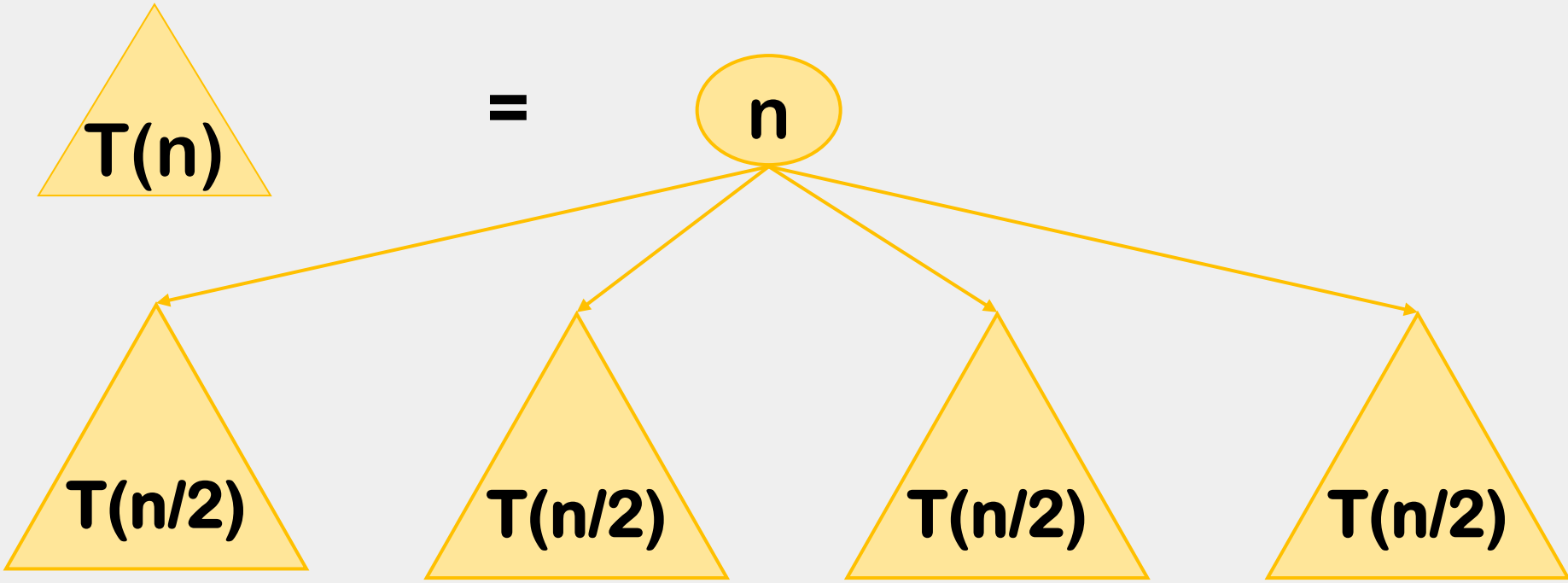
第10章　递归与分治策略

第一节 算法总体思想

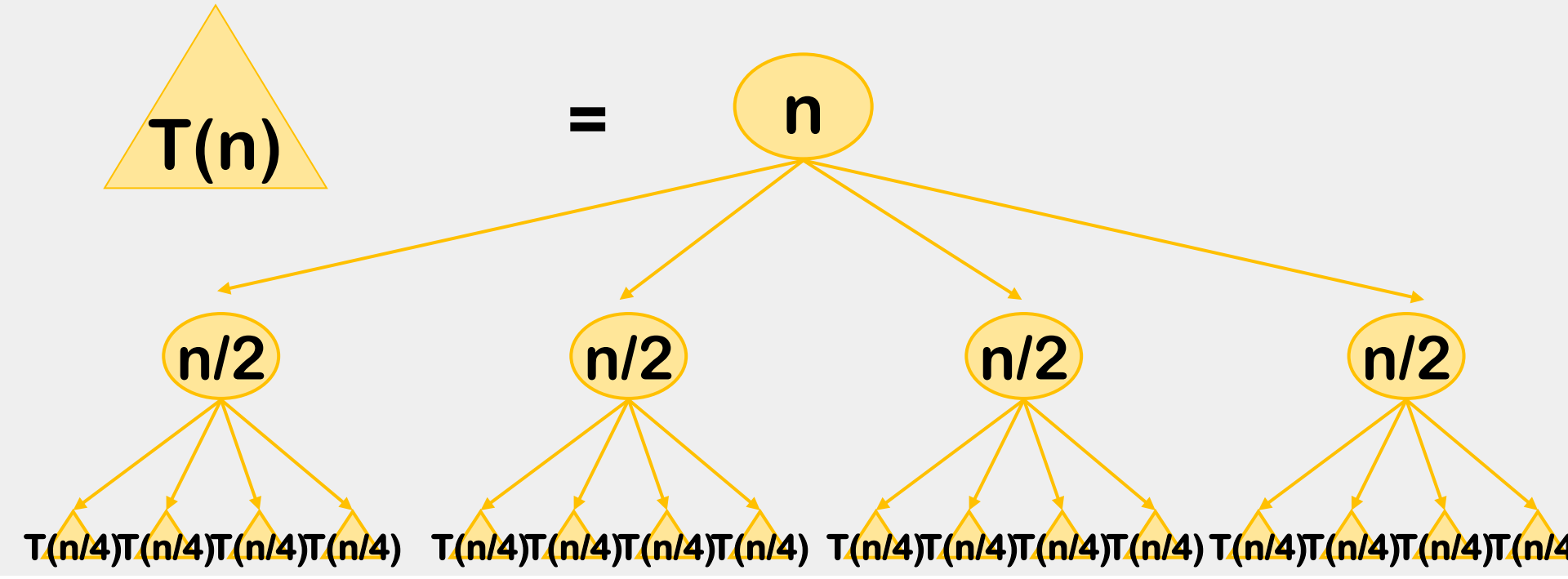
算法总体思想

- 将要求解的较大规模的问题分割成k个更小规模的子问题。



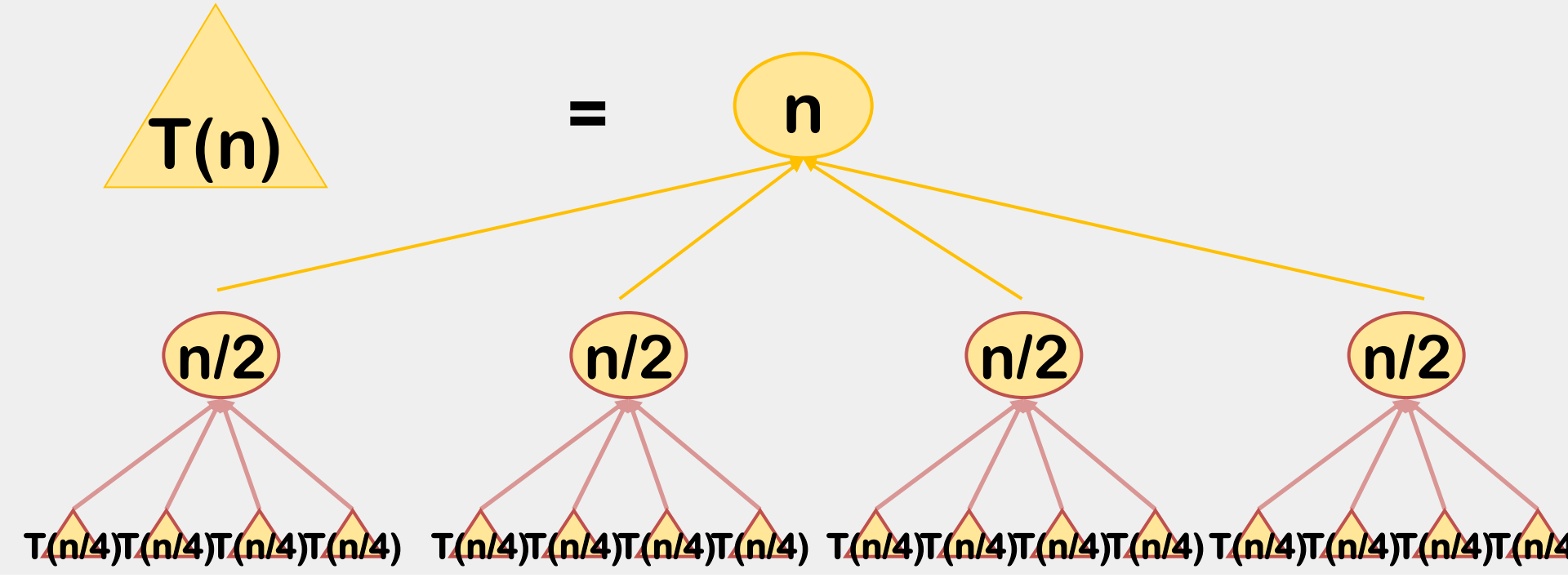
算法总体思想

对这k个子问题分别求解。如果子问题的规模仍然不够小，则再划分为k个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。



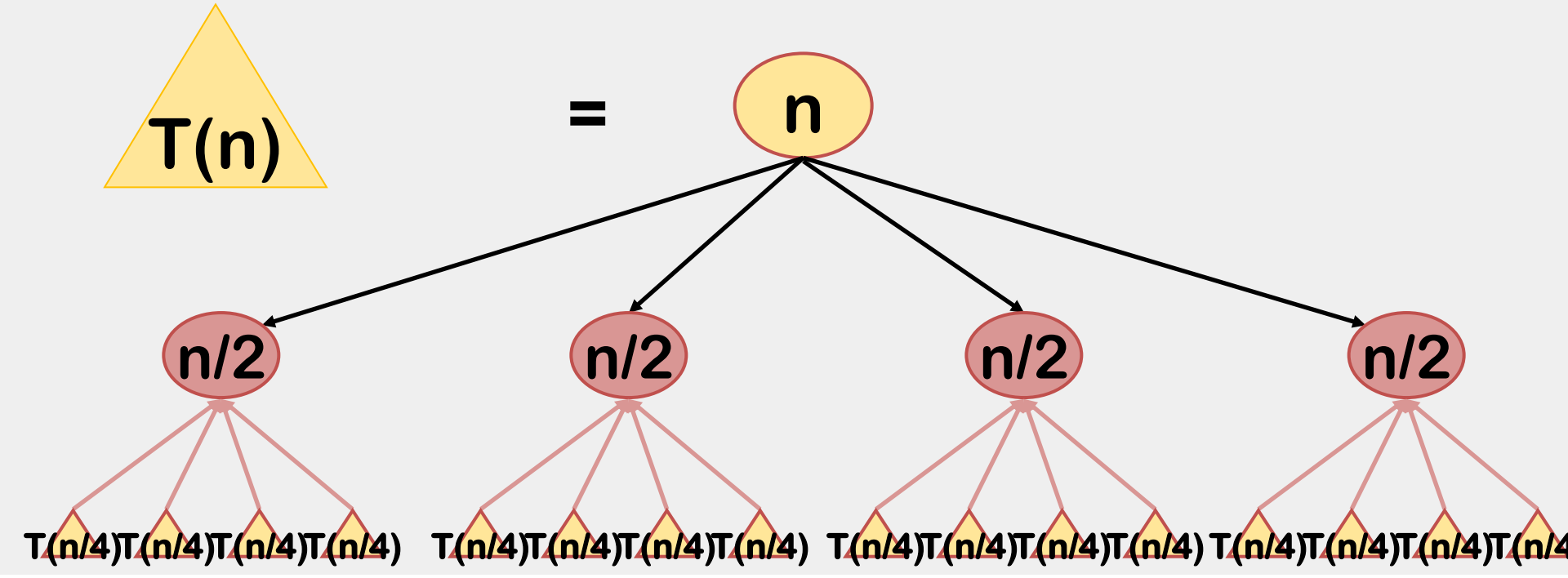
算法总体思想

将求出的小规模的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来问题的解。



算法总体思想

分治法的设计思想是，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。



第二节 递归

（一）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。
- 由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。
- 分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

（一） 递归的概念

- 例1 阶乘函数

阶乘函数可递归地定义为：

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

边界条件

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。

（一） 递归的概念

- 例2 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

递归方程

第n个Fibonacci数可递归地计算如下：

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

（一） 递归的概念

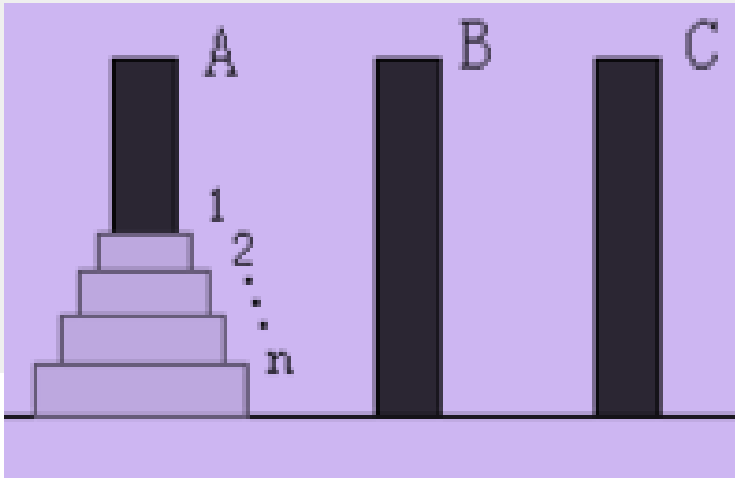
例3 Hanoi塔问题

设a, b, c是3个塔座。开始时，在塔座a上有一叠共n个圆盘，这些圆盘自下而上，由大到小地叠在一起。各圆盘从小到大编号为1, 2, …, n, 现要求将塔座a上的这一叠圆盘移到塔座b上，并仍按同样顺序叠置。在移动圆盘时应遵守以下移动规则：

规则1：每次只能移动1个圆盘；

规则2：任何时刻都不允许将较大的圆盘压在较小的圆盘之上；

规则3：在满足移动规则1和2的前提下，可将圆盘移至a, b, c中任一塔座上。



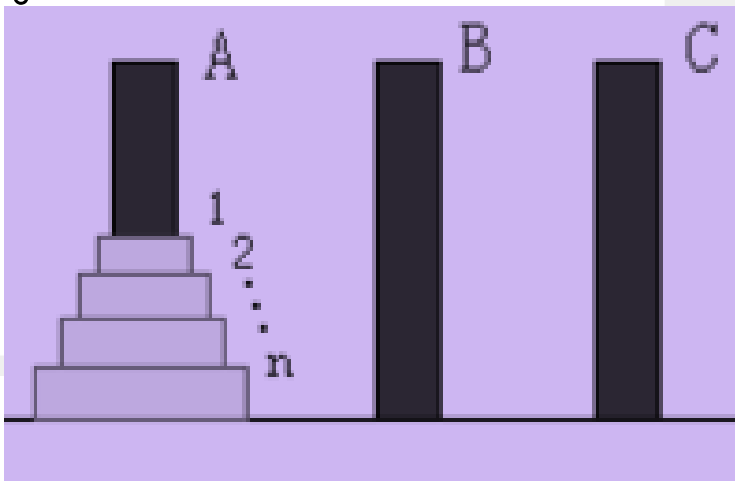
（一） 递归的概念

- Hanoi塔问题　在问题规模较大时，较难找到一般的方法，因此我们尝试用递归技术来解决这个问题。

当n=1时，问题比较简单。此时，只要将编号为1的圆盘从塔座a直接移至塔座b上即可。

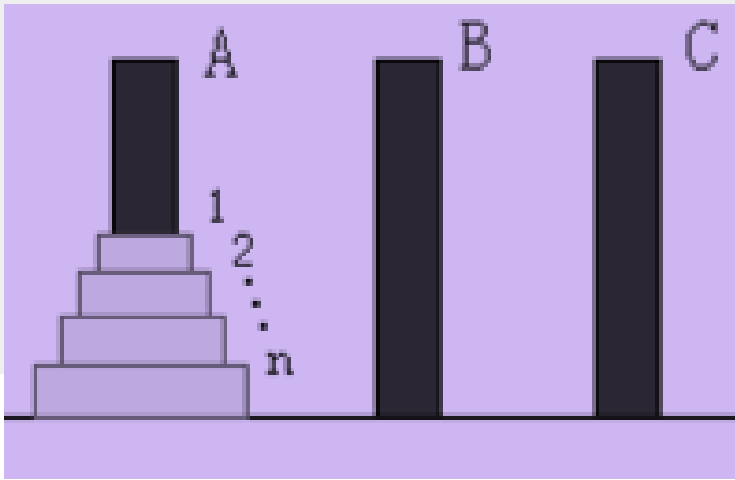
当n＞1时，需要利用塔座c作为辅助塔座。此时若能设法将n−1个较小的圆盘依照移动规则从塔座a移至塔座c，然后，将剩下的最大圆盘从塔座a移至塔座b，最后，再设法将n−1个较小的圆盘依照移动规则从塔座c移至塔座b。

由此可见，n个圆盘的移动问题可分为2次n−1个圆盘的移动问题，这又可以递归地用上述方法来做。由此可以设计出解Hanoi塔问题的递归算法如下。



（一） 递归的概念

```
void hanoi(int n, int a, int b, int c)
  {
     if (n > 0)
     {
        hanoi(n-1, a, c, b);
        move(a,b);
        hanoi(n-1, c, b, a);
     }
  }
```



（一） 递归的概念

```c
int fact(int n) {
    if (n < 0)
        return 0;
    else if(n == 0 || n == 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

```c
int height(BTree *p)
{
    int hi = 0,lh = 0,rh = 0;
    if (p == NULL)
        hi = 0;
    else
    {
        if (p->lchild ==NULL)
            lh = 0;
        else
            lh = height(p->lchild);//递归求解左子树的高度
        if (p->rchild ==NULL)
            rh = 0;
        else
            rh = height(p->rchild);//递归求解右子树的高度
        hi = lh>rh ? (lh + 1) : (rh + 1);
    }
    return hi;
}
```

（一） 递归的概念

```c
int fact(int n) {
    if (n < 0)
        return 0;
    else if(n == 0 || n == 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

```c
int fact (int n)
 {
    if( n < 0)
       return 0;
    if( n == 0)
       return 1;
    int a=1;
    for(int i=2; i++; i<=n)
       a=a*i;
    return (a);
 }
```

```c
int height(BTree *p)
{
    int hi = 0,lh = 0,rh = 0;
    if (p == NULL)
        hi = 0;
    else
    {
        if (p->lchild ==NULL)
            lh = 0;
        else
            lh = height(p->lchild);//递归求解左子树的高度
        if (p->rchild ==NULL)
            rh = 0;
        else
            rh = height(p->rchild);//递归求解右子树的高度
        hi = lh>rh ? (lh + 1) : (rh + 1);
    }
    return hi;
}
```

（一） 尾递归的概念

```c
int fact(int n) {
    if (n < 0)
        return 0;
    else if(n == 0 || n == 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

```c
int facttail(int n, int res)
{
    if (n < 0)
        return 0;
    else if(n == 0)
        return 1;
    else if(n == 1)
        return res;
    else
        return facttail(n - 1, n *res);
}
```

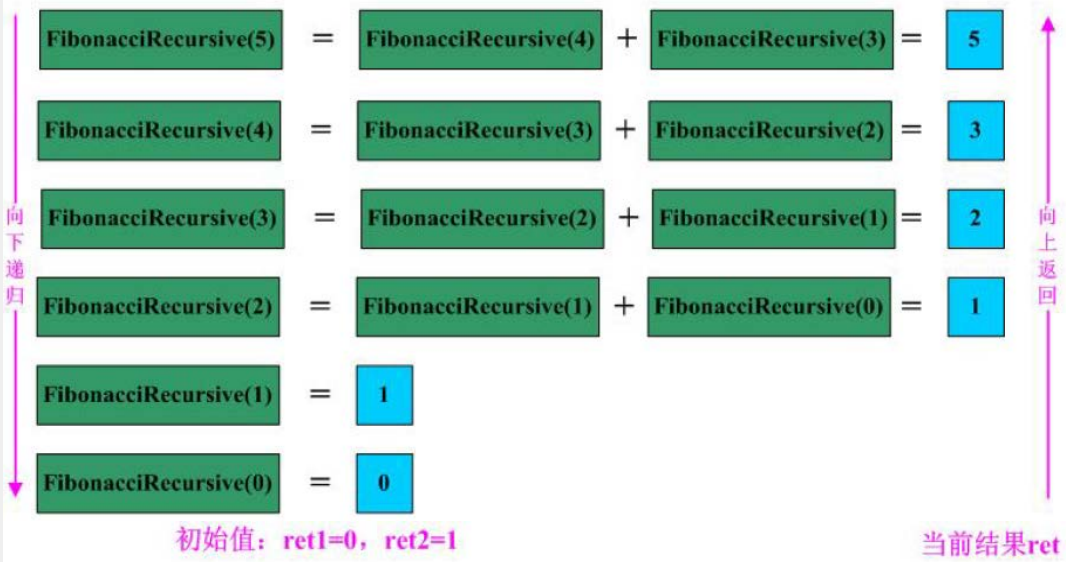
如果一个函数中所有递归形式的调用都出现在函数的末尾，我们称这个递归函数是**尾递归**的。当编译器检测到一个函数调用是尾递归的时候，它就覆盖当前的活动记录而不是在栈中去创建一个新的。不用尾递归，函数的堆栈耗用难以估量，需要保存很多中间函数的堆栈。其**关键就是通过参数传递结果，达到不压栈的目的**。

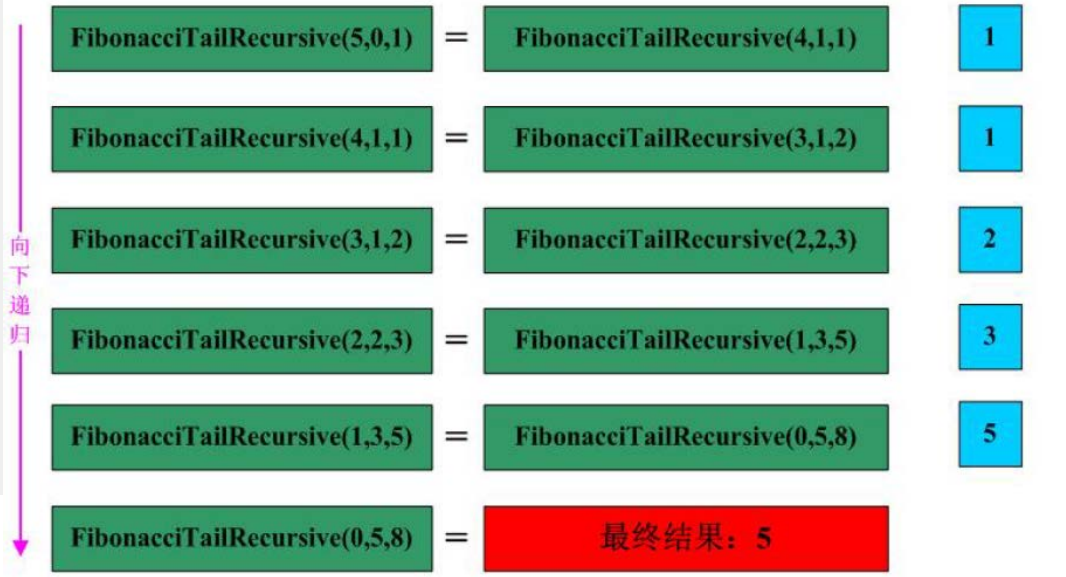
（一） 尾递归的概念

0、1、1、2、3、5、8、13、21、34

```
int Fibo (int n)
 {
     if( n < 2)
         return n;
     return (Fibo (n-1)+Fibo (n-2));
 }
```

```
int FiboTail (int n,int ret1,int ret2)
{
   if(n==0)
      return ret1;
    return FiboTail (n-1,ret2,ret1+ret2);
}
```





（二）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。

（二）递归小结

- 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。

1. 采用一个用户定义的栈来模拟系统的递归调用工作栈。该方法通用性强，但本质上还是递归，只不过人工做了本来由编译器做的事情，优化效果不明显。
2. 用递推来实现递归函数。
3. 通过变换能将一些递归转化为尾递归，从而迭代求出结果。

- 后两种方法在时空复杂度上均有较大改善，但其适用范围有限。

第三节　分治法

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；

因为问题的计算复杂性一般是随着问题规模的增加而增加，因此大部分问题满足这个特征。

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；

这条特征是应用分治法的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用

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；

能否利用分治法完全取决于问题是否具有这条特征，如果具备了前两条特征，而不具备第三条特征，则可以考虑**贪心算法**或**动态规划**。

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。

分治法的基本步骤

```
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++)
      yi=divide-and-conquer(Pi);  //递归的解各子问题
    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。

分治法的基本步骤

```
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++)
      yi=divide-and-conquer(Pi);  //递归的解各子问题
    return merge(y1,...,yk);  //将各子问题的解合并为原问题的解
  }
```

采用分治法将一个规模为n的问题分成a个规模为n/b的子问题进行求解。

复杂度分析

$$T(n)=\begin{cases} O(1) & n=1 \\ aT(n/b)+f(n) & n>1 \end{cases}$$

代入递推得

$$T(n)=n^{\log_b^a}+\sum_{j=0}^{\log_b^n -1} a^j f(n/b^j)$$

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。
- 分析：
  - **该问题的规模缩小到一定的程度就可以容易地解决；**

分析：如果n=1即只有一个元素，则只要比较这个元素和x就可以确定x是否在表中。因此这个问题满足分治法的第一个适用条件

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。
- 分析：
  - **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；**

分析：比较x和a的中间元素a[mid]，若x=a[mid]，则x在L中的位置就是mid；如果x<a[mid]，由于a是递增排序的，因此假如x在a中的话，x必然排在a[mid]的前面，所以我们只要在a[mid]的前面查找x即可；如果x>a[i]，同理我们只要在a[mid]的后面查找x即可。无论是在前面还是后面查找x，其方法都和在a中查找x一样，只不过是查找的规模缩小了。这就说明了此问题满足分治法的第二个和第三个适用条件。

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。
- 分析：
  - **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。**

分析：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，因此满足分治法的第四个适用条件。

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。
- 据此容易设计出二分搜索算法：l:left,r:right;

```
template<class Type>
int BinarySearch(Type a[], const
Type& x, int l, int r)
{
      while (r >= l){
          int m = (l+r)/2;
          if (x == a[m]) return m;
          if (x < a[m]) r = m-1; else
l = m+1;
          }
     return -1;
}
```

算法复杂度分析：

每执行一次算法的while循环，待搜索数组的大小减少一半。因此，在最坏情况下，while循环被执行了O(logn) 次。循环体内运算需要O(1) 时间，因此整个算法在最坏情况下的计算时间复杂性为O(logn) 。

大整数的乘法

- 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
  - 小学的方法：0(n2)　　×效率太低
  - 分治法：

X =	a	b
Y =	c	d

$X = a\ 2^{n/2} + b$　　$Y = c\ 2^{n/2} + d$

$XY = ac\ 2^{n} + (ad+bc)\ 2^{n/2} + bd$

复杂度分析

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

$T(n)=O(n^2)$　　✘没有改进

大整数的乘法

- 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
  - 小学的方法：$O(n^2)$ ×效率太低
  - 分治法： $XY = ac\ 2^n + (ad+bc)\ 2^{n/2} + bd$

    为了降低时间复杂度，必须减少乘法的次数。

    1. $XY = ac\ 2^n + ((a-c)(b-d)+ac+bd)\ 2^{n/2} + bd$
    2. $XY = ac\ 2^n + ((a+c)(b+d)-ac-bd)\ 2^{n/2} + bd$

复杂度分析

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

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

细节问题：两个XY的复杂度都是$O(n^{\log 3})$，但考虑到a+c, b+d可能得到n+1位的结果，使问题的规模变大，故不选择第2种方案。

大整数的乘法

- 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
  - 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。

合并排序

- **基本思想**：将待排序元素分成大小大致相同的2个子集合，分别对2个子集合进行排序，最终将排好序的子集合合并成为所要求的排好序的集合。

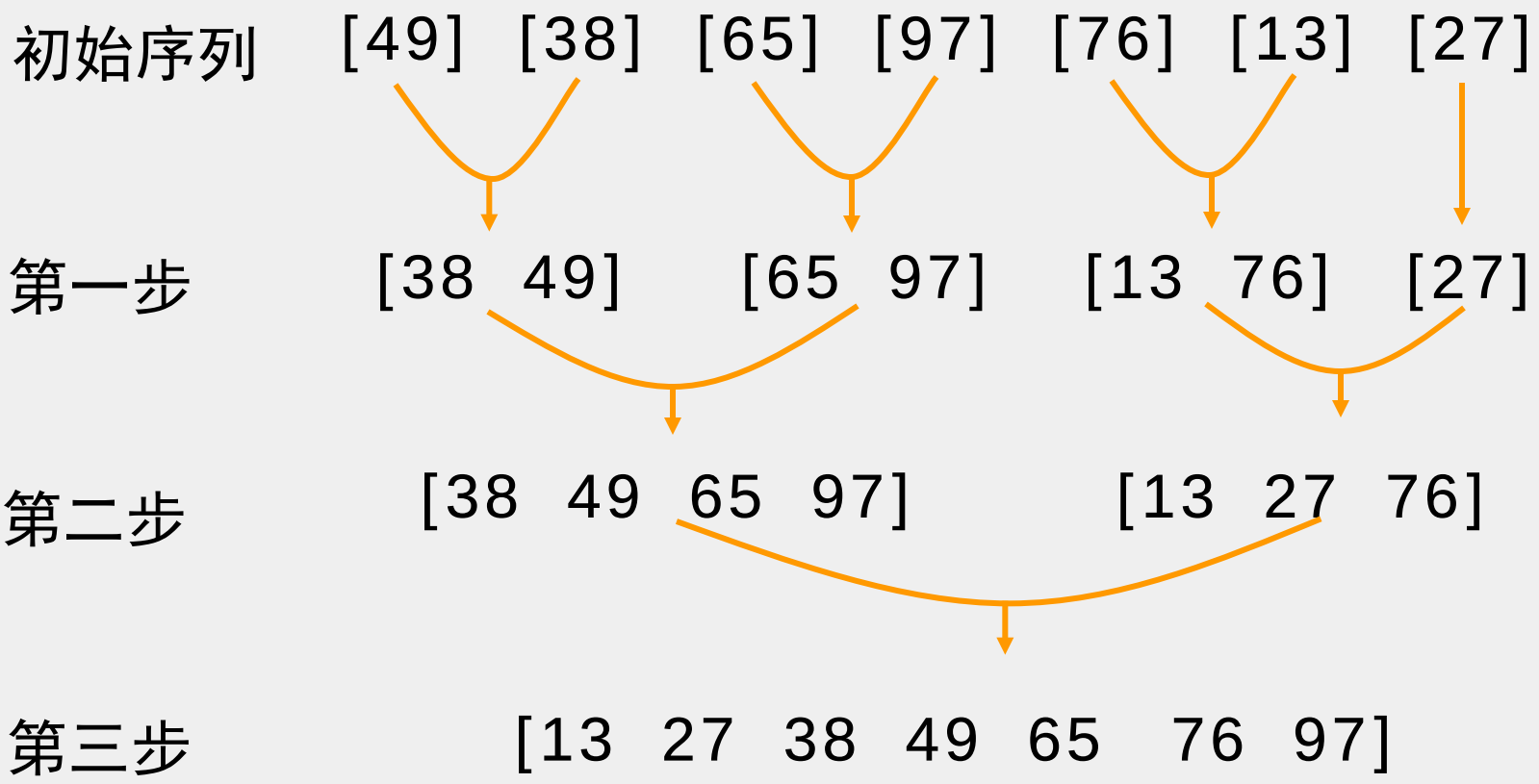
```
void MergeSort(Type a[], int left, int right)
  {
     if (left<right) {//至少有2个元素
     int i=(left+right)/2;  //取中点
     mergeSort(a, left, i);
     mergeSort(a, i+1, right);
     merge(a, b, left, i, right);  //合并到数组b
     copy(a, b, left, right);    //复制回数组a
     }
  }
```

复杂度分析 $T(n)=\begin{cases} O(1) & n\le 1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$

T(n)=O(nlogn) 渐进意义下的最优算法

合并排序

- 算法mergeSort的递归过程可以消去。



合并排序

- 最坏时间复杂度：O(nlogn)
- 平均时间复杂度：O(nlogn)
- 辅助空间：O(n)

快速排序

```
int qusort(int s[],int start,int end)    //自定义函数 qusort()
{
    int i,j;    //定义变量为基本整型
    i=start;    //将每组首个元素赋给i
    j = end;    //将每组末尾元素赋给j
    s[0]=s[start];    //设置基准值
    while(i<j)
    {
        while(i<j&&s[0]<s[j])
        j--;    //位置左移
        if(i<j)
        {
            s[i]=s[j];    //将s[j]放到s[i]的位置上
            i++;    //位置右移
        }
        while(i<j&&s[i]<=s[0])
            i++;    //位置左移
        if(i<j)
        {
            s[j]=s[i];    //将大于基准值的s[j]放到s[i]位置
            j--;    //位置左移
        }
    }
    s[i]=s[0];    //将基准值放入指定位置
    if (start<i)
        qusort(s,start,j-1);    //对分割出的部分递归调用qusort()函数
    if (i<end)
        qusort(s,j+1,end);
    return 0;
}
```

- **最坏时间复杂度：O(n²)**

  每次极不对称划分（1，n-1）

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

- **最好时间复杂度：O(nlogn)**

  每次对称划分

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

循环赛日程表

- 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；
  - (3) 循环赛一共进行n-1天。

按分治策略，将所有的选手分为两半，n个选手的比赛日程表就可以通过为n/2个选手设计的比赛日程表来决定。递归地对选手进行分割，直到只剩下2个选手时。这时只要让这2个选手进行比赛就可以了。

循环赛日程表

- 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；
  - (3) 循环赛一共进行n-1天。

按分治策略，将所有的选手分为两半，n个选手的比赛日程表就可以通过为n/2个选手设计的比赛日程表来决定。递归地对选手进行分割，直到只剩下2个选手时。这时只要让这2个选手进行比赛就可以了。

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

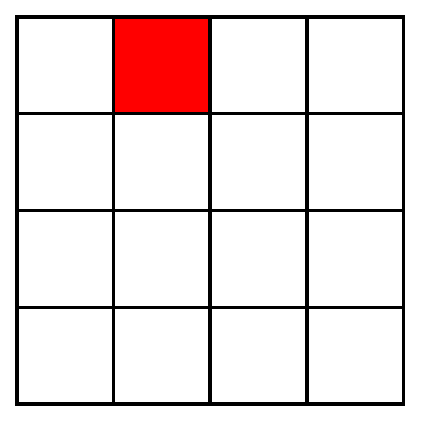
1 2 3 4 2 1 4 3 3 4 1 2 4 3 2 1	5 6 7 8 6 5 8 7 7 8 5 6 8 7 6 5
5 6 7 8 6 5 8 7 7 8 5 6 8 7 6 5	1 2 3 4 2 1 4 3 3 4 1 2 4 3 2 1

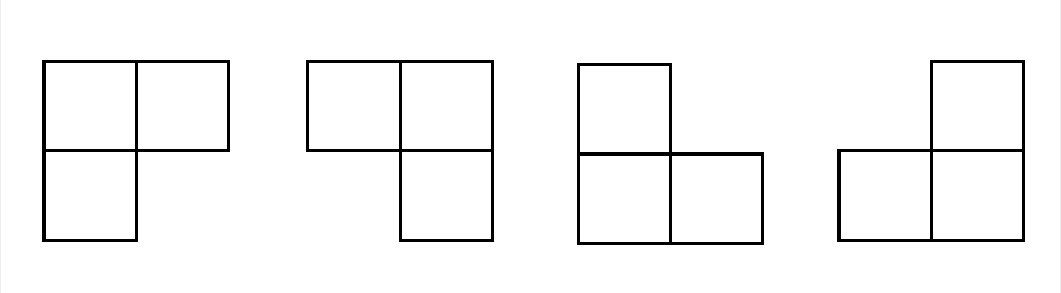
(c) $2^k(k=3)$个选手比赛

- 求解过程是自底向上的迭代过程，其中图(c)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
- 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。

棋盘覆盖

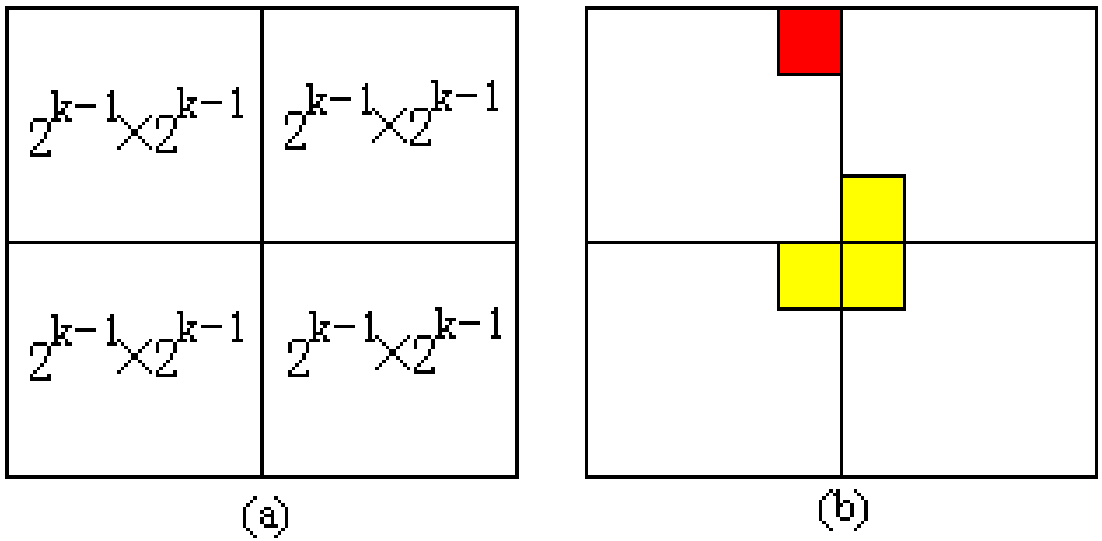
- 在一个2k×2k 个方格组成的棋盘中，恰有一个方格与其它方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。





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。



棋盘覆盖

复杂度分析

$$T(k)=\begin{cases} O(1) & k=0 \\ 4T(k-1)+O(1) & k>0 \end{cases}$$

T(n)=O(4^k) 渐进意义下的最优算法