数据结构

第 12 周 – 递归：把问题归结为它自身的较小实例

Koczka Ferenc 博士 · 埃斯特哈齐·卡罗伊大学 · 2026

复习

- **调用栈**（第 7 周）：每次函数调用都把一个栈帧压入栈中，返回时弹出
- **链表：**一个结点 + 链表的其余部分（它也是一个链表！）
- **二分查找：**把问题缩小一半，然后用同样的方法继续

今天的问题

- 什么是递归？它在计算机中是怎样工作的？
- 怎样写出正确的递归函数？
- 什么时候适合用递归，什么时候递归很危险？

什么是递归?

递归：函数在问题的**较小**实例上**调用自身**。

每个递归函数都由两部分组成：

1. **基本情况（递归出口）：**最简单的情况，直接求解 – 递归在这里**停止**
2. **递归情况：**把问题归结为同一问题的一个**较小**实例

阶乘： $n! = 1 \cdot 2 \cdot \ldots \cdot n$

$0! = 1$ **（基本情况）**

$n! = n \cdot (n - 1)!$ **（递归情况）**

在数学中与之对应的是**数学归纳法**：如果基本情况成立，并且由较小的情况能推出较大的情况，那么对所有情况都成立。

阶乘

伪代码

```
Function Fact(n)
  If n ≤ 1 then
    Return 1                    // 基本情况
  else
    Return n · Fact(n − 1)      // 递归情况
  End if
End function
```

Java

```java
static long fact(int n) {
    if (n <= 1) return 1;
    return n * fact(n - 1);
}
```

Go

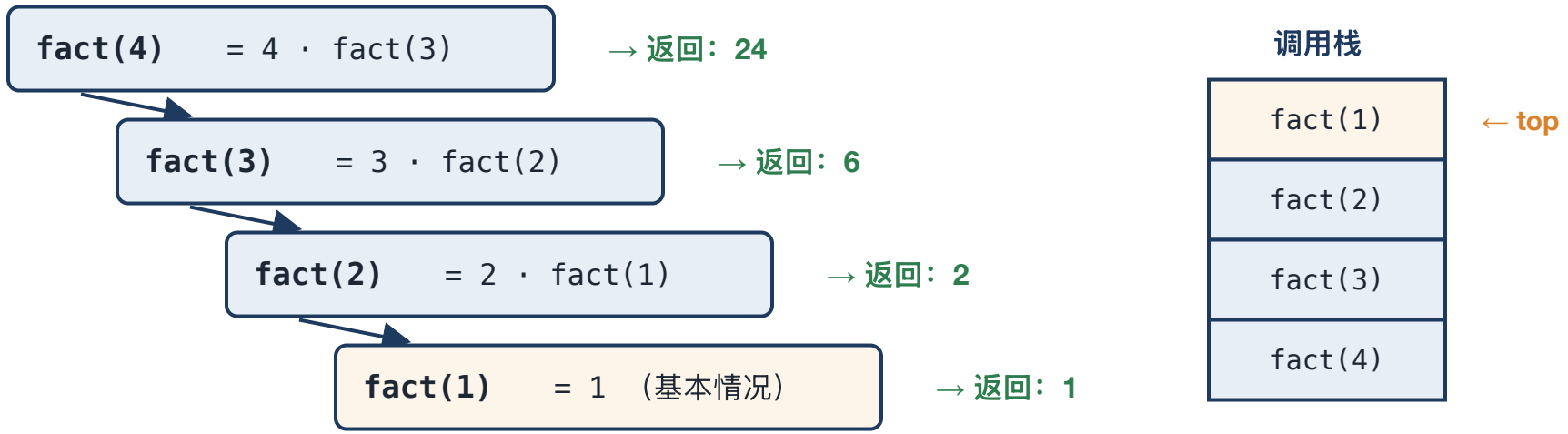
```go
func fact(n int) int64 {
    if n <= 1 {
        return 1
    }
    return int64(n) * fact(n-1)
}
```

程序的输出（`Factorial`）：

```
fact(4) called
  fact(3) called
    fact(2) called
      fact(1) called
      fact(1) = 1
    fact(2) = 2
  fact(3) = 6
fact(4) = 24
```

它是怎样工作的？调用栈

计算 fact(4)：向下调用（栈增长），向上返回（栈缩小）



- 每次调用都有自己的**栈帧**：自己的参数 **n**、自己的返回地址
- 乘法只在**返回的过程中**（倒着）执行
- 空间：**O(n)** – 栈中有 n 个栈帧

正确的递归 – 三条规则

1. **有基本情况**，它不用递归就能给出结果
2. 每次递归调用都**更接近**基本情况（更小的 n、更短的链表、更小的范围）
3. **相信递归：**假定对较小的问题调用会返回正确的结果，只需考虑怎样由它构造出较大问题的解

没有基本情况，或者不向它靠近：**无穷递归** → 栈被填满
Java: `StackOverflowError`（大约 10 000–30 000 次调用之后）· Go：栈会增长，但到 1 GB 时崩溃

递归的种类

种类	含义	例子
直接递归	f 调用自身	阶乘
间接递归	f 调用 g,g 调用 f	even(n) → odd(n – 1)
线性递归	一次递归调用	阶乘、求和、链表长度
树形（多重）递归	多次递归调用	斐波那契、汉诺塔、树的遍历
尾递归	递归调用是最后一个操作	`gcd(b, a mod b)`

尾递归可以机械地改写成循环（有些编译器会自动这样做 – Java 和 Go 不会）。

效率：斐波那契的例子

斐波那契数

定义： $F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

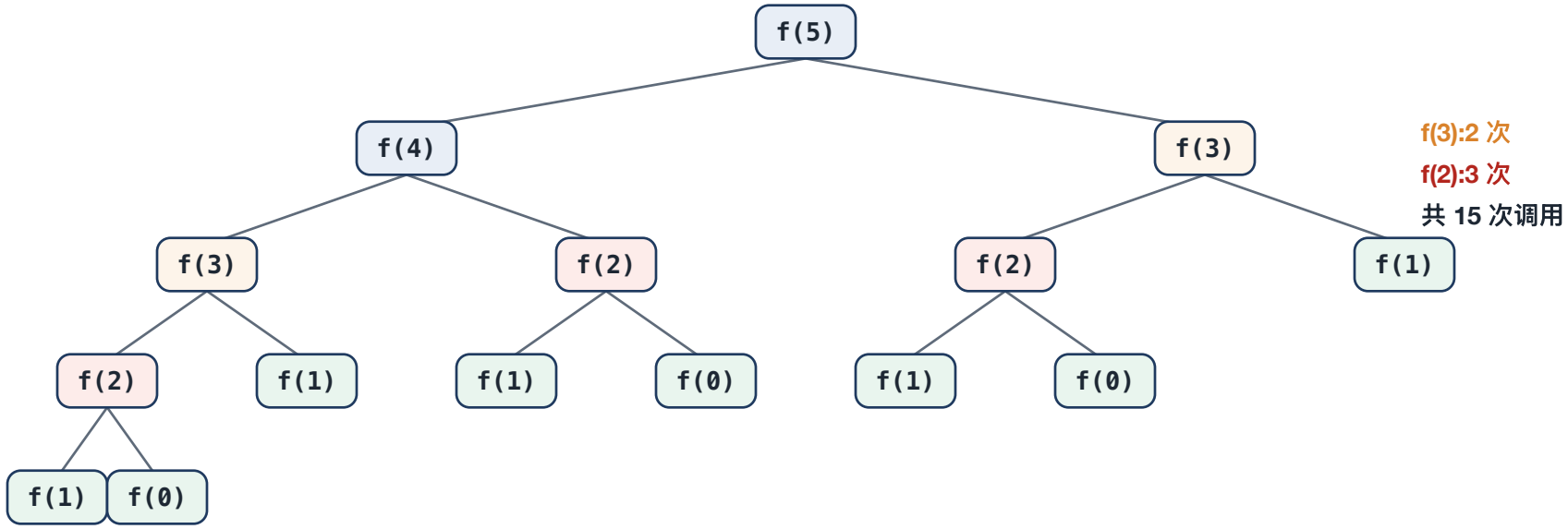
```
Function Fib(n)
  If n < 2 then
    Return n
  else
    Return Fib(n - 1) + Fib(n - 2)
  End if
End function
```

```java
static long fib(int n) {
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);
}
```

简单，而且完全遵循定义 – 但是…

朴素斐波那契的调用树

fib(5) 的调用树：相同的值被一次又一次地重复计算



实测结果

n	fib(n)	朴素调用次数	记忆化	时间（朴素）
10	55	177	19	0.0 ms
20	6 765	21 891	39	0.1 ms
30	832 040	2 692 537	59	2.6 ms
40	102 334 155	**331 160 281**	79	**320 ms**

- 朴素版本是**指数级**的：$O(\phi^n) \approx O(1.618^n)$ – n = 50 时要几分钟，n = 100 时要几百万年
- **记忆化**：把已经算出的值存放在**哈希表**中 → **O(n)**
- **迭代**：用两个变量，O(n) 时间，**O(1)** 空间

改进的版本

记忆化（Java）

```java
static HashMap<Integer, Long> memo = new HashMap<>();

static long fibMemo(int n) {
    if (n < 2) return n;
    Long cached = memo.get(n);
    if (cached != null) return cached;
    long r = fibMemo(n - 1) + fibMemo(n - 2);
    memo.put(n, r);
    return r;
}
```

迭代（Go）

```go
func fibIter(n int) int {
    a, b := 0, 1
    for i := 0; i < n; i++ {
        a, b = b, a+b
    }
    return a
}
```

记忆化是**动态规划**的基本思想：每个子问题只求解一次。

经典例子

递归思维：“第一个 + 其余部分”

数组求和：第一个元素 + 其余部分的和

```java
static int sum(int[] a, int from) {
    if (from == a.length) return 0;
    return a[from] + sum(a, from + 1);
}
```

字符串逆序：其余部分逆序 + 第一个字符

```java
static String reverse(String s) {
    if (s.isEmpty()) return s;
    return reverse(s.substring(1)) + s.charAt(0);
}
```

回文：第一个 = 最后一个，并且中间部分也是回文

```go
func isPalindrome(r []rune) bool {
    if len(r) < 2 {
        return true
    }
    return r[0] == r[len(r)-1] &&
        isPalindrome(r[1:len(r)-1])
}
```

链表长度：1 + 其余部分的长度

```go
func length(p *Node) int {
    if p == nil {
        return 0
    }
    return 1 + length(p.Next)
}
```

快速幂 – 分治

计算 x^{13}，而不用 12 次乘法：

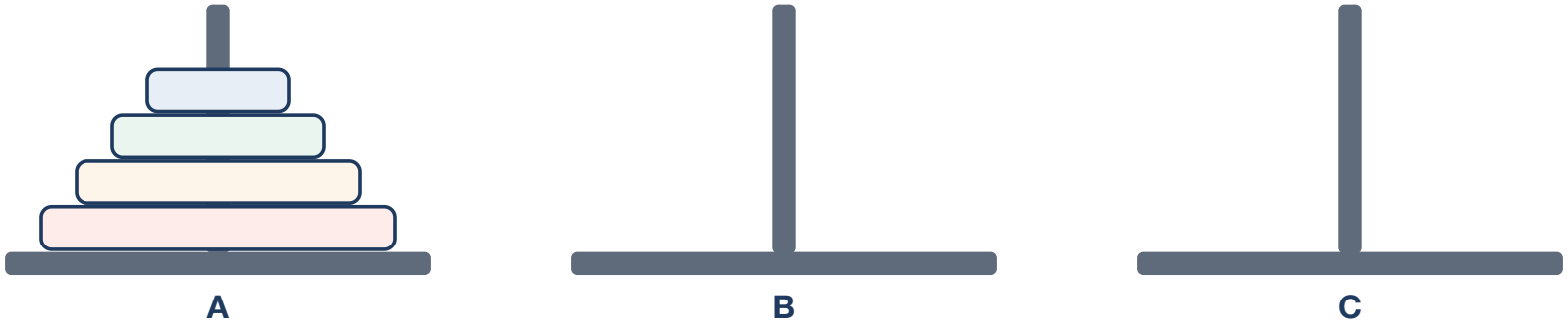
- $x^{13} = (x^6)^2 \cdot x$　$x^6 = (x^3)^2$　$x^3 = (x^1)^2 \cdot x$　→ 总共 **5** 次乘法

```
Function Power(x, n)
  If n = 0 then Return 1
  half ← Power(x, n div 2)
  If n mod 2 = 0 then Return half · half
  else Return half · half · x
End function
```

- 每次调用都把 n **减半** → **O(log n)** 次调用（和二分查找一样）
- 同样的模式：**递归的二分查找**、归并排序、快速排序（第 13 周）

汉诺塔

汉诺塔：把 n 个盘子从 A 移到 C，大盘不能放在小盘上




```
Procedure Hanoi(n, from, to, via)
  If n > 0 then
    Hanoi(n − 1, from, via, to)
    Move(from, to)
    Hanoi(n − 1, via, to, from)
  End if
End procedure
```

- 移动次数：$\mathbf{2^n - 1} \rightarrow O(2^n)$
- n = 3:7 次 · n = 20:1 048 575 次
- n = 64：每秒移动一次，大约需要 **5850 亿年**

用迭代来写要难得多！

回溯：全排列

`A, B, C` 的所有排列：ABC、ACB、BAC、BCA、CAB、CBA

```
static void permute(List<Character> prefix, boolean[] used, char[] items) {
    if (prefix.size() == items.length) { print(prefix); return; }  // 基本情况
    for (int i = 0; i < items.length; i++) {
        if (used[i]) continue;
        used[i] = true;  prefix.add(items[i]);          // 做出选择
        permute(prefix, used, items);                   // 递归处理其余部分
        prefix.remove(prefix.size() - 1);  used[i] = false;   // 回溯
    }
}
```

- n 个元素有 **n!** 个排列 – O(n · n!)
- 同样的模式：数独、迷宫、八皇后、子集

递归还是迭代?

	递归	迭代
代码	简短，遵循定义	通常更长
空间	栈中 O（深度）	通常 O(1)
速度	有函数调用的开销	更快
危险	栈溢出	–
自然的应用	**树**、分治、回溯	简单循环、链表

每个递归都可以改写成迭代（必要时使用自己的**栈**），反之亦然。

经验法则：如果问题是**树形**的（多次递归调用），递归要简单得多；如果是线性的，最好写成循环。

小结

- **递归**：函数在一个**较小**的问题上调用自身
- 需要：**基本情况** + 调用要**接近**基本情况
- 工作原理：**调用栈** – 每次调用都有自己的栈帧，空间 O（深度）
- **树形递归**很容易变成指数级（朴素斐波那契） → **记忆化**或迭代
- 经典例子：阶乘、快速幂（$O(\log n)$）、汉诺塔（$2^n - 1$）、全排列
- 是**树**（第 14–15 周）和快速排序算法（第 13 周）的自然工具

下周：排序 II – 归并排序和快速排序。

复习题

1. 调用 `Fact(5)` 时调用栈的最大深度是多少?
2. 有什么错误? `Function F(n): Return n + F(n - 1)`
3. 计算 `Fib(5)` 时，朴素的 `Fib` 函数被调用了多少次?
4. 输出什么? `Procedure P(n): If n > 0 then P(n - 1); Print(n)` – `P(3)`
5. 如果把 `Print(n)` 移到 `P(n - 1)` 之前呢?
6. 汉诺塔中移动 5 个盘子需要多少步?

实践课

✎ 示例程序

程序	Java	Go	演示内容
阶乘	`Factorial.java`	`w12/factorial`	缩进打印调用过程、溢出
斐波那契	`Fibonacci.java`	`w12/fibonacci`	调用次数、记忆化、时间
例子	`RecursionExamples.java`	`w12/examples`	求和、乘方、逆序、回文、二分查找
汉诺塔	`Hanoi.java`	`w12/hanoi`	打印移动步骤，$2^n - 1$
栈深度	`StackDepth.java`	`w12/stackdepth`	Java 中的 `StackOverflowError`，Go 中增长的栈
全排列	`Permutations.java`	`w12/permutations`	回溯

✎ 练习

用**递归**（不用循环）求解！

1. **数字和：** `2026 → 10`
2. **二进制表示：** `50 → "110010"`（与第 7 周的栈比较！）
3. **最大值：**数组中最大的元素
4. **计数：** **x** 在数组中出现了多少次?
5. **反向输出链表：**按相反的顺序输出链表的元素！
6. 递归求**链表长度**
7. **二项式系数：**C(n, k) = C(n – 1, k – 1) + C(n – 1, k) – 为什么 n 较大时很慢?

参考答案：`Exercises12.java`、`w12/exercises`

✎ 课后作业

1. 用**记忆化**实现二项式系数，并测量计算 C(30, 15) 时的加速效果！
2. 编写递归函数，输出一个集合的**所有子集**！共有多少个？
3. **迷宫：**在字符地图上（`#` 表示墙，`.` 表示路），用回溯法找出从左上角到右下角的路径！
4. 用自己的栈把汉诺塔的解法改写成**迭代**形式！（提示：把“还需要完成的任务”压入栈中。）