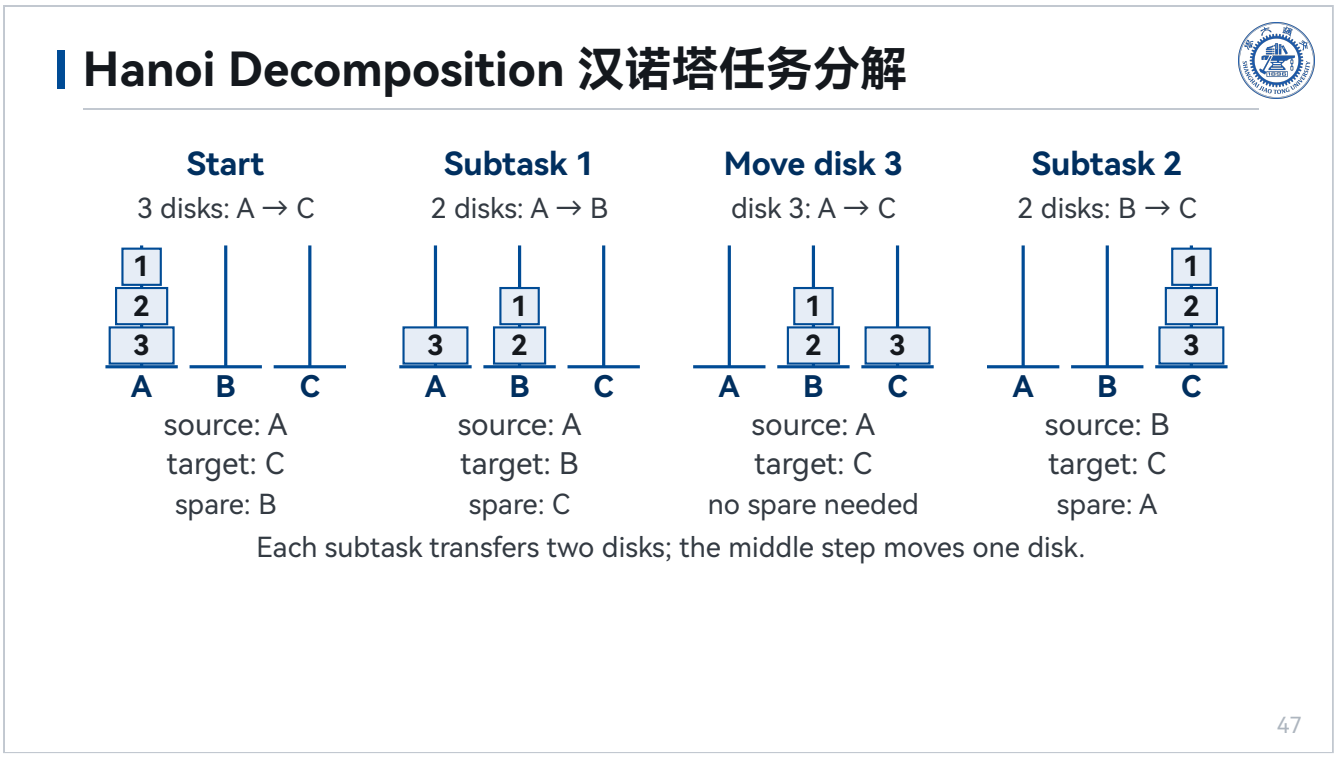
No larger disk on a smaller disk

Goal: A → C

Disk number increases with disk size.

第 46 页 · Hanoi Rules

> 三根柱子固定命名为 A、B、C。盘号越大，盘子越大；初始从下到上是 3、2、1，全部位于 A。目标是把全部盘子合法移到 C。
>
> 每次只移动一个柱顶盘，大盘不能放在小盘上。左侧为往年汉诺塔实物照片，右侧为按本讲规则重画的三盘示意。照片中的盘数不影响任务定义；本页示例以右侧三盘为准。



第 47 页 · Hanoi Decomposition 汉诺塔任务分解

> 最大盘 3 要从 A 移到 C，先要把上面的两个盘从 A 移到 B。完成后，盘 3 从 A 移到 C，再把 B 上面的两个盘移到 C。
>
> 两个两盘转移都遵守相同规则，分别使用当时空出的柱子作为备用柱。图中的子任务箭头代表完整的一段合法序列，不代表把两个盘一次搬动。

Source, Target and Spare 三种任务角色

Role 当前角色	Meaning 含义	hanoi(3, A, C, B)
source	Start pillar 起点	**A**
target	Destination 终点	**C**
spare	Temporary pillar 备用柱	**B**

1	**First subtask**	`hanoi(n - 1, source, spare, target)`
2	**Largest disk**	Move disk n: **source → target**
3	**Second subtask**	`hanoi(n - 1, spare, target, source)`

第 48 页 · Source, Target and Spare 三种任务角色

先用表格核对当前角色：hanoi(3, A, C, B) 中，起点是 A，终点是 C，备用柱是 B。柱名固定，角色随子任务改变。

再按下面三步说明移动顺序。第一个子任务将两个小盘从 A 转移到 B，此时 C 作为备用柱；所以调用为 hanoi(2, A, B, C)。不能只记参数交换顺序。

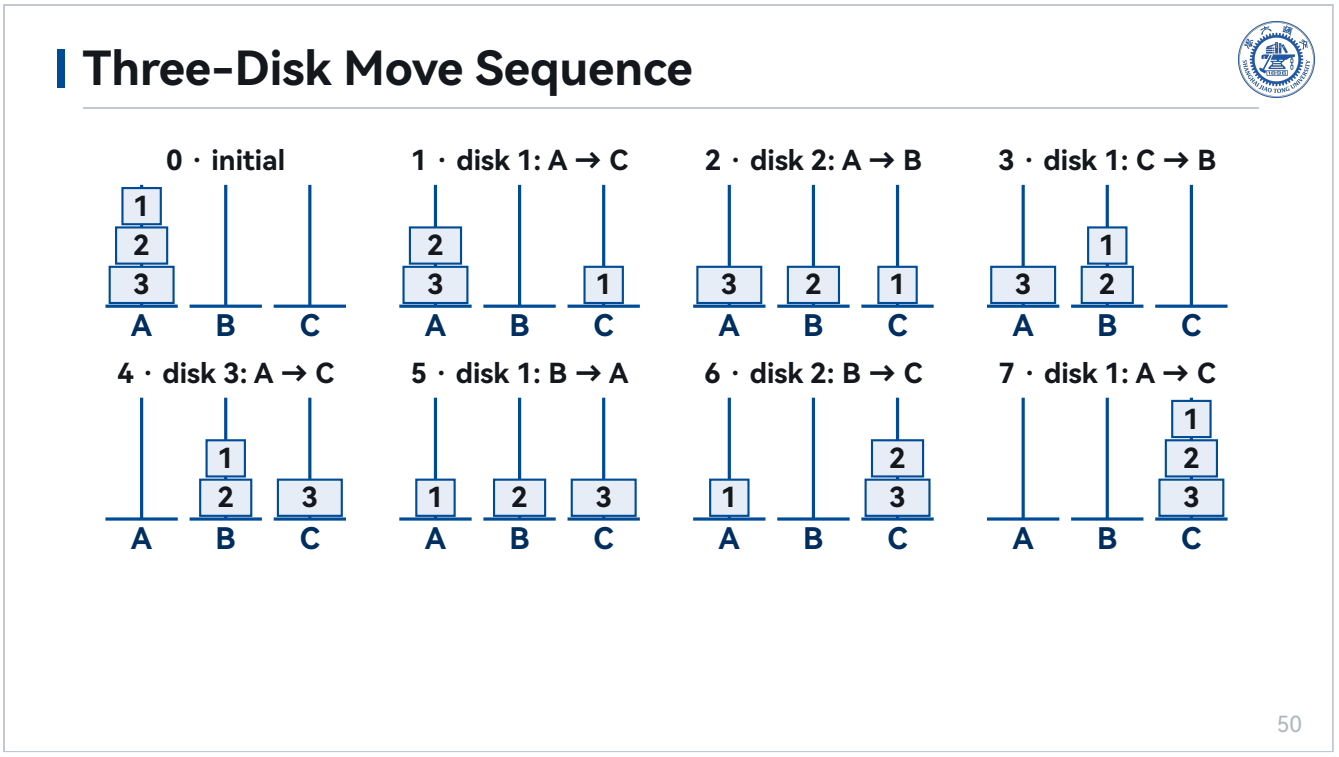
Recursive Hanoi Instructions

```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(3, "A", "C", "B")
```


第 49 页 · Recursive Hanoi Instructions

> n 为 0 时没有盘子，直接 return，不输出指令。非零时，先递归转移上面 n - 1 个盘，再打印最大盘的移动，最后递归完成另一段转移。
>
> 两个子调用都减少盘数，因此有限非负整数输入能够到达基准。这个版本的作用是打印指令，并没有维护三根柱子的列表状态。输出正确与实际执行移动需要分别验证。



第 50 页 · Three-Disk Move Sequence

> 图包含初态和七次移动后的状态，共八个状态。指令依次为盘 1 A 到 C、盘 2 A 到 B、盘 1 C 到 B、盘 3 A 到 C、盘 1 B 到 A、盘 2 B 到 C、盘 1 A 到 C。
>
> 各状态由同一份移动数据逐步执行得到，并检查每次移走的是顶盘，以及目标柱没有更小盘。七条指令与八个状态不要混淆；最后全部盘按大小顺序位于 C。

Hanoi Move Count 移动次数

$T(n) = 2T(n-1) + 1$ · $T(0) = 0$

- **Two** transfers of n − 1 disks + **one** move of the largest disk

Disks 盘数 n	Moves 移动次数 T(n)
1	1
2	3
3	7
4	15
5	31

$T(n) = 2^n - 1$ counts moves, not function calls. Maximum depth grows with n.

> 移动指令数满足 T(0) = 0、T(n) = 2T(n - 1) + 1，展开得到 2 的 n 次方减 1。三盘为七次移动，五盘为三十一次。
>
> 这是移动次数，不是函数调用次数。调用 hanoi(0) 也算一次函数调用，却不产生移动。最大递归深度随 n 增加，总移动量增长快得多；两种计数不能混用。最少步数的论证保留在参考区。

Output and Return Values 输出与返回值

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)

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

output

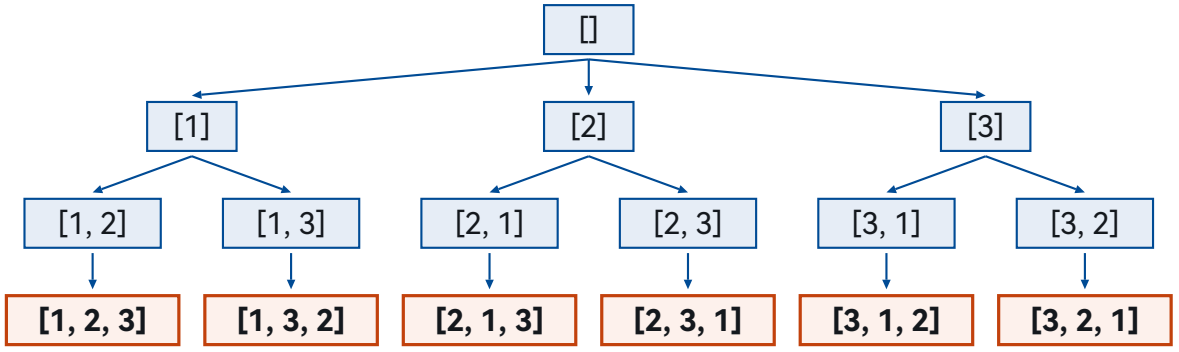
```
disk 1: A -> C
None
```


> 先打印 disk 1: A -> C，之后 print(result) 打印 None。带有 -> None 的函数仍然返回一个值 None；裸 return 和执行到函数末尾都会返回 None。
>
> 打印指令不等于修改盘子状态。要验证移动序列，需要另写状态检查器，根据指令更新柱子，并检查顶盘、大小限制和最终位置。

V · Enumeration and Backtracking 枚举与回溯

> 排列问题需要尝试多个候选选择。皇后问题也要尝试候选列，但可以提前排除已经发生冲突的部分解。
>
> 完整枚举与回溯都用递归组织分支。区别在于是否能够根据当前部分解提前判断某些分支不可能产生合法结果。

Permutation Choices 排列选择



A node records the chosen prefix; six leaves give six permutations.

> 三个互异元素共有六种排列。先选首项，首项可以是 1、2、3；每种选择留下两个元素，再递归求剩余元素的全部排列。
>
> 把选定的首项放在每个子排列前面，就得到当前分支的结果。图中六个叶子是六个完整排列。第一层有三个分支，并不意味着只产生三个结果。

Recursive Permutations

code

```python
def permutations(
    items: list[int]
) -> list[list[int]]:
    if len(items) <= 1:
        return [items.copy()]
    result = []
    for i in range(len(items)):
        rest = items[:i] + items[i + 1:]
        for p in permutations(rest):
            result.append([items[i]] + p)
    return result

p = permutations([1, 2, 3])
print(p[:2])
print(len(p))
```

output

```
[[1, 2, 3], [1, 3, 2]]
6
```


> 函数返回全部排列的列表。每次按位置选择一个首项，rest 是移除该位置后的新列表；对子排列 p 加上首项，形成当前结果。
>
> 空输入返回 [[]]，表示唯一的空排列。基准返回副本，不把输入原列表直接交给调用者。重复元素按位置枚举，可能产生相同内容的结果，本版本不自动去重。

Partial Solutions 部分解

	0	1	2	3
0		Q		
1				Q
2	Q			
3			Q	

queens = [1, 3, 0, 2]

queens[row] = column

row 0 → column 1
row 1 → column 3
row 2 → column 0
row 3 → column 2

One queen per row, column and diagonal.

queens[row] = column；zero-based indices 零基下标。

> 列表 queens 的第 r 项表示第 r 行皇后所在的列。图中的 [1, 3, 0, 2] 是一个合法的四皇后布局。逐行放置天然避免同行冲突，还需要检查同列与两条对角线。
>
> 部分解只记录已经完成的前几行。每次递归增加一行，剩余行数减少；达到 n 行时得到一个完整解。八皇后使用相同规则，结果规模更大。

Queen Placement 皇后位置检查

Same column or diagonal? 同列：c == col；对角线：距离相等

```python
def is_safe(queens: list[int], col: int) -> bool:
    row = len(queens)
    for r, c in enumerate(queens):
        if c == col or abs(row - r) == abs(col - c):
            return False
    return True
```


> 准备放置的新皇后位于 row = len(queens) 这一行，候选列为 col。enumerate 给出已放皇后的行号 r 与列号 c。列号相等意味着同列。
>
> 两格在同一对角线上，当且仅当行差绝对值等于列差绝对值。任一已有皇后发生冲突就返回 False；全部检查通过才返回 True。实际调用还需确保已有部分解本身合法。

Counting Queens Solutions

Excerpt · see complete program

```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
```

Complete runnable example 完整程序

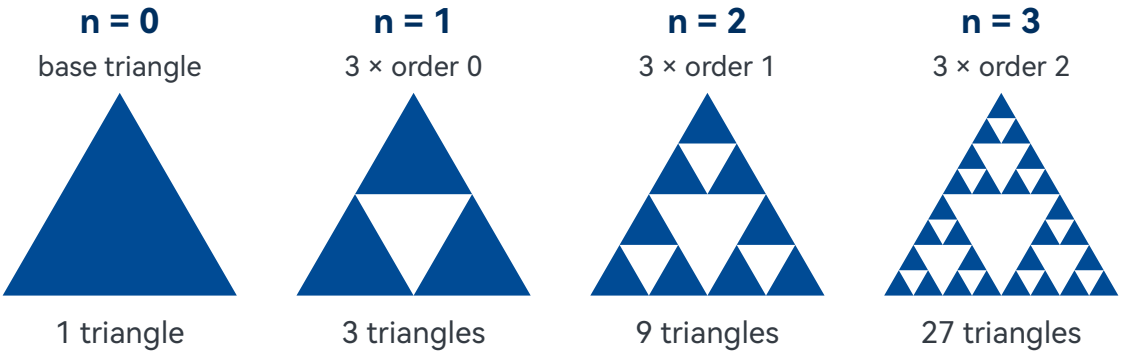
Actual output 实跑输出

```
2
92
```

- **Requires is_safe**: conflicts are checked before recursion.
- **queens + [col]** creates a new partial list.
- **Pruning 剪枝**: invalid branches stop early.

> 完整程序实跑得到四皇后 2 个解、八皇后 92 个解。当前部分解完成 n 行时返回 1；否则尝试每个列号，只对安全候选继续递归，再把各分支的解数相加。
>
> queens + [col] 创建新部分解，不修改原列表。本页是依赖 is_safe 的静态片段，完整讲义入口包含两个函数和调用示例，不要求先运行其他页面。

Self-Similar Fractals 自相似分形



Three copies of the previous order, each at half scale.

> 谢尔宾斯基三角形每一阶由三份上一阶结构组成，每份边长缩小一半。本讲使用文本程序构造近似图，阶数从 0 到 3。
>
> 递归子任务返回上一阶字符串列表，当前层把它们拼成上方一份、下方两份。分形并不都严格由完全相同的缩小副本组成；这里展示的是这种自相似结构的具体例子。完整代码和计数放在参考页。

Recursion Design and Choice

1	**Define 设计**	**Task → base case → progress → combination**
2	**Match the structure**	**Nested or branching tasks** may suit recursion.
3	**Check repeated work**	Identify **repeated subproblems** and opportunities for reuse.
4	**Check scale and state**	**Depth, input size, state and storage** all matter.

> Sequential accumulation often suits a loop; other conversions may require an **explicit stack**.

第 60 页 · Recursion Design and Choice

设计递归时明确函数任务、基准、进展与组合。选择实现方式时，还需要看任务是否自然包含层级或分支，是否重复求同一个子问题，深度是否适合输入规模，以及要保存多少状态。

顺次累加通常用循环更直接。递归可以改为显式维护状态的循环，但可能需要栈或更复杂的数据结构，不能承诺把函数调用简单换成一个 for 就足够。

Quiz 2 · Recursive Reasoning

扫码作答 · 四题

1 基准情形能否到达
2 每次调用的局部参数
3 同一子问题的重复调用
4 汉诺塔的三个任务角色

All four topics have been covered. Submit once.

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

> 第二轮四题分别检查终止性、局部参数、结果复用和汉诺塔角色。全部内容已经讲过，不再次提交开场题。
>
> 每题先根据规则推理，再用代码或图核对。基准写在代码里不代表可达，内层同名参数不会覆盖外层参数，两个相同表达式也不会自动合并成一次函数调用。

Quiz Review: Termination and Local State

Question 题目	Reasoning 推理	Outcome 结果
Q2 · reachability	`3 → 1 → −1 → −3 →` … Zero is never reached.	**RecursionError**
Q3 · local n	`total(0) → 0` `total(1) → 1 + 0` `total(2) → 2 + 1`	**0 → 1 → 3**

> A decreasing number may skip the base case. Each call keeps **its own local n**.

> 第二题从 3 开始，每次减 2，不会等于 0，默认环境最终报 RecursionError。数值减少与基准可达是两项不同的检查。
>
> 第三题 total(0) 返回 0，total(1) 用自己保留的 n = 1 得到 1，total(2) 用 n = 2 得到 3。返回之后恢复的是调用者的执行过程，不是用内层参数覆盖外层参数。

Quiz Review: Reuse and Hanoi Roles

Q4 · result reuse

```
half = power(a, n // 2)
return half * half

# Reuse a value, not two calls
```

- **One calculation, two uses** of its value

Q5 · first Hanoi subtask

```
hanoi(3, "A", "C", "B")

# Before moving disk 3:
hanoi(2, "A", "B", "C")
```

- **A → B**: move the two smaller disks first.
- **C** is the spare pillar in this subtask.

第 63 页 · Quiz Review: Reuse and Hanoi Roles

> 第四题把同一个 power 调用写两次，得到相同结果，却执行两次子问题。half 保存已经计算出的结果，乘法只复用这个值。
>
> 第五题先把上面两个盘从当前起点 A 转移到当前备用柱 B，此时 C 成为子任务的备用柱，所以第一调用是 hanoi(2, A, B, C)。先说明子任务，再对应参数。

Lab 7 · Recursion and Tools

Part 实验段	Learning tasks 任务	Checks 核对
A · recursion	Digits, reversal, palindrome, flatten, digit sum and coin combinations	**Zero / empty** inputs; recursive and loop digit sums
B · Josephus	Simulate and recurse 约瑟夫环	**Small cases** and numbering conventions
C · tools	`grep` / `find` / `wc` `sort` / `uniq` / `pipes`	**Search and summarize**; submit operation records

> Each code or explanation goes to its own entry. Large digit inputs also require **depth checks**.

A 段有六个递归学习任务，其中数位和要求同时写递归与循环版本，并在文字卡比较结果。位数、反转、回文、压平与硬币组合也都有各自入口。

B 段用约瑟夫环比较模拟与递推，核对编号约定和小输入。C 段先练习 grep、find、wc、sort、uniq 和管道，再提交操作记录。每个入口的代码和说明按实验页要求填写，不把文字回答写进别题。10**1000 有 1001 位，递归版本可能超过当前深度上限。

Solo Assignment 1 · Restricted Hanoi

Movement rule 规则	Permitted moves 移动	End-to-end count 次数
Ordinary Hanoi	Any pair of pillars	**$2^n - 1$**
Adjacent pegs only	**A ↔ B ↔ C**; no direct A ↔ C	**$3^n - 1$**

- **Move lists**, not just a count
- **Legality and final-state checks**; explain the recurrence

Course grade: **5%** · due **10 Nov, 23:59 GMT+8**.

个人作业一包含普通汉诺塔与相邻杆限制下的转移。变体只能 A 与 B、B 与 C 之间直接移动，A 到 C 不能一步完成。两种规则下都需要保存并验证移动序列。

提交要求包括代码、合法性和终态检查，以及变体最少步数的解释，不能只提交一个公式。整份作业占总成绩的 5%，与另外两次个人作业合计 15%；截止时间从网站统一校历读取。具体输入范围、函数名和各题分值以实验页为准。

Code Review and Verification

- **Same task, smaller or nested input**: recursive decomposition
- **Separate local state**: each call keeps its own variables
- **Combination**: returned results or ordered actions

L08 · Code Review and Verification

- **Input assumptions → base cases → progress → results and state changes**

> How can these claims be checked against a program?

> 回到开场的嵌套列表：每一层使用相同的逐项处理任务，遇到内层列表就再次调用，能够处理有限、没有循环引用的任意嵌套深度，但实际深度仍受解释器限制。
>
> 一个程序看起来合理，还需要检查输入前提、基准、终止、返回值和状态变化。下一讲从这些具体检查进入代码审查与验证。下面的参考页供课后查阅，不要求在课堂剩余时间全部讲完。

AFTER CLASS

Reference 参考

> 参考区保留完整追踪、程序接口、核验方法以及往年扩展知识。课堂主线到上一页结束，下面按学习需要查阅。
>
> 标有 Excerpt 的页面展示片段，不是独立运行入口。对应讲义的完整程序包含所有定义和调用。图、输出和数值均经过实际运行核对。

Factorial Trace

Complete output 完整输出

```
enter 3
  enter 2
    enter 1
      enter 0
      return 1
    return 1
  return 2
return 6
6
```

- **enter**: begin a new call
- **return**: finish that call
- **Indentation**: one level deeper per indent
- **Returned values**: 1, 1, 2, 6
- **Last 6**: the outer print()

> 完整输出先显示四次进入，再显示四次返回，最后打印整个函数调用的结果。enter 0 后立刻 return 1，随后才轮到 n 为 1 的调用继续执行。
>
> 两行 return 1 属于不同调用：第一行是零的阶乘，第二行是一的阶乘。输出中的缩进对应 depth，图中的调用帧帮助核对这些局部状态。

Explicit Stack Traversal 显式栈遍历

code

```python
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

print(sum_stack([1, [2, 3], [4, [5]]]))
```

output

```
15
```


> pending 保存尚未处理的元素，每次 pop 取出末尾一项。整数累加，列表则把其元素加入 pending。输入仍限定为整数与有限无环列表。
>
> 这是把递归中的待处理状态显式存放到列表中的一种写法。由于从末尾取项，遍历次序与前面的递归版本不同；整数总和不受次序影响，但有顺序要求的任务不能直接套用。原输入不修改。

Binary Search: Complete Interface

Excerpt · see complete program

```python
def binary_search(
    nums: list[int], target: int
) -> int:
    hi = len(nums) - 1
    return search_range(
        nums, target, 0, hi)

# search_range: page 38
```

Actual output 实跑输出

```
[] 1 -1
[3] 3 0
[3] 2 -1
[2, 2, 2] 2 1
```

- **Requires search_range**; the complete example defines both functions.
- **Sorted input** is required.
- **Any matching index** is allowed for duplicates.

Complete runnable example 完整程序

> 公开接口 binary_search 负责建立初始闭区间，辅助函数 search_range 负责递归。讲义的完整入口同时包含两个定义和这些边界测试，不依赖先运行第 38 页。
>
> 空输入未命中；单元素输入能分别处理命中和缺失；重复值返回一个正确下标。这些测试不能证明全部正确性，但能发现常见的空区间、下标和接口问题。

Merge: Comparison Loop

Code / excerpt 代码或片段

```
def merge(
    a: list[int], b: list[int]
) -> list[int]:
    result = []
    i = 0
    j = 0
    while (i < len(a)
           and j < len(b)):
        if a[i] <= b[j]:
            result.append(a[i])
            i += 1
        else:
            result.append(b[j])
            j += 1
```

- **Sorted prefix**: result stays sorted.
- **Consumed**: a[:i] and b[:j].
- **Next candidates**: a[i] and b[j].
- **Choose the smaller** unused element.

第 71 页 · Merge: Comparison Loop

> 比较循环中，result 始终有序，a 的前 i 项与 b 的前 j 项已经使用。两个输入本身有序，所以各自下一个元素是这一侧剩余元素的最小值。
>
> 选取两侧较小值不会跳过更小候选，因此加入 result 后仍有序。这里只说明主循环；耗尽一侧后的处理和递归排序在完整示例中衔接。

Merge Sort: Complete Example

Excerpt · see complete program

```
    # End of merge:
    result.extend(a[i:])
    result.extend(b[j:])
    return result

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, 2, 3, 3, 4, 5, 7]
```

- **Complete merge** is required; this page shows its ending.
- **Empty → []**; duplicates are retained.
- **Inputs unchanged**; copies, slices and result lists use storage.

> merge 主循环结束后，最多一侧仍有剩余元素。接上两段切片并返回，然后由 merge_sort 在递归结果上调用它。完整依赖较长，因此本页继续使用明确标注的片段。
>
> 讲义提供一个独立入口，可检查空列表、重复值、逆序输入与原列表是否保持不变。基准复制、切片和 result 都会创建列表，需要考虑相应空间。

Permutation Verification 排列结果核验

Excerpt · see complete program

```python
from itertools import (
    permutations as ref_perm)

items = [1, 1, 2]
mine = permutations(items)
ref = [list(p)
       for p in ref_perm(items)]
print(sorted(mine) == sorted(ref))

# Requires our permutations.
```

Actual output 实跑输出

```
[] True
[1] True
[1, 2, 3] True
[1, 1, 2] True
```

- **Requires our permutations** from the complete example.
- **Convert tuples to lists** before comparison.
- **Keep multiplicities**: a set alone loses repeated results.

Complete runnable example 完整程序

第 73 页 · Permutation Verification 排列结果核验

> 核验时真正调用自写 permutations，而不是把某个预先写好的结果当作程序输出。标准库的排列结果是元组，这里先转成列表，才与自写返回类型一致。
>
> 互异输入可以比较集合和数量。重复输入不能只转成集合，否则会丢失重复项数量。排序后的完整列表比较会保留重数，空输入也得到一个空排列。

Coin Combinations 硬币组合

Branch 分支	Subtask 子任务	Progress 终止度量
Use current coin	`ways(amount - coin, index)` Requires amount ≥ coin; reuse is allowed.	**Amount decreases**
Skip current coin	`ways(amount, index + 1)` Do not return to earlier coins; order is ignored.	**Fewer denominations remain**

- **amount = 0 → one combination**; positive amount with no coins → zero
- **Distinct positive denominations**; nonnegative amount

> 硬币组合可以分成两类：至少使用当前面额，或完全不使用当前面额。前者减少金额，仍允许使用同一面额；后者进入下一种面额，不再返回之前的面额。
>
> 金额为 0 时只有一种组合，即不再选币；正金额却没有可用面额时为零种。面额必须是互异正整数，金额非负。这里解释分解结构，完整实现由实验独立完成，不公开实验答案。

Queens: Search and Verification

Method 方法	Candidates 候选	Conflict check 冲突检查
Partial search 回溯	Extend a legal prefix	**Before recursion**; new lists need no in-place undo
Full enumeration 枚举	**8! = 40320** column permutations	Rows/columns distinct; check **every diagonal pair**

n	4	5	6	7	8
Solutions 解数	**2**	**10**	**4**	**40**	**92**

Both complete checks give **92** eight-queens arrangements.

主线方法逐行扩展，并在递归之前排除同列和对角线冲突。由于构造新列表，不需要原地添加后再撤销；原地回溯则必须正确恢复状态。

往年另一种方法先枚举列号 0 到 7 的全部排列，共 40320 个候选，再检查任意两行皇后是否位于同一对角线，最终也是 92 个解。列号排列已经保证不同列，但不能保证没有对角线冲突。完整核验代码在讲义。

Text Sierpinski Triangle

Code / excerpt 代码或片段

```python
def sierpinski(
    n: int
) -> list[str]:
    if n == 0:
        return ["*"]
    smaller = sierpinski(n - 1)
    pad = " " * (2 ** (n - 1))
    top = []
    bottom = []
    for line in smaller:
        top.append(
            pad + line + pad)
        doubled = line + " " + line
        bottom.append(doubled)
    return top + bottom
```

- **n = 0**: one star
- **smaller**: solve the previous order once
- **Top**: one centered subtriangle
- **Bottom**: two subtriangles
- **Result**: a list of text lines

Complete runnable example 完整程序

> 函数返回字符串列表，每个字符串是一行。零阶只有一个星号。得到上一阶 smaller 后，上半部逐行在两端补空格，下半部把相同行的两个副本用一个空格连接。
>
> 较长完整入口还包括逐行打印与计数测试，因此本页标作片段并链接讲义。字符串中的空格决定图形宽度，不能把显示时的换行或字体比例误当成算法变化。

Sierpinski Output and Counts

n = 3 · actual output

```
        *
       * *
      *   *
     * * * *
    *       *
   * *     * *
  *   *   *   *
 * * * * * * * *
```

n	Rows 行数	Stars 星号
0	1	1
1	2	3
2	4	9
3	8	27
4	16	81

Rows: 2^n · Stars: 3^n

> 零阶为一行一个星号，每升一阶，行数变成两倍，星号数变成三倍。三阶有八行、二十七个星号。右侧计数对每个实际生成的字符串进行统计。
>
> 文本显示使用等宽字体。程序构造的是离散字符近似，不能据此说有限阶文本图具有无限分形的全部几何性质。

Hanoi: Minimum Move Count

1	**Before largest disk**	Transfer all **n − 1 smaller disks** to the spare pillar.
2	**Largest disk**	At least **one move** to the target.
3	**After largest disk**	Transfer the **n − 1 smaller disks** onto it.

Lower bound 下界	Construction 构造达到
$T(n) \geq 2T(n-1) + 1$	$T(0) = 0;\ T(n) = 2^n - 1$

> The recursive algorithm uses exactly these stages. Extra largest-disk transfers cannot reduce the bound.

第 78 页 · Hanoi: Minimum Move Count

普通汉诺塔中，最大盘直接从起点移到终点之前，上面 n - 1 个盘必须全部位于备用柱；最大盘移到终点之后，还要将它们移到最大盘上。这样的解至少需要两次 n - 1 盘转移和一次最大盘移动。

若最大盘经备用柱分两次到达终点，会迫使小盘组额外转移，不会减少下界。递归构造达到两次子任务加一次移动的下界，因此给出最少步数 $2^n - 1$。

Input Size 输入规模与递归上限

`10**1000` has **1001 digits** · a valid finite input

- **One digit per call** may exceed the current depth limit.
- **Accepted input sizes** must be documented.
- **Suitable loop or stack**: consider the scale of the task.

> Current build limit: **1000**. 实验须比较输入规模与实现限制，不以调高递归上限代替分析。

> 10 的 1000 次方是 1 后面跟 1000 个零，共 1001 位，不是 1000 位。每次删除一位的递归算法虽然最终能到达基准，调用深度仍可能超过当前环境限制。
>
> 输入域与实现能够处理的实际规模需要分别说明。递归正确性不能替代规模检查。实验应比较递归与循环的边界，而不是通过任意调高递归上限掩盖问题。

Operation Costs 运算次数与整数大小

power(2, 10) · exponent chain: 10 → 5 → 2 → 1 → 0

Exponent 指数	10	5	2	1
Multiplications 乘法	1	2	1	2

- **Six multiplications** across four non-base calls
- **power(2, 100)** · actual result:

Actual integer result 实际整数值

```
1267650600228229401496703205376
```

> Fewer arithmetic operations do not make each operation fixed-time; **integer size also grows**.

> 指数链 10、5、2、1、0 中，偶数层各一次乘法，奇数层各两次，总共六次。这里数的是代码中的乘法操作，不是所有函数调用或机器指令。
>
> 右侧是 2^{100} 的完整真实值。Python 整数可以增长，但大整数乘法的时间和存储随位数变化。指数减半带来的是算术操作次数减少，不能进一步声称实际总耗时只取决于递归层数。

Binary Representation 二进制表示

code

```python
def binary_text(n: int) -> str:
    if n < 2:
        return str(n)
    return binary_text(n // 2) + str(n % 2)

for n in [0, 1, 2, 5, 13]:
    print(n, binary_text(n))
```

output

```
0 0
1 1
2 10
5 101
13 1101
```


> 对非负整数不断除以 2，余数给出当前最低位。先递归生成商的二进制表示，再接上当前余数，得到从高位到低位的字符串。
>
> n 为 0 或 1 时直接返回对应单字符。零的表示是字符串 "0"，不是空字符串。这个版本展示整数分解与字符串组合，不处理负数。

Counting Set Bits 二进制中 1 的个数

code

```python
def count_bits(n: int) -> int:
    if n == 0:
        return 0
    return count_bits(n // 2) + n % 2

for n in [0, 1, 4, 5, 7]:
    print(n, count_bits(n))
```

output

```
0 0
1 1
4 1
5 2
7 3
```


> 二进制中的一位等于 n % 2，其他更高位对应 n // 2。把各位的 0 或 1 相加，就得到二进制中 1 的数量。
>
> 整数 4 的二进制是 100，长度为 3，但只有一个 1。零得到计数 0，和二进制字符串 "0" 的长度 1 不是同一个任务。输入仍限定为非负整数。

Decimal Digits and Reversal

Decomposition 整数分解	Meaning 含义
n % 10	Final digit 末位
n // 10	Remaining prefix 前缀

- **Digits(n) = 1** for 0 ≤ n < 10
- **Digits(n) = 1 + Digits(n // 10)** for n ≥ 10
- **120 → 21** as an integer; string reversal can preserve the leading zero.

> 比较整数与字符串逆序的结果；同时检查反复统计前缀位数带来的额外计算。

> 十进制整数可以拆成末位 n % 10 与剩余前缀 n // 10。0 到 9 都只有一位，位数的基准应覆盖这十个输入；零输入的位数也为 1。
>
> 按整数逆序时，120 的结果是 21，前导零不能作为整数的一部分保留；按字符串逆序则可以保留字符 0。往年的算术逆序在每层重新计算前缀位数，会产生额外工作。这里保留任务差别与递推关系，实验实现由学生完成。

Digit Sum 数位和

Code / excerpt 代码或片段

```
# String traversal:
for ch in str(n):
    total += int(ch)

# Arithmetic loop:
while n > 0:
    total += n % 10
    n //= 10
```

- **Recursive decomposition** for nonnegative integers:
- **sum_digits(0) = 0**
- **sum_digits(n) = n % 10 + sum_digits(n // 10)**
- **Compare the same outputs** across both loops and recursion.
- **Large inputs** require a depth check.

循环片段须补齐 total 初始化与完整上下文；实验实现独立完成。

> 字符串循环把每个字符转成整数并累加；算术循环反复取末位并更新商；递归按相同的商与余数关系组合结果。三种方法对非负整数应该返回相同数位和。
>
> 两段循环标作核心片段，需要初始化 total 等完整上下文，不能单独运行。零输入和很长整数都要检查，递归还受深度限制。实验要求独立完成递归与循环函数，并记录验证结果，不直接从课件复制完整答案。

A Third-Order Recurrence 三阶递推

code

```python
def third_order(n: int) -> int:
    if n <= 3:
        return n
    return (3 * third_order(n - 1)
            - 2 * third_order(n - 2)
            + third_order(n - 3))

values = [third_order(n)
          for n in range(1, 9)]
print(values)
```

output

```
[1, 2, 3, 6, 14, 33, 77, 179]
```


> 往年原式为 $a_1 = 1$、$a_2 = 2$、$a_3 = 3$，其他项由 3 倍前一项减 2 倍前两项再加前三项得到。这里的输入为正整数，前三个输入直接返回。
>
> 前八项实际是 1、2、3、6、14、33、77、179。不能把这条式子改写成三个前项相加；系数和符号决定了完全不同的序列。这个朴素版本也会重复计算子问题。

A Two-Dimensional Recurrence 二维递推

code

```python
def grid_value(m: int, n: int) -> int:
    if m == 0:
        return 1
    if n == 0:
        return m
    return (grid_value(m - 1, n)
            + grid_value(m, n - 1)
            - grid_value(m - 1, n - 1))

for m in range(5):
    print([grid_value(m, n) for n in range(5)])
```

output

```
[1, 1, 1, 1, 1]
[1, 1, 1, 1, 1]
[2, 2, 2, 2, 2]
[3, 3, 3, 3, 3]
[4, 4, 4, 4, 4]
```


> 旧材料同时写 f(0, n) = 1 和 f(m, 0) = m，原点会分别得到 1 和 0，存在冲突。这里明确采用修订：第一边界适用于 n ≥ 0，第二边界只适用于 m ≥ 1。
>
> 因此 f(0, 0) 取 1，代码先检查 m == 0。递推仅适用于 m、n 都为正的输入。本页采用修订后的约定，不声称原题本来就没有歧义。

Ackermann Function 阿克曼函数

code

```python
def ackermann(m: int, n: int) -> int:
    if m == 0:
        return n + 1
    if n == 0:
        return ackermann(m - 1, 1)
    return ackermann(m - 1, ackermann(m, n - 1))

print(ackermann(0, 1))
print(ackermann(1, 1))
print(ackermann(2, 2))
```

output

```
2
3
7
```


> Ackermann 函数展示一种增长很快的递归定义。m 为 0 时返回 n + 1；否则根据 n 是否为 0 使用两种递归关系。输入约定为非负整数。
>
> 这里只运行三个小输入，结果为 2、3、7。A(4, 2) 的结果可用 2**65536 - 3 表示，约有 19729 位，但本页没有执行这个巨大递归求值，也不提高递归上限。数学定义能够给出结果，不表示朴素递归实现能够实际算完。

Turtle Drawing 绘图基本操作

Code / excerpt 代码或片段

```python
import turtle

pen = turtle.Turtle()
pen.forward(80)
pen.left(90)
pen.forward(40)
turtle.done()
```

- **forward(d)**: move along the current heading.
- **left(a)**: turn anticlockwise by a degrees.
- **Start**: (0, 0), heading 0°
- **After forward(80)**: (80, 0)
- **After left(90), forward(40)**: (80, 40), heading 90°

Local Python only: requires a graphics window; no browser-run button.

> turtle 创建画笔对象。forward 按当前方向前进指定距离，left 按角度逆时针转向；位置与方向是两个不同状态。默认起点为原点，方向向右。
>
> 先前进 80，再左转 90 度并前进 40，最终位置为 (80, 40)，方向为 90 度。这个例子需要本机图形窗口，不提供浏览器运行按钮。完整本机程序和依赖说明见讲义。

Recursive Star Drawing 递归五角星

Code / excerpt 代码或片段

```python
def star(pen, size: float) -> None:
    if size <= 10:
        return
    for i in range(5):
        pen.forward(size)
        star(pen, size / 3)
        pen.left(216)

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

- **Size chain**: 360 → 120 → 40 → 13⅓ → stop
- **Five segments** per non-base call
- **780 drawn segments**; **781 calls** including base cases
- **Position and heading** return to the starting state.
- **Pen stub verified**; not a GUI screenshot.

第 89 页 · Recursive Star Drawing 递归五角星

> 往年第 25 页是递归五角星。每个非基准调用重复五次：前进 size，递归绘制 size / 3 的子图，左转 216 度。小于等于 10 时停止。
>
> size 为 360 时，共绘制 780 条线段，调用 781 次，最终位置和朝向恢复。轨迹计数由替身画笔核验，没有冒充实际打开 GUI 的结果。这不是递归树，也不是旧圆形自相似照片对应的绘图程序。

Permutations by Insertion 插入法排列

code

```python
def insert_perms(n: int) -> list[tuple[int, ...]]:
    if n == 0:
        return [()]
    result = []
    for p in insert_perms(n - 1):
        for i in range(n):
            result.append(p[:i] + (n,) + p[i:])
    return result

for p in insert_perms(3):
    print(p)
```

output

```
(3, 2, 1)
(2, 3, 1)
(2, 1, 3)
(3, 1, 2)
(1, 3, 2)
(1, 2, 3)
```


> 这是一种与主线不同的分解：先得到 1 到 n - 1 的全部排列，再把 n 插入每个可能位置。零元素仍有唯一空排列，用空元组表示。
>
> 三元素实际输出六个排列，顺序不同于主线按首项枚举。结果集合相同不表示遍历次序相同。这里保留插入方法，帮助区分算法任务与输出顺序。

Lexicographic Permutations 字典序排列

1	**Input 输入**	**Distinct, sorted** items
2	**Branch order**	Choose the first item **in order**.
3	**Subproblem**	Recurse on the **ordered remainder**.

Actual lexicographic order 实际字典序

```
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

Compare sequences at the **first differing position**. Not every result is internally increasing.

> 字典序按第一个不同位置比较序列。对互异且升序的输入，主线按下标顺序选择首项、再递归保留剩余顺序，就按字典序输出排列。
>
> 旧标题 Increasing permutations 容易误解，这里明确为字典序全排列，不是每个结果内部都严格递增的组合。输入未排序或包含重复值时，不能不加条件地套用这里的次序结论。

Partial Permutations and Combinations

Order matters 排列

```python
from itertools import (
    permutations)

items = [1, 2, 3]
for p in permutations(items, 2):
    print(p)
```

output

```
(1, 2)
(1, 3)
(2, 1)
(2, 3)
(3, 1)
(3, 2)
```

Order ignored 组合

```python
from itertools import (
    combinations)

items = [1, 2, 3]
for c in combinations(items, 2):
    print(c)
```

output

```
(1, 2)
(1, 3)
(2, 3)
```


> 从三个互异元素取两个，排列计顺序，因此 (1, 2) 与 (2, 1) 是两个结果，共六个。组合不计顺序，只保留一个代表，共三个。
>
> itertools 中两个接口名称分别是 permutations 和 combinations。排列与组合的递归扩展可以根据是否允许返回较早位置来组织分支，不能只对排列输出排序或去重就当成原算法。

Displaying Nested Results with pprint

Excerpt · display only

```python
from pprint import pprint

pprint(permutations([1, 2, 3]),
       width=20)

# Complete definition in the notes.
```

Actual formatted output 实际排版输出

```
[[1, 2, 3],
 [1, 3, 2],
 [2, 1, 3],
 [2, 3, 1],
 [3, 1, 2],
 [3, 2, 1]]
```

- **Display only**: no sorting, deduplication or new permutations
- **width=20** guides formatting; it is not a hard truncation limit.

> pprint 来自标准库，按结构与宽度对较长对象进行排版。本页在完整 permutations 定义上实际调用，得到右侧输出。
>
> pprint 只改变显示，不改变排列顺序，不去重，也不计算新的排列。宽度参数不是必须遵守的硬截断；过长单个元素仍可能超出指定宽度。浏览器显示与程序内容需要分别理解。

Fibonacci with an Array

code

```python
def fib_array(n: int) -> int:
    if n == 0:
        return 0
    values = [0] * (n + 1)
    values[1] = 1
    for i in range(2, n + 1):
        values[i] = values[i - 1] + values[i - 2]
    return values[n]

print(fib_array(0))
print(fib_array(20))
```

output

```
0
6765
```


> 循环按下标从小到大计算 Fibonacci 项，values 保存全部已经得到的结果。n 为 0 时必须先返回，否则只有一项的列表访问 values[1] 会越界。
>
> 本版修正往年数组实现的零输入问题，实际结果是 0 和 6765。保存数组避免重复计算，也保存了不再用于最终答案的早期项；只需最终值时可以减少状态。

Fibonacci with Two Variables

code

```
def fib_pair(n: int) -> int:
    previous = 0
    current = 1
    for i in range(n):
        previous, current = current, previous + current
    return previous

print(fib_pair(0))
print(fib_pair(20))
```

output

```
0
6765
```


> 循环开始前 previous 是 F(0)，current 是 F(1)。完成一次更新后，它们分别是 F(1) 与 F(2)。执行 n 次后，previous 为 F(n)。
>
> 右侧表达式先计算，再把两个结果分别绑定到左侧名字，因此加法使用更新前的两个值。n 等于 0 时循环不执行，返回 0。往年双变量 for 版本本来正确，这里换名解释状态，不把它写成旧错误。

Loop State Updates

Recursive parameters 递归参数

```python
def gcd(a: int, b: int) -> int:
    if b == 0:
        return a
    return gcd(b, a % b)

print(gcd(48, 18))
```

output

```
6
```

Loop state 循环状态

```python
def gcd_loop(
    a: int, b: int
) -> int:
    while b != 0:
        a, b = b, a % b
    return a

print(gcd_loop(48, 18))
```

output

```
6
```


> 递归版本把下一组 (b, a % b) 交给新的调用，循环版本在同一次调用里更新 a 和 b。两段都得到最大公约数 6，终止依据也相同：非零第二参数对应的余数严格减小。
>
> while 中不能漏掉状态更新。循环仍要进行判断、取余和保存变量；递归还增加调用管理与深度要求。能改为循环不表示所有任务都能用这两个变量表示全部状态。

Adjacent-Peg Hanoi 相邻杆汉诺塔

A ↔ B ↔ C · no direct A ↔ C move

Step 步骤	Transfer 移动任务
1 · smaller disks	Transfer **n − 1** disks A → C.
2 · largest disk	Move disk n: **A → B**.
3 · smaller disks	Transfer **n − 1** disks C → A.
4 · largest disk	Move disk n: **B → C**.
5 · smaller disks	Transfer **n − 1** disks A → C.

n	0	1	2	3	4	5
Moves	0	2	8	26	80	242

$T(0) = 0; T(n) = 3T(n-1) + 2 = 3^n - 1$. 核验移动序列，并证明三次小盘组转移都不可省略。

97

第 97 页 · Adjacent-Peg Hanoi 相邻杆汉诺塔

要把最大盘从 A 经 B 移到 C，先把上面 n - 1 个盘从 A 转移到 C，给 A 到 B 的移动腾出位置；之后把小盘组从 C 移回 A，清空 C；最大盘 B 到 C 后，最后把小盘组从 A 移到 C。

这构成三次 n - 1 盘端点转移与两次最大盘移动，因此 $T(n) = 3T(n - 1) + 2$，零盘为 0，得到 $3^n - 1$。教师验证序列在 n = 0 到 5 满足相邻、顶盘、大小与终态要求。这里保留推导，不公开个人作业完整实现；最少步数还需说明必要的三次转移，不能只凭构造宣称最优。