第2章 递归与分治策略

- 本章主要知识点：
  - 2.1 递归的概念
  - 2.2 分治法的基本思想
  - 2.3 二分搜索技术
  - 2.4 大整数的乘法
  - 2.5 Strassen矩阵乘法
  - 2.6 棋盘覆盖
  - 2.7 合并排序
  - 2.8 快速排序
  - 2.9 线性时间选择
  - 2.10 最接近点对问题
  - 2.11 循环赛日程表
- 计划授课时间：6～8课时

2.1 递归的概念

➢ 直接或间接地调用自身的算法称为**递归算法**。用函数自身给出定义的函数称为**递归函数**。

一个递归问题

调用自己

- 从前有座山，山上有座庙，庙里有个老和尚讲故事，讲的是
  - 从前有座山，山上有座庙，庙里有个老和尚讲故事，讲的是
    - 从前有座山，山上有座庙，庙里有个老和尚讲故事，讲的是

……　　……

2.1 递归的概念---阶乘函数

- 例1 阶乘函数
- 可递归地定义为：

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

- 其中：
  - $n=0$时，$n!=1$为边界条件
  - $n>0$时，$n!=n(n-1)!$为递归方程
- 边界条件与递归方程是递归函数的二个要素，递归函数只有具备了这两个要素，才能在有限次计算后得出结果。

任何大于1的自然数n阶乘表示方法：
n!=1×2×3×……×n

2.1 递归的概念---阶乘函数

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

```
int Factorial(int n)
{
    if (n==0) return 1;
    return n*Factorial(n - 1);
}
```

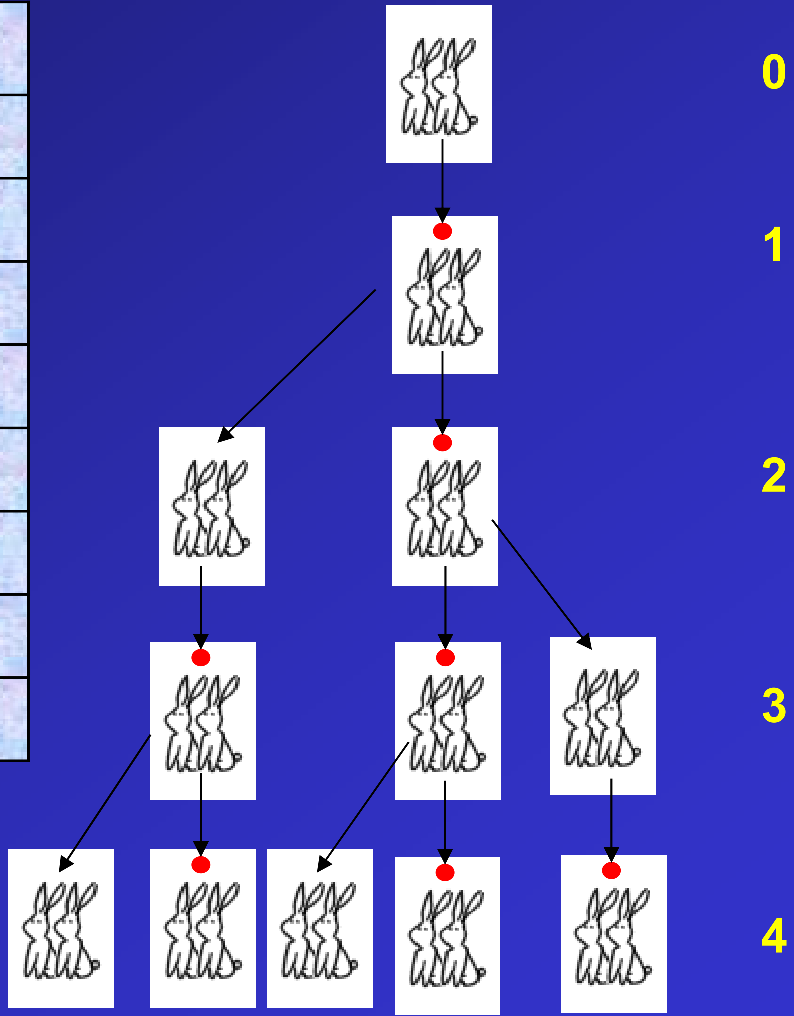
2.1 递归的概念--*Fibonacci*数列

- 例2：*Fibonacci*数列
  - 问题引入
    - 裴波那契（*Fibonacci leonardo*，约1170-1250）是意大利著名数学家.
    - 在他的著作《算盘书》中许多有趣的问题，最富成功的问题是著名的“**兔子繁殖问题**”：如果每对兔子每月繁殖一对子兔，而子兔在出生后第二个月就有生殖能力，试问一对兔子一年能繁殖多少对兔子？
  - 问题分析

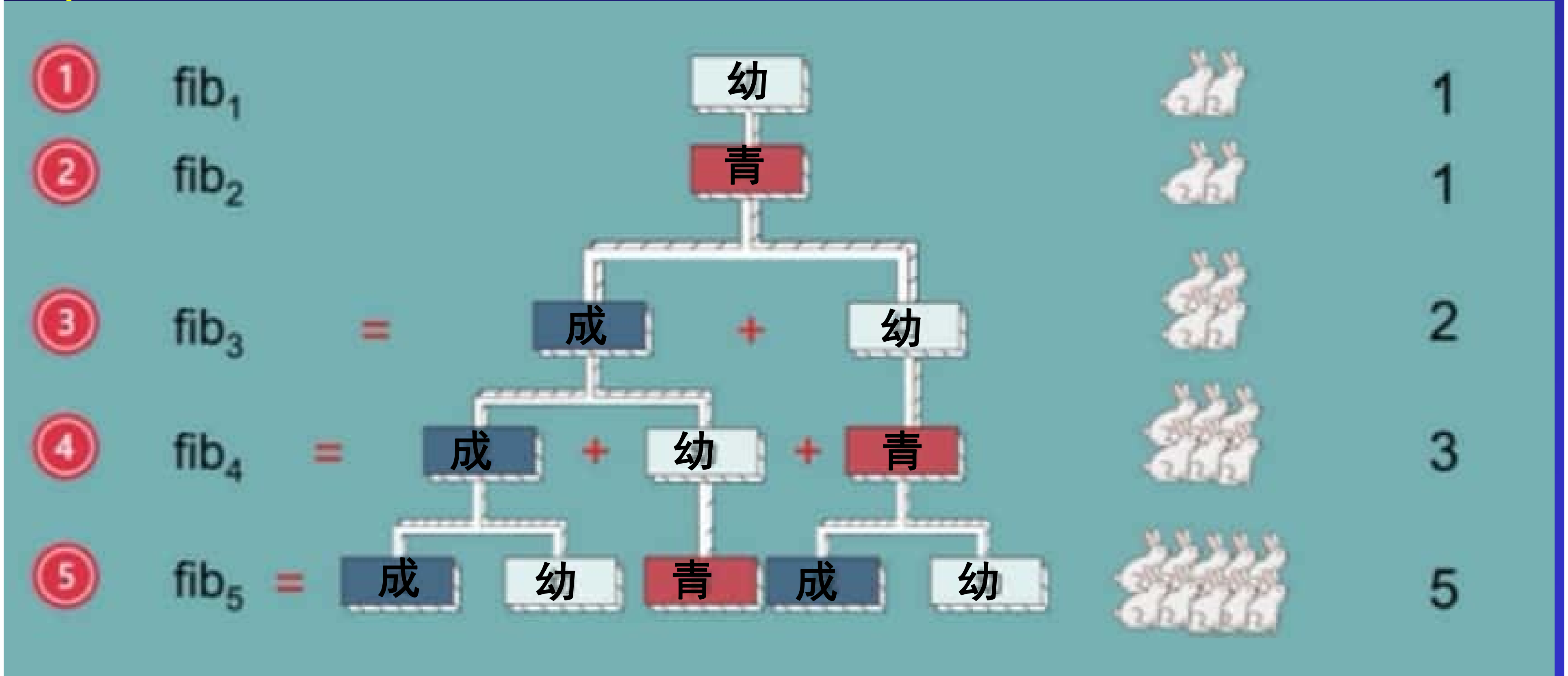
2.1 递归的概念--*Fibonacci*数列

月份	初生	成熟	总数
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3	5	8
6	5	8	13
……	……	……	……

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



2.1 递归的概念--*Fibonacci*数列



月份	初生	成熟	总数
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3	5	8
6	5	8	13
……	……	……	……

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

2.1 递归的概念

- 数列的特点
  - 数列的增长速度
  - 构造一个新数列
  - 自然科学中的若干实例

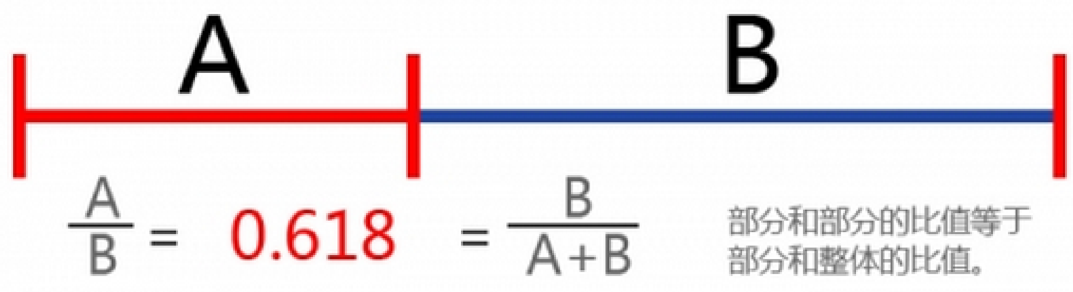
$$\frac{1}{1},\frac{1}{2},\frac{2}{3},\frac{3}{5},\frac{5}{8},\frac{8}{13},\frac{13}{21},\frac{21}{34}\cdots\cdots\frac{F_n}{F_{n+1}}\cdots\cdots$$

$$\lim_{n\to\infty}\frac{f(n)}{f(n+1)}=\frac{\sqrt{5}-1}{2}=0.618\ldots(\text{黄金分割数})$$

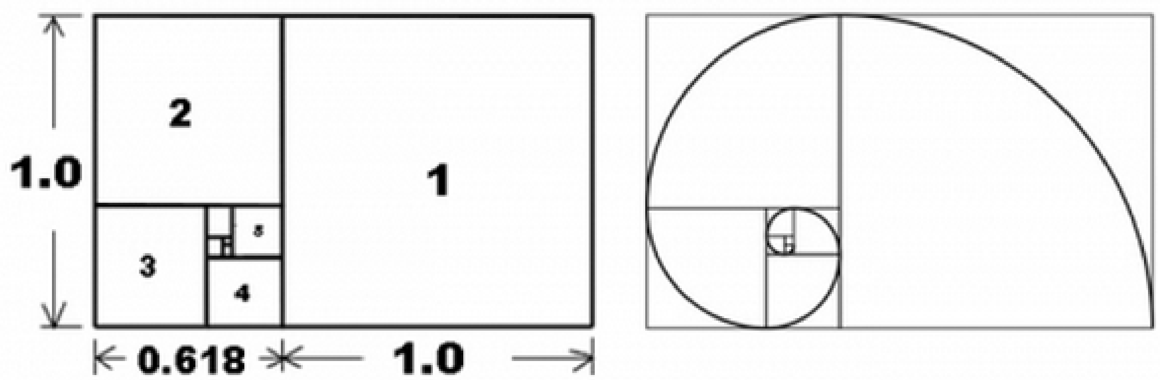
PS：黄金分割的美学价值

- <u>黄金分割率：</u>

0.618或者1.618，这个数字是否觉得似曾相识。这其实是一个数学比例关系（说到数学，不要先着急晕哦，知道咱们做设计得对计算都不敏感，呵呵），即把一条线段分为两部分，此时短段与长段之比恰恰等于长段与整条线之比，其数值比为1:1.618或0.618:1。

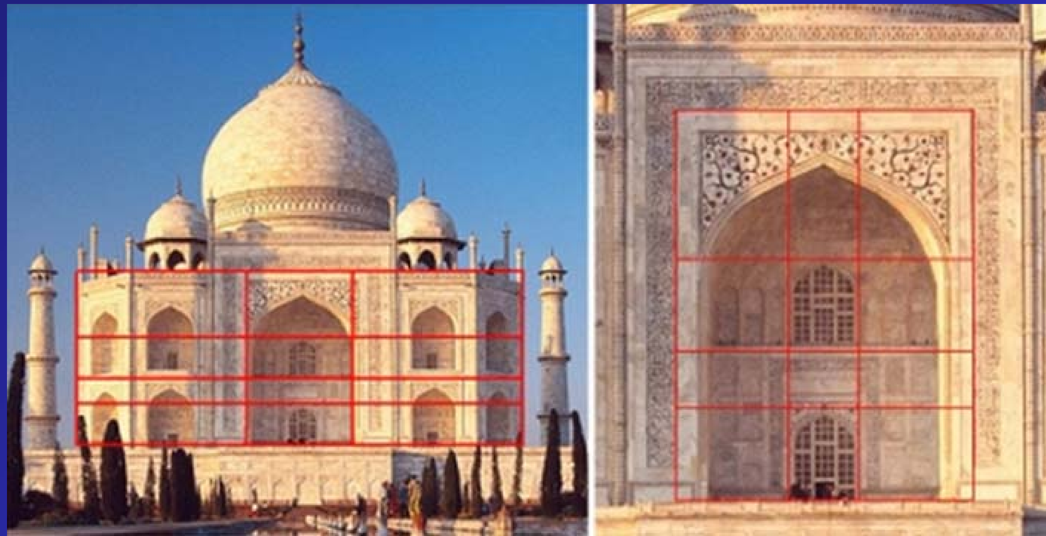


- <u>黄金矩形</u>：长宽之比为黄金分割率，即矩形的长边为短边 1.618倍



PS：黄金分割的美学价值

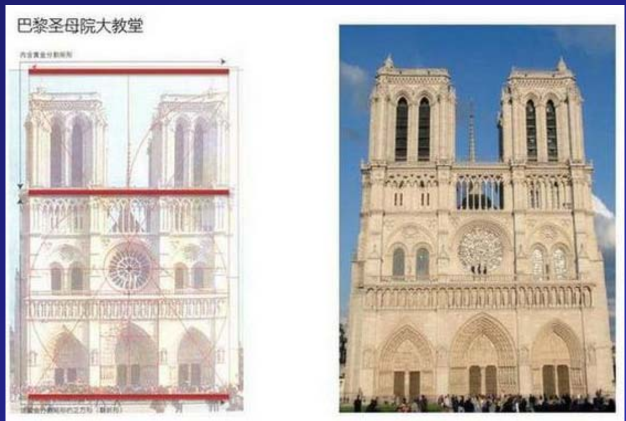
- 印度的泰姬陵：

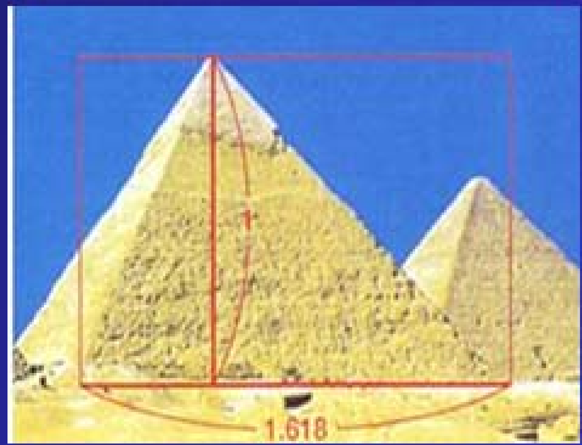


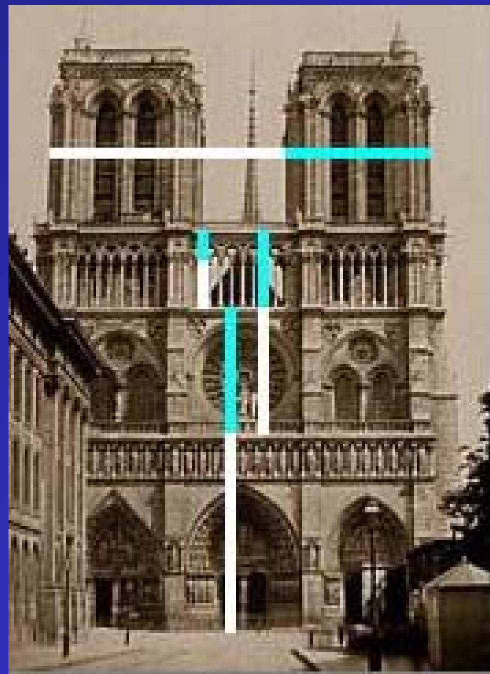
- 希腊雅典的巴特农神庙：

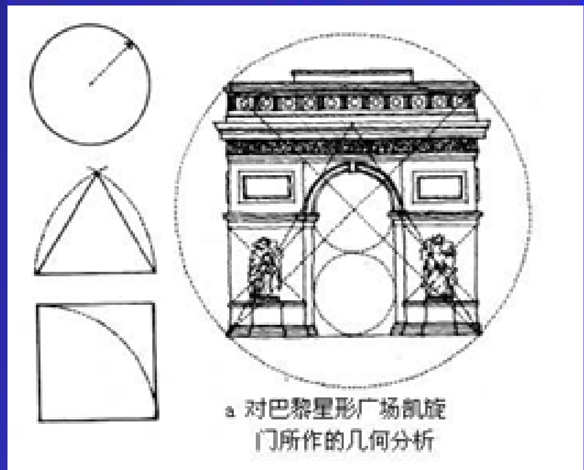
PS：黄金分割的美学价值

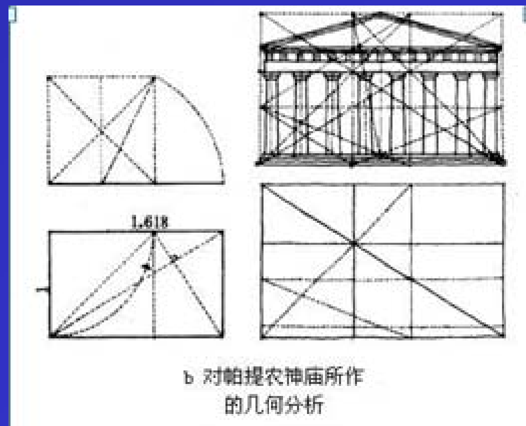
- 古埃及金字塔
- 巴黎圣母院
- 法国埃菲尔铁塔







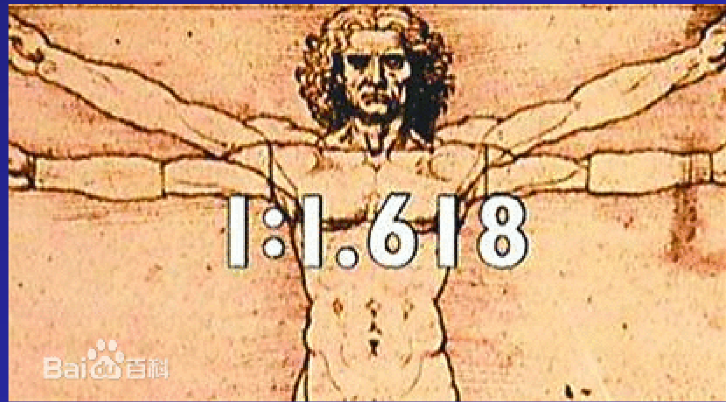


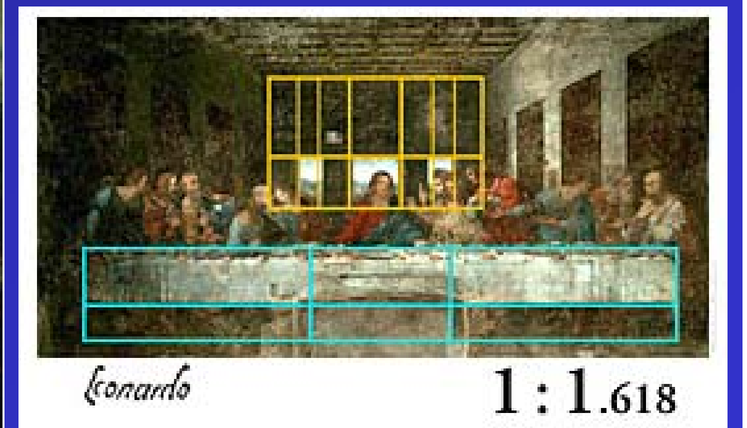


- 大多数门窗的宽长之比也是0.618；

PS：黄金分割的美学价值

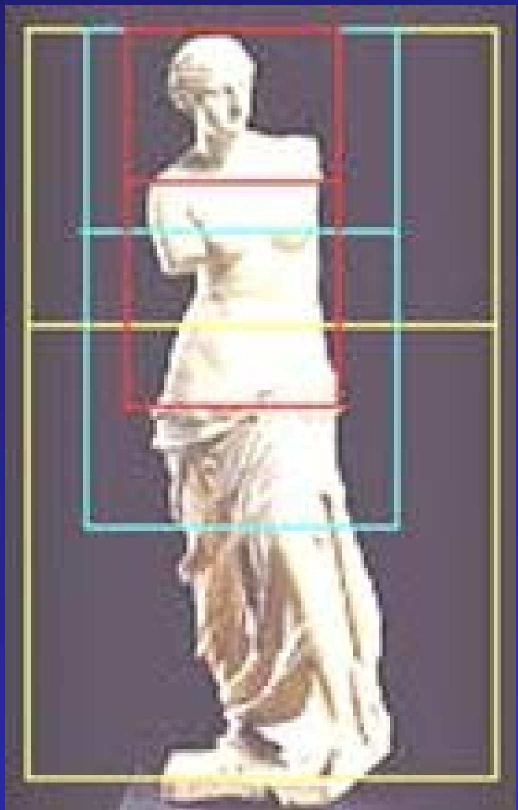
- 《维特鲁威人》
- 《蒙娜丽莎》
- 《最后的晚餐》

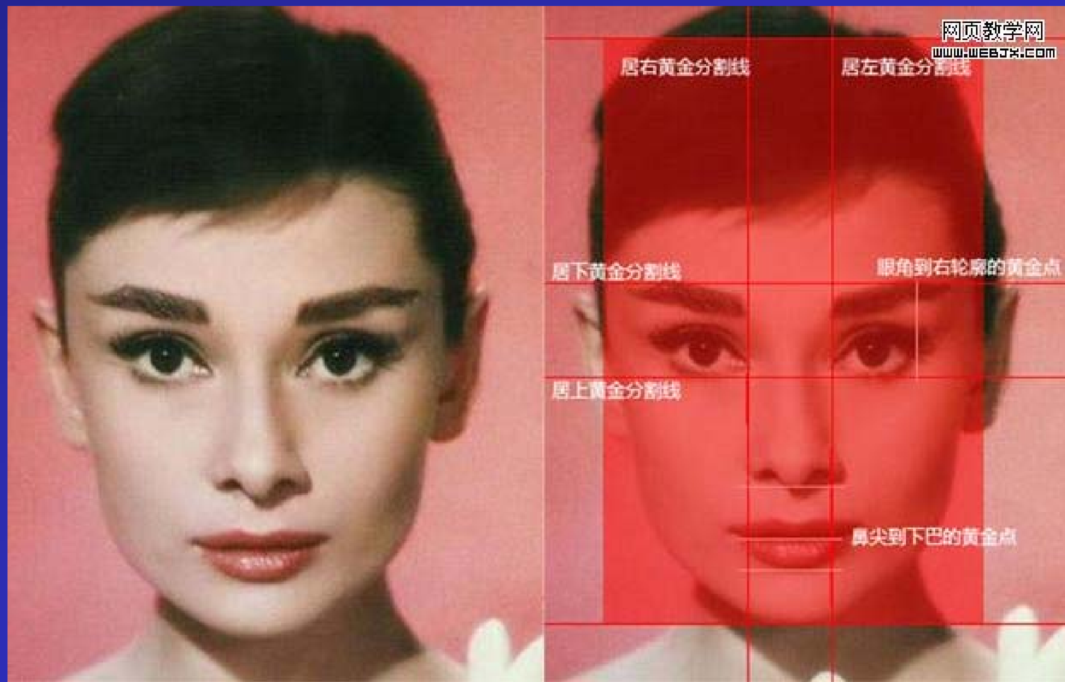




PS：黄金分割的美学价值

- 古希腊维纳斯女神塑像及太阳神阿波罗的形象都通过故意延长双腿，使之与身高的比值为0.618，从而创造艺术美。

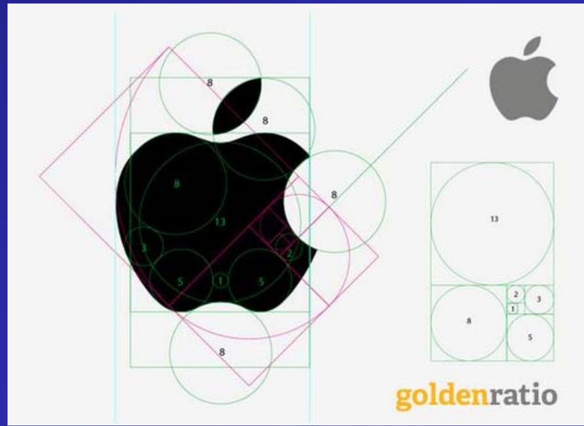




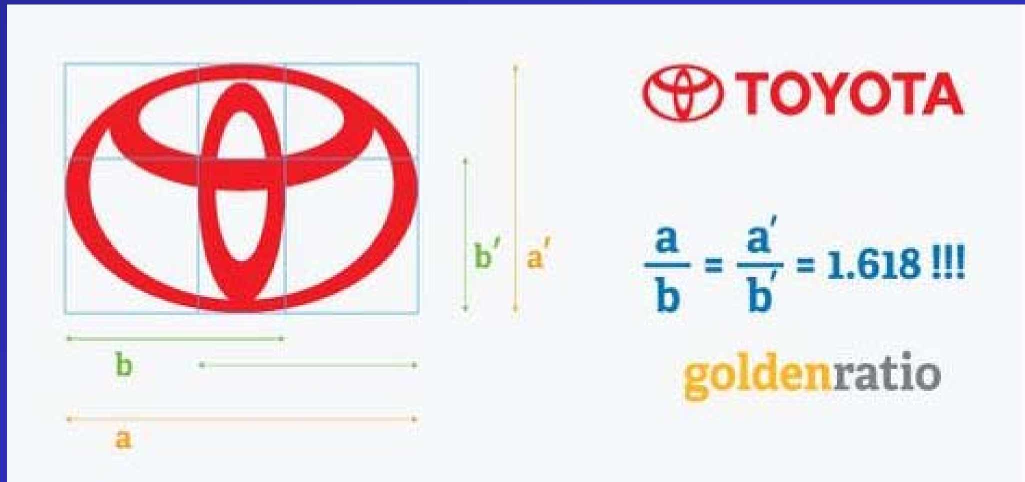
奥黛丽赫本

PS：黄金分割的美学价值

- **Apple**　Apple的Logo具有完美的平衡，映射到Logo上的轮廓是在直径上遵循斐波那契数列的圆形。



- **丰田**　Toyota的Logo由三个椭圆组成。一个基于φ的网格。该网格的网格线间隔遵循黄金分割率φ



2.1 递归的概念--*Fibonacci*数列

- 定义及解法

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

$$F(n)=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$$

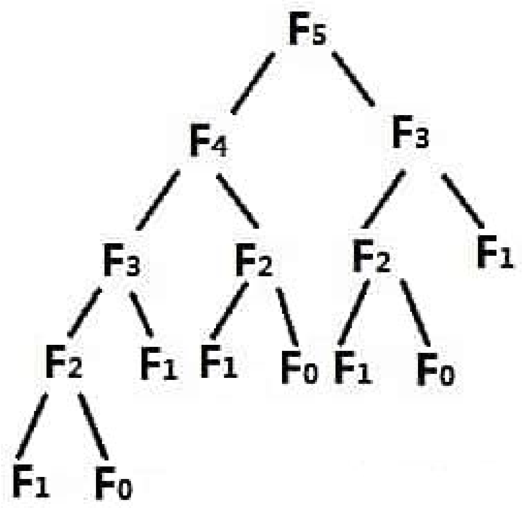
$$\begin{vmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{vmatrix} = \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}^n$$

2.1 递归的概念--*Fibonacci*数列

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

```
long long Fib(int n)//递归
{
        if (n < 2)
        {
                return n;
        }
        return Fib(n - 1) + Fib(n - 2);
}
```

我们把 F_5 作为树的根节点， F_4 和 F_3 作为左右两个叶子节点，继续向下递归，左节点 F_4 继续向下分解为 F_3 和 F_2 ，右节点 F_3 继续向下分解为 F_2 和 F_1 ，依此类推，如下图所示：



因此，该算法的时间复杂度为 $O(2^n)$。

2.1 递归的概念--*Fibonacci*数列

➢非递归解法：

```
int Fibonacci(int n) {
        if (n<=0) {
            return 0;
        }
        if (n==1) {
            return 1;
        }
        int min=0;
        int max=1;
        int i=2;
        int result=0;
        while (i<=n) {
            result=min+max;
            min=max;
            max=result;
            ++i;
        }
        return result;
    }
```

如果说前面的递归解法是自顶向下将大问题拆解成小问题求解，那么循环解法则是逆向思维，自底向上，先求出小问题的解，再向上一步一步向上求取最终问题的解。

单层循环，时间复杂度为 $O(n)$

2.1 递归的概念--*Fibonacci*数列

➢矩阵连乘法：

根据斐波那契数列自身的性质，我们可以构造如下方等式关系：

$$\begin{cases} F_2 = F_1 + F_0 \\ F_1 = F_1 \end{cases}$$

使用矩阵表示上述等式关系，即

$$\begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

那么

$$\begin{bmatrix} F_3 \\ F_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

依次乘下去，可得一般形式

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

现在求$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$的特征值。使

$$\mathbf{Det}\left(\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} - \lambda\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right)$$
$$= \lambda^2 - \lambda - 1 = 0$$

解出$\lambda_1 = \frac{1+\sqrt{5}}{2}$，$\lambda_2 = \frac{1-\sqrt{5}}{2}$

所以矩阵$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$的特征向量为

$$\vec{x}_1 = \begin{bmatrix} \frac{1+\sqrt{5}}{2} \\ 1 \end{bmatrix}$$

$$\vec{x}_2 = \begin{bmatrix} \frac{1-\sqrt{5}}{2} \\ 1 \end{bmatrix}$$

线性无关

两个特征值为 λ_1=1.618，λ_2=-0.618

→ O(1.618^n)

2.1 递归的概念--*Fibonacci*数列

➤矩阵连乘法：

根据斐波那契数列自身的性质，我们可以构造如下方等式关系：

$$\begin{cases} F_2 = F_1 + F_0 \\ F_1 = F_1 \end{cases}$$

使用矩阵表示上述等式关系，即

$$\begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

那么

$$\begin{bmatrix} F_3 \\ F_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

依次乘下去，可得一般形式

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

现在寻找方法使$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = T\Lambda^n T^{-1}$

这里Λ为对角阵

由相似对角化原理知

$$T = \begin{bmatrix} \frac{1+\sqrt{5}}{2} & \frac{1-\sqrt{5}}{2} \\ 1 & 1 \end{bmatrix}$$

$$T^{-1} = \begin{bmatrix} \frac{1}{\sqrt{5}} & \frac{-1+\sqrt{5}}{2\sqrt{5}} \\ -\frac{1}{\sqrt{5}} & \frac{1+\sqrt{5}}{2\sqrt{5}} \end{bmatrix}$$

此时$T^{-1}AT = \Lambda = \begin{bmatrix} \frac{1+\sqrt{5}}{2} & 0 \\ 0 & \frac{1-\sqrt{5}}{2} \end{bmatrix}$

从而 $\begin{bmatrix} a_{n+2} \\ a_{n+1} \end{bmatrix} = T\Lambda^n T^{-1} \begin{bmatrix} a_2 \\ a_1 \end{bmatrix}$

$$= \begin{bmatrix} \frac{1+\sqrt{5}}{2} & \frac{1-\sqrt{5}}{2} \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \frac{1+\sqrt{5}}{2} & 0 \\ 0 & \frac{1-\sqrt{5}}{2} \end{bmatrix}^n \begin{bmatrix} \frac{1}{\sqrt{5}} & \frac{-1+\sqrt{5}}{2\sqrt{5}} \\ -\frac{1}{\sqrt{5}} & \frac{1+\sqrt{5}}{2\sqrt{5}} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

换元，取一个行得到

$$a_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

$n \in N_+$

这就是斐波那契数列的通项公式。

我们也可以发现一个全是整数的数列通项公式竟然有无理数。而且和黄金分割率有关。

$\rightarrow$ O(logn)

2.1 递归的概念--*Fibonacci*数列

- 三种解法的比较
  - 解法1： $O(1.618^n)$
  - 解法2： $O(n)$
  - 解法3： $O(logn)$

fib(110)：

$O(1.618^n)$	→	10^{22} 次运算
$O(n)$	→	111 次运算
$O(logn)$	→	7 次运算

2.1 递归的概念

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

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

- 第n个Fibonacci数可递归地计算如下：

```
public static int fibonacci(int n)
  {
     if (n <= 1) return 1;
     return fibonacci(n-1)+fibonacci(n-2);
  }
```

2.1 递归的概念--*Fibonacci*数列

- 思考：
  - 楼梯问题
    - 有一楼梯共有n阶，上楼可以一步上一阶，也可以一步上两阶。
    - 编一个程序，计算共有多少种不同的走法？

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

➢思考：

- 数字转换字符串问题 **百度面试真题**
  - 将给定的数转换为字符串，原则如下：1对应 a，2对应b，…..26对应z，例如12258可以转换为"abbeh", "aveh", "abyh", "lbeh" and "lyh"，个数为5，编写一个函数，给出可以转换的不同字符串的个数。

```
Please input the serial numbers:
11020

Total number of strings: 6
aab
kb
ajb
aat
kt
ajt
```

```
Please input the serial numbers:
1234

Total number of strings: 3
abcd
lcd
awd
```

```
Please input the serial numbers:
12235

Total number of strings: 5
abbce
lbce
avce
abwe
lwe
```

- 数字转换字符串问题

百度面试真题

➢每个位置两种选择：

1. 将当前单位数字翻译
2. 当前和下一位数字集合成两位数翻译（两位数需要小于26）

➢递归结束条件：

1. *最后一个数字时：只能以单个字符翻译，结束，返回1；*
2. *对于其他情况：起码都还剩余两个数字，所以我们先判断两位数字是不是满足[10,25]，满足的话说明有两种选择，返回f(i+1) + f(i+2)，否则只有一种选择，返回f(i+1);*
3. *特殊情况：当i为倒数第二个数时，即下标为n-2，此时如果最后两位数字构成的两位数满足要求（[10,25]），则有两种选择：分别是两个数字单独翻译以及两个数字合成一个翻译，应该返回2，即返回f(i+1) + f(i+2)，这时f(i+1)为1，而i+2超出n的范围了，应该返回1。递归结束条件进行修改，i>=n-1退出递归，返回1。*

2.1 递归的概念----*Ackerman*函数

- 例3 Ackerman函数
- 当一个函数及它的一个变量是由函数自身定义时，称这个函数是**双递归函数**。Ackerman函数A(n，m)定义如下：
- 前2例中的函数都可以找到相应的非递归方式定义。
- 但本例中的Ackerman函数却无法找到非递归的定义。

$$
\begin{cases}
A(1,0)=2 & \\
A(0,m)=1 & m\ge 0 \\
A(n,0)=n+2 & n\ge 2 \\
A(n,m)=A(A(n-1,m),m-1) & n,m\ge 1
\end{cases}
$$

$$n!=1\cdot 2\cdot 3\cdot\dots\cdot(n-1)\cdot n$$

$$F(n)=\frac{1}{\sqrt{5}}\left[\left[\frac{1+\sqrt{5}}{2}\right]^{n+1}-\left[\frac{1-\sqrt{5}}{2}\right]^{n+1}\right]$$

2.1 递归的概念----*Ackerman*函数

- *Ackerman*函数
  - $A(n,0)=n+2$
  - $A(n,1)=2n$
  - $A(n,2)=2^n$。
  - $A(n,3)=\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n}$

$$A(n,m)=\begin{cases}2 & n=1,m=0\\ 1 & n=0,m\geq 0\\ n+2 & n\geq 2,m=0\\ A(A(n-1,m),m-1) & n,m\geq 1\end{cases}$$

  - $A(n,4)$的增长速度非常快，以致于没有适当的数学式子来表示这一函数。

$$A(3,4)=\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{65536}$$

2.1 递归的概念----*Ackerman*函数

```
int Ackerman(int n,int m)
{
  if( n= =1&&m= =0 )
        return 2;
  else if (n= =0&&m>=0)
            return 1;
        else if(n>=2&&m= =0)
                return n + 2;
             else if(n>=1&&m>=1)
                    return Ackerman(Ackerman(n - 1,m),m - 1);
}
```

$$A(n,m)=\begin{cases} 2 & n=1,m=0 \\ 1 & n=0,m\ge 0 \\ n+2 & n\ge 2,m=0 \\ A(A(n-1,m),m-1) & n,m\ge 1 \end{cases}$$

应用：路径压缩算法中，在集合的查找过程中将树的深度降低。

2.1 递归的概念----排列问题

- 设A={$a_1,a_2,\ldots,a_n$}是要进行排列的n个元素的集合,
- n=1　　输出a_1
- n=2　　输出a_1a_2

　　　　　　a_2a_1
- n=3　　输出$a_3a_1a_2$

　　　　　　$a_3a_2a_1$

　　　　　　$a_1a_2a_3$

　　　　　　$a_1a_3a_2$

　　　　　　$a_2a_1a_3$

　　　　　　$a_2a_3a_1$

分析n=3，排列按如下步骤进行：

(1) a_3之后跟a_1,a_2的所有全排列；

(2)在上述全排列里，a_3和a_1位置互换；

(3)在上述全排列里，a_3和a_2位置互换。

2.1 递归的概念----排列问题

- **range(A)**

$\boxed{A\text{-}a_1}$ $\boxed{A\text{-}a_2}$ $\boxed{A\text{-}a_n}$

$$= a_1 range(A_1),\ a_2 range(A_2), \ldots,\ a_n range(A_n)$$

集合A用数组实现

range(A,1,*n*)：

递归出口：range(A, *n*, *n*)

递归调用：使得集合所有元素都可以作为前缀出现

2.1 递归的概念----排列问题

procedure range(A, k, n)

if k=n then print(A)

else for $i \leftarrow k$ to n do

A(k) $\leftrightarrow$ A(i)

call range(A,k+1,n)

A(k) $\leftrightarrow$ A(i)

repeat

endif

end range

递归出口，打印整个数组**A**。

A(*i*)与**A(*k*)**值互换

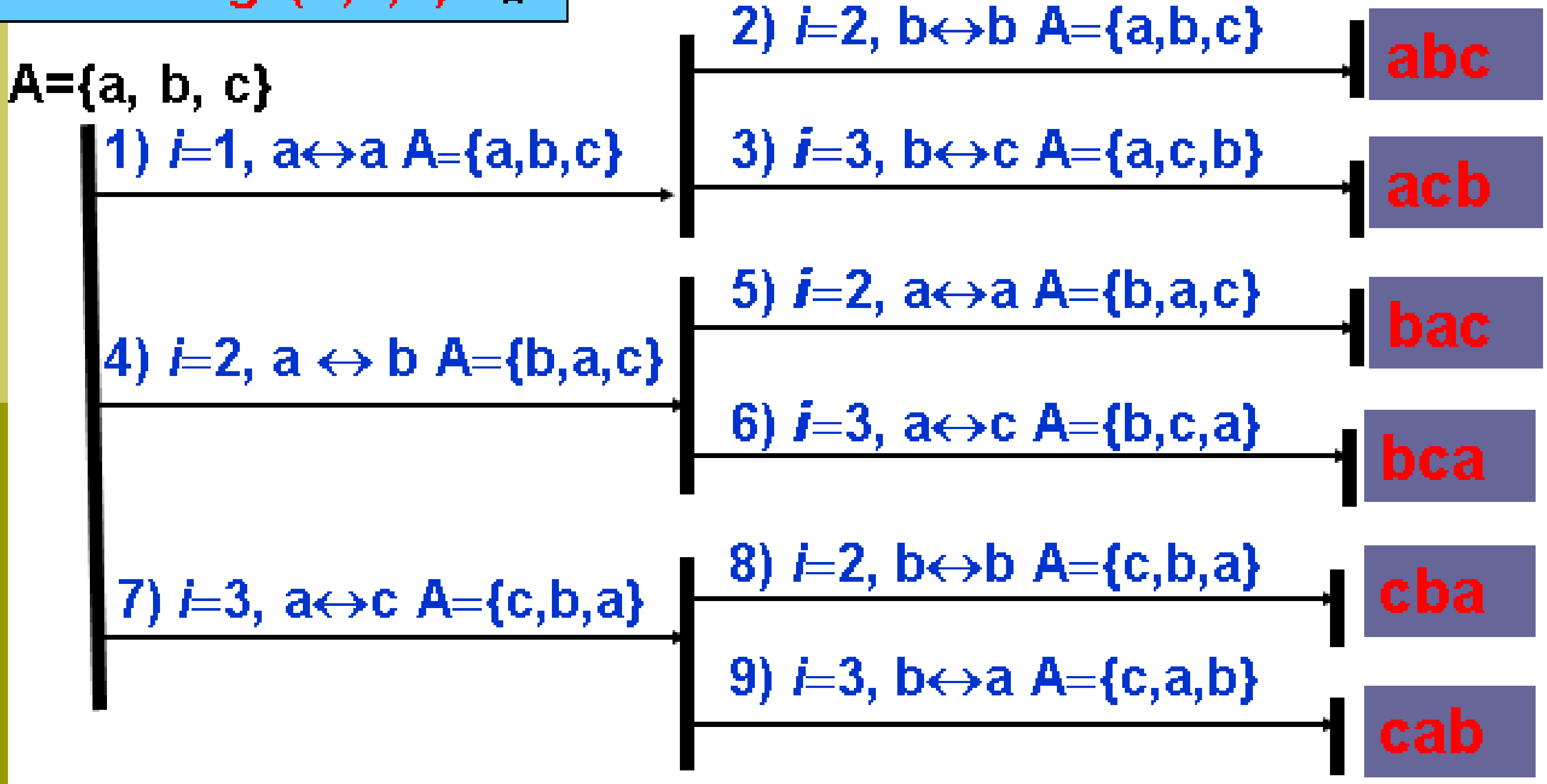
缺省值变化时不回传，交换返回原始状态。

call range(A,1,n)

range(A,1,3)
If 1=3 then print(A)
else
  for $i \leftarrow 1$ to 3 do
    $A(1) \leftrightarrow A(i)$;
    call range(A,2,3) 略

range(A,2,3)…
for $i \leftarrow 2$ to 3 do
    $A(2) \leftrightarrow A(i)$;
    call range(A,3,3) 略

range(A,3,3)
If 3=3 print(A) 略



2.1 递归的概念----整数划分

- 任何一个大于1的自然数*n*，总可以拆分成若干个小于*n*的自然数之和，试求*n*的所有拆分。

- *n*=2　2=1+1
- *n*=3　3=1+2

　　　=1+1+1
- *n*=4　4=1+3

　　　=1+1+2

　　　=1+1+1+1

　　　=2+2

2.1 递归的概念----整数划分

- 分析：
- 将最大加数n_1不大于m的划分个数记作$q(n,m)$

$$q(n,m)=\begin{cases}1\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots m=1\vee n=1\\ q(n,n)\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots n<m\\ 1+q(n,n-1)\cdots\cdots\cdots\cdots\cdots\cdots n=m\\ q(n,m-1)+q(n-m,m)\cdots\cdots 1<m<n\end{cases}$$

正整数n的划分数
p(n)=q(n,n)。

6；
5+1；
4+2，　　4+1+1；
3+3，　　3+2+1，　　3+1+1+1；
2+2+2，　　2+2+1+1，　　2+1+1+1+1；
1+1+1+1+1+1;

2.1 递归的概念----整数划分

$$q(n,m)=\begin{cases}1\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots & m=1\vee n=1\\ q(n,n)\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots & n<m\\ 1+q(n,n-1)\cdots\cdots\cdots\cdots\cdots\cdots\cdots & n=m\\ q(n,m-1)+q(n-m,m)\cdots\cdots\cdots & 1<m<n\end{cases}$$

```
int q(int n,int m)
{   if((n<1)||(m<1)) return 0;
    if(n==1||m==1)  return 1;
    if(n<m)         return q(n,n);
    if(n==m)         return q(n,m-1)+1;
    return q(n,m-1) + q(n-m,m);
 }
```

2.1 递归的概念----简单的0/1背包问题

例：$m=20$, $n=5$,

$(m_1, m_2, m_3, m_4, m_5) = (3,5,8,9,10)$

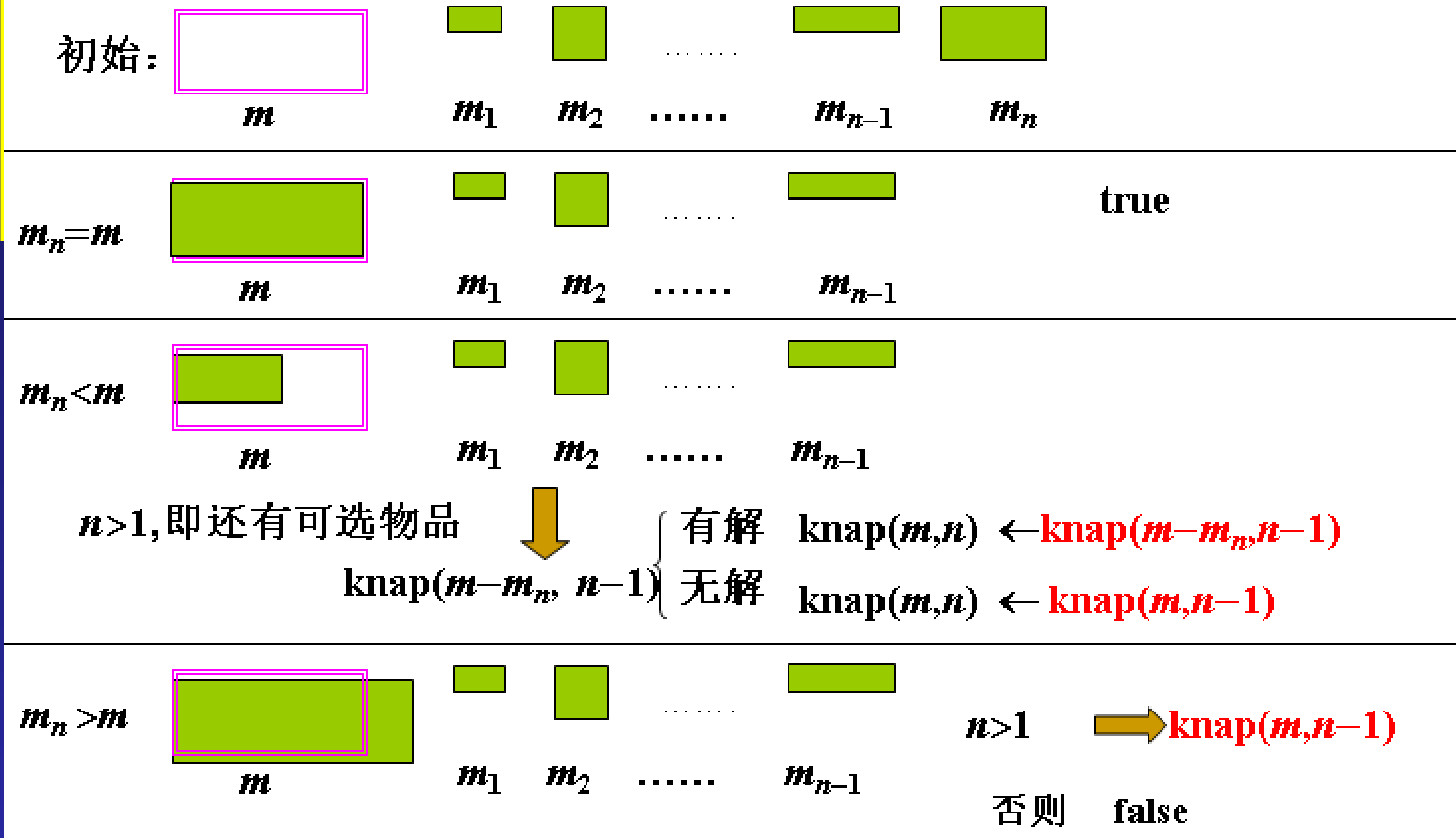
$(x_1,x_2,x_3,x_4,x_5)=(1,0,1,1,0)$

$m=18$?　　　$m=28$?

注：对于第*i*件物品要么取，要么舍，不能取一部分，因此这个问题可能有解，也可能无解。

布尔函数

问题分析 knap(m, n)



初始：
m
m_1 m_2 …… m_{n-1} m_n
$m_n=m$
m
m_1 m_2 …… m_{n-1}
true
$m_n<m$
m
m_1 m_2 …… m_{n-1}
$n>1$,即还有可选物品
knap($m-m_n$, $n-1$)
有解 knap(m,n) ←knap($m-m_n$,$n-1$)
无解 knap(m,n) ← knap(m,$n-1$)
$m_n>m$
m
m_1 m_2 …… m_{n-1}
$n>1$
knap(m,$n-1$)
否则 false

2.1 递归的概念---Hanoi塔传说

- 在贝拿勒斯（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(Tower of Hanoi）。
- 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。

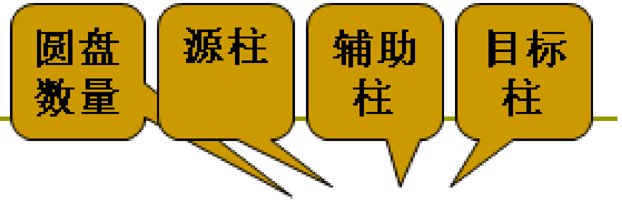
假如每秒钟一次，移完这些金片需要**5845**亿年以上。

n阶Hanoi塔问题

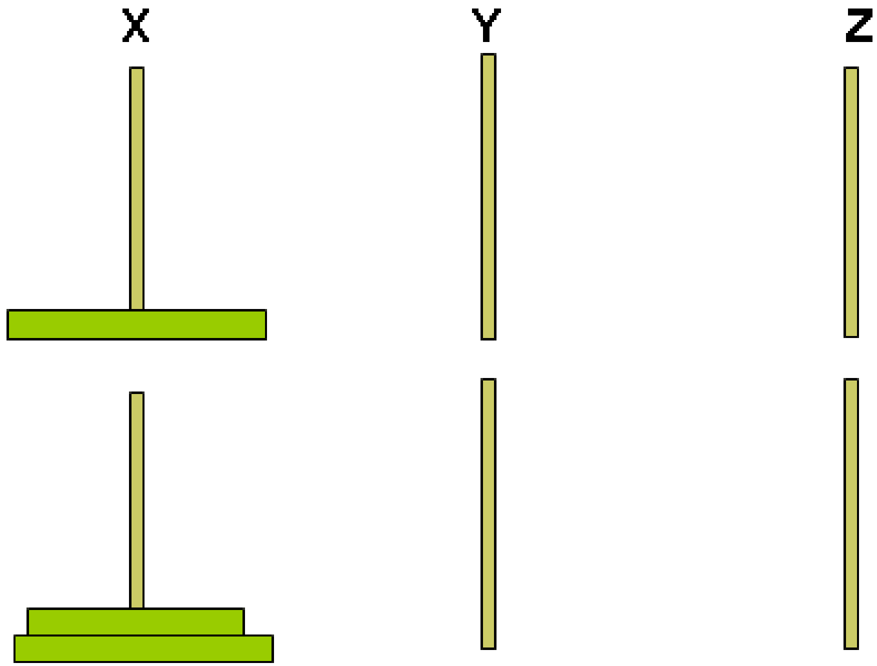
- 有n个圆盘依半径从小到大自上而下地套在柱子X上，柱子Y和Z没有圆盘。要求将X上的盘子换到Z上，每次只移动一个，且不允许将大圆盘压在小圆盘的上面。

2.1 递归的概念---Hanoi塔问题

寻找递归出口



Hanoi(n,X,Y,Z)



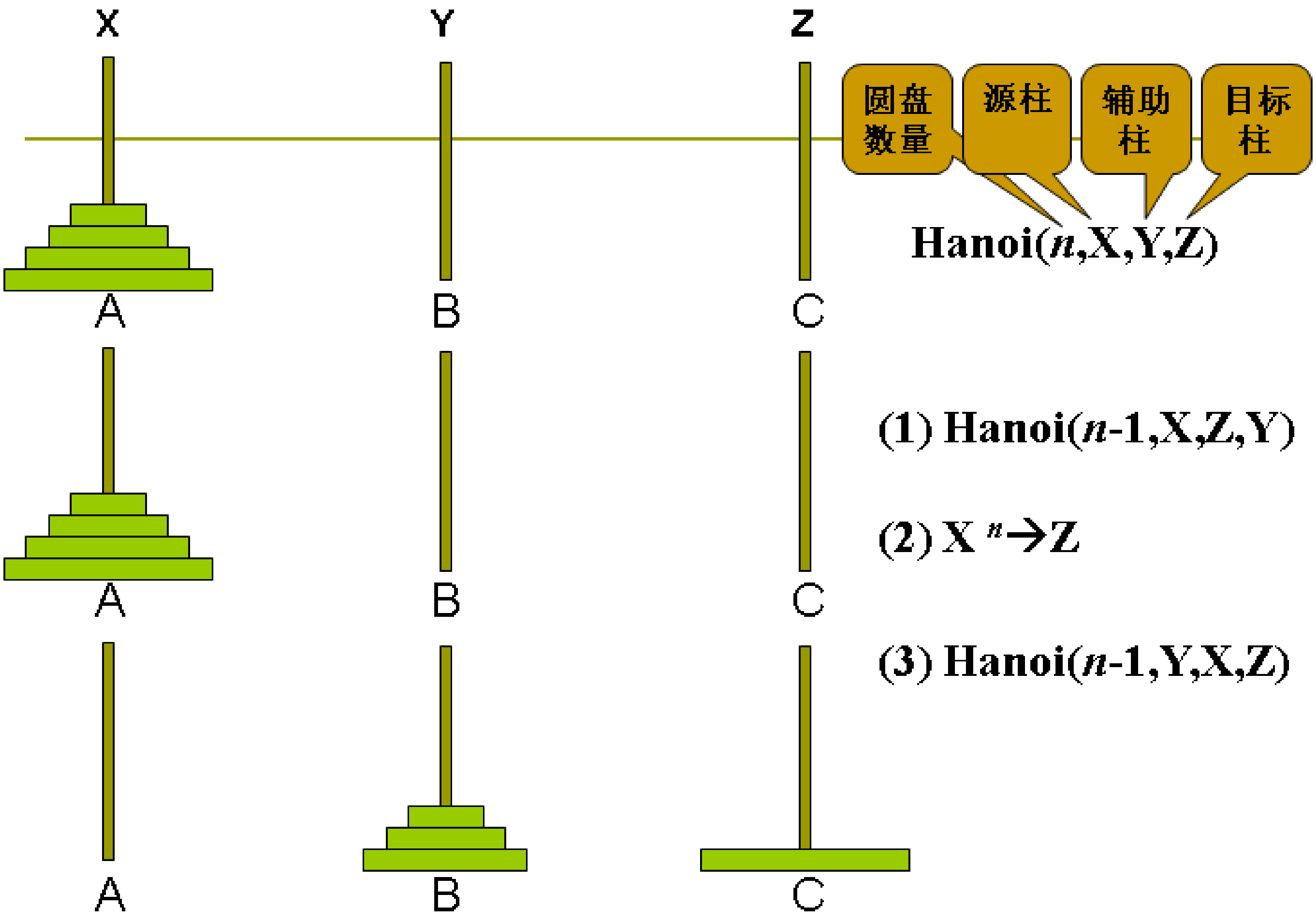
(1) n=1, $X^1 \rightarrow Z$

(2) n=2, $X^1 \rightarrow Y$

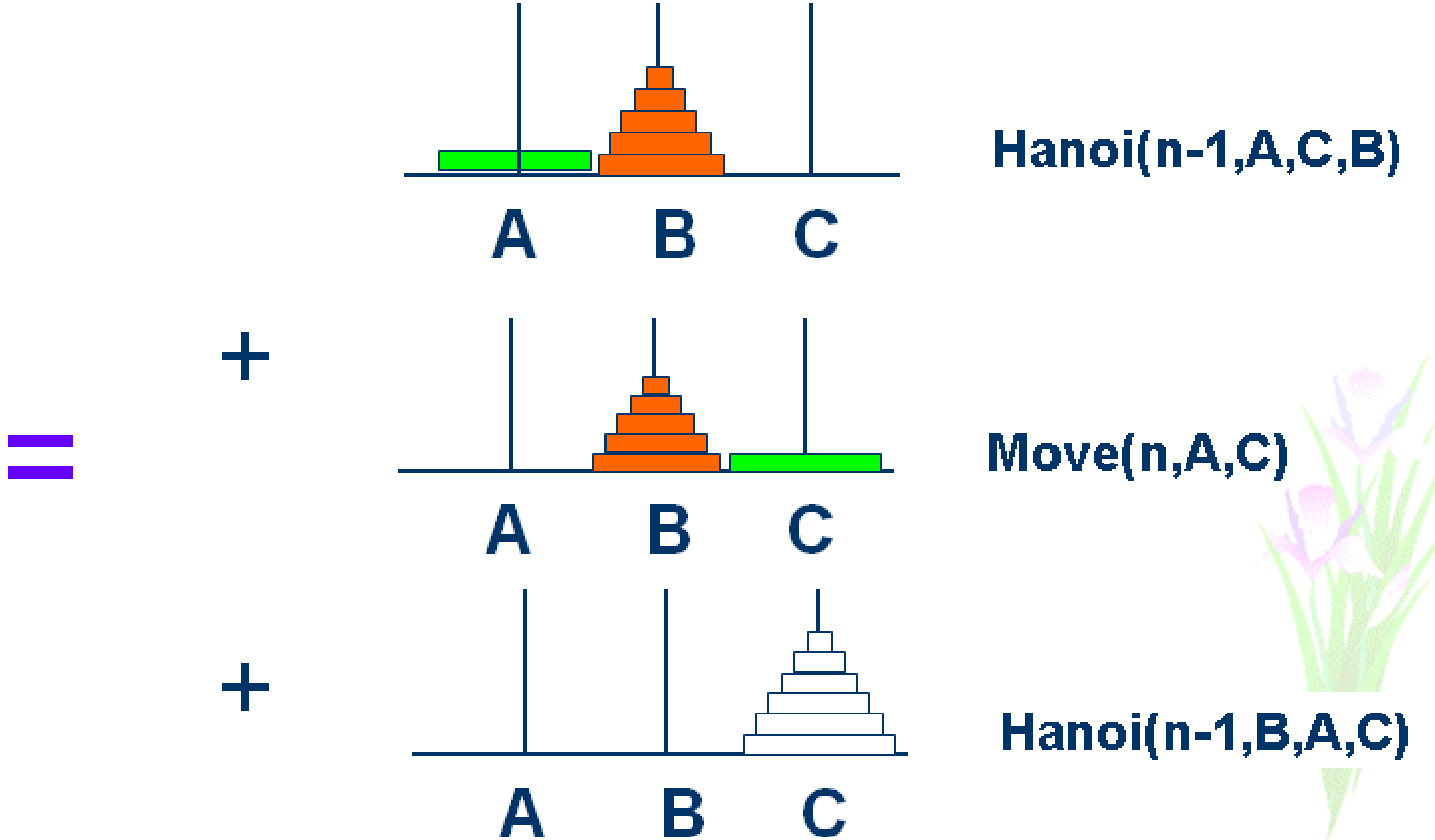
$X^2 \rightarrow Z$

$Y^1 \rightarrow Z$

2.1 递归的概念---Hanoi塔问题

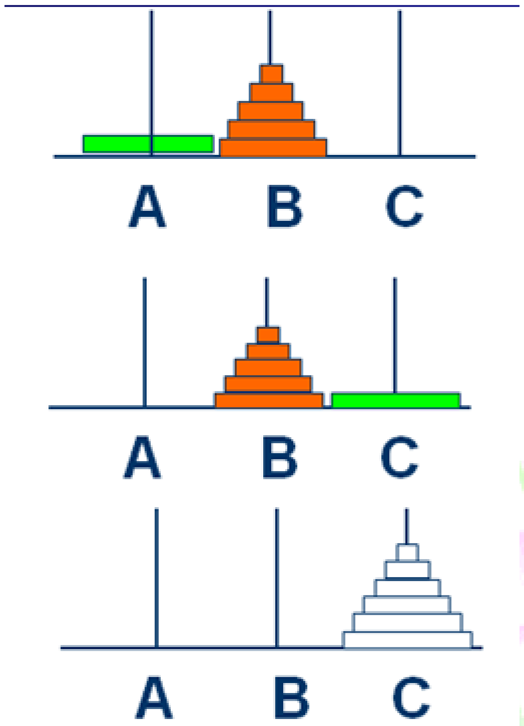


2.1 递归的概念---Hanoi塔问题

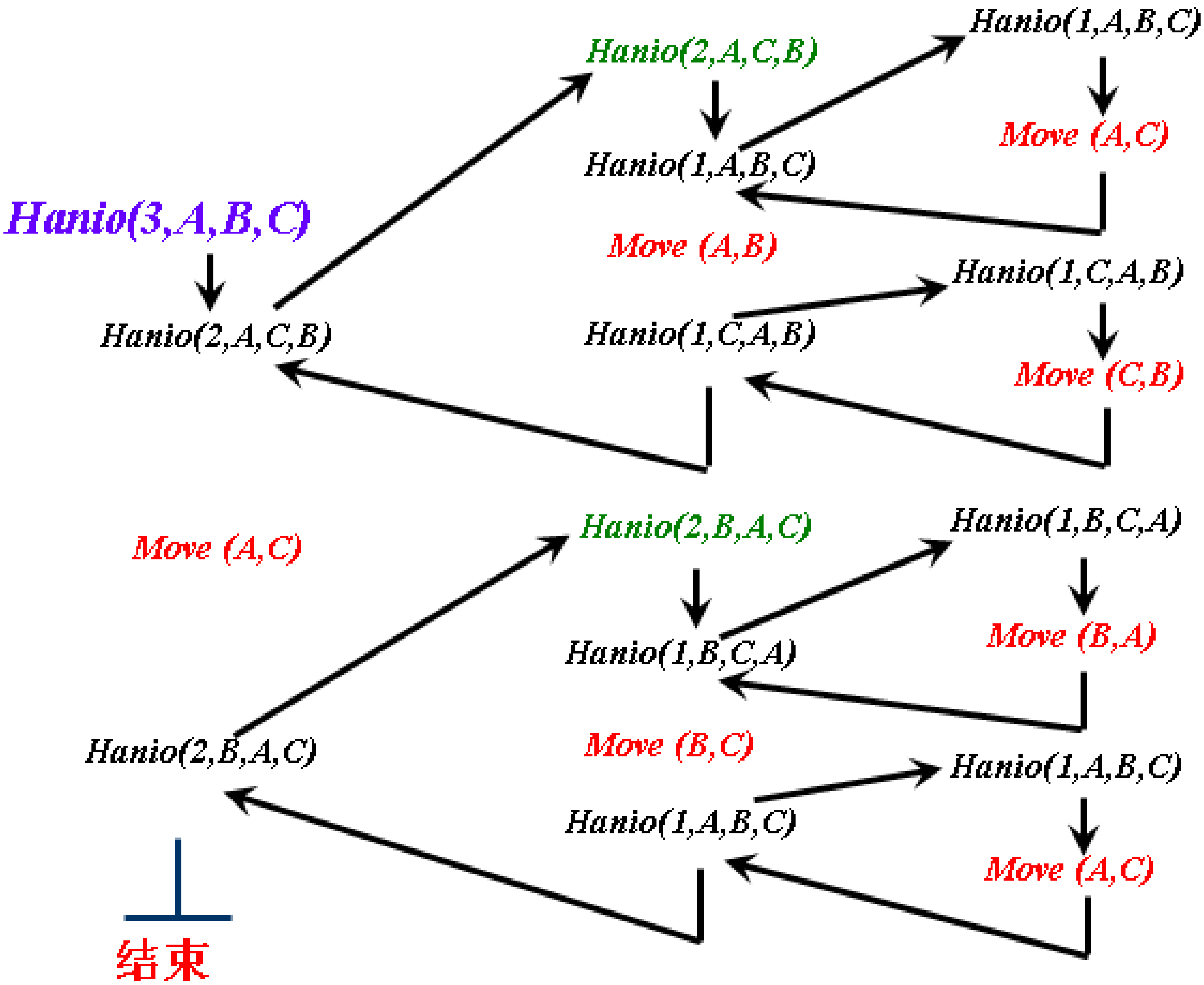


递归求解

```
void hanoi(int n, int a, int b, int c)
  {  if   (n== 1)          move(a,b);
     else  {
                  hanoi(n-1, a, c, b);
                  move(a,b);
                  hanoi(n-1, c, b, a);
           }
  }
```



递归函数的运行轨迹


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

2.1 递归的概念---Hanoi塔问题

　　这个问题有一个简单的解法。假设塔座A、B、C排列成一个三角形，A——>B——>C——>A构成一顺时针循环。在移动圆盘的过程中，若是奇数次移动，则将最小的圆盘移到顺时针方向的下一塔座上；若是偶数次移动，则保持最小的圆盘不动。而其他两个塔座之间，将较小的圆盘移到另一塔座上去。

2.1 递归的概念---Hanoi塔问题

时间复杂性分析:

- 规模为*n*的*Hanoi(n)*问题，可以分解为2个规模为*n-1*的*Hanoi(n-1)*问题和一个*Move* 操作。
- 所以，*n*个盘子的移动次数为:

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

$$\Rightarrow T(n)=2^n-1$$

若*n*=64，则移动次数为$2^{64}-1$

$2^{64}-1=18,446,744,073,709,551,615$

2.1 递归的概念---Hanoi塔问题

- $2^{64}-1=18,446,744,073,709,551,615$是个什么概念？
  - 实例1：
    - 假设每秒钟移动一次，一年约31556926秒，
    - 计算表明：移动64个盘子需要5800多亿年。
  - 实例2：
    - 国王的麦子问题
      - 一个高4米、宽10米的粮仓装麦子，这个粮仓有3000万公里长，能绕地球赤道700圈，可以把地球全部表面（包括海洋）铺上2米厚的小麦层！它相当于全世界2000多年小麦产量的总和.

2.1 递归的概念---查找数组最大数据项

- 练习：在数组a[0],……, a[n-1]的n个项中找出最大数据项。用递归算法来实现。

```
item max(item a[],int l,int r)
{ item u,v; int m=(l+r)/2;      //取数组下标的中点
  If (l= =r)
     return a[l]; //只有一个元素的情况，直接返回
  u=max(a,l,m);                  //递归地对左半部分取最大值
  v=max(a,m+1,r);                //递归地对右半部分取最大值
  if (u>v)
     return u;
  else return v;
}
```

算法是将数组a[1],…,a[r]分成a[1],…a[m]和a[m+1],…a[r]两部分，分别求出每一部分的最大元素（递归地），并返回较大的那一个作为整个数组的最大元素。

2.1 递归的概念

- 在运行递归算法时：系统需要在运行被调用算法之前完成三件事：

1. 将所有实参指针，返回地址等信息传递给被调用算法；
2. 为被调用算法的局部变量分配存储区；
3. 将控制转移到被调用算法的入口。

- 在从被调用算法返回调用算法时：系统也相应地要完成三件事：

1. 保存被调用算法的计算结果；
2. 释放分配给被调用算法的数据区；
3. 依照被调用算法保存的返回地址将控制转移到调用算法。

2.1 递归的概念

- 递归小结
  - 优点：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、调试程序带来很大方便。
  - 缺点：递归算法的运行效率较低，无论是耗费的计算时间还是占用的存储空间都比非递归算法要多。
  - 解决方法：在递归算法中消除递归调用，使其转化为非递归算法。

思考：

扰乱字符串问题

2.1 递归的概念

下图是字符串 *s1* = "great" 的一种可能的表示形式。

爱奇艺面试真题

```
    great
   /    \
  gr    eat
 / \    /  \
g   r  e   at
           / \
          a   t
```

在扰乱这个字符串的过程中，我们可以挑选任何一个非叶节点，然后交换它的两个子节点。

例如，如果我们挑选非叶节点 "gr" ，交换它的两个子节点，将会产生扰乱字符串 "rgeat" 。

```
    rgeat
   /    \
  rg    eat
 / \    /  \
r   g  e   at
           / \
          a   t
```

我们将 "rgeat" 称作 "great" 的一个扰乱字符串。

思考：

扰乱字符串问题

2.1 递归的概念

同样地，如果我们继续交换节点 "eat" 和 "at" 的子节点，将会产生另一个新的扰乱字符串 "rgtae" 。

```
    rgtae
   /    \
  rg    tae
 / \    / \
r   g  ta  e
       / \
      t   a
```

爱奇艺面试真题

我们将 "rgtae" 称作 "great" 的一个扰乱字符串。

给出两个长度相等的字符串 *s1* 和 *s2*，判断 *s2* 是否是 *s1* 的扰乱字符串。

示例 1:

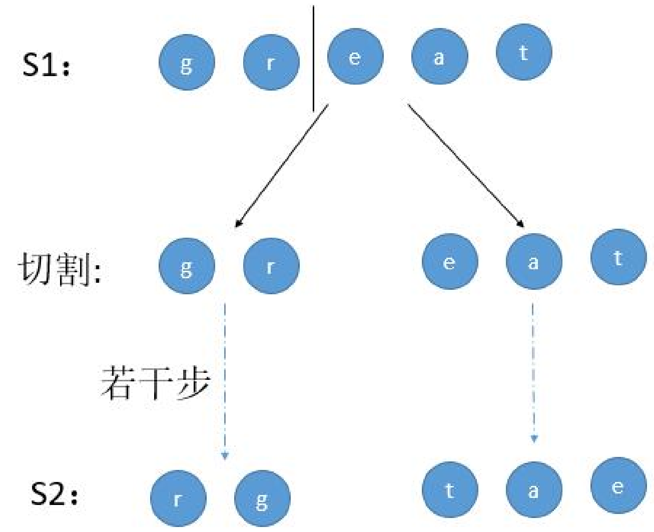
```
输入: s1 = "great", s2 = "rgeat"
输出: true
```

示例 2:

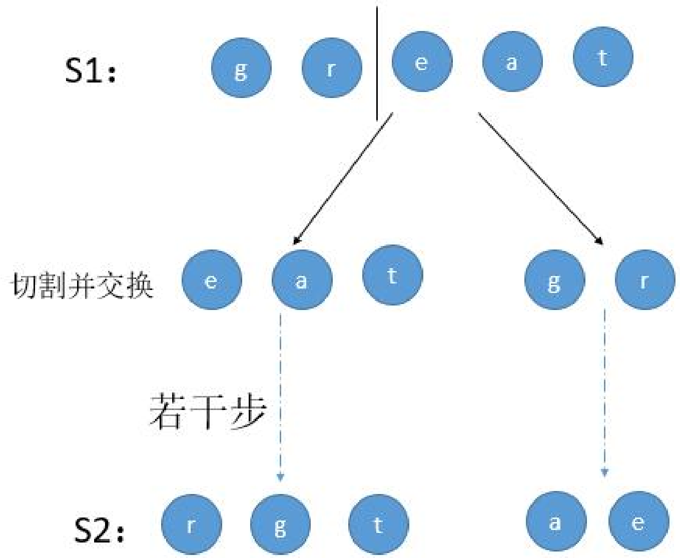
```
输入: s1 = "abcde", s2 = "caebd"
输出: false
```

递归的概念

第 1 种情况：S1 切割为两部分，然后进行若干步切割交换，最后判断两个子树分别是否能变成 S2 的两部分。



第 2 种情况：S1 切割并且交换为两部分，然后进行若干步切割交换，最后判断两个子树是否能变成 S2 的两部分。



思考：

扰乱字符串问题

思考：

扰乱字符串问题

2.1 递归的概念

首先，这两个字符串长度要相等；
其次，这两个字符串组成要一样；
然后，S1和S2中，至少存在一个i，使得S1的前半段和S2的前半段扰动，S1后半段和S2后半段扰动；或者S1前半段和S2后半段扰动，S1后半段和S2前半段扰动；

递归方程

1. length(S1)==length(S2)

2. compose(S1)==compose(S2)

3. 至少存在一个整数i在[1, length(S2)], 使得(--符号表示是扰动字符串)：

S1(0:i)-- S2(0:i) && S1(i:end)-- S2(i:end)

or S1(0:i)-- S2(end-i:end) && S1(end-i:end)-- S2(0:i)

```java
public boolean isScramble(String s1, String s2)
{
   if (s1.length() != s2.length())
   {
      return false;
   }

   if (s1.equals(s2))
   {
      return true;
   }
```

```java
//判断两个字符串每个字母出现的次数是否一致
int[] letters = new int[26];
for (int i = 0; i < s1.length(); i++)
{
   letters[s1.charAt(i) - 'a']++;
   letters[s2.charAt(i) - 'a']--;
}

//如果两个字符串的字母出现不一致直接返回 false
for (int i = 0; i < 26; i++)
{
   if (letters[i] != 0)
   {
      return false;
   }
}
```

```java
   //遍历每个切割位置
   for (int i = 1; i < s1.length(); i++)
   {
    //对应情况 1，判断 S1 的子树能否变为 S2 相应部分
    if (isScramble(s1.substring(0, i), s2.substring(0, i)) && isScramble(s1.substring(i), s2.substring(i)))
    {
       return true;
    }
    //对应情况 2，S1 两个子树先进行了变换，然后判断 S1 的子树能否变为 S2 相应部分
    if (isScramble(s1.substring(i), s2.substring(0, s2.length() - i)) &&
      isScramble(s1.substring(0, i), s2.substring(s2.length() - i)) )
    {
       return true;
    }
   }
   return false;
}
```

爱奇艺面试真题

思考：

扰乱字符串问题

思考：

外观数列问题

2.1 递归的概念

给定一个正整数 `n` ，输出外观数列的第 `n` 项。

「外观数列」是一个整数序列，从数字 1 开始，序列中的每一项都是对前一项的描述。

你可以将其视作是由递归公式定义的数字字符串序列：

- `countAndSay(1) = "1"`
- `countAndSay(n)` 是对 `countAndSay(n-1)` 的描述，然后转换成另一个数字字符串。

华为面试真题

前五项如下：

```
1.     1
2.     11
3.     21
4.     1211
5.     111221
第一项是数字 1
描述前一项，这个数是 1 即 “ 一 个 1 ”，记作 "11"
描述前一项，这个数是 11 即 “ 二 个 1 ” ，记作 "21"
描述前一项，这个数是 21 即 “ 一 个 2 + 一 个 1 ” ，记作 "1211"
描述前一项，这个数是 1211 即 “ 一 个 1 + 一 个 2 + 二 个 1 ” ，记作 "111221"
```

思考:

外观数列问题

求字符串中连续出现的数字的次数,并通过次数+数字组成新的字符串
通过递归的方法求解:
str = countAndSay(n - 1)
然后对字符串str进行遍历,通过count进行计数
如果str的第i个和第i + 1个数字不相等那么就可以写入结果中,并且把count置为1,
如果相等就继续计数count++
直到递归终止,也就是n==1的时候

华为面试真题

```
String countAndSay(int n)
  {
        if (n == 1) {
            return "1";
        }
        StringBuffer res = new StringBuffer();
        String str = countAndSay(n - 1);
        int count = 1;
        for (int i = 0; i < str.length(); i++) {
            if (i == str.length() - 1 || str.charAt(i) != str.charAt(i + 1)) {
                res.append(count).append(str.charAt(i));
                count = 1;
            } else {
                count++;
            }
        }
        return res.toString();
   }
```

2.2 分治法的基本思想

➢ 分治法的基本思想

- 将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

分治法（*Divide and Conquer*）的基本思想:

- **分解(*Divide*):**
  - 将一个规模为n的问题，分解为k个规模较小的子问题，这些子问题互相独立且与原问题形式相同。
- **求解(*Conquer*):**
  - 若子问题规模较小而容易被解决则直接解，否则递归地解这些子问题。
- **合并(*Merge*):**
  - 将各个子问题的解合并得到原问题的解。

➢2.2 分治法

Problem 1

[找伪币问题]给你一个装有1 6个硬币的袋子。1 6个硬币中有一个是伪造的，并且那个伪造的硬币比真的硬币要轻一些。你的任务是找出这个伪造的硬币。为了帮助你完成这一任务，将提供一台可用来比较两组硬币重量的仪器，利用这台仪器，可以知道两组硬币的重量是否相同。

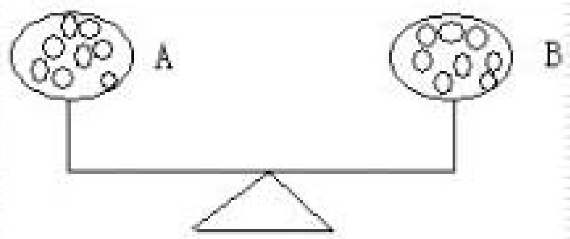
2.2 分治法的基本思想

Solution

- 一种方式
  - 两两对比，找到轻者，最差比较8次。
- 另外一种

1) 将16个硬币分成A、B两半；

2) 将A放仪器的一边，B放另一边，如果A袋轻，则表明伪币在A，解子问题A即可，否则，解子问题B。



2.2 分治法的基本思想

Problem 2

例2:[金块问题]有一个老板有一袋金块。每个月将有两名雇员会因其优异的表现分别被奖励一个金块。按规矩，排名第一的雇员将得到袋中最重的金块，排名第二的雇员将得到袋中最轻的金块。假设有一台比较重量的仪器，我们希望用最少的比较次数找出最轻和最重的金块。

2.2 分治法的基本思想---金块问题

Solution(1)

- 1. （两两比较）假设袋中有**n** 个金块，通过**n-1**次比较找到最重的金块。找到最重的金块后，可以从余下的**n-1**个金块中用类似的方法通过**n-2**次比较找出最轻的金块。
- 比较的总次数为**2n-3**。

Solution(2)

- **2.** *（分治）* ***n***≤2时，做一次比较即可。

  n＞2时，

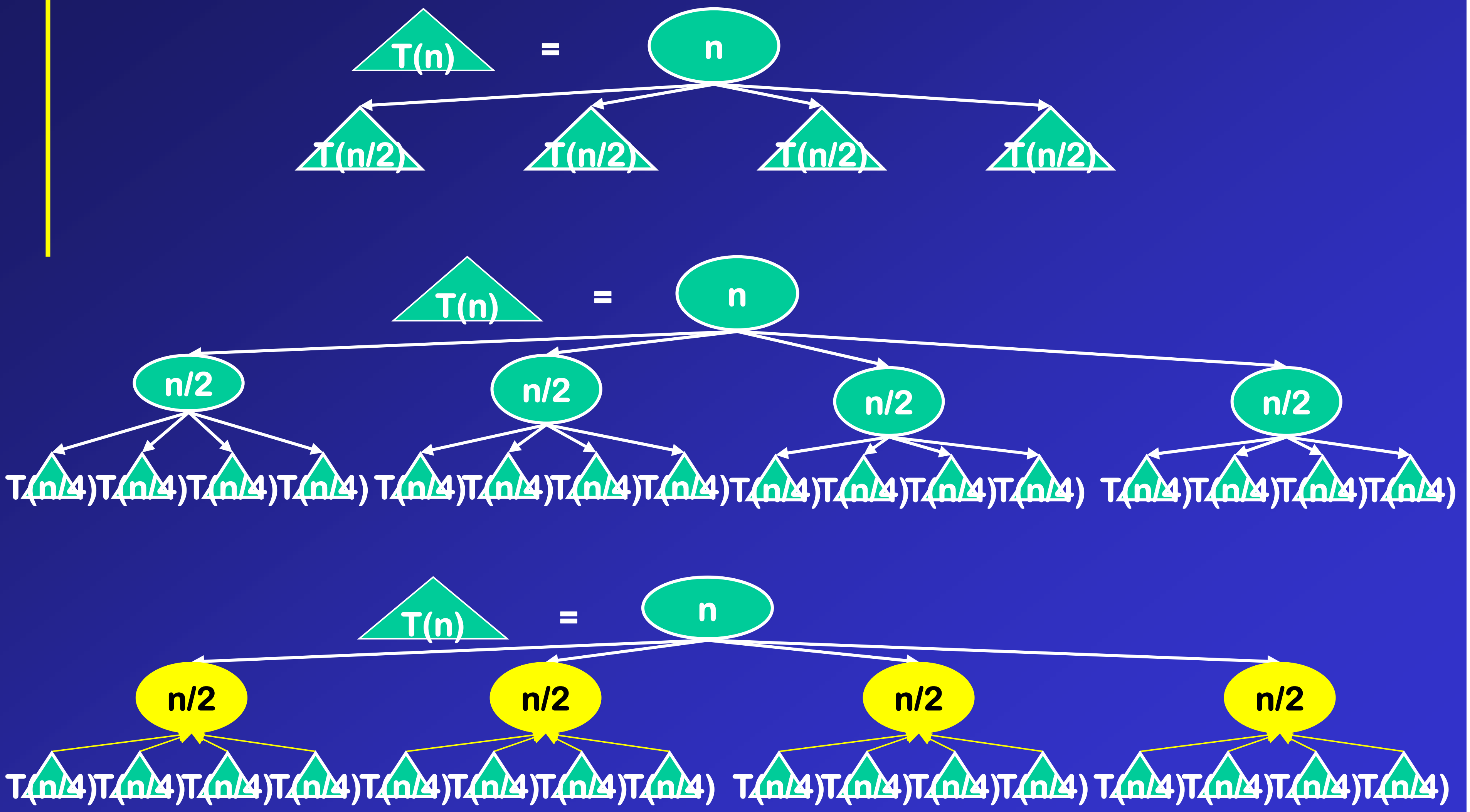
  第一步，把这袋金块平分成两个小袋***A***和***B***。

  第二步，分别找出在***A***和***B***中最重和最轻的金块：***HA*** 与***LA***，***HB*** 和***LB***。

  第三步，通过比较***HA*** 和***HB***，可以找到所有金块中最重的；通过比较***LA*** 和***LB***，可以找到所有金块中最轻的。

- 设**c(n)**为比较次数。假设**n**是**2**的幂。
  - 当**n= 2**时，**c(n) = 1**;
  - 当**n>2**时，**c(n) = 2c(n/ 2) + 2**。当**n**是**2**的幂时，可知**c(n) = 3n/ 2 - 2**。
- 使用分而治之方法比逐个比较的方法少用了**2 5**%的比较次数。

2.2 分治法的基本思想



T(n) = n
T(n/2) T(n/2) T(n/2) T(n/2)
T(n) = n
n/2 n/2 n/2 n/2
T(n/4)T(n/4)T(n/4)T(n/4) T(n/4)T(n/4)T(n/4)T(n/4)T(n/4)T(n/4)T(n/4)T(n/4) T(n/4)T(n/4)T(n/4)T(n/4)
T(n) = n
n/2 n/2 n/2 n/2
T(n/4)T(n/4)T(n/4)T(n/4)T(n/4)T(n/4)T(n/4)T(n/4) T(n/4)T(n/4)T(n/4)T(n/4) T(n/4)T(n/4)T(n/4)T(n/4)

2.2 分治法的基本思想

- 分治法的适用条件
  - 分治法所能解决的问题一般具有以下几个特征：
    - 该问题的规模缩小到一定的程度就可以容易地解决；
    - 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质
    - 利用该问题分解出的子问题的解可以合并为该问题的解；
    - 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。
  - **这条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然也可用分治法，但一般用动态规划较好。**

2.2 分治法的基本思想

分治法的设计模式

```
divide-and-conquer(P) {
    if ( | P | <= n0) adhoc(P);    //解决小规模的问题
    divide P into smaller subinstances P1,P2,...,Pk;
                                   //分解问题
    for (i=1;i<=k;i++)             //递归地解各子问题
        yi=divide-and-conquer(Pi);
    return merge(y1,...,yk);       //将各子问题的解合并
                                   //为原问题的解
}
```

- 人们从大量实践中发现，**在用分治法设计算法时，最好使子问题的规模大致相同。**即将一个问题分成大小相等的k个子问题的处理方法是行之有效的。这种使子问题规模大致相等的做法是出自一种平衡(balancing)子问题的思想，它几乎总是比子问题规模不等的做法要好。

2.3 二分搜索技术

- 给定已按升序排好序的n个元素a[0:n-1]，现要在这n个元素中找出一特定元素x。
- 适用分治法求解问题的基本特征：
  - 该问题的规模缩小到一定的程度就可以容易地解决；
  - 该问题可以分解为若干个规模较小的相同问题;
  - 分解出的子问题的解可以合并为原问题的解；
  - 分解出的各个子问题是相互独立的。
- 很显然此问题分解出的子问题相互独立，即在a[i]的前面或后面查找x是独立的子问题，因此满足分治法的四个适用条件。

2.3 二分搜索算法

➢ 据此容易设计出二分搜索算法：

```
public static int binarySearch(int [] a, int x, int n)
{
    // 在 a[0] <= a[1] <= ... <= a[n-1] 中搜索 x
    // 找到x时返回其在数组中的位置，否则返回-1
    int left = 0; int right = n - 1;
    while (left <= right) {
        int middle = (left + right)/2;
        if (x == a[middle]) return middle;
        if (x > a[middle]) left = middle + 1;
        else right = middle - 1;
    }
    return -1; // 未找到x
}
```

2.4 大整数的乘法

- 大数
  - 计算机存储数据是按类型分配空间的。
  - 例如：在微型机上

数据类型	存贮字节	数据范围
int	2	－32768～32767
long int	4	－2147483648～2147483647
float	4（精确位数：6~7位）	-3.4e+38~3.4e+38
double	8（精确位数：15~16位）	-1.79e+308~1.79e+308

2.4 大整数的乘法

大数的存储方案

- 数组
  - 数值数组
    - 从计算的方便性考虑，决定将数据是由低到高还是由高到低存储到数组中；
    - 可以每位占一个数组元素空间，也可几位占一个数组元素空间。
  - 字符型数组
    - 从键盘输入要处理的高精度数据，无需对高精度数据进行分段输入，但计算是需要类型转换的操作。

2.4 大整数的乘法

- 问题描述：
  - 设X和Y都是n位的二进制整数，请设计一个有效的算法，可以进行两个n位大整数的乘法运算。
- 分析：
  - 1.小学的方法：$O(n^2)$ 效率太低

			1	0	1	1
		×	1	0	0	1
			1	0	1	1
		0	0	0	0	
	0	0	0	0		
1	0	1	1			
1	1	0	0	0	1	1

2.4 大整数的乘法

- 2.可以用分治法的原理设计一个更有效的算法。

将n位的二进制整数分为2段：

	$n/2$位	$n/2$位			$n/2$位	$n/2$位
$X=$	A	B		$Y=$	C	D

则： $X=A2^{n/2}+B$ （乘$2^{n/2}$，相当于左移$n/2$位）

$Y=C2^{n/2}+D$

于是： $XY=(A2^{n/2}+B)(C2^{n/2}+D)$

$$=AC2^n+(AD+BC)2^{n/2}+BD \quad (1)$$

2.4 大整数的乘法

- 效率：
  - 4次$n/2$位整数乘法(AC, AD, BC, BD)；
    - 3次不超过n位整数加法；
    - 2次移位(分别对应乘以2^n和$2^{n/2}$)
  - 所有加法和移位共用O(n)步计算。

时间复杂性分析

$$T(n) = \begin{cases} O(1) & n = 1 \\ 4T(n/2) + O(n) & n > 1 \end{cases}$$

$T(n)=O(n^2)$ ✘没有改进

$XY = (A2^{n/2}+B)\ (C2^{n/2}+D)$

$= AC2^n + (AD+BC)\ 2^{n/2} + BD$

(AD – AC – BD + BC + AC +BD)

2.4 大整数的乘法

改进：把(1)式稍作修改：

$$XY = AC2^n + ((A-B)(D-C)+AC+BD)\ 2^{n/2} + BD \quad (2)$$

效率：

- 3次n/2位整数乘法(AC, BD, (A −B)(D −C))；
- 6次不超过n位整数加、减法和2次移位；

时间复杂性分析

$$T(n) = \begin{cases} O(1) & n = 1 \\ 3T(n/2) + O(n) & n > 1 \end{cases}$$

$T(n) = O(n^{\log 3}) = O(n^{1.59})$ ✔ 较大的改进

2.4 大整数的乘法

```
Char *Mult(char X[ ],char  Y[ ],int n)
{  //两个n位整数相乘
   S=sign(X)*sign(Y);            //取乘积的符号
   X=abs(X);    Y=abs(Y);
   if(n==1)
        return (S*X*Y);
   else
   {   A=X的左边n/2位;           B=X的右边n/2位;
       C=Y的左边n/2位;           D=Y的右边n/2位;
       m1=Mult(A, C, n/2);      m2=Mult(A-B, D-C, n/2);
       m3=Mult(B, D, n/2);
       S=S*(m1* 2^n +(m1+m2+m3)*2^(n/2)+m3);
       return S;
   }
}
```

2.4 大整数的乘法

- 小学的方法：$O(n^2)$——效率太低
- 分治法: $O(n^{1.59})$——较大的改进
- 更快的方法？
  - 如果将大整数分成更多段，用更复杂的方式把它们组合起来，将有可能得到更优的算法。
  - 最终的，这个思想导致了快速傅利叶变换(Fast Fourier Transform)的产生。该方法也可以看作是一个复杂的分治算法，对于大整数乘法，它能在O(nlogn)时间内解决。
  - 是否能找到线性时间的算法？目前为止还没有结果。

2.5 Strassen矩阵乘法

- $n \times n$矩阵A和B的乘积矩阵C中的元素C[i,j]定义为：

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

- 若依此定义来计算A和B的乘积矩阵C，则每计算C的一个元素C[i][j]，需要做n次乘法和n-1次加法。因此，算出矩阵C的元素所需的计算时间为$O(n^3)$

2.5 Strassen矩阵乘法----简单分治法求矩阵乘积

- 首先假定n是2的幂。使用与上例类似的技术，将矩阵A，B和C中每一矩阵都分块成4个大小相等的子矩阵。由此可将方程C=AB重写为：

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

- 由此可得：

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

- 复杂度分析（需要8次乘法和4次加法）

$$T(n) = \begin{cases} O(1) & n = 2 \\ 8T(n/2) + O(n^2) & n > 2 \end{cases}$$

- T(n)=O(n3) ✘没有改进☹

2.5 Strassen矩阵乘法

- Strassen提出了一种用7次乘法运算和18次加减法的方法（以增加加减法的次数来减少乘法次数）：

$$M_1=A_{11}(B_{12}-B_{22})$$
$$M_2=(A_{11}+A_{12})B_{22}$$
$$M_3=(A_{21}+A_{22})B_{11}$$
$$M_4=A_{22}(B_{21}-B_{11})$$
$$M_5=(A_{11}+A_{22})(B_{11}+B_{22})$$
$$M_6=(A_{12}-A_{22})(B_{21}+B_{22})$$
$$M_7=(A_{11}-A_{21})(B_{11}+B_{12})$$

$$C_{11}=M_5+M_4-M_2+M_6$$
$$C_{12}=M_1+M_2$$
$$C_{21}=M_3+M_4$$
$$C_{22}=M_5+M_1-M_3-M_7$$

比较

$$C_{11}=A_{11}B_{11}+A_{12}B_{21} \qquad C_{12}=A_{11}B_{12}+A_{12}B_{22}$$
$$C_{21}=A_{21}B_{11}+A_{22}B_{21} \qquad C_{22}=A_{21}B_{12}+A_{22}B_{22}$$

2.5 Strassen矩阵乘法

- 做了这7次乘法后，在做若干次加/减法就可以得到：

$C_{11}=M_5+M_4-M_2+M_6$

$C_{12}=M_1+M_2$

$C_{21}=M_3+M_4$

$C_{22}=M_5+M_1-M_3-M_7$

验证： $C_{22} = M_5+M_1-M_3-M_7$

$$\begin{aligned}
&= (A_{11}+A_{22})(B_{11}+B_{22}) + A_{11}(B_{12}-B_{21}) \\
&\quad -(A_{21}+A_{22})B_{11} - (A_{11}-A_{21})(B_{11}+B_{12}) \\
&= A_{11}B_{11}+A_{11}B_{22}+A_{22}B_{11}+A_{22}B_{22}+A_{11}B_{12} \\
&\quad -A_{11}B_{22}-A_{21}B_{11}-A_{22}B_{11}-A_{11}B_{11} \\
&\quad -A_{11}B_{12}+A_{21}B_{11}+A_{21}B_{12} \\
&= A_{21}B_{12}+A_{22}B_{22}
\end{aligned}$$

- 复杂度分析

$$T(n)=\begin{cases} O(1) & n=2 \\ 7T(n/2)+O(n^2) & n>2 \end{cases}$$

比较

$$C_{11}=A_{11}B_{11}+A_{12}B_{21} \quad C_{12}=A_{11}B_{12}+A_{12}B_{22}$$
$$C_{21}=A_{21}B_{11}+A_{22}B_{21} \quad C_{22}=A_{21}B_{12}+A_{22}B_{22}$$

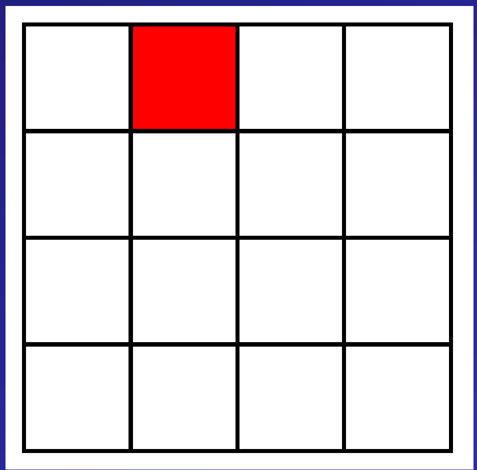
- $T(n)=O(n^{\log 7})=O(n^{2.81})$ ✓较大的改进☺

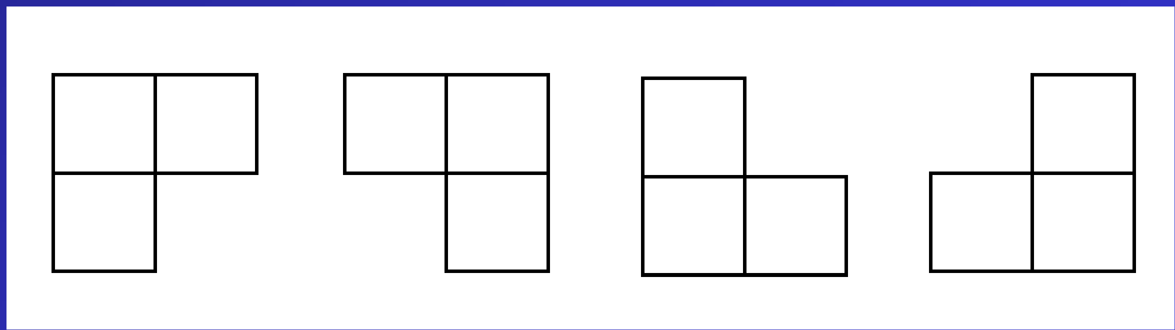
2.5 Strassen矩阵乘法

- Hopcroft和Kerr已经证明(1971)，计算2个2×2矩阵的乘积，7次乘法是必要的。因此，要想进一步改进矩阵乘法的时间复杂性，就不能再基于计算2×2矩阵的7次乘法这样的方法了。或许应当研究3×3或5×5矩阵的更好算法。
- 在Strassen之后又有许多算法改进了矩阵乘法的计算时间复杂性。
- 目前最好的计算时间上界是 $O(n^{2.376})$
- 是否能找到$O(n^2)$的算法？目前为止还没有结果。

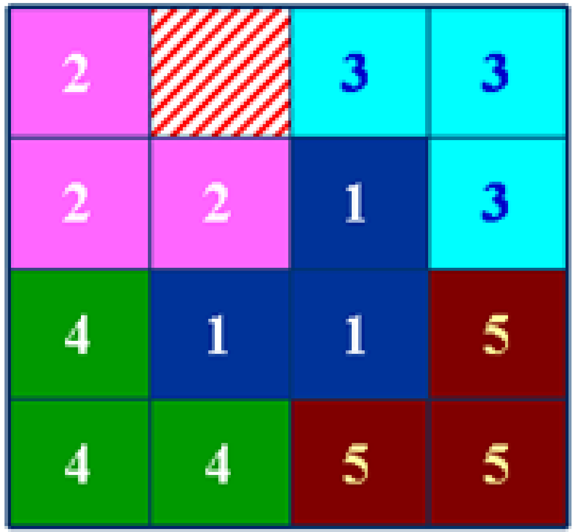
2.6 棋盘覆盖

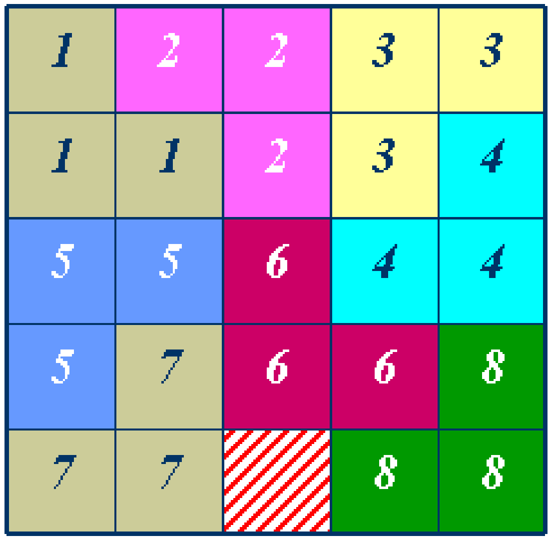
- 在一个$2^k \times 2^k$个方格组成的棋盘中，恰有一个方格与其他方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。
- 易知，覆盖任意一个$2^k \times 2^k$的特殊棋盘，用到的骨牌数恰好为$(4^K-1)/3$。





2.6 棋盘覆盖



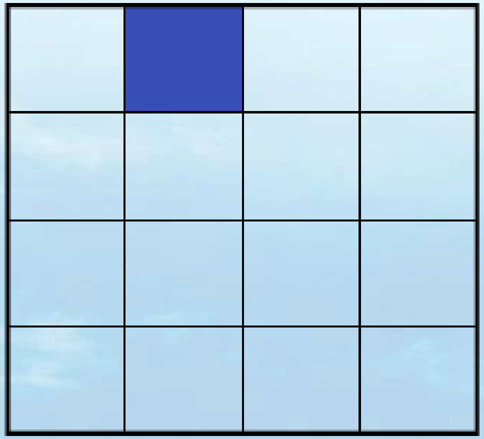


输入：第一行为k（棋盘的尺寸），第二行为 x, y ($1 \le x , y \le 2^k$)，分别表示特殊方格所在行与列。

输出：共2^k行，每行2^k个数，分别表示覆盖该格的L型的编号（特殊格用0表示）。

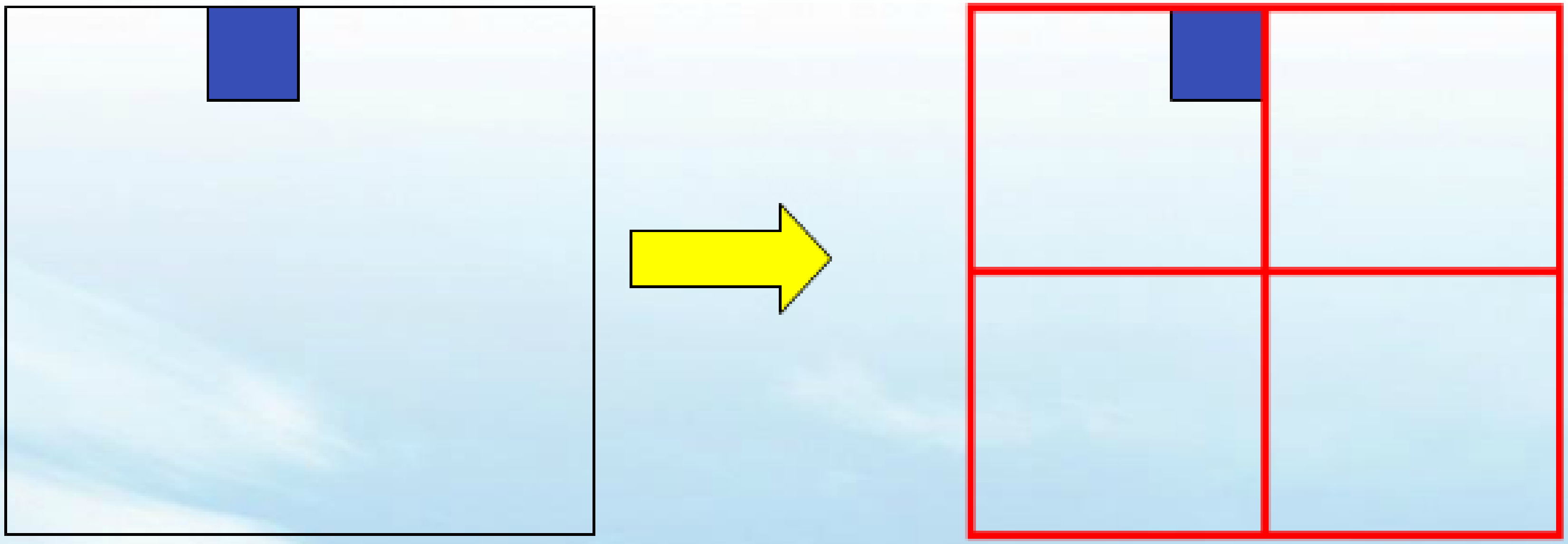
样例：

输入：	输出：
2	1 0 2 2
1 2	1 1 3 2
	4 3 3 5
	4 4 5 5



2.6 棋盘覆盖

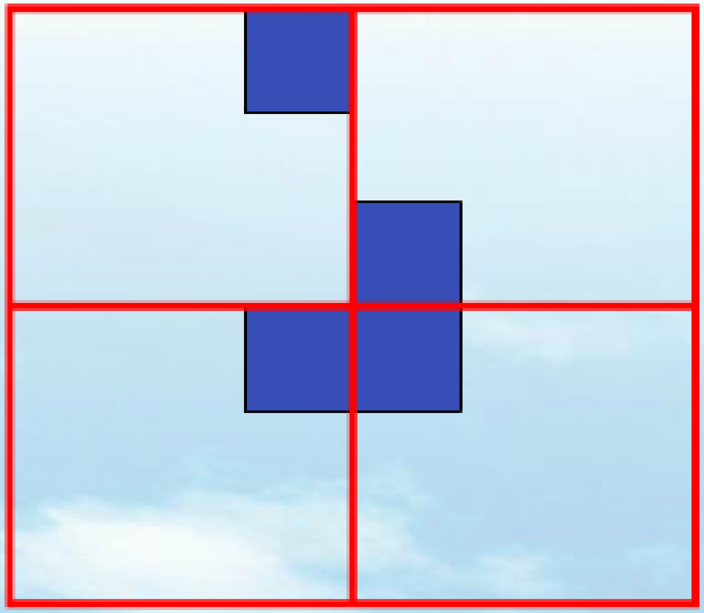
由棋盘尺寸为$2^k \times 2^k$，我们可以想到将其分割成四个尺寸为$2^{k-1} \times 2^{k-1}$的子棋盘。



可是，由于含特殊方格的子棋盘与其它子棋盘不同，问题还是没有解决。

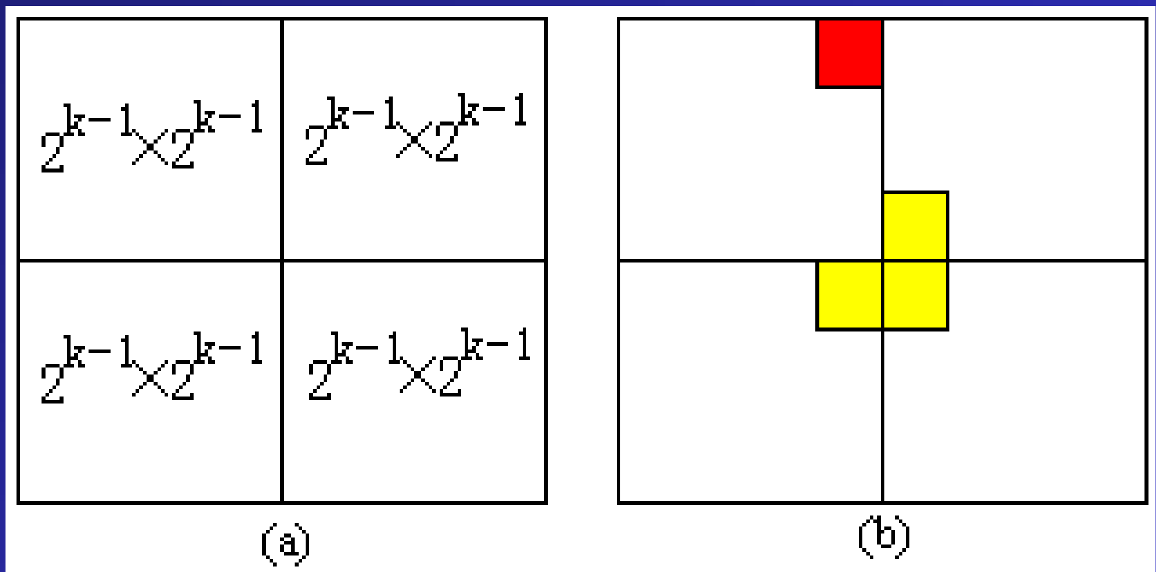
2.6 棋盘覆盖

经过思考，我们发现，只要将L型如图放置在棋盘的中央，就可以使四个子棋盘都变成特殊棋盘。此时问题也变成了四个相同的子问题，运用递归就可以解决这个问题了。如下图：



2.6 棋盘覆盖

➢ 当k>0时，将2k×2k棋盘分割为4个2k-1×2k-1 子棋盘(a)所示。特殊方格必位于4个较小子棋盘之一中，其余3个子棋盘中无特殊方格。为了将这3个无特殊方格的子棋盘转化为特殊棋盘，可以用一个L型骨牌覆盖这3个较小棋盘的会合处，如 (b)所示，从而将原问题转化为4个较小规模的棋盘覆盖问题。递归地使用这种分割，直至棋盘简化为棋盘1×1。



(a)棋盘分割 (b) 构造相同子问题

下面介绍棋盘覆盖问题中数据结构的设计：

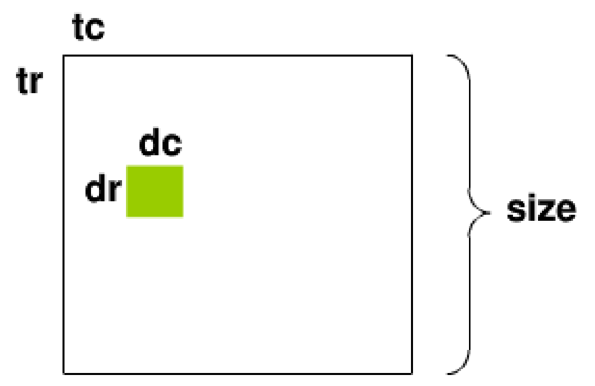
（1）棋盘：用二维数组**board[size][size]**表示一个棋盘，其中，**size=2^k**。为了在递归处理的过程中使用同一个棋盘，将数组**board**设为全局变量；

（2）子棋盘：在棋盘数组**board[size][size]**中，由子棋盘左上角的下标**tr**、**tc**和棋盘边长**s**表示；

（3）特殊方格：用**board[dr][dc]**表示，**dr**和**dc**是该特殊方格在棋盘数组**board**中的下标;

（4）**L**型骨牌：一个$2^k \times 2^k$的棋盘中有一个特殊方格，所以，用到**L**型骨牌的个数为$(4^k-1)/3$，将所有**L**型骨牌从**1**开始连续编号，用一个全局变量**t**表示

2.6 棋盘覆盖



棋盘覆盖问题中的数据结构

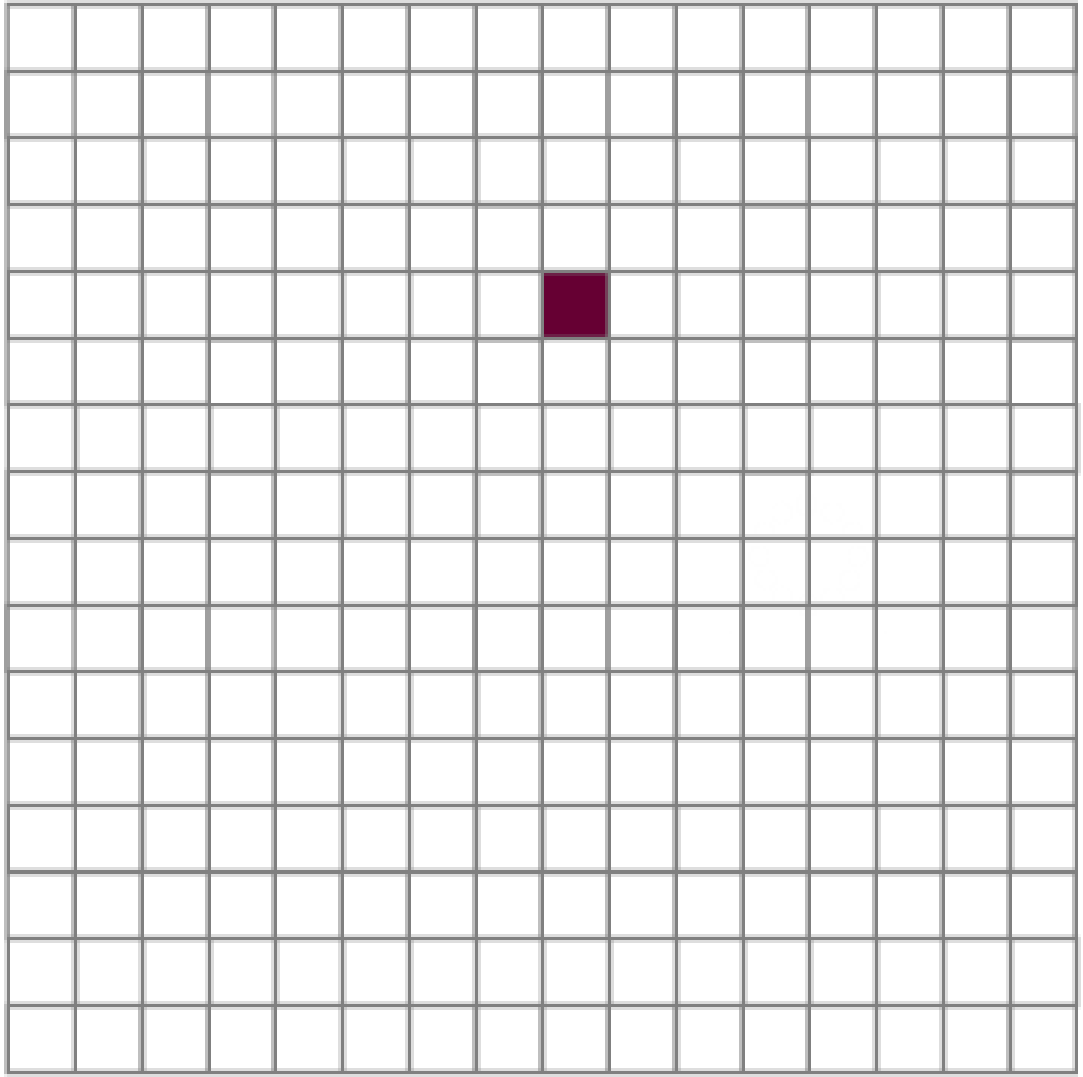
➢ 说明：

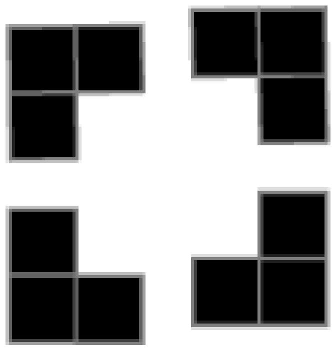
- 整形二维数组Board表示棋盘，Borad[0][0]使棋盘的左上角方格。
- tile是一个全局整形变量，用来表示L形骨牌的编号，初始值为0。
- tr：棋盘左上角方格的行号；
- tc：棋盘左上角方格的列号；
- dr：特殊方各所在的行号；
- dc：特殊方各所在的列号；
- size：size=2^k，棋盘规格为$2^k \times 2^k$。

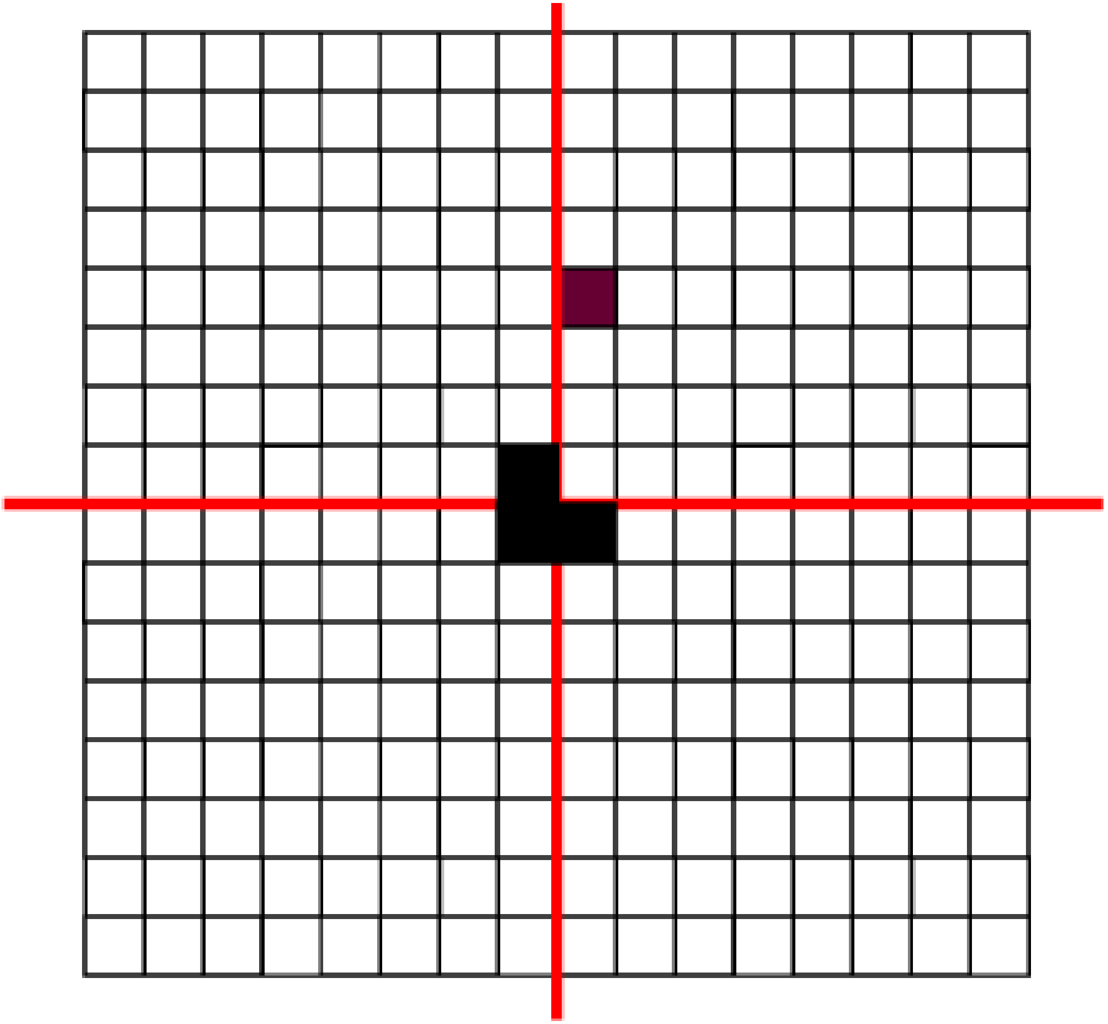
算法——棋盘覆盖

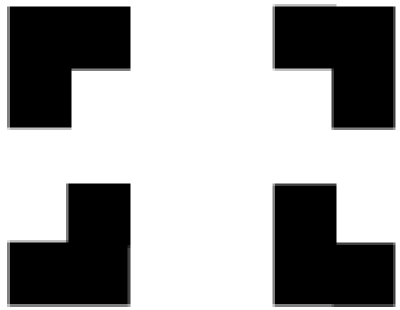
```
void  ChessBoard(int tr, int tc, int dr, int dc, int size)
// tr和tc是棋盘左上角的下标，dr和dc是特殊方格的下标，
// size是棋盘的大小，t已初始化为0
{
     if (size = = 1) return;  //棋盘只有一个方格且是特殊方格
     t++;  // L型骨牌号
     s = size/2;  // 划分棋盘
     // 覆盖左上角子棋盘
     if (dr < tr + s && dc < tc + s)   // 特殊方格在左上角子棋盘中
       ChessBoard(tr, tc, dr, dc, s);          //递归处理子棋盘
     else{      // 用 t 号L型骨牌覆盖右下角，再递归处理子棋盘
      board[tr + s - 1][tc + s - 1] = t;
      ChessBoard(tr, tc, tr+s-1, tc+s-1, s);
     }
```

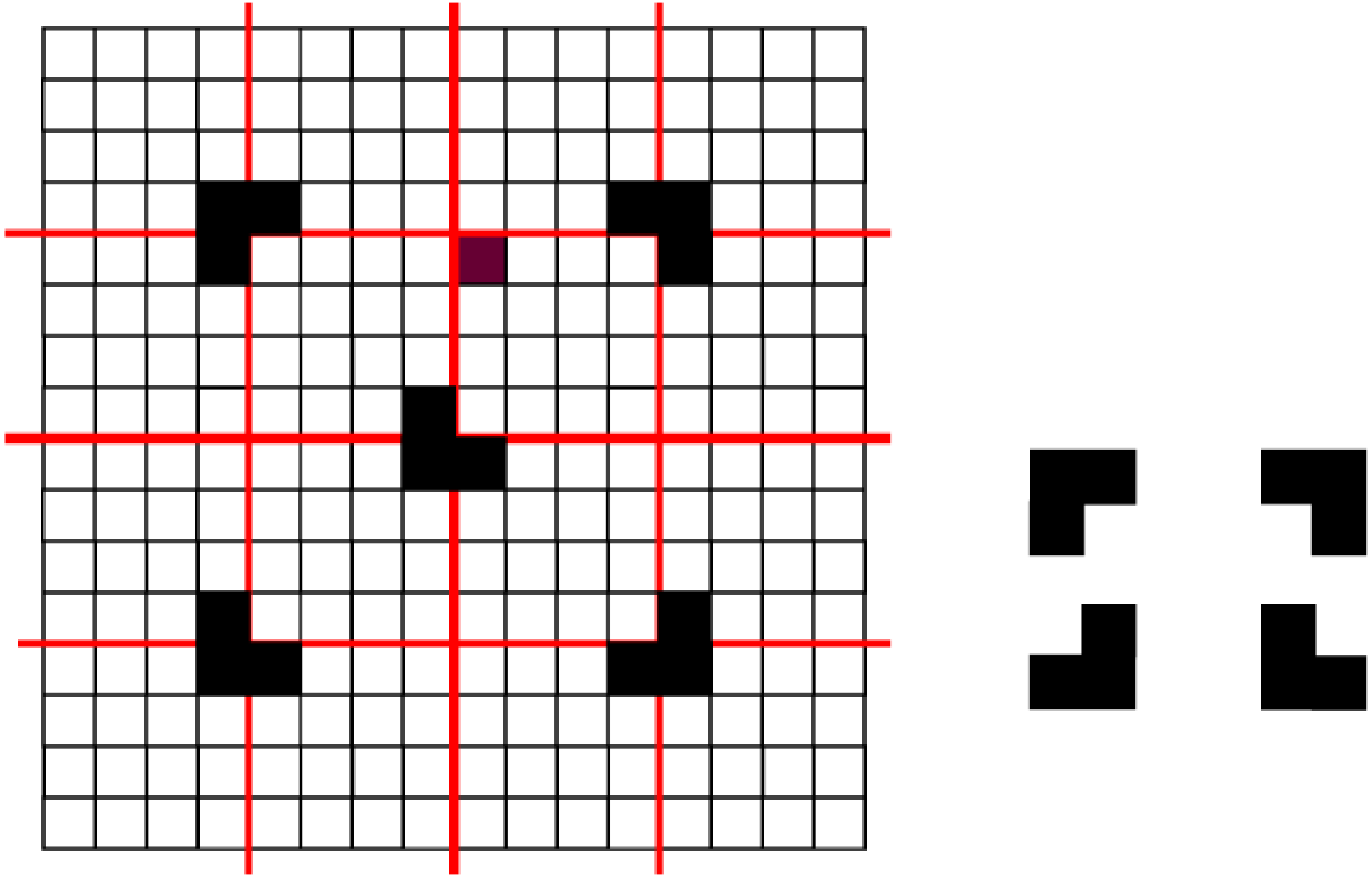
```
    // 覆盖右上角子棋盘
    if (dr < tr + s && dc >= tc + s)    // 特殊方格在右上角子棋盘中
      ChessBoard(tr, tc+s, dr, dc, s);          //递归处理子棋盘
    else {        // 用 t 号L型骨牌覆盖左下角，再递归处理子棋盘
      board[tr + s - 1][tc + s] = t;
      ChessBoard(tr, tc+s, tr+s-1, tc+s, s); }
    // 覆盖左下角子棋盘
    if (dr >= tr + s && dc < tc + s)   // 特殊方格在左下角子棋盘中
      ChessBoard(tr+s, tc, dr, dc, s);          //递归处理子棋盘
    else {        // 用 t 号L型骨牌覆盖右上角，再递归处理子棋盘
      board[tr + s][tc + s - 1] = t;
      ChessBoard(tr+s, tc, tr+s, tc+s-1, s); }
    // 覆盖右下角子棋盘
    if (dr >= tr + s && dc >= tc + s)  // 特殊方格在右下角子棋盘中
      ChessBoard(tr+s, tc+s, dr, dc, s);        //递归处理子棋盘
    else {        // 用 t 号L型骨牌覆盖左上角，再递归处理子棋盘
      board[tr + s][tc + s] = t;
      ChessBoard(tr+s, tc+s, tr+s, tc+s, s); }
}
```

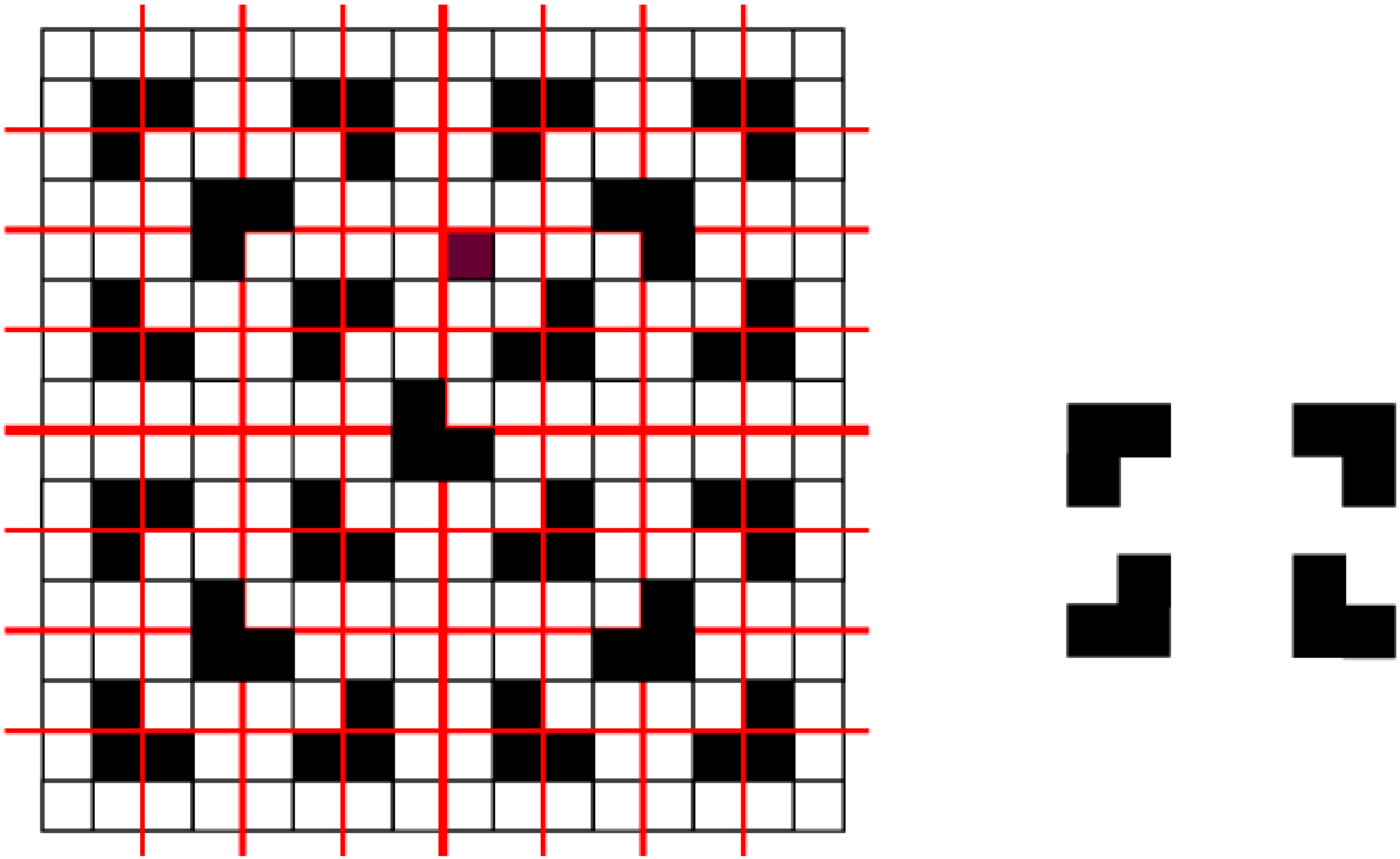


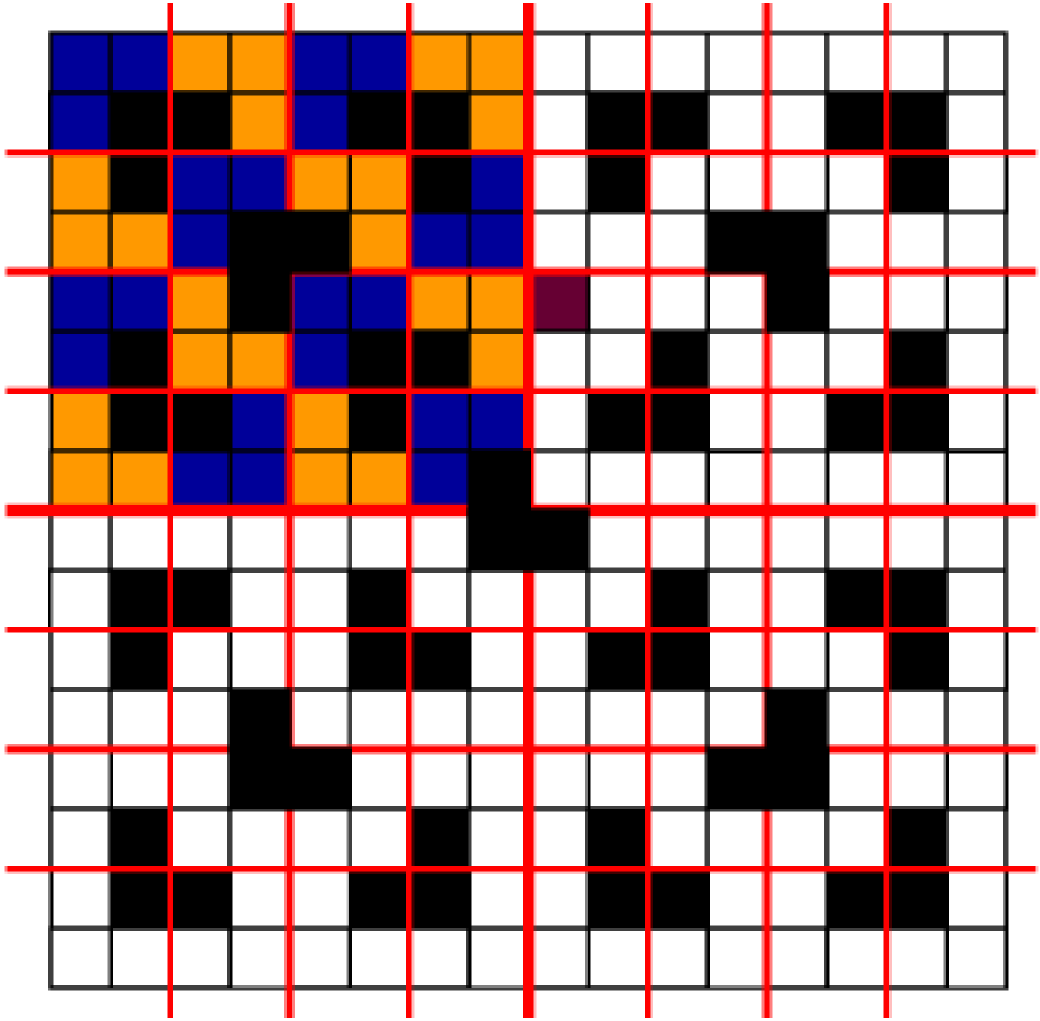


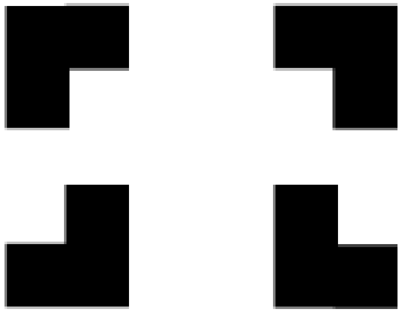












问题描述：在一个$2^k \times 2^k$ 个方格组成的棋盘中，恰有一个方格与其它方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。

算法思想：

用分治策略，可以设计出解棋盘覆盖问题的简洁算法。

(1)当k>0时，将2的k次幂乘以2的k次幂棋盘分割为4个2的k-1次幂乘以2的k-1次幂子棋盘。

(2)特殊方格必位于4个较小棋盘之一中，其余3个子棋盘中无特殊方格。

(3)为了将这3个无特殊方格的子棋盘转化为特殊棋盘，可以用一个L型骨牌覆盖这3个较小棋盘的会合处，这3个子棋盘上被L型骨牌覆盖的方格就成为该棋盘上的特殊方格，从而将原问题转化为4个较小规模的棋盘覆盖问题。递归地使用这种分割，直至棋盘简化为1*1棋盘。