递归与分治

递归函数设计

- 阶乘的计算公式

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

$$N! = N \times (N-1) \times \ldots \times 2 \times 1$$

- 换一种思路可以写成

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1$$

递归函数设计

- 阶乘公式可以写成

$$\begin{cases} \mathrm{N}! = N \times (N-1)! \\ 1! = 1 \end{cases}$$

- 这其实就是递归的思想，即函数的定义中使用函数自身的方法
- 除了计算机科学中，还有更多的递归的例子

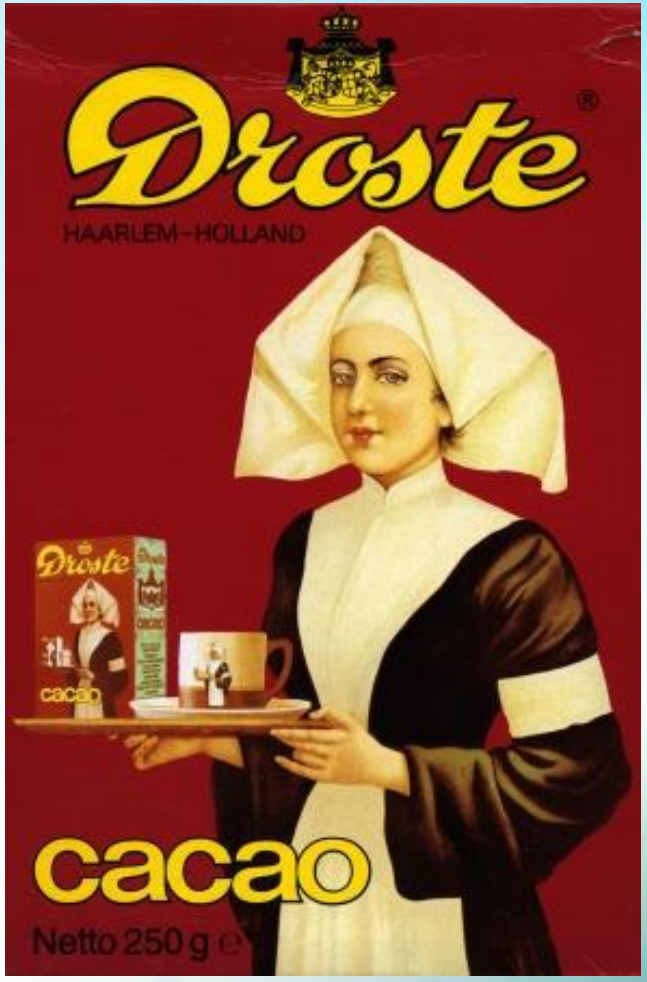
递归的例子

从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？“从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？“从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？……””

递归的例子

- 德罗斯特效应
  - Droste effect
  - 是递归的一种视觉形式
  - 是指一张图片的某个部分与整张图片相同，如此产生无限循环。

德罗斯特效应



德罗斯特效应

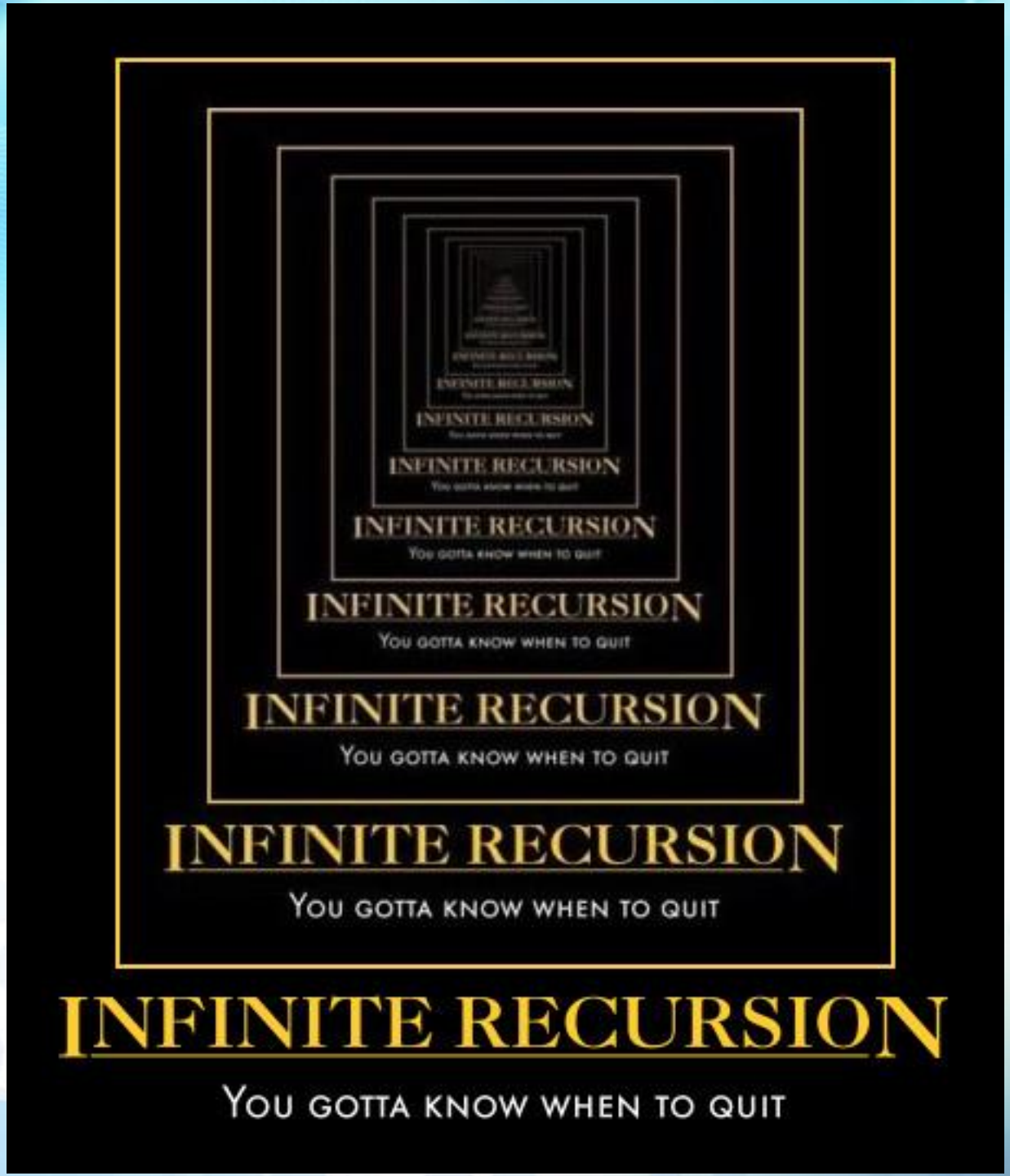
设计递归函数

- 程序设计中如何设计这样的函数呢？

$$\begin{cases} \mathrm{N!} = N \times (N-1)! \\ 1! = 1 \end{cases}$$

```
1   def fac(N):
2       return N * fac(N-1)
```

这个函数没有边界条件，函数fac的调用将无法终止。一定要注意首先递归函数要有一个边界条件，之后再考虑递归的步骤。


INFINITE RECURSION
YOU GOTTA KNOW WHEN TO QUIT
INFINITE RECURSION
YOU GOTTA KNOW WHEN TO QUIT
INFINITE RECURSION
YOU GOTTA KNOW WHEN TO QUIT
INFINITE RECURSION
YOU GOTTA KNOW WHEN TO QUIT
INFINITE RECURSION
YOU GOTTA KNOW WHEN TO QUIT
INFINITE RECURSION
YOU GOTTA KNOW WHEN TO QUIT

设计递归函数

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

边界条件

递归步骤

函数幕后

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

```
>>> fac(1)
```

Result: 1

初始条件没有问题！

函数幕后

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

---

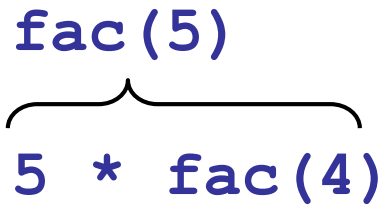
fac(5)

函数幕后

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

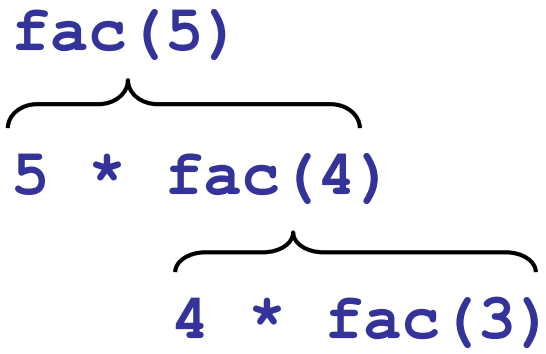


函数幕后

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

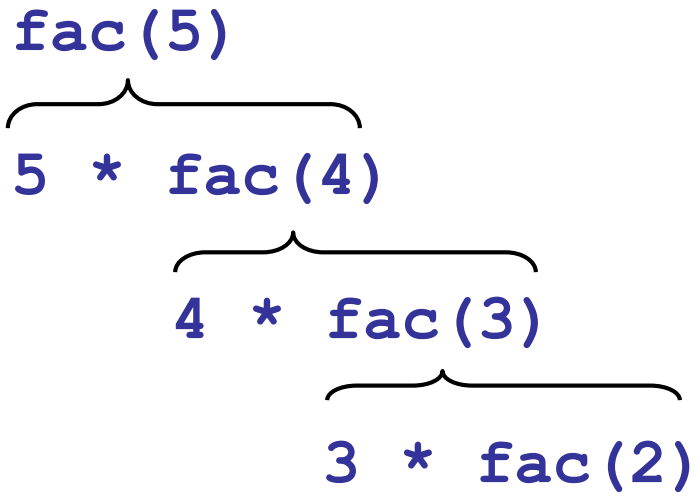


函数幕后

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

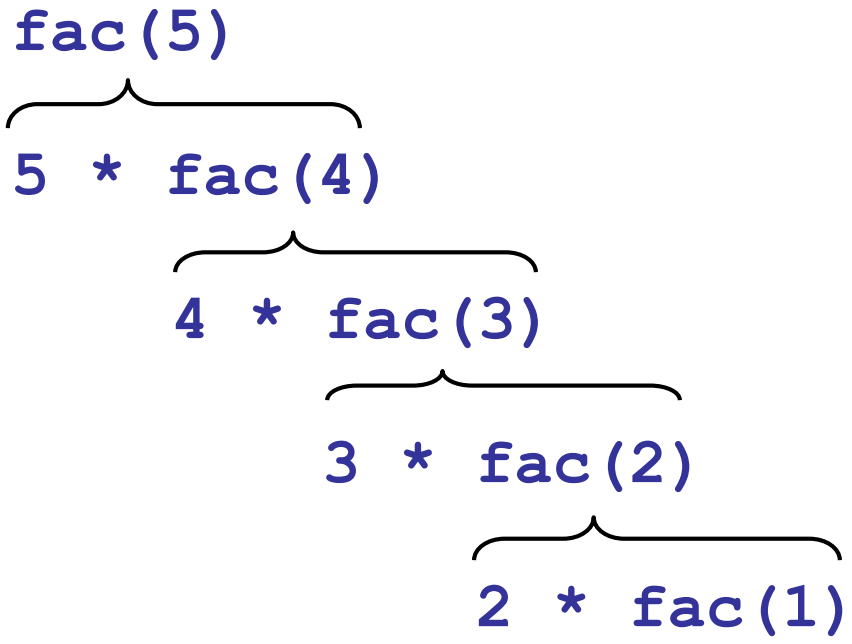


函数幕后

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

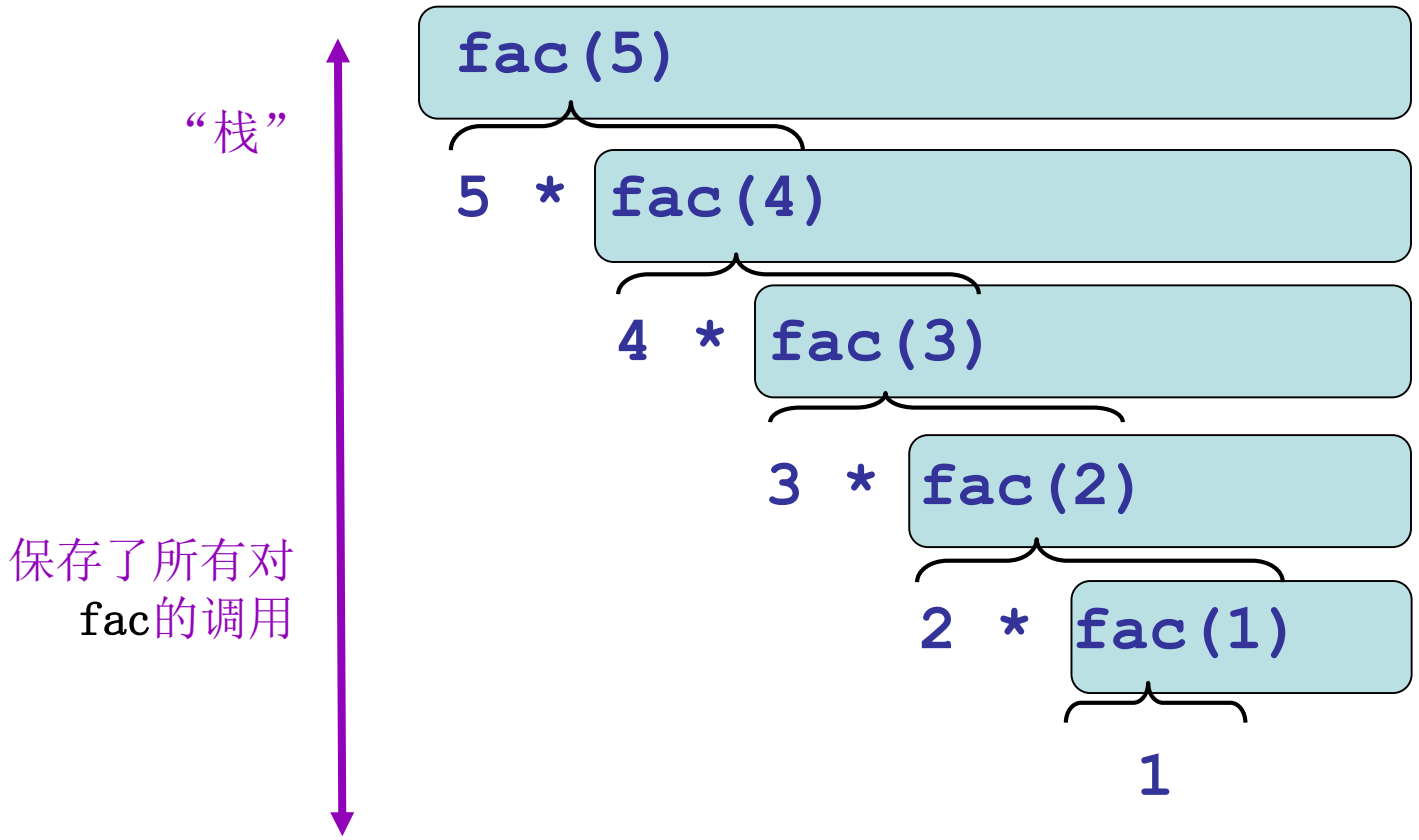


```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

函数幕后

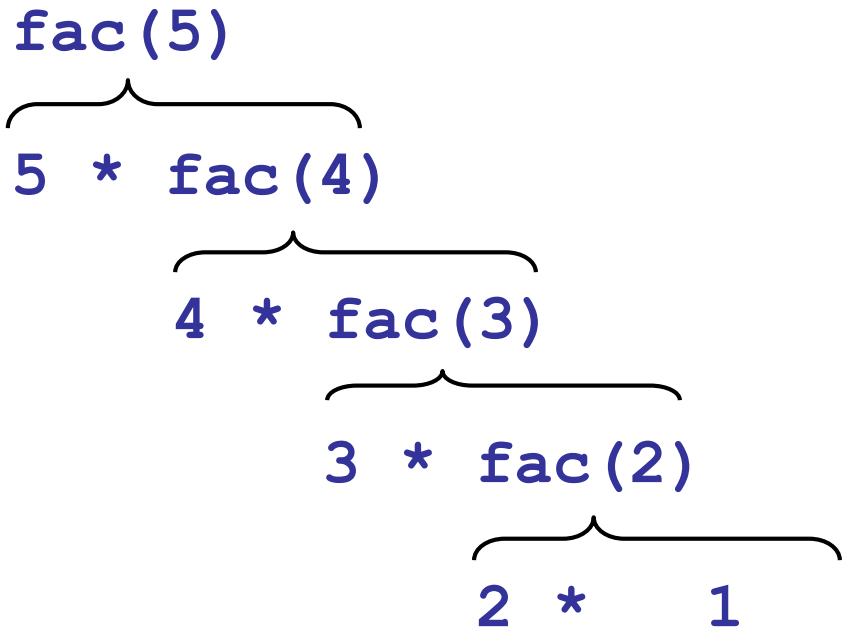


函数幕后

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

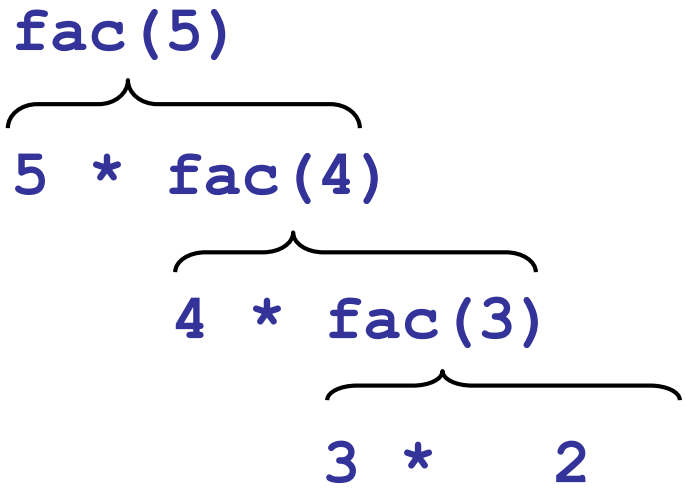


函数幕后

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

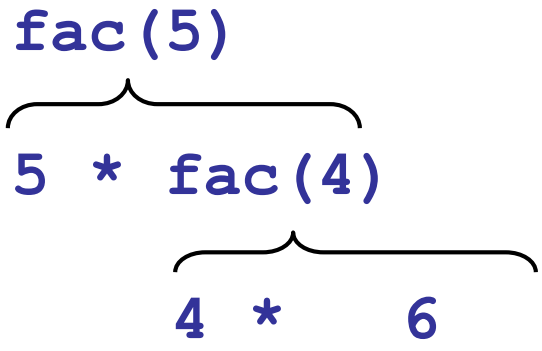


函数幕后

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```



函数幕后

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

```
fac(5)
5 *    24
```

函数幕后

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

---

fac(5)

Result: **120**

递归函数设计

挖掘自相似的特性
产生简介、优雅的代码

} 更少的工作！

```
def fac(N):

    if N <= 1:
        return 1
    else:
        return N * fac(N-1)
```

你需要控制边界条件——也就是你能想到的最简单的情况！

递归完成了几乎所有的剩余工作！

递归函数设计

- 但是你也必须自己完成一步

```
def fac(N):

    if N <= 1:
        return 1
    else:
        return fac(N)
```

这样是无法工作的！

递归定义

- 递归（Recursion）就是子程序（或函数）直接调用自己或通过一系列调用语句间接调用自己，是一种描述问题和解决问题的基本方法。
- 递归有两个基本要素：
  1. **边界条件**：确定递归到何时终止；
  2. **递归模式**：大问题是如何分解为小问题的。

Fibonacci数列

- 无穷数列1，1，2，3，5，8，13，21，34，55，……，称为Fibonacci数列。
- 它可以递归地定义为：

$$F(n)=\begin{cases}1 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

边界条件

递归模式

Fibonacci数列

```
def Fibonacci(N):

    if N <= 1:
        return 1
    else:
        return Fibonacci(N-2) + Fibonacci(N-1)
```

这段程序中什么是边界条件？什么是递归模式？你能模拟一下这个程序的运行吗？

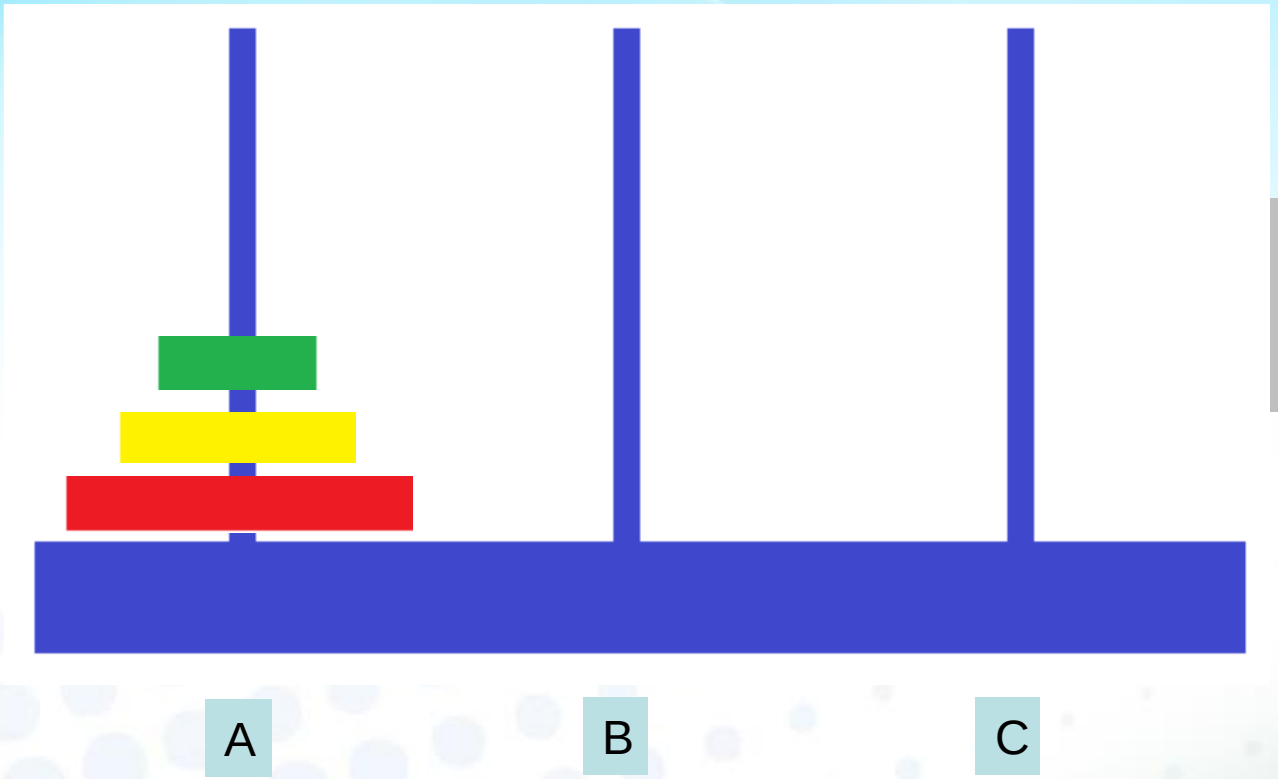
Hanoi塔问题

在印度，有这么一个古老的传说：在世界中心贝拿勒斯（在印度北部）的圣庙里，一块黄铜板上插着三根宝石针。印度教的主神梵天在创造世界的时候，在其中一根针上从下到上地穿好了由大到小的64片金片，这就是所谓的汉诺塔。不论白天黑夜，总有一个僧侣在按照下面的法则移动这些金片：一次只移动一片，不管在哪根针上，小片必须在大片上面。僧侣们预言，当所有的金片都从梵天穿好的那根针上移到另外一根针上时，世界就将在一声霹雳中消灭，而梵塔、庙宇和众生也都将同归于尽。

Hanoi塔问题

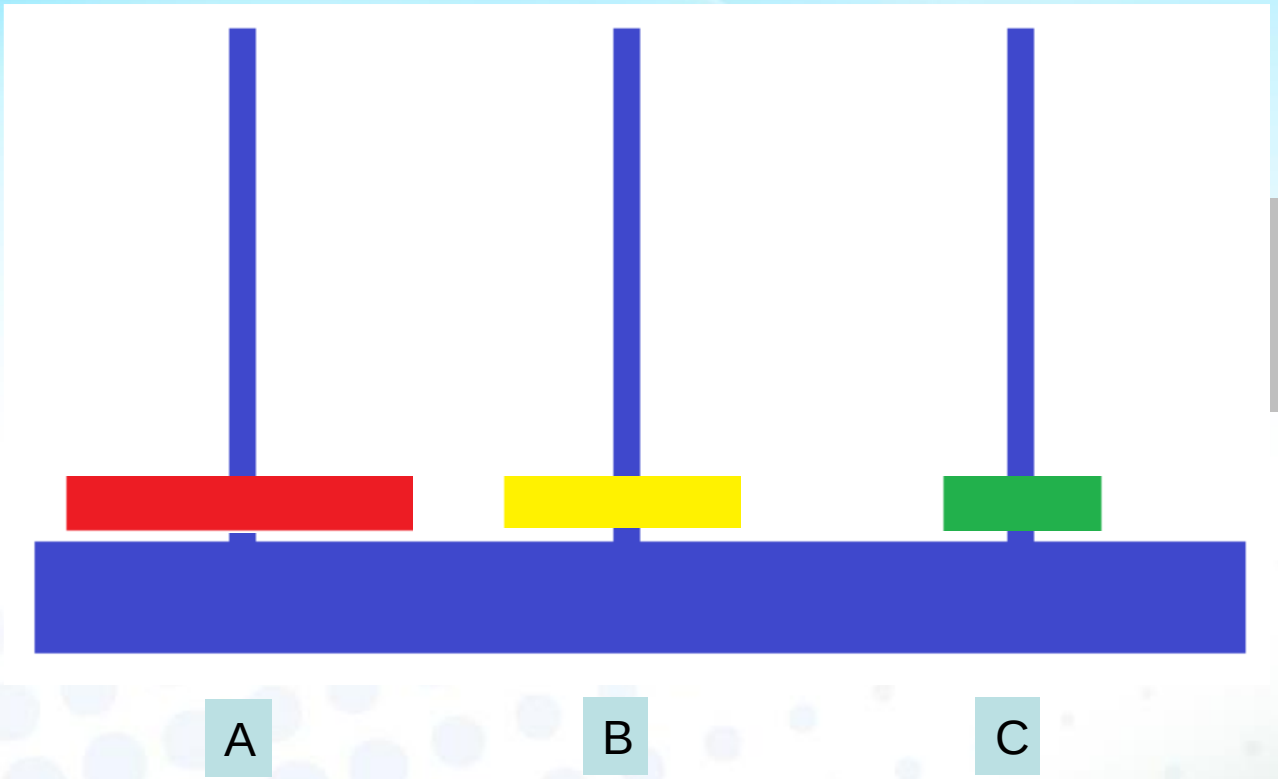
- 一共有3个塔座，在一个塔座上有一叠圆盘，这些圆盘自下而上，由大到小地叠在一起。要求将塔座上的这一叠圆盘移到另一塔座上，并仍按同样顺序叠置。
- 并遵守以下移动规则：
  1. 每次只能移动1个圆盘；
  2. 任何时刻都不允许将较大的圆盘压在较小的圆盘之上；
  3. 在满足移动规则1和2的前提下，可将圆盘移至任一塔座上。

Hanoi塔问题



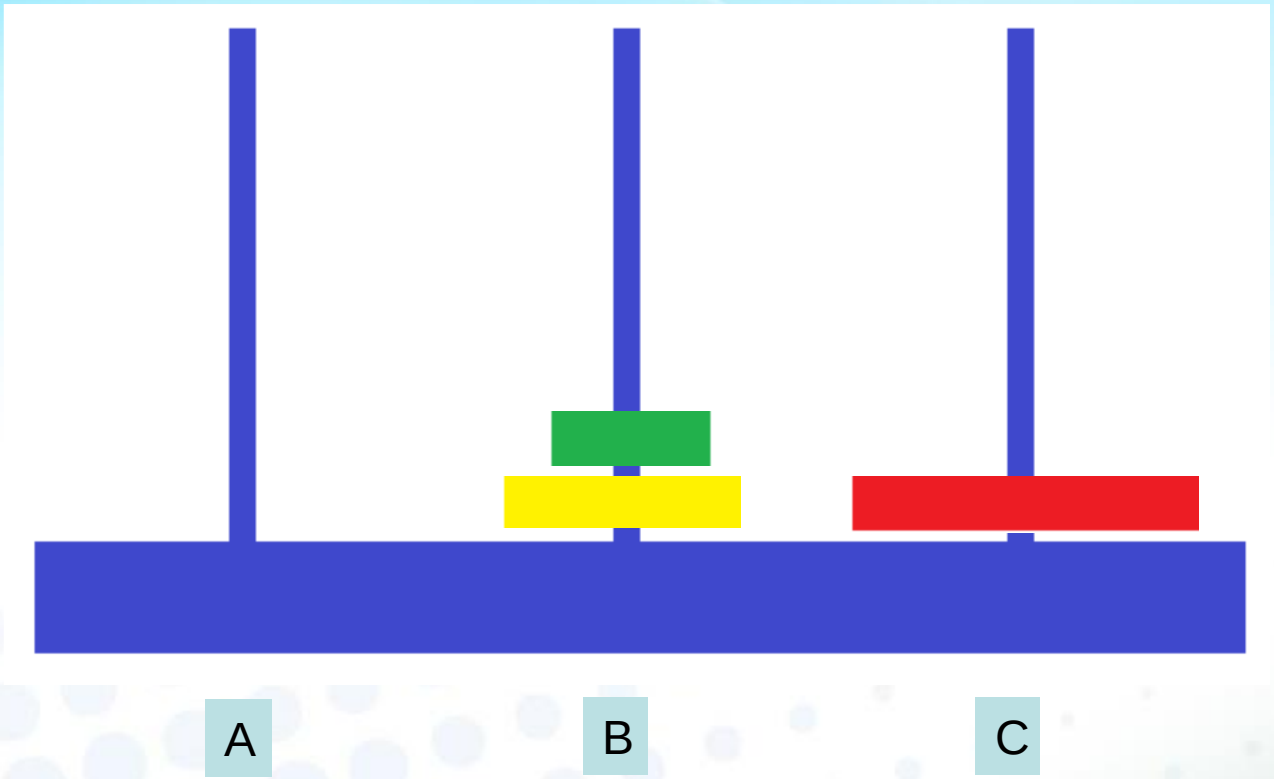
首先将A上的黄色和绿色通过C移动到B上

Hanoi塔问题



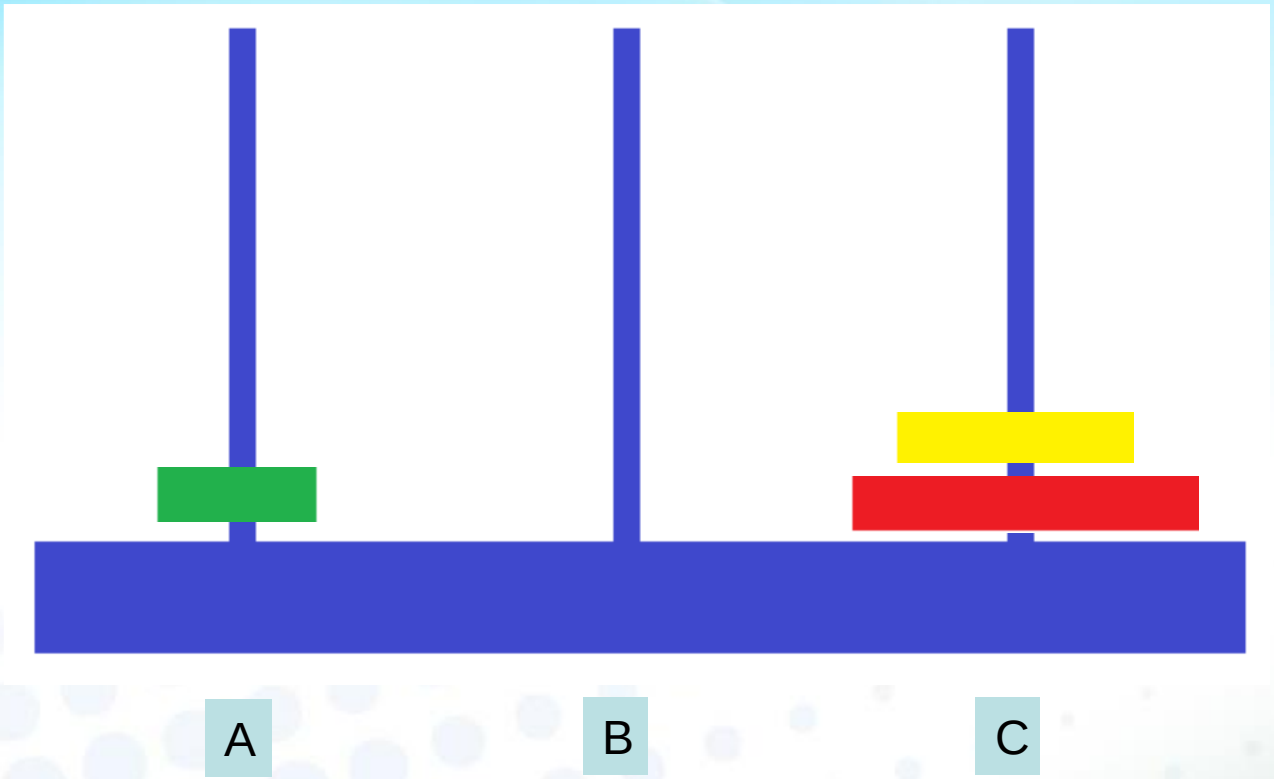
接着将A上的红色移动到C柱上

Hanoi塔问题



最后将B柱上的黄色和绿色通过A柱移动到C上，完成！

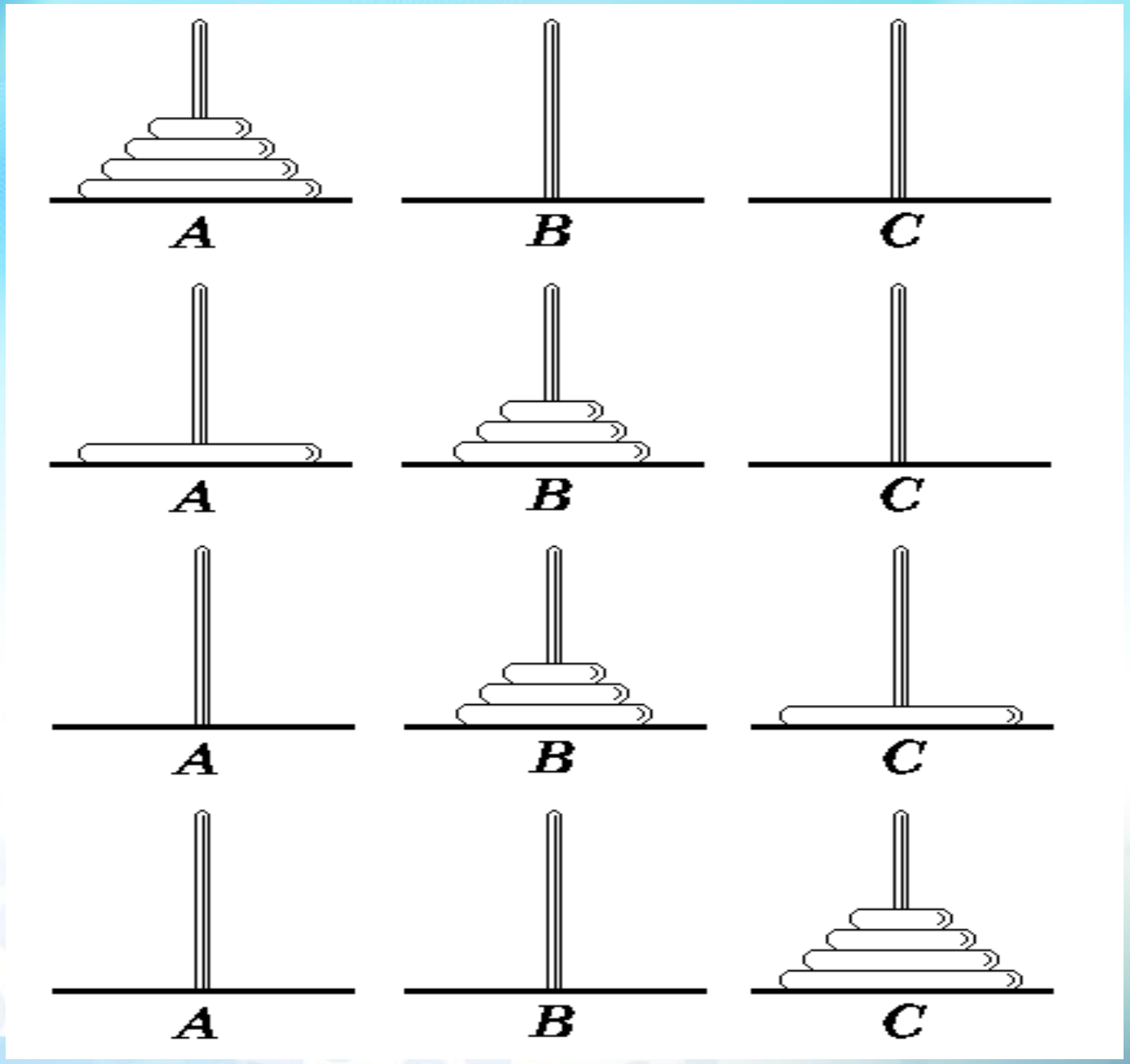
Hanoi塔问题



最后将B柱上的黄色和绿色通过A柱移动到C上，完成！

Hanoi塔问题

- 汉诺塔问题可以通过以下三个步骤实现：
  1. 将塔A上的n-1个碟子借助塔C先移到塔B上。
  2. 把塔A上剩下的一个碟子移到塔C上。
  3. 将n-1个碟子从塔B借助塔A移到塔C上。
- 显然，这是一个递归求解的过程

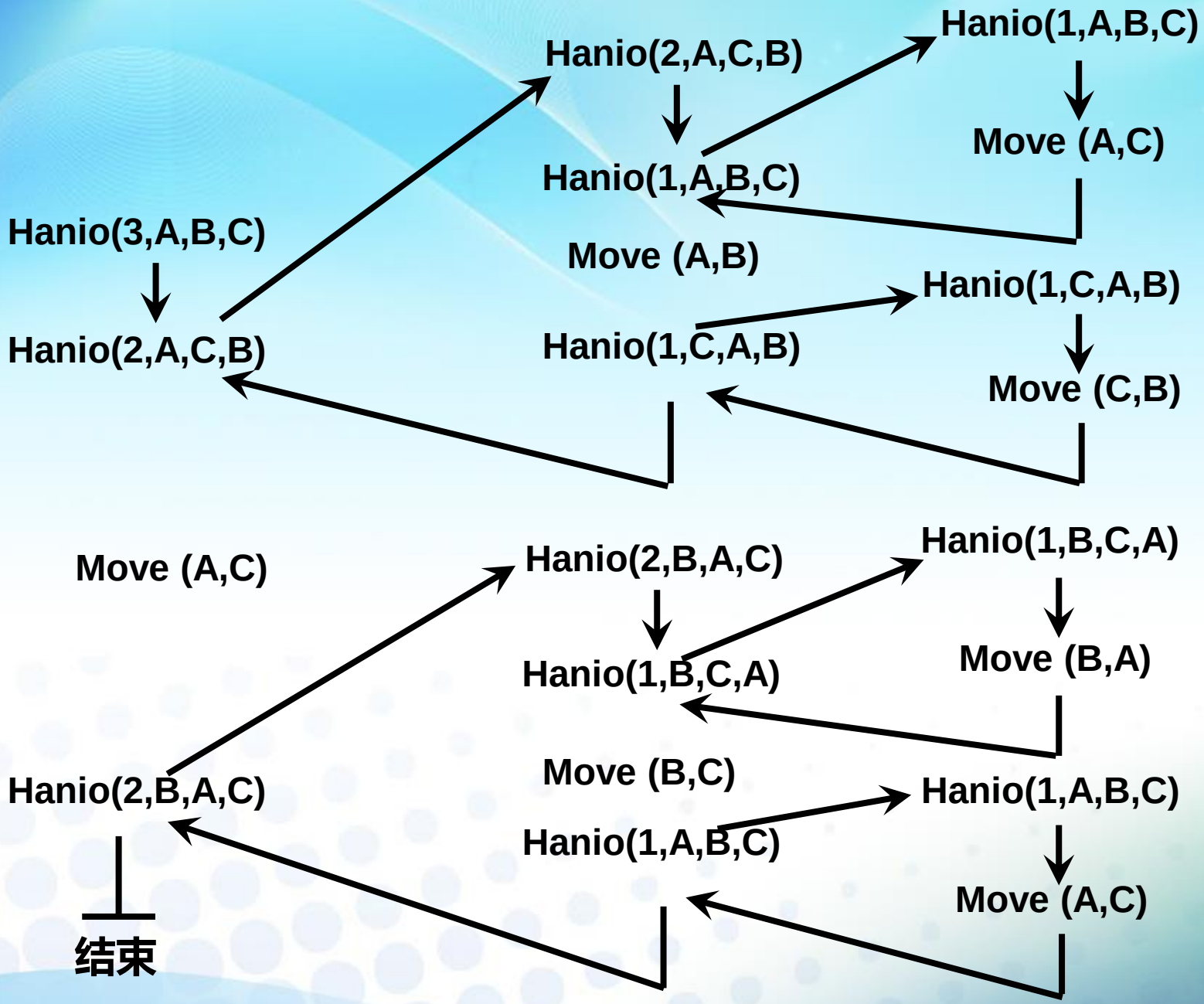

A
B
C
A
B
C
A
B
C
A
B
C

Hanoi塔问题

```
1 def Hanoi(N, A, B, C):
2
3     if N == 1:
4         Move(A,C)
5
6     else:
7         Hanoi(N-1, A, C, B)
8         Move(A, C)
9         Hanoi(N-1, B, A, C)
```

递归函数的运行轨迹

在递归函数中，调用函数和被调用函数是同一个函数，需要注意的是递归函数的调用层次，如果把调用递归函数的主函数称为第0层，进入函数后，首次递归调用自身称为第1层调用；从第i层递归调用自身称为第i+1层。反之，退出第i+1层调用应该返回第i层。采用图示方法描述递归函数的运行轨迹，从中可较直观地了解到各调用层次及其执行情况。


Hanio(3,A,B,C)
Hanio(2,A,C,B)
Hanio(2,A,C,B)
Hanio(1,A,B,C)
Move (A,B)
Hanio(1,A,B,C)
Move (A,C)
Hanio(1,C,A,B)
Hanio(1,C,A,B)
Move (C,B)
Hanio(1,A,B,C)
Hanio(2,B,A,C)
Move (A,C)
Hanio(2,B,A,C)
Hanio(1,B,C,A)
Move (B,C)
Hanio(1,B,C,A)
Move (B,A)
Hanio(1,A,B,C)
Hanio(1,A,B,C)
Move (A,C)
结束

Hanoi塔问题

- 由上面的递归函数我们可以推得

$$f(n) = 2f(n-1) + 1$$

- *f*(*n*)是n个金片时需要移动的次数，并且

$$f(1) = 1$$

- 我们可以得到

$$f(n) = 2^n - 1$$

Hanoi塔问题

那么把64片金片都移动到位的话，需要多长的时间呢？

$$f(64) = 2^{64} - 1 = 18446744073709551615$$

如果假设每秒移动一次，一年有31536000秒，那么需要

$$18446744073709551615 \div 31536000$$
$$= 584942417355.07\text{年}$$
$$= 5849\text{亿年}$$

而地球存在至今不过45亿年，太阳系的预期寿命据说也就是数百亿年。真的过了5845亿年，不说太阳系和银河系，至少地球上的一切生命，连同梵塔、庙宇等，都早已经灰飞烟灭。

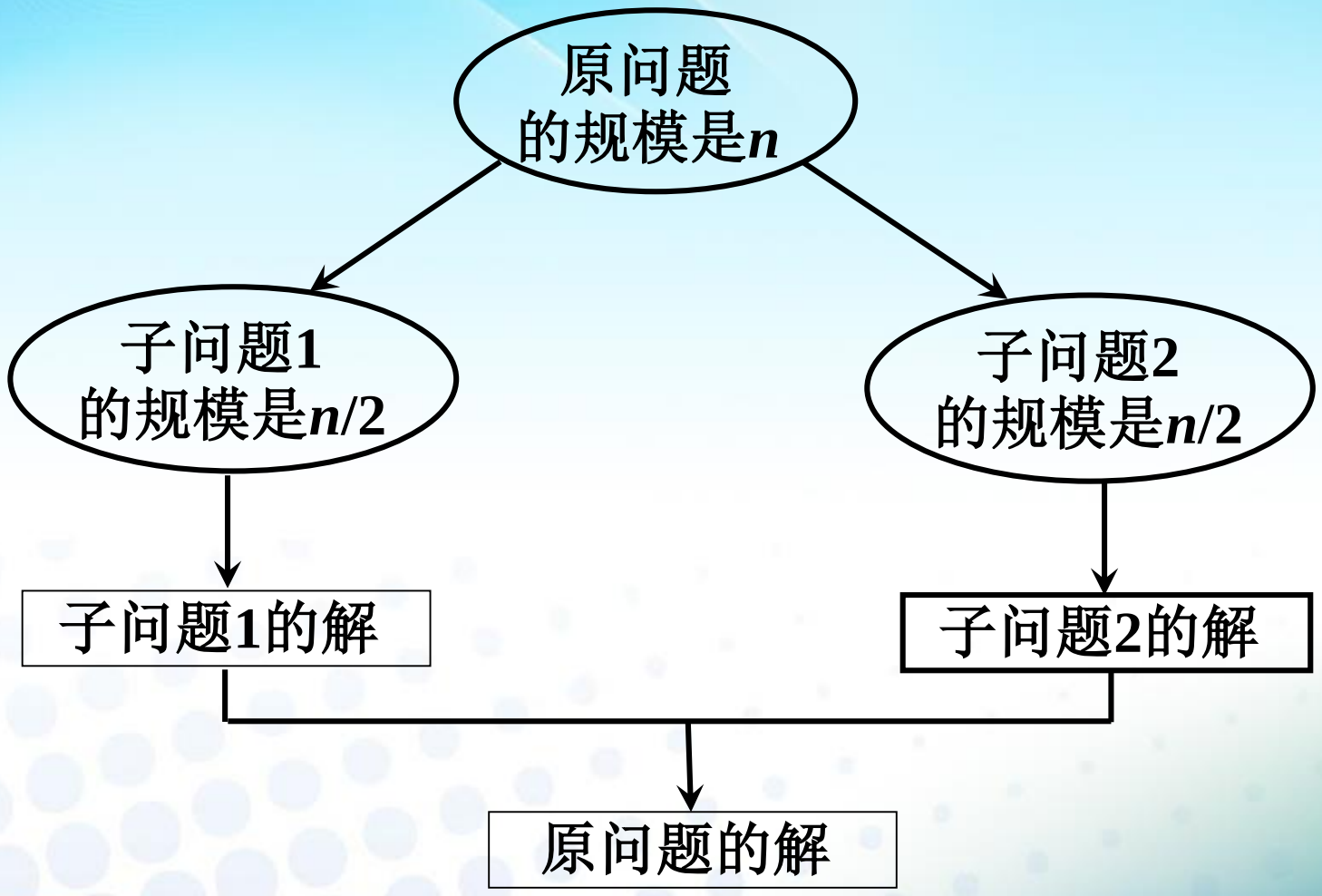
总结递归

递归算法结构清晰，可读性强，而且容易用数学归纳法来证明算法的正确性，因此，它为设计算法和调试程序带来很大方便，是算法设计中的一种强有力的工具。但是，因为递归算法是一种自身调用自身的算法，随着递归深度的增加，工作栈所需要的空间增大，递归调用时的辅助操作增多，因此，递归算法的运行效率较低。

分治法

将一个难以直接解决的大问题，划分成一些规模较小的子问题，以便各个击破，分而治之。更一般地说，将要求解的原问题划分成k个较小规模的子问题，对这k个子问题分别求解。如果子问题的规模仍然不够小，则再将每个子问题划分为k个规模更小的子问题，如此分解下去，直到问题规模足够小，很容易求出其解为止，再将子问题的解合并为一个更大规模的问题的解，自底向上逐步求出原问题的解。

分治法的典型情况



分治法的求解过程

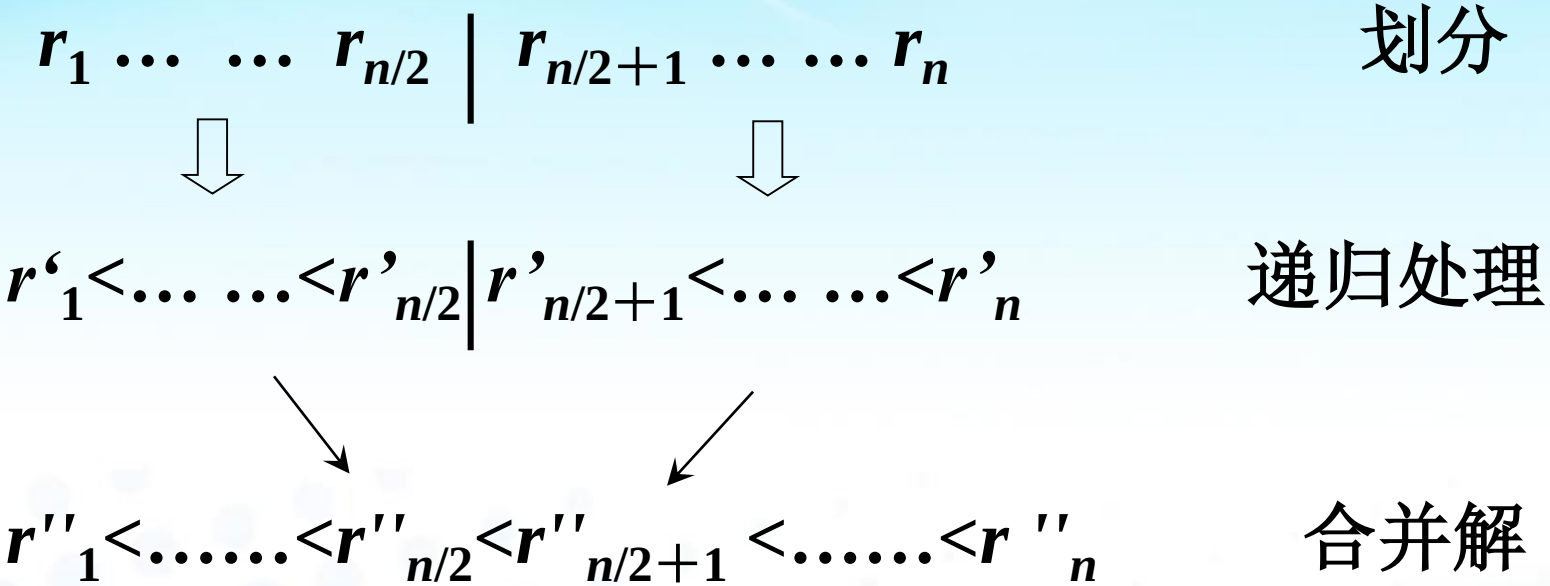
- 分治法的求解过程由以下三个阶段组成：
  1. **划分**
     - 把规模为n的原问题划分为k个规模较小的子问题，并尽量使这k个子问题的规模大致相同。
  2. **求解子问题**
     - 各子问题的解法与原问题的解法通常是相同的，可以用递归的方法求解各个子问题，有时递归处理也可以用循环来实现。
  3. **合并**
     - 把各个子问题的解合并起来，合并的代价因情况不同有很大差异，分治算法的有效性很大程度上依赖于合并的实现。

归并排序

- 二路归并排序的分治策略是：

1. **划分**：将待排序序列$r_1, r_2, \ldots, r_n$划分为两个长度相等的子序列$r_1, \ldots, r_{n/2}$和$r_{n/2+1}, \ldots, r_n$；
2. **求解子问题**：分别对这两个子序列进行排序，得到两个有序子序列；
3. **合并**：将这两个有序子序列合并成一个有序序列。

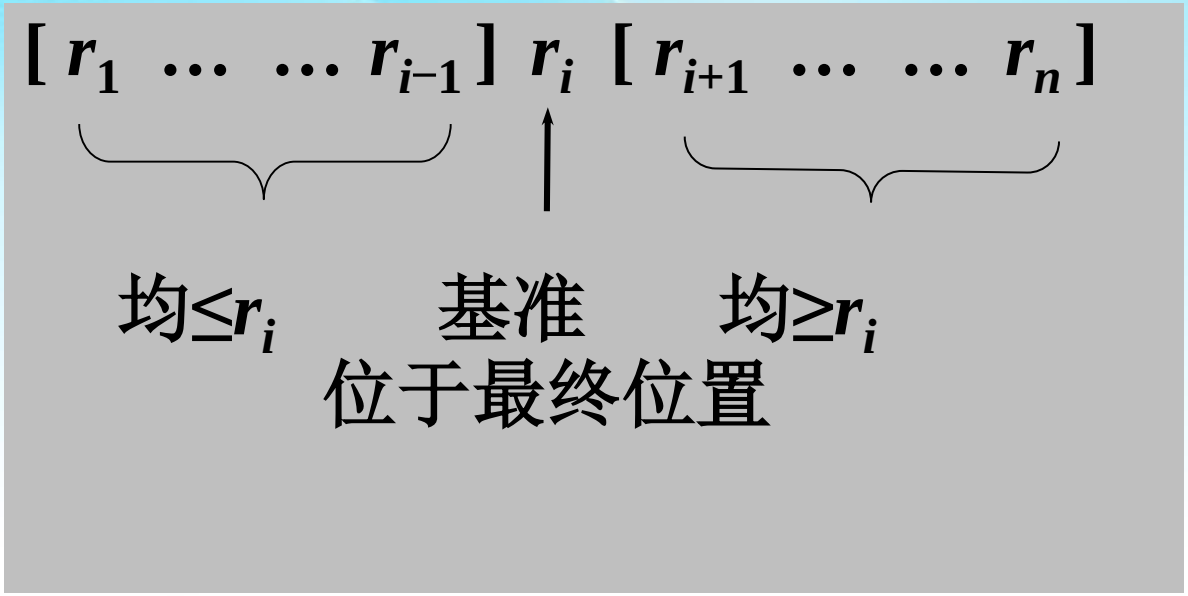
归并排序


$r_1 \cdots \cdots r_{n/2} \mid r_{n/2+1} \cdots \cdots r_n$
划分
$r'_1 < \cdots \cdots < r'_{n/2} \mid r'_{n/2+1} < \cdots \cdots < r'_n$
递归处理
$r''_1 < \cdots\cdots < r''_{n/2} < r''_{n/2+1} < \cdots\cdots < r''_n$
合并解

快速排序

- 快速排序的分治策略是：
  1. **划分**：选定一个记录作为基准，将整个序列划分为两个子序列$r_1 \dots r_{i-1}$和$r_{i+1} \dots r_n$，前一个子序列中记录的值均小于或等于基准值，后一个子序列中记录的值均大于或等于基准值；
  2. **求解子问题**：分别对划分后的每一个子序列递归处理；
  3. **合并**：由于对子序列$r_1 \dots r_{i-1}$和$r_{i+1} \dots r_n$的排序是就地进行的，所以合并不需要执行任何操作。

快速排序



- 归并排序按照记录在序列中的位置对序列进行划分，
- 快速排序按照记录的值对序列进行划分。

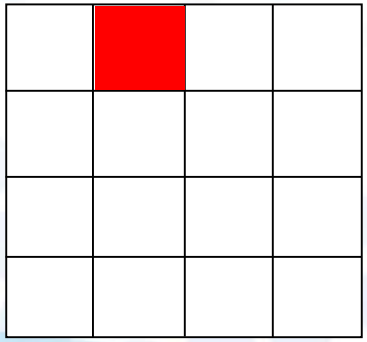
二分查找法

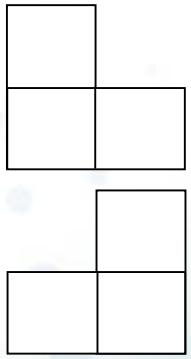
- 二分查找法的分治策略是：

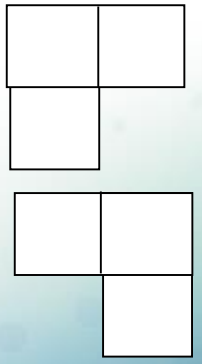
1. **划分**：取排过序的序列r_1, r_2, …, r_n中间一个数分为两个长度相等的子序列；
2. **求解子问题**：判断是否是要查找的数字，以及查找的数字在哪个序列中；
3. **合并**：返回最终是否查找了到了相应的值。

棋盘覆盖问题

在一个$2^k \times 2^k$（$k \geq 0$）个方格组成的棋盘中，恰有一个方格与其他方格不同，称该方格为特殊方格。棋盘覆盖问题要求用4种不同形状的L型骨牌覆盖给定棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。







棋盘覆盖问题

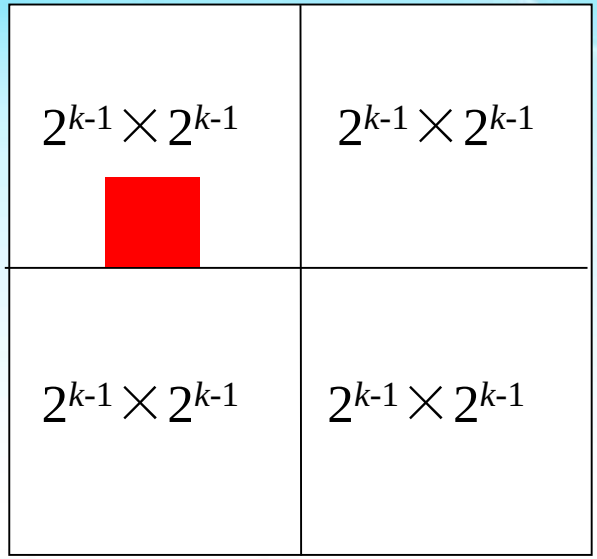
如何利用分治的思想来解决棋盘覆盖问题呢？

分治法求解棋盘覆盖问题的技巧在于划分棋盘，使划分后的子棋盘的大小相同，并且每个子棋盘均包含一个特殊方格，从而将原问题分解为规模较小的棋盘覆盖问题。

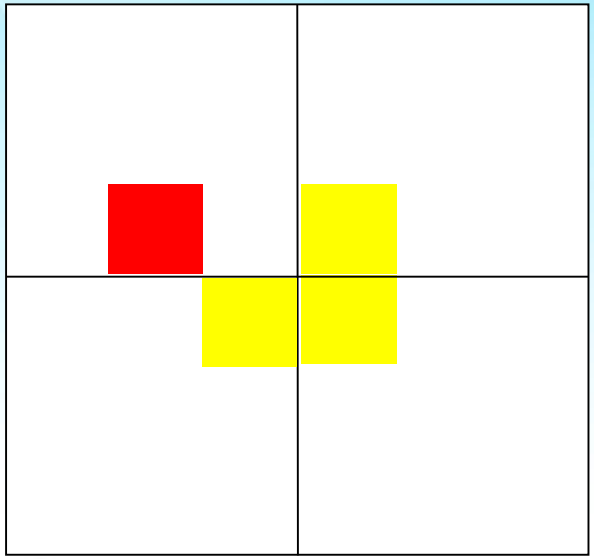
棋盘覆盖问题

$k>0$时，可将$2^k \times 2^k$的棋盘划分为4个$2^{k-1} \times 2^{k-1}$的子棋盘，这样划分后，由于原棋盘只有一个特殊方格，所以，这4个子棋盘中只有一个子棋盘包含该特殊方格，其余3个子棋盘中没有特殊方格。为了将这3个没有特殊方格的子棋盘转化为特殊棋盘，以便采用递归方法求解，可以用一个L型骨牌覆盖这3个较小棋盘的会合处，从而将原问题转化为4个较小规模的棋盘覆盖问题。递归地使用这种划分策略，直至将棋盘分割为1×1的子棋盘。

棋盘覆盖问题



棋盘分割



构造相同子问题

棋盘覆盖问题

- 采用分治法解决棋盘覆盖问题比单纯的穷举法要快速很多
- 想一想你能不能写出像解决Hanoi塔问题的递归程序吗？

循环赛日程安排问题

- 设有$n=2^k$个选手要进行网球循环赛，要求设计一个满足以下要求的比赛日程表：
  1. 每个选手必须与其他n-1个选手各赛一次；
  2. 每个选手一天只能赛一次。
- 按此要求，可将比赛日程表设计成一个 n 行n-1列的二维表，其中，第 i 行第 j 列表示和第 i 个选手在第 j 天比赛的选手。

循环赛日程安排问题

按照分治的策略，可将所有参赛的选手分为两部分，$n = 2^k$个选手的比赛日程表就可以通过为$n/2 = 2^{k-1}$个选手设计的比赛日程表来决定。递归地执行这种分割，直到只剩下2个选手时，比赛日程表的制定就变得很简单：只要让这2个选手进行比赛就可以了。

循环赛日程安排问题

1	2
2	1

(a) 2^k(k=1)个选手比赛

加2

1 2 2 1	3 4 4 3
3 4 4 3	1 2 2 1

(b) 2^k(k=2)个选手比赛

加4

1 2 3 4 2 1 4 3 3 4 1 2 4 3 2 1	5 6 7 8 6 5 8 7 7 8 5 6 8 7 6 5
5 6 7 8 6 5 8 7 7 8 5 6 8 7 6 5	1 2 3 4 2 1 4 3 3 4 1 2 4 3 2 1

(c) 2^k(k=3)个选手比赛

循环赛日程安排问题

显然，这个求解过程是自底向上的迭代过程，其中左上角和左下角分别为选手1至选手4以及选手5至选手8前3天的比赛日程，据此，将左上角部分的所有数字按其对应位置抄到右下角，将左下角的所有数字按其对应位置抄到右上角，这样，就分别安排好了选手1至选手4以及选手5至选手8在后4天的比赛日程，如图(c)所示。具有多个选手的情况可以依此类推。

循环赛日程安排问题

- 这种解法是把求解2^k个选手比赛日程问题划分成依次求解2^1、2^2、…、2^k个选手的比赛日程问题
- 换言之，2^k个选手的比赛日程是在2^{k-1}个选手的比赛日程的基础上通过迭代的方法求得的。

循环赛日程安排问题

- 在每次迭代中，将问题划分为4部分：
  1. **左上角**：左上角为2^{k-1}个选手在前半程的比赛日程；
  2. **左下角**：左下角为另2^{k-1}个选手在前半程的比赛日程，由左上角加2^{k-1}得到，例如2^2个选手比赛，左下角由左上角直接加2得到，2^3个选手比赛，左下角由左上角直接加4得到；
  3. **右上角**：将左下角直接抄到右上角得到另2^{k-1}个选手在后半程的比赛日程；
  4. **右下角**：将左上角直接抄到右下角得到2^{k-1}个选手在后半程的比赛日程；