分治法的设计思想

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。

由分治法产生的子问题往往是原问题的较小模式，可以采用递归技术解决这些子问题。

要点分析

- **分解:**将规模为n的问题分解为k个规模较小的子问题.
- **解决:**对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，停止递归,直接求解.
- **合并:**将子问题的解组合成原问题.

适用问题

- ✓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。

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**。
- 由分治法产生的子问题往往是原问题的较小模式，为使用递归技术提供了方便。
- 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，自然导致递归过程的产生。

相关概念

- ✓当子问题足够大,需要递归求解时,称为递归情况;
- ✓当子问题足够小时,不再需要递归时,递归进入“触底”,进入基本情况;
- ✓用函数自身给出定义的函数称为递归函数;
- ✓递归式是一个等式或不等式,它通过更小的输入上的函数值来描述一个函数.

例2-1　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（**基础步**）

递归方程（**归纳步**）

<u>说明</u>：边界条件与递归方程是递归函数的两个要素，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。

例如: 计算3!

求解过程: 3!=3*2! → 2!=2*1! → 1!=1*0! → 0!

6 ← 2 ← 1 ← 1

可递归地计算如下:

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

也可以采用非递归方式定义: $n!=1\cdot 2\cdot 3\cdot \quad \cdot(n-1)\cdot n$

递归函数的内部执行过程

- （1）运行开始时，为递归调用建立一个工作栈，其结构包括实参、局部变量和返回地址；
- （2）每次执行递归调用之前，把递归函数的实参和局部变量的当前值以及调用后的返回地址压栈；
- （3）每次递归调用结束后，将栈顶元素出栈，使相应的实参和局部变量恢复为调用前的值，然后转向返回地址指定的位置继续执行

例2-2 Fibonacci数列

问题:有雌雄一对小兔,假设过两个月便可以繁殖一对雌雄小兔,以后每月生一对,问过n各月后有多少对兔子?

分析:此问题可以用下列数列表示:

1，1，2，3，5，8，13，21，34，55，……，

从第三项起,任意一项为前两项和.

- 它可以递归定义为：

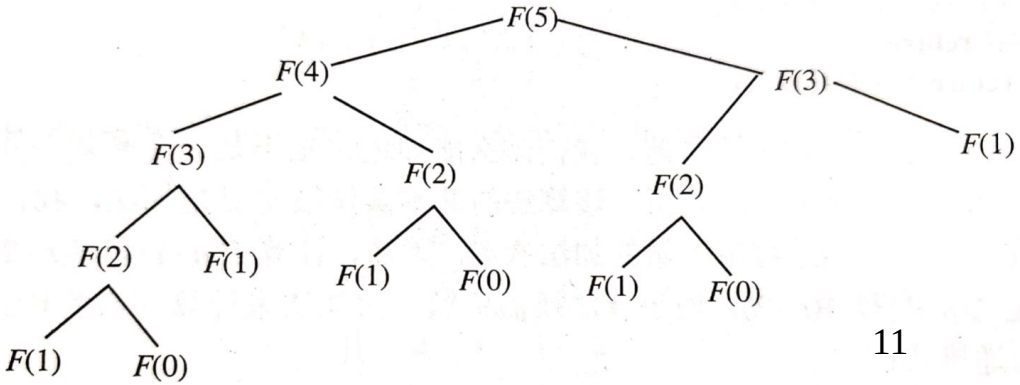
$$F(n)=\begin{cases}1 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

边界条件

递归方程

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

时间复杂度是$O(2^n)$



采用非递归方法定义:

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

方法二:循环法

```
int f(int n){
    int s1=1,s2=1,s=1;
    for(int i=3;i<=n;i++){
        s=s1+s2;
        s2=s1;
        s1=s;
    }
    return s;
}
```

时间复杂度O(n)

方法三：矩阵连乘法

$$\begin{pmatrix} f(n) \\ f(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f(n-1) \\ f(n-2) \end{pmatrix} = \cdots = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} \begin{pmatrix} f(2) \\ f(1) \end{pmatrix}$$

- 计算f(n)就简化为了计算矩阵的(n-2)次方
- 对于计算矩阵的(n-2)次方的问题采用分治法，对问题进行分解，即计算矩阵(n-2)/2次方的平方，逐步分解下去。
- 由于折半计算矩阵次方，时间复杂度为O(log n)

例2-3 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：

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

该变量是由函数自身定义

分析:

A(n,m)的自变量m的每一个值都定义了一个单变量函数:

- **M=0时**， A(n,0)=n+2
- **M=1时** **A(1,1)=2** 和 A(n,1)=A(A(n-1,1),0)=A(n-1,1)+2，
  故A(n,1)=2*n （上式是一个递推公式，每当n-1便加2）
- **M=2**时，A(1,2)=A(A(0,2),1)=A(1,1)=2
  A(n,2)=A(A(n-1,2),1)=2A(n-1,2)， 故A(n,2)= 2^n 。
- **M=3**时，类似的可以推出 $\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{n}$
- **M=4**时，A(n,4)的增长速度非常快，以至于没有适当的数学式子来表示这一函数。

1、证明 A(1,1)=2

因为n,m>=1使用递推式（4） 得 A(1,1)=A(A(0,1),0)

由A(0,m)=1,故 A(1,1）=A(1,0)

由（1）式 A(1,0)=2 所以A(1,1)=2

2、证明A(n,1)=2n (n>=1)

因为n, m>=1 使用递归式(4) 得A(n,1)=A(A(n-1,1),0)

由公式(3) A(n,1)=A(n-1,1)+2

由此式看出m=1 时，n每降一阶就增加2， 一直降到n=1
即A(1,1)共增加n个2，n 个2相加和为2n

所以A(n,1)=2n

3、证明当m=2时 $A(n,2)=2^n$

由递推式(4) A(n,2)=A(A(n-1,2),1),此时已经演变为m=1的情况，式中A(n-1,2) 相当于2节中的n的位置，因此利用2节的结论有 A(n,2)=2A(n-1,2)

可以看出随着n的降阶会增长2的倍数，那么当n降为1时即

A(1,2)的情况如何呢？

根据递推式(4)有

A(1,2)=A(A(0,2),1)，再由公式(2) A(0,m)=1

故A(1,2)=A(1,1)=2

所以A(n,2)随着n的降阶每次乘2，共乘n次得$A(n,2)=2^n$

公式得证

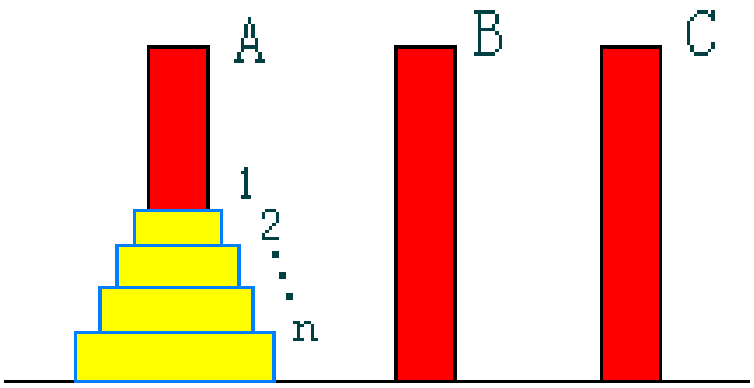
例2-6：Hanoi塔问题

- 问题定义：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：

（1）每次只能移动1个圆盘；

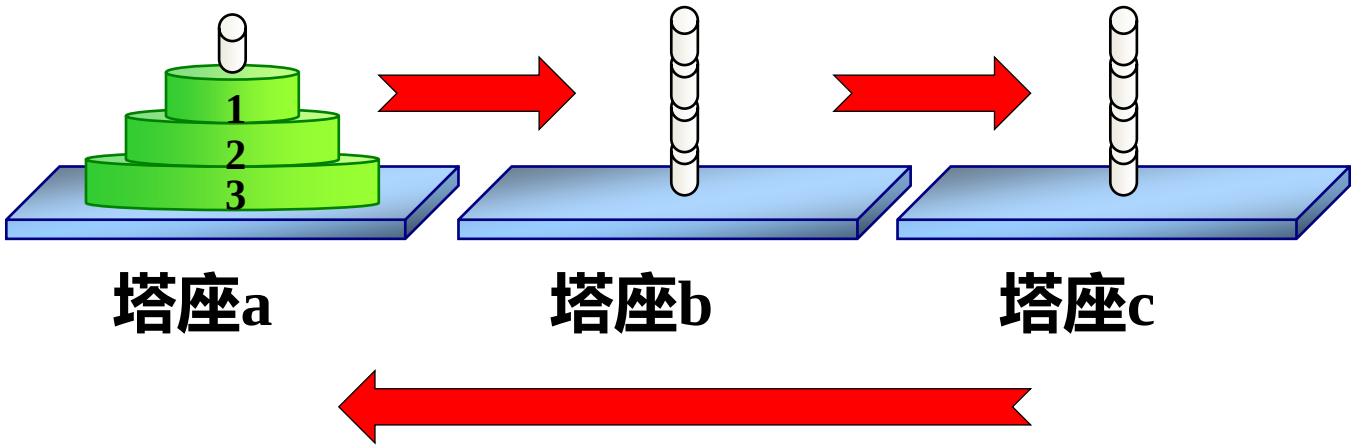
（2）任何时刻都不允许将较大的圆盘压在较小的圆盘之上；

（3）在满足移动规则1和2的前提下，可将圆盘移至a,b,c中任一塔座上。



简单解法

- 将A、B、C、A构成一个顺时针循环A->B->C->A。
- 在移动盘子的过程中，若是奇数次移动，则将最小的盘子移到顺时针方向的下一塔座上；
- 若是偶数次移动，则保持最小盘子不动，而在其他两个塔座之间，将较小的盘子移到另一塔座上。

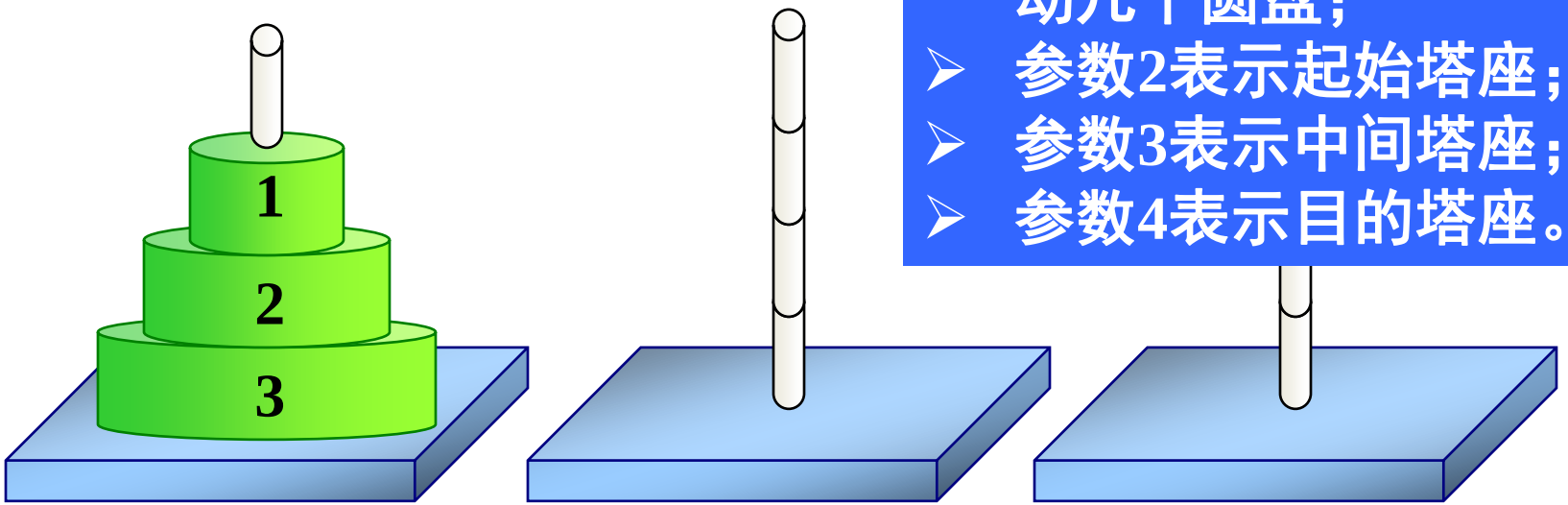


- 递归思路：
  - 当n=1时，只要将编号1的盘子从A移到B即可；
  - 当n>1时，以C为辅助，
    - 设法将n-1个较小的盘子从A移到C；
    - 将剩下的最大盘子从A移到B；
    - 最后，再设法将n-1个较小的盘子从C移到B。
    - n个盘子的移动就可以分解为两次n-1个盘子的移动。

特例：三阶Hanoi塔问题

需要完成：Hanoi(3,a,c,b)；

- 参数1表示一共需要移动几个圆盘；
- 参数2表示起始塔座；
- 参数3表示中间塔座；
- 参数4表示目的塔座。

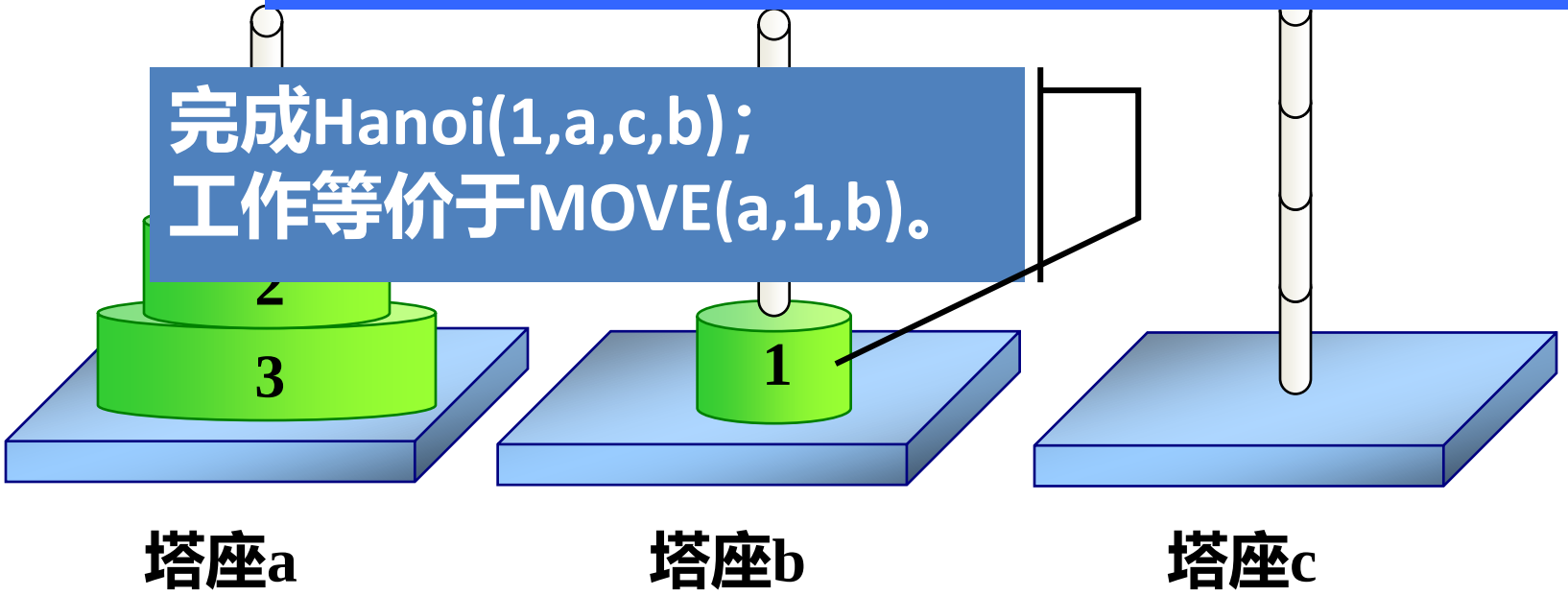


- 分析：起始塔座a上有三个圆盘，所以该问题被称为三阶Hanoi塔问题。

需要先完成：Hanoi(2,a,b,c)；

该工作又可分解成：① Hanoi(1,a,c,b)；② MOVE(a,2,c)；③ Hanoi(1,b,a,c)。

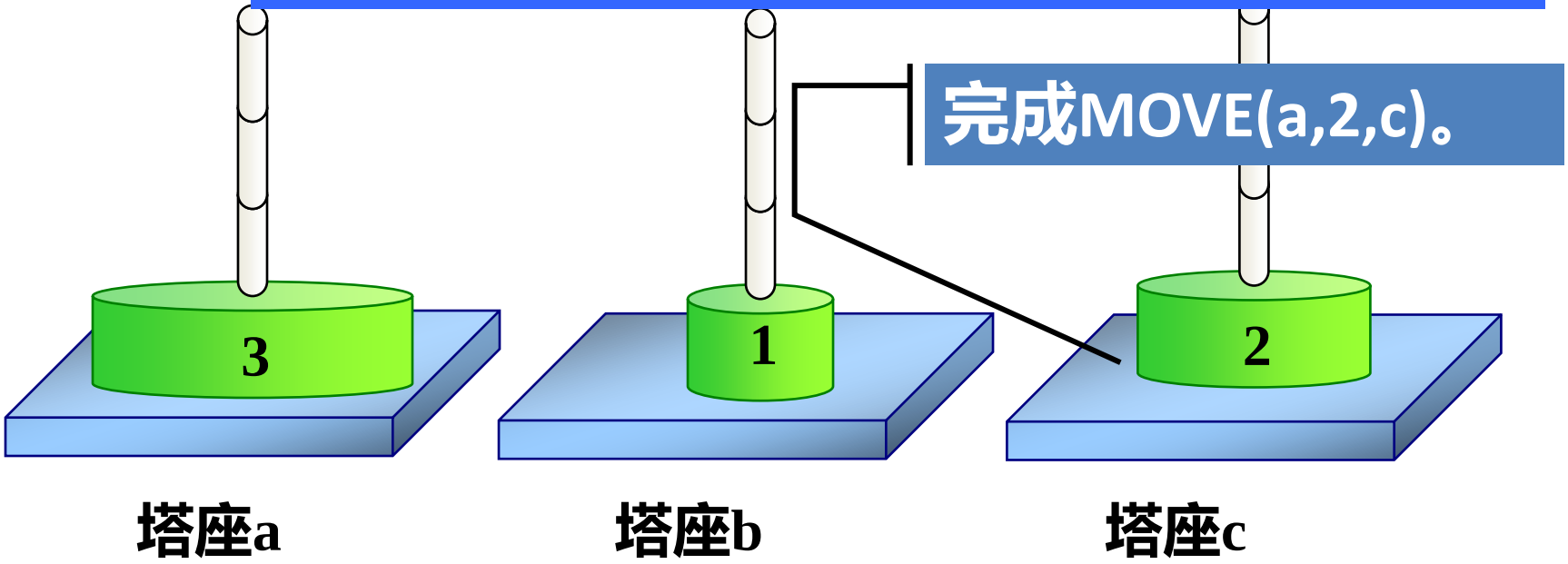
完成Hanoi(1,a,c,b)；
工作等价于MOVE(a,1,b)。



- 分析：完成Hanoi(3,a,c,b)的工作可以分解成如下三个步骤：Hanoi(2,a,b,c)、MOVE(a,3,b)和Hanoi(2,c,a,b)。

需要先完成：Hanoi(2,a,b,c)；

该工作又可分解成：① Hanoi(1,a,c,b)；② MOVE(a,2,c)；③ Hanoi(1,b,a,c)。



- 分析：完成Hanoi(3,a,c,b)的工作可以分解成如下三个步骤：Hanoi(2,a,b,c)、MOVE(a,3,b)和Hanoi(2,c,a,b)。

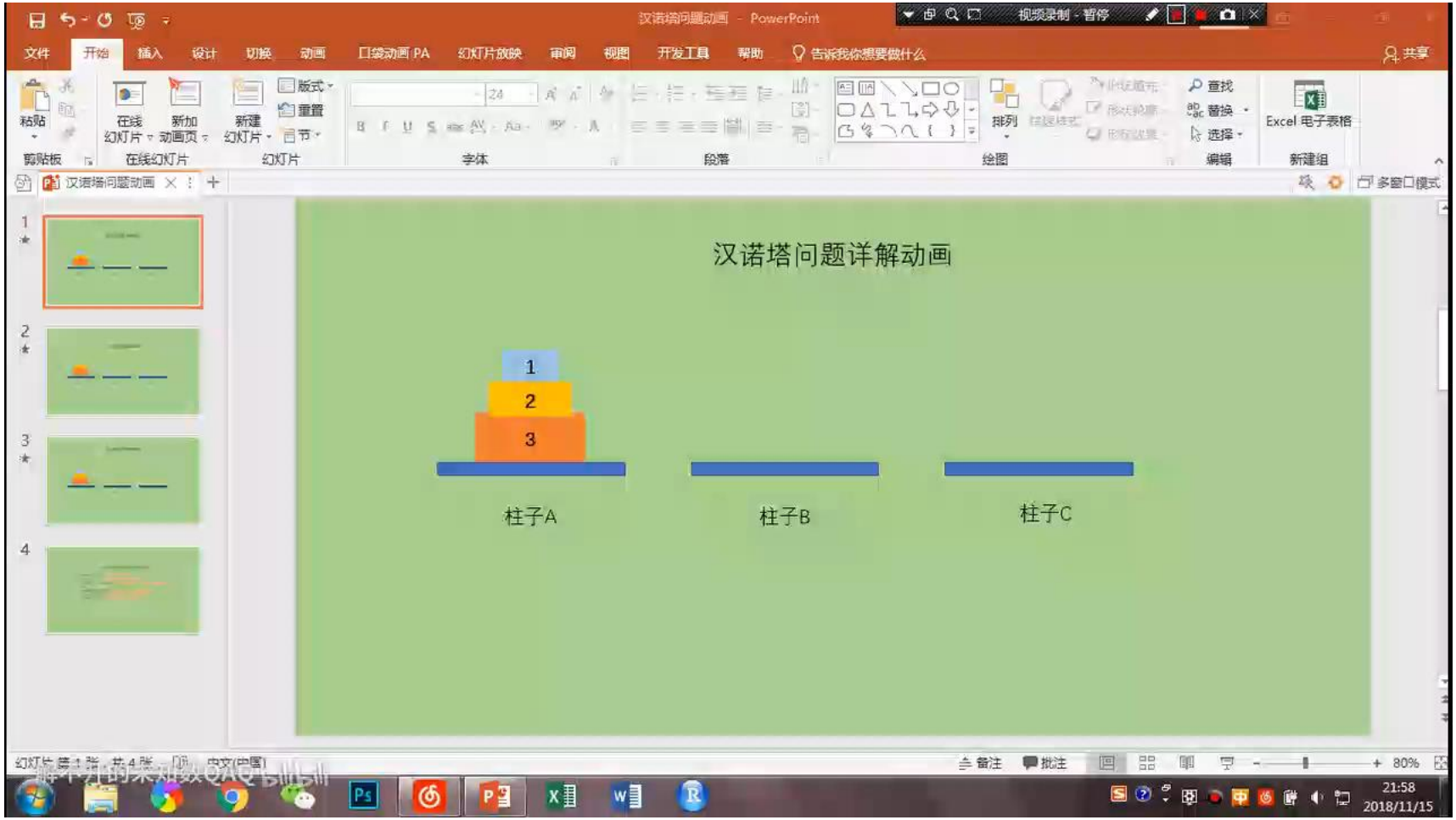
需要先完成：Hanoi(2,a,b,c)；

该工作又可分解成：① Hanoi(1,a,c,b)；② MOVE(a,2,c)；③ Hanoi(1,b,a,c)。

完成Hanoi(1,b,a,c)；
工作等价于MOVE(b,1,c)。

Hanoi(2,a,b,c)
完成!

- 分析：完成Hanoi(3,a,c,b)，可分解成如下三个步骤：Hanoi(2,a,b,c)、MOVE(a,3,c)和Hanoi(2,c,a,b)。


汉诺塔问题详解动画
1
2
3
柱子A
柱子B
柱子C

总结：三阶Hanoi塔问题

- 将一个复杂的问题（三阶Hanoi塔问题）归结为若干个较简单的问题（二阶Hanoi塔问题）；
- 然后将这些较简单的每一个问题（二阶Hanoi塔问题）再归结为更简单的问题，直到最简单的问题（一阶Hanoi塔问题）为止。

n阶Hanoi塔问题

- **求解的步骤：**

1. 如果n=1，则可直接将这一个圆盘移动到目的柱上，过程结束。如果n>1，则进行**步骤2**。
2. 设法将**起始柱**的上面*n-1个*圆盘（编号1到n-1）按移动原则移动到**中间柱**上。
3. 将**起始柱**上的*最后一个圆盘*（编号为n）移到**目的柱**上。
4. 设法将**中间柱**上的*n-1*圆盘按移动原则移到**目的柱**上。

- **步骤2**与**步骤4**实际上还是Hanoi塔问题，只不过其规模小一些而已。
- 如果最原始的问题为n阶Hanoi塔问题，且表示为 Hanoi(n, a, c, b)
- 则**步骤2**与**步骤4**为n-1阶Hanoi塔问题分别表示为：

  Hanoi(n-1, a, b, c)

  Hanoi(n-1, c, a, b)

  其中第一个参数表示问题的阶数，第二、三、四个参数分别表示起始柱、中间柱与目的柱。
- 将a塔座上的n号圆盘移到b塔座上

  Move(a, n, b)

- 算法：求解n阶Hanoi塔问题。

```
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);
    }
  }
```

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

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

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。

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。

尾递归是在递归调用时“积累”之前调用的结果，并将其传入下一次递归调用中,将递归方法中的需要的“所有状态”通过方法的参数传入下一次调用中.

```
int fib_1(int n) {
        if (n <= 1) {
                return 1 ;
        }
        else {
                return fib_1(n - 1) + fib_1(n - 2) ;
        }
}
```

尾递归只会占用恒量的内存,形式上只要最后一个return语句是单纯函数就可以

```
int fib_4(int n) {
        int fib_iter(int a , int b , int n) {
                if (n == 0) {
                        return a;
                }
                else {
                        return fib_iter(b , a + b , n - 1) ;
                }
        }
        return fib_iter(1 , 1 , n) ;
}
```

思考题：将P7 1-8算法改为递归算法。

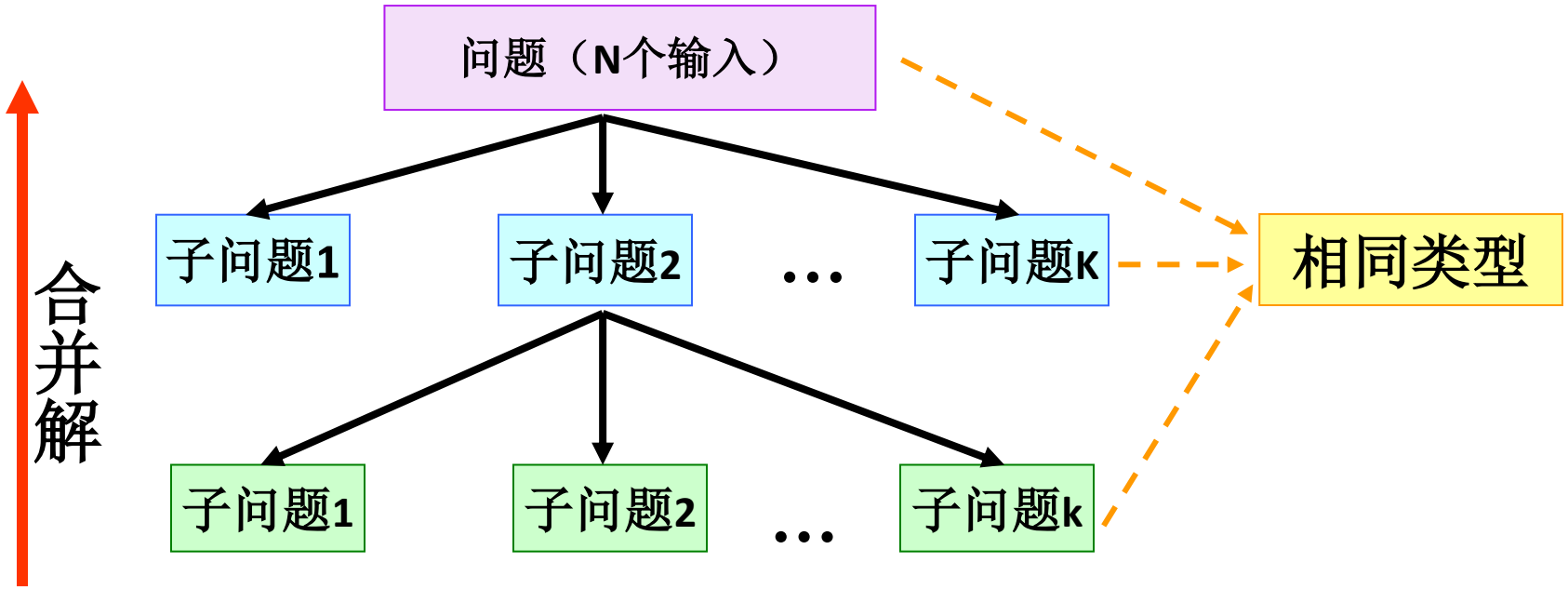
该问题是由L. Collatz在1937年提出的，也被称为3n+1问题、角谷猜想或冰雹猜想。

```
#include "pch.h"
#include <iostream>
# include<cstdio>
using namespace std;
int f(int n, int deep)
{
    if (n == 1)     return deep;
    if (n % 2 == 0)
    {   cout <<"执行次数：  "<<deep<< "  当前n值:" << n << en
        return (f(n / 2, deep + 1));    }
    else
    {   cout << "执行次数：  " << deep<< "   当前n值:" << n <
        return (f(3 * n + 1, deep + 1));    }
}
int main(void) {
    int x;
    cin >> x;
    cout << f(x, 0) << endl;
    return 0;
}
```

```
100
执行次数:  0  当前n值:100
执行次数:  1  当前n值:50
执行次数:  2   当前n值:25
执行次数:  3  当前n值:76
执行次数:  4  当前n值:38
执行次数:  5   当前n值:19
执行次数:  6  当前n值:58
执行次数:  7   当前n值:29
执行次数:  8  当前n值:88
执行次数:  9  当前n值:44
执行次数:  10   当前n值:22
执行次数:  11   当前n值:11
执行次数:  12  当前n值:34
执行次数:  13   当前n值:17
执行次数:  14  当前n值:52
执行次数:  15  当前n值:26
执行次数:  16   当前n值:13
执行次数:  17  当前n值:40
执行次数:  18  当前n值:20
执行次数:  19  当前n值:10
执行次数:  20   当前n值:5
执行次数:  21  当前n值:16
执行次数:  22  当前n值:8
执行次数:  23  当前n值:4
执行次数:  24  当前n值:2
25
```

2.2分治法的基本思想

- 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*原问题相同*。
- 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。
- 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。



- 分治法要求分解成**同类子问题**，并允许不断分解，使问题规模逐步减小，最终可用已知的方法求解足够小的问题。

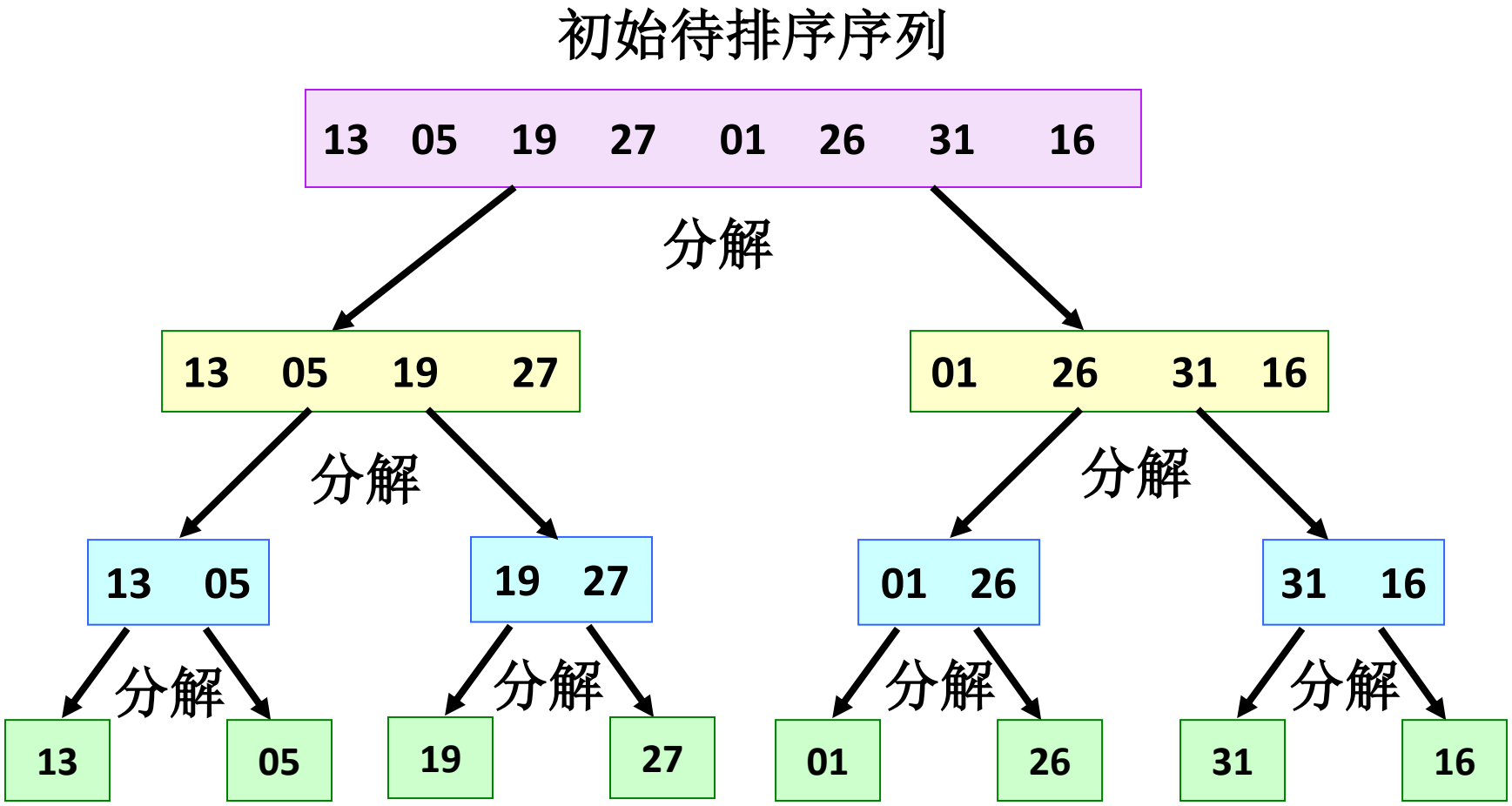
举例：二路归并排序

- 分解过程：对任意长度为*n*的序列排序，首先将问题不断分解，直至问题可直接求解。
  - 长度为*n*的序列分解为2个*n*/2的子序列，依次循环，直至分解为*n*个长度为1的序列，显然长度为1的序列是有序表。
- 合并过程：将已排序的的子序列不断合并，形成长度为*n*的排序序列。
  - 然后反复进行两两归并为*n*/2个有序表；再对*n*/2个有序表两两归并，循环直到得到长度为*n*的有序线性表。

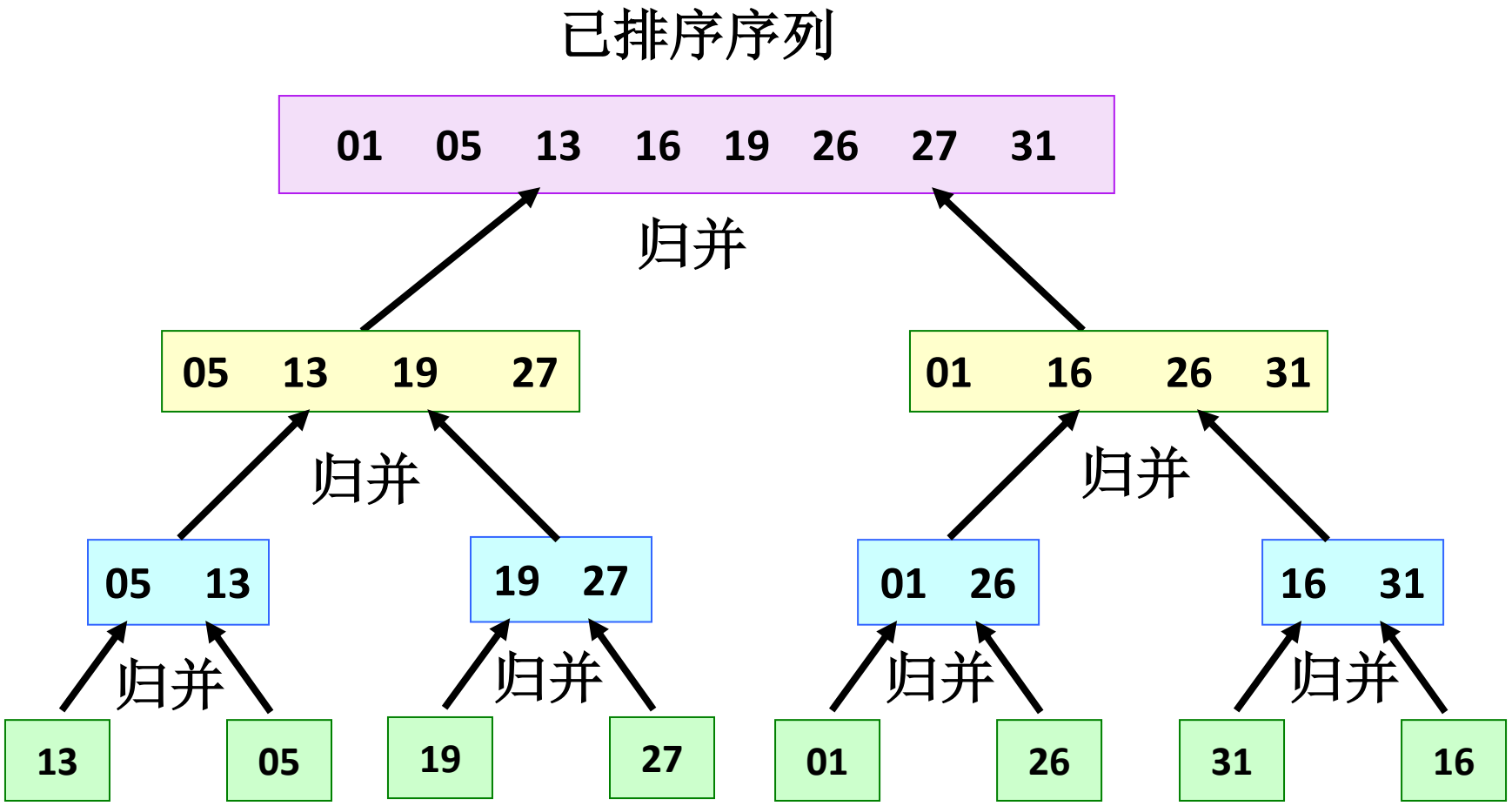
```
template < class Type >
void MergeSort( Type a[], int left, int right )
{
   if( left<right )  {          //至少有2个元素
      int i = ( left + right )/2;
      MergeSort( a, left, i );
      MergeSort( a, i+1, right );     //对后半部分排序
      Merge( a, b, left, i, right );  //合并到数组b
      Copy( a, b, left, right );      //复制回数组a
   }
}
```

问题分解：规模为原来的一半；问题性质不变。

合并步，将有序表a[left,i]和a[i+1,right]合并到b[left,right]




```
MergeSort( a, left, i );
MergeSort( a, i+1, right );
```



Merge(a, b, left, i, right);

Copy(a, b, left, right);

分析：二路归并排序的时空复杂性

```
template < class Type >
void MergeSort( Type a[], int left, int right )
{
  if( left<right )  {
    int i = ( left + right )/2;
    MergeSort( a, left, i );
    MergeSort( a, i+1, right );
    Merge( a, b, left, i, right );
    Copy( a, b, left, right );
  }
}
```

时间复杂度
T(n)
c
T(n/2)
T(n/2)
O(n)

```
void Merge(Type c[], Type d[], int i, int m, int r)
{  int la,lb,lc;
   la=i;lb=m+1;lc=i;
   while(la<=m&&lb<=n)
  {  if(c[la]<c[lb])    d[lc++]=c[la++];
     else     d[lc++]=c[lb++];}
   while(la<=m) d[lc++]=c[la++];
   while(lb<=n)   d[lc++]=c[lb++];
}
```

- 二路归并排序算法存在以下递推式：

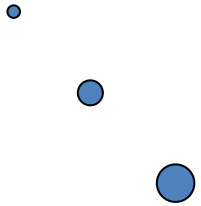
$$T(n)=\begin{cases}O(1) & n\le 1\\ 2T(n/2)+O(n) & n>1\end{cases}$$

- 解此递归方程，二路归并排序的**时间复杂度是O(nlogn)**。
- 二路归并排序其**空间复杂性为O(n)**。

求解递归式(递归树)

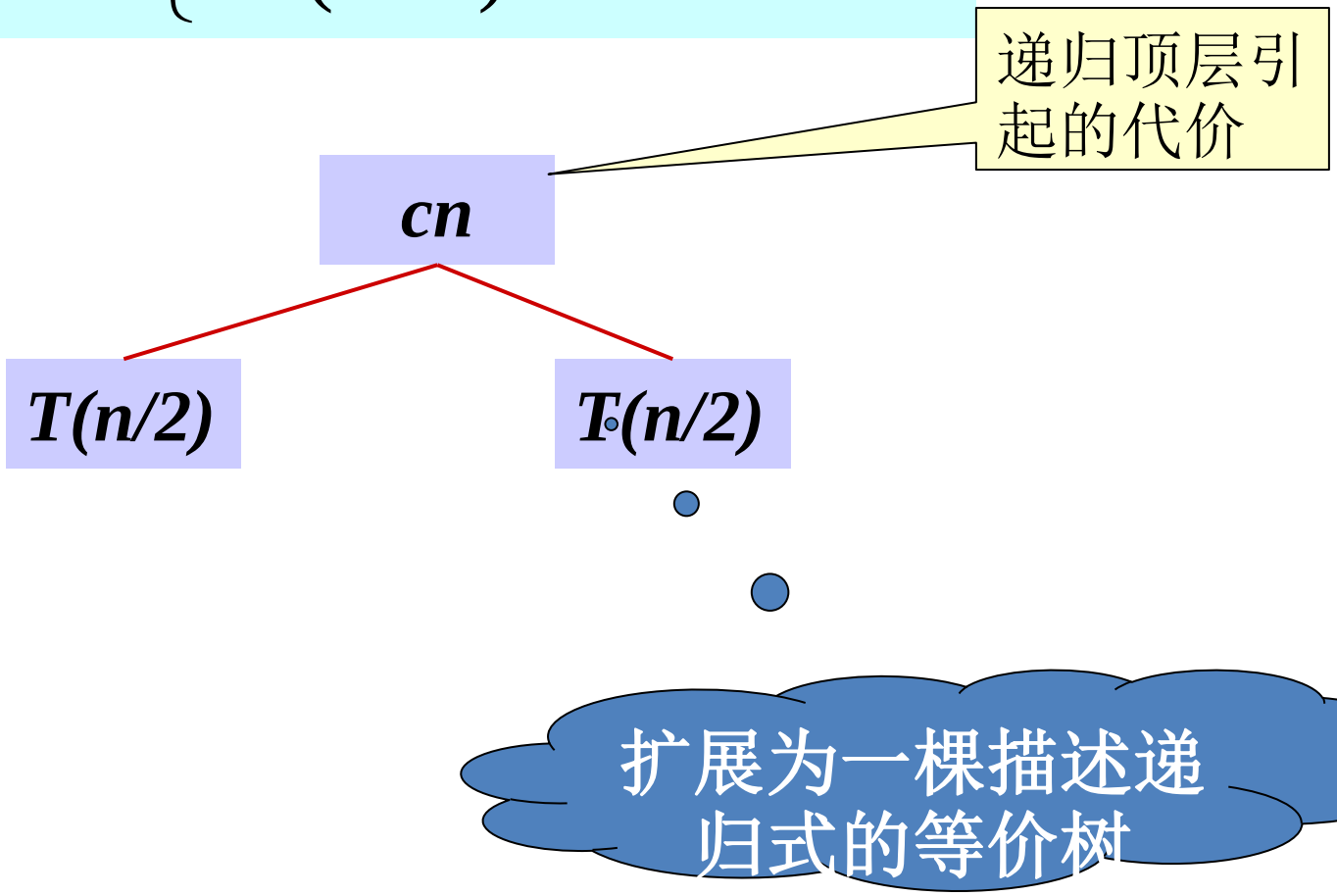
$$T(n)=\begin{cases} c & n \le 1 \\ 2T(n/2)+cn & n>1 \end{cases}$$

T(n)

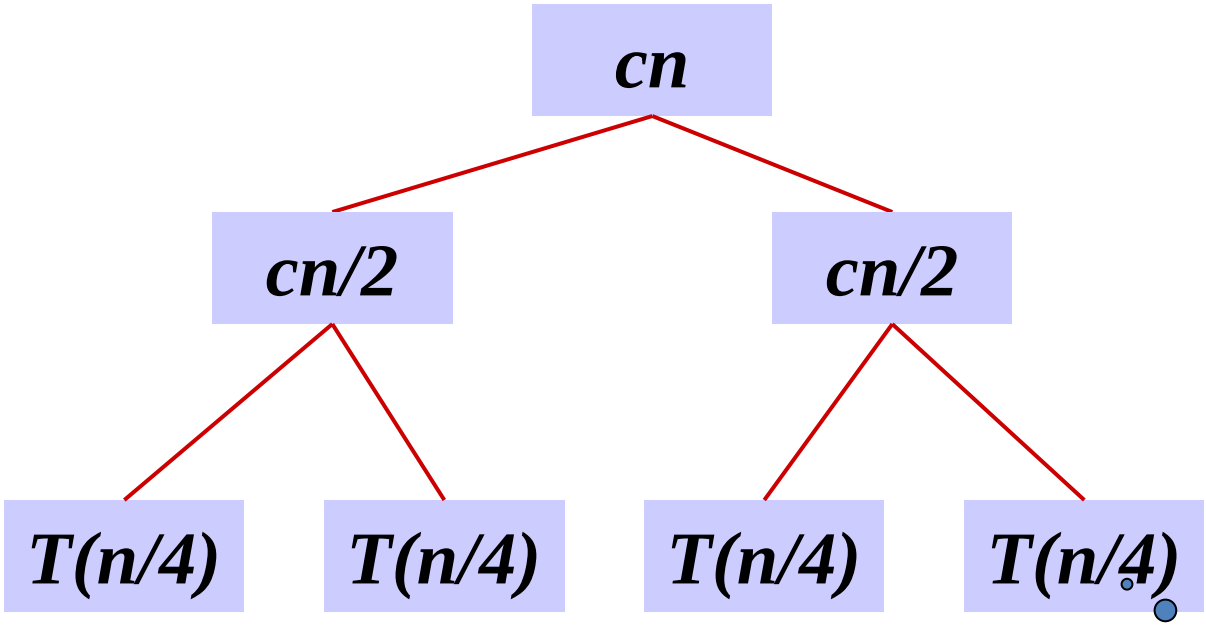


构造一颗递归树

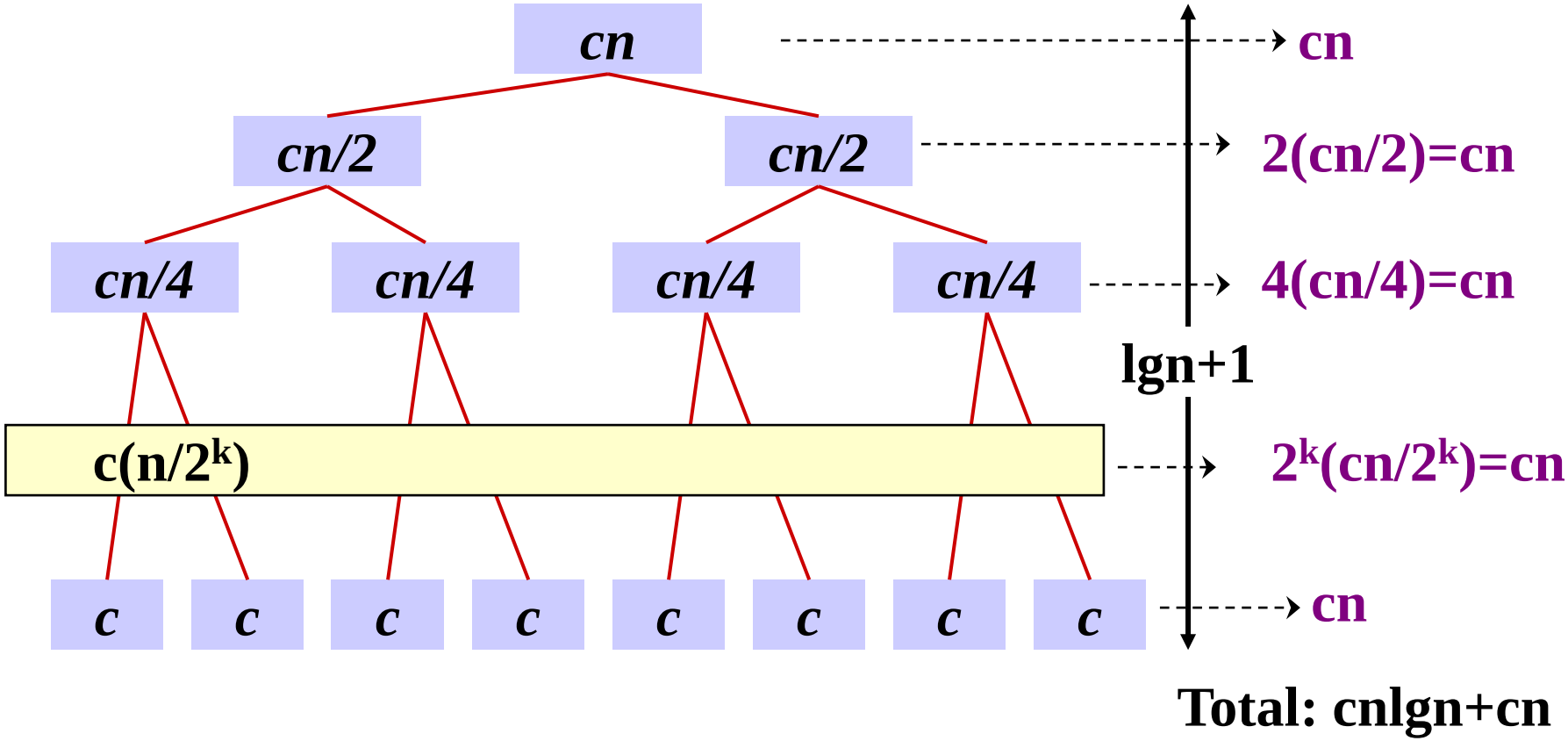
$$T(n)=\begin{cases} c & n\le 1 \\ 2T(n/2)+cn & n>1 \end{cases}$$



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



进一步推进



完全扩展的递归树具有lgn+1 层(如图所示其高度为lgn),每层将贡献总代价*cn*. 所以总代价为*cn*lgn+*cn*,就是 **Θ(nlgn)**

- 自然合并算法：将数组a中相邻元素两两配对，用合并算法将它们排序，构成n/2组长度为2的子数组段，循环这个过程

```
template<class Type>
void MergeSort(Type a[], int n) {
   Type  *b = new Type [n];
   int  s = 1;
   while (s < n) {
      MergePass(a, b, s, n);                    // 合并到数组 b
      s += s;
      MergePass(b, a, s, n);                    // 合并到数组 a
      s += s;
   }
}
```

$T(n)=O(n)$

总结：分治法的适用条件

- 分治法所能解决的问题一般具有以下特征：

1. 该问题规模缩小到一定的程
解决；
2. 该问题可以分解为若干个规
问题，即该问题具有**最优子结构性质**；
3. 利用该问题分解出的子问题的解可以合并为
该问题的解；
4. 该
是**相互独立**的，
即
问题。

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。

关键特征，能否利用分治法完全取决于问题是否具有第四条特征，如果具备了第一条和第二条特征，而不具备第四条特征，则可以考虑贪心法或动态规划法。

分治法的基本步骤：

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，有时也可用循环来实现。
3. **合并：**把各个子问题的解合并起来。

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
```

|P|问题P的规模
n0为阈值
adhoc(P)基本子算法

- 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**（许多问题取**k=2**）。
- 使子问题规模大致相等的做法是出自一种**平衡子问题**的思想，这通常比子问题规模不等的做法要好。

分治法的复杂性分析

- 从一般设计模式看，用分治法设计的程序通常是一个递归算法。
- 分治法将规模为n的问题分成**k个规模为n/m的子问题**：
  - 设n0=1,且adhoc解问题耗费**1个单位时间**。
  - 再设将原问题分解为k个子问题以及用merge将k个子问题的解合并为原问题的解需用***f (n)*个单位时间**。

用T(n)（递归方程）表示该分治法求解规模为|P|=n的问题所需的计算时间：

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

通过迭代法求得方程解：

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

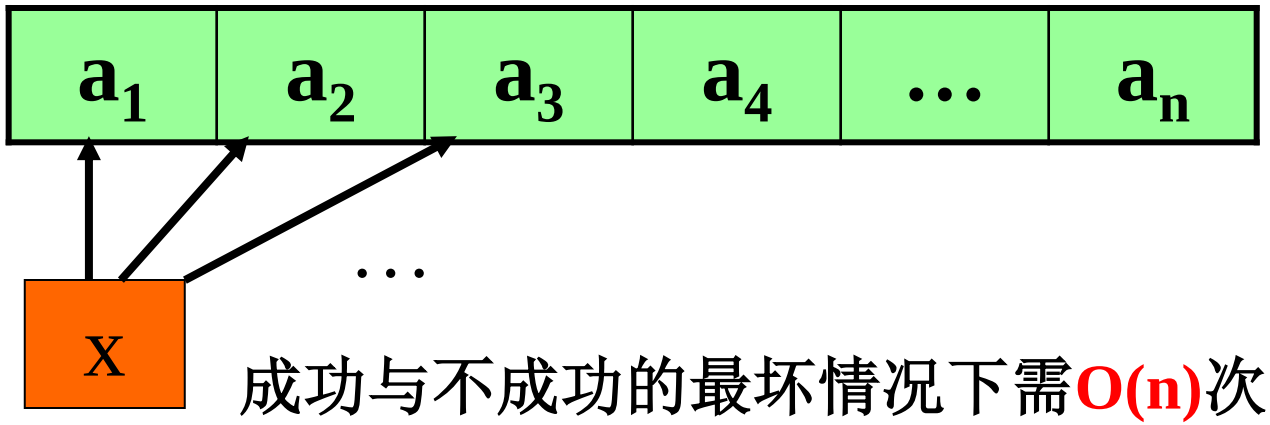
合并步花费时间

$$T(n)=n^{\log_m k}+\sum_{j=0}^{\log_m n-1} k^j f(n/m^j)$$

基本子问题花费时间

2.3 二分搜索技术

- **问题描述：**已知一个按非降次序排列的元素表 $a_1,a_2,\ldots,a_n$，判定某个给定元素x是否在该表中出现，若是，则找出该元素在表中的位置，并置于j，否则，置j为0。
- **一般解决方法：**从头到尾查找一遍。



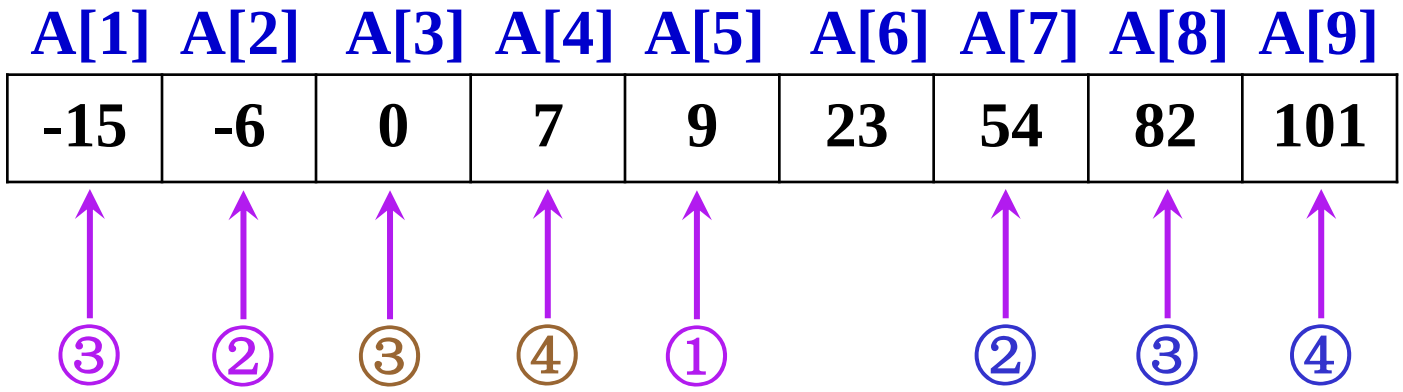
成功与不成功的最坏情况下需O(n)次比较

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，因此满足分治法的适用条件。

实例： 设在A(1:9)中顺序放了以下9个元素：



检索：

X=9　　9= =A[5], 一次比较就成功, **最好情况。**

X=-15　15<A[5], -15<A[2], -15= =A[1], 3次比较, **成功。**

X=101　101>A[5], 101>A[7], 101>A[8], 101= =A[9], 4次比较,**成功。**

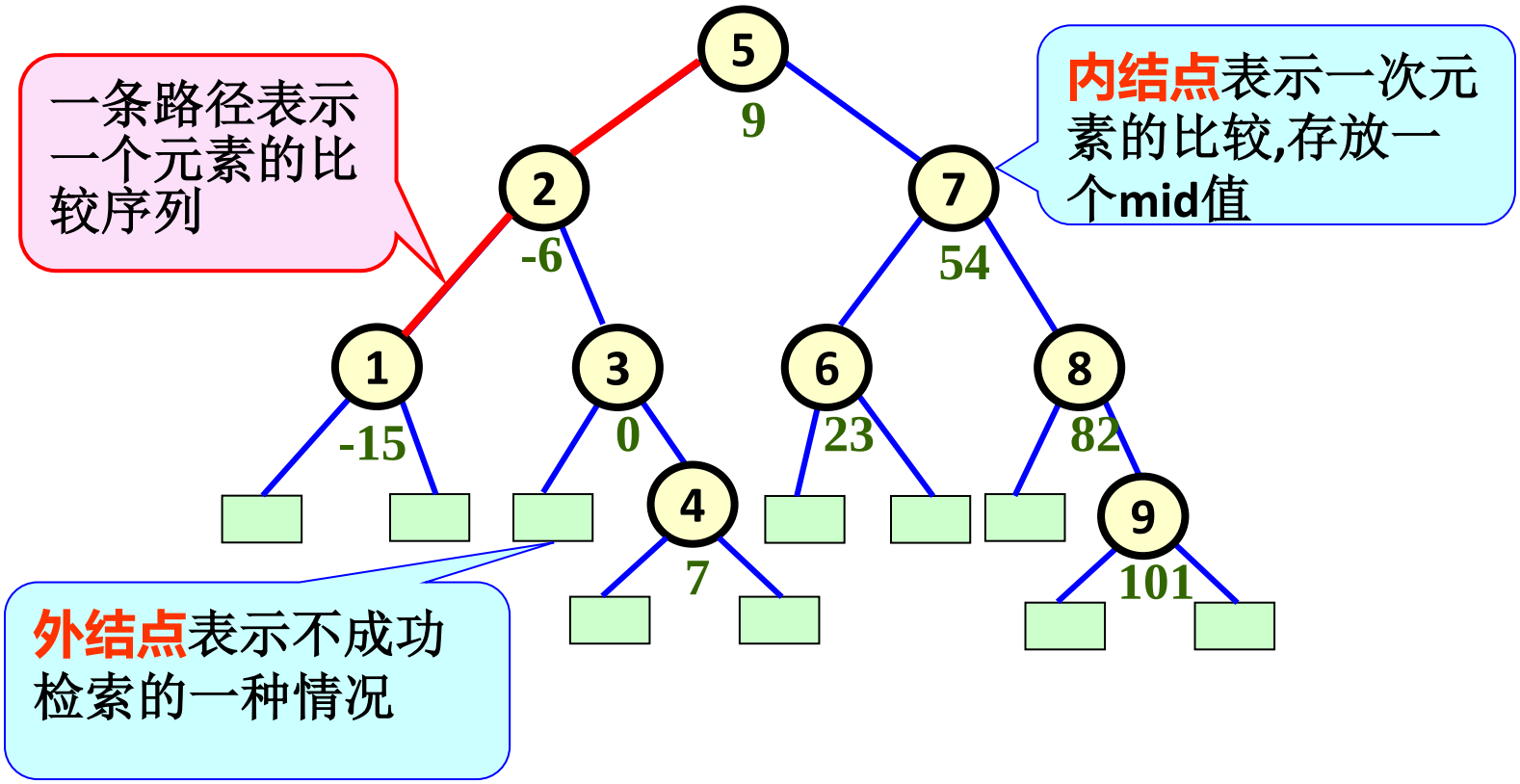
X=8　　8<A[5], 8>A[2], 8>A[3], 8>A[4], 4次比较, **检索不成功。**

- 成功检索有n种情况，不成功检索有n+1种情况：

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
	-15	-6	0	7	9	23	54	82	101
成功：	3	2	3	4	1	3	2	3	4
不成功：	3	3	3	4	4	3	3	3	4	4

- 不论检索是否成功，算法的执行过程都是x与一系列中间元素A[mid]的比较过程，可以用一棵**二叉判定树**来描述算法的执行过程。

二叉判定树



- 具有n（n>0）个内结点的二叉搜索判定树的高度为⌊logn⌋+1。
- 二分搜索算法在成功搜索的平均时间复杂度为Θ(logn)。

二分搜索算法：

```
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; el
        }
   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。

◆在最坏情况下，while循环被执行O(logn) 次，循环体内运算需要O(1)时间。

◆算法在最坏情况下的计算时间复杂性为O(logn) 。

2.8 快速排序

[算法思路] 对**A[p:r]**,按以下步骤进行排序:

(1)分解: 取A中一元素为支点，将A[p:r]分成3段:

A[p:q-1], A[q], A[q+1:r], 其中

- **A[p:q-1]中元素≤A[q],**
- **A[q+1: r]中元素 ≥A[q];**

(2)求解:通过递归调用分别对A[p:q-1]和A[q+1:r]排序。

(3)合并: 合并A[p:q-1], A[q], A[q+1:r]为A[p:r]。

例如：数组A[1:8]=[8, 4, 1, 7, 11, 5, 6, 9]，利用快速排序算法排序

① 选取元素8作为支点，分解：

A[q]=8; A[p: q-1]=[4, 1, 7, 5, 6]; A[q+1:r]=[11,9];

q=6

② 递归调用

③ 求解：A[p:q-1]=[1, 4, 5, 6, 7]; A[q+1:r]=[9, 11]

③ 合并：[1, 4, 5, 6, 7] [8] [9,11]

④ 结果：[1, 4, 5, 6, 7, 8, 9, 11]

[快速排序算法]

```
template <class T>
void QuickSoft(Ta[ ], int p, int r)
{  if(p<r){
   int q=Partition(a, p, r)
   QuickSort(a, p, q-1); //对左半段排
   QuickSoft(a, q+1, r); //对右半段排
```

```
template<class T>
int Partion(T a[ ],int p, int r
{  int i=p;  j=r+1;
   t  x=a[p];   //取支点
   //将≥x的元素交换到左边
   //将≤x的元素交换到右边
   while (true) {
        while(a[++i] < x);
        while(a[--j] > x);
        if (i>=j ) break;
        swap(a[i],a[j]); }
   a[p] = a[j];
   a[j] = x;
   return j   }
```

[复杂性分析]

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

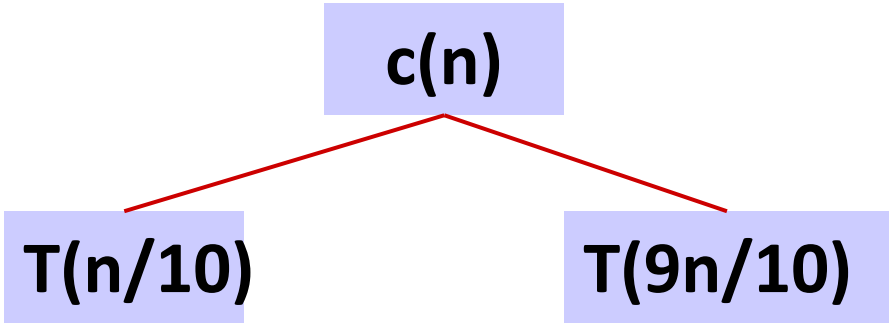
$$T_{min}(n)=\begin{cases} O(1) & n<=1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

得: $T_{min}(n)=O(nlogn)$

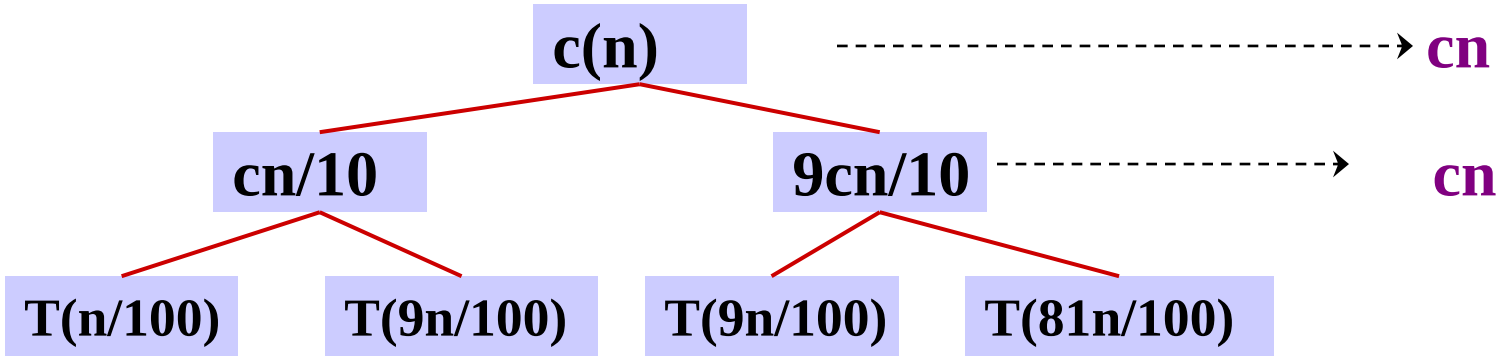
思考题

- 假设划分算法总是产生9：1的划分，快速排序算法的时间复杂度是多少？
- 平均情况下快速排序的时间复杂度是多少？

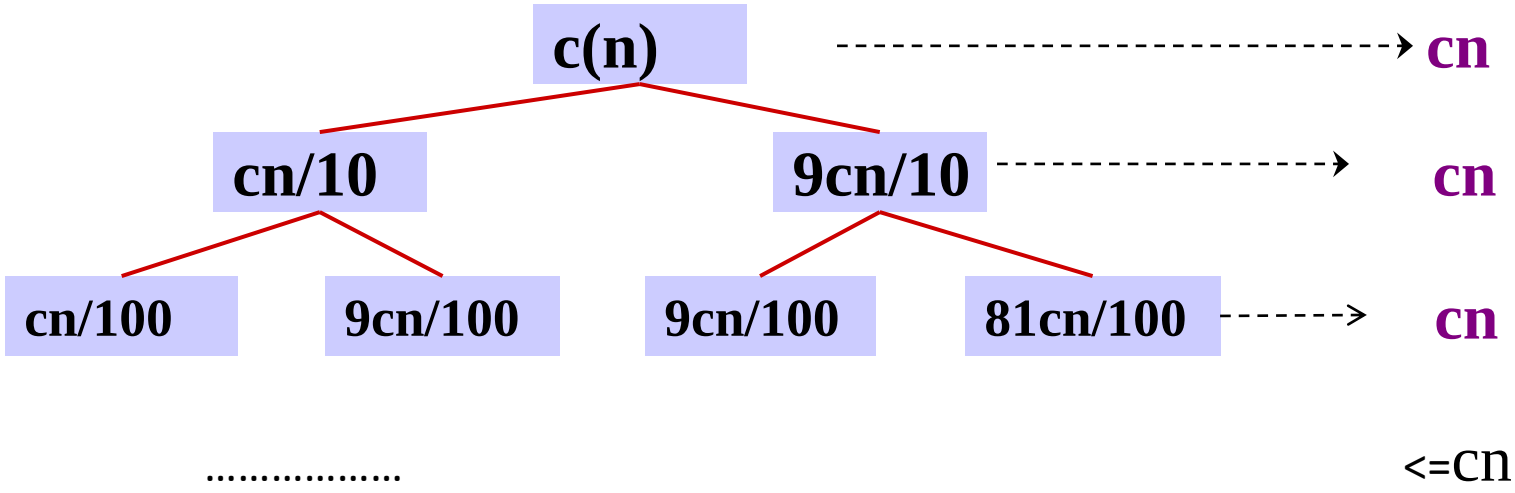
- 快速排序划分 1/10n和9/10n



$$T(n)=\begin{cases} O(1) & n<=1 \\ T(n/10)+T(9n/10)+O(n) & n>1 \end{cases}$$



依次循环 左枝深度为$\log_{10} n$, 右枝深度为$\log_{10/9} n$



$$T(n) \le cn \log_{10/9} n + \Theta(n)$$

快速排序算法性能取决于划分的对称性

[随机选择支点的快速排序]

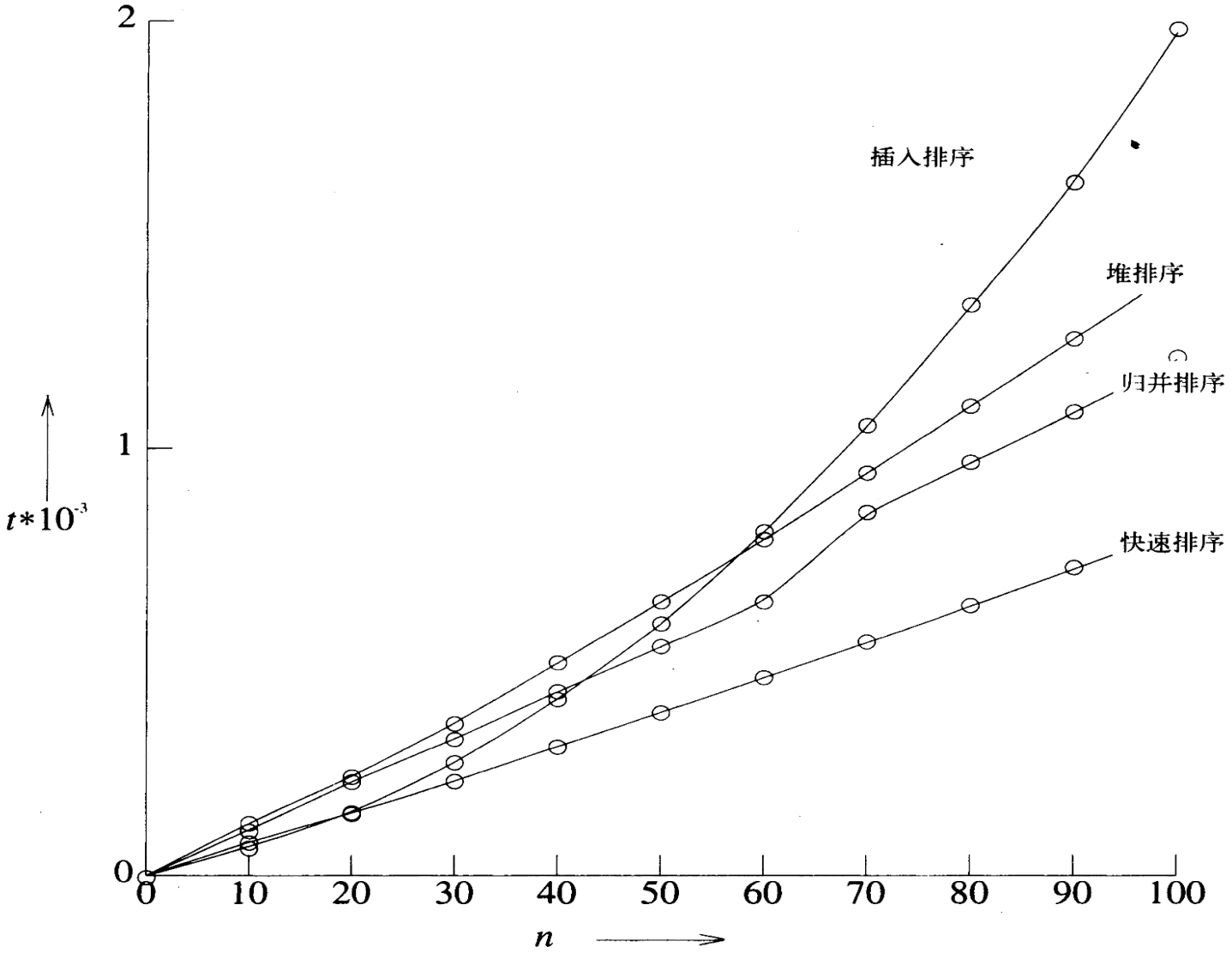
```
template <class T>
void randomizedQuickSoft(T a[], int p, int r)
{  if (p<r) {
   int q=randomizedPartition(a, p, r)
   randomizedQuickSort(a, p, q-1); //对左半段排序
   randomizedQuickSoft(a, q+1, r); //对右半段排序
   }}
```

```
template <class T>
int randomizedPartition(T a[], int p, int r)
{ int i=random( p, r)
    swap( a[i], a[p] )
return Partition (a,p,r)
```

[排序算法效率比较]

排序方法	平均时间	最坏情况	辅存
选择排序	$O(n^2)$	$O(n^2)$	O(1)
插入排序	…	…	…
冒泡排序	…	…	…
希尔排序			
快速排序	O(nlogn)	$O(n^2)$	O(logn)
堆排序	O(nlogn)	O(nlogn)	O(1)
归并排序	O(nlogn)	O(nlogn)	O(n)
基数排序	O(k(n+m))	O(k(n+m))	O(m)

排序算法效率比较



2.9 线性时间选择

[问题陈述] 求n个元素中的第k小元素.

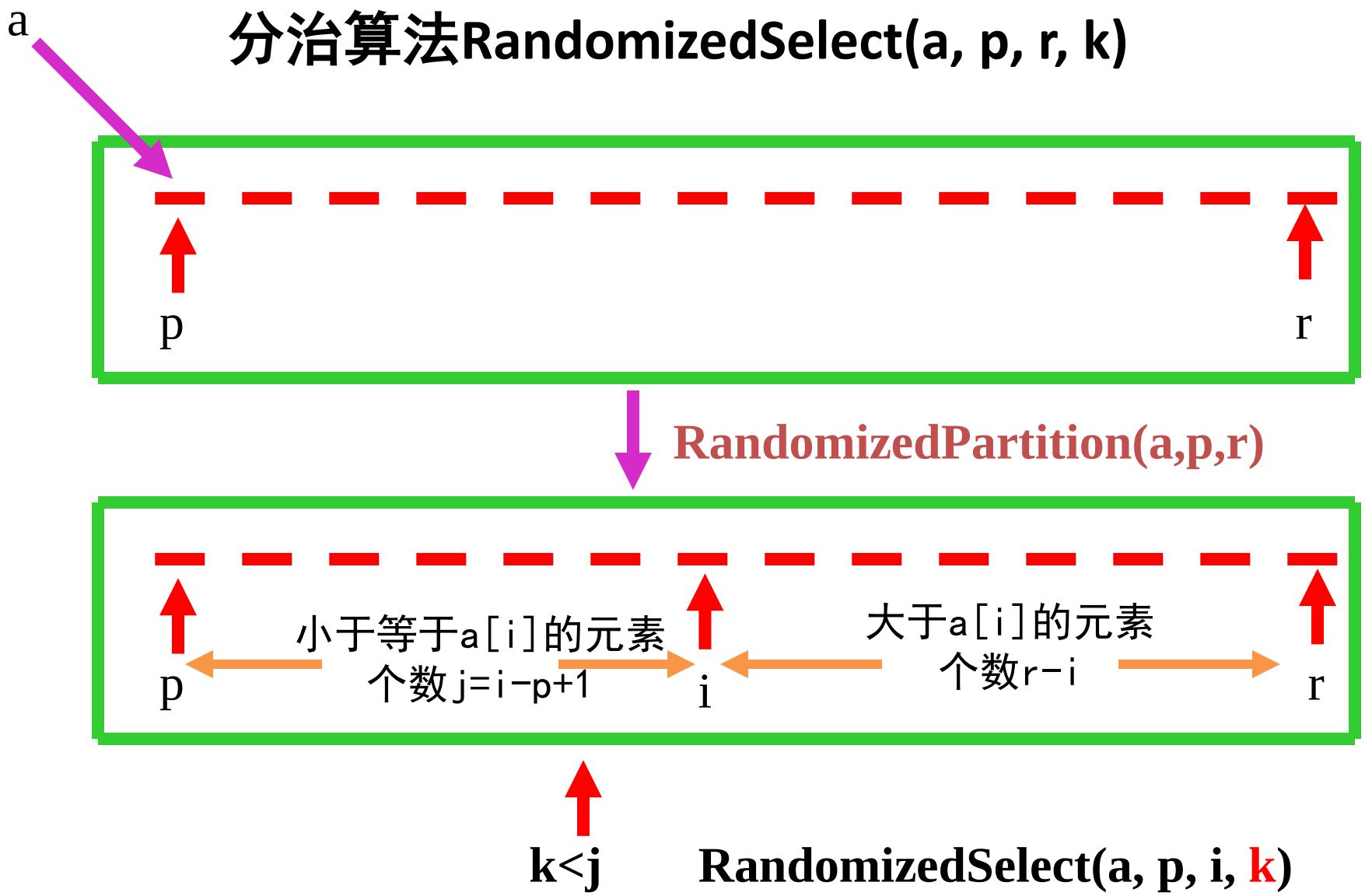
➢给定线性序集中n个不同的元素和一个整数k，1≤k≤n要求找出这n个元素中第k小的元素，即如果将这n个元素依其线性序排列时，排在第k个位置的元素即为我们要找的元素。

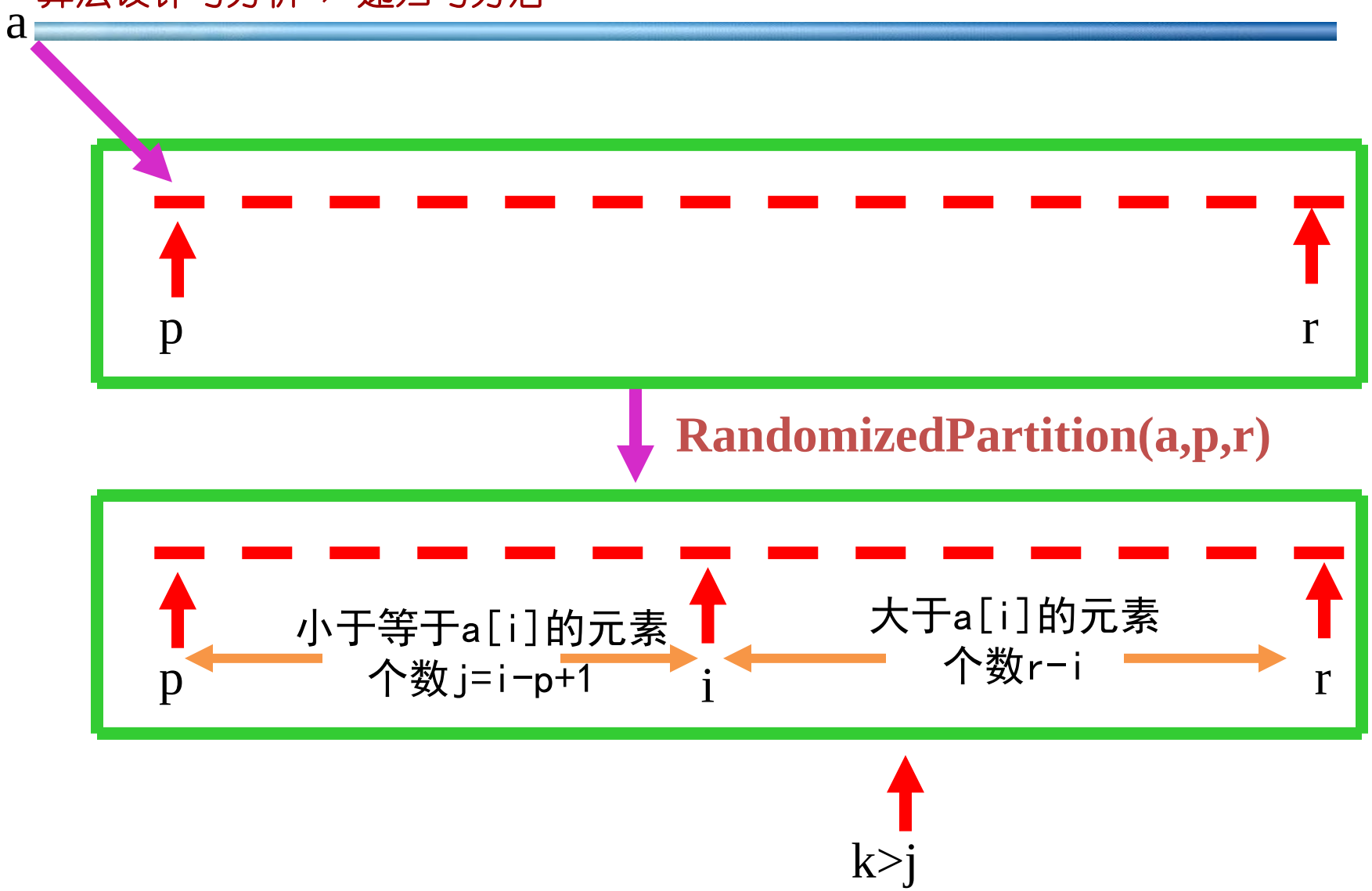
- **当k=1时,找最小元;**
- **当k=n时,找最大元素;**
- **当k=(n+1)/2时,找中位数.**

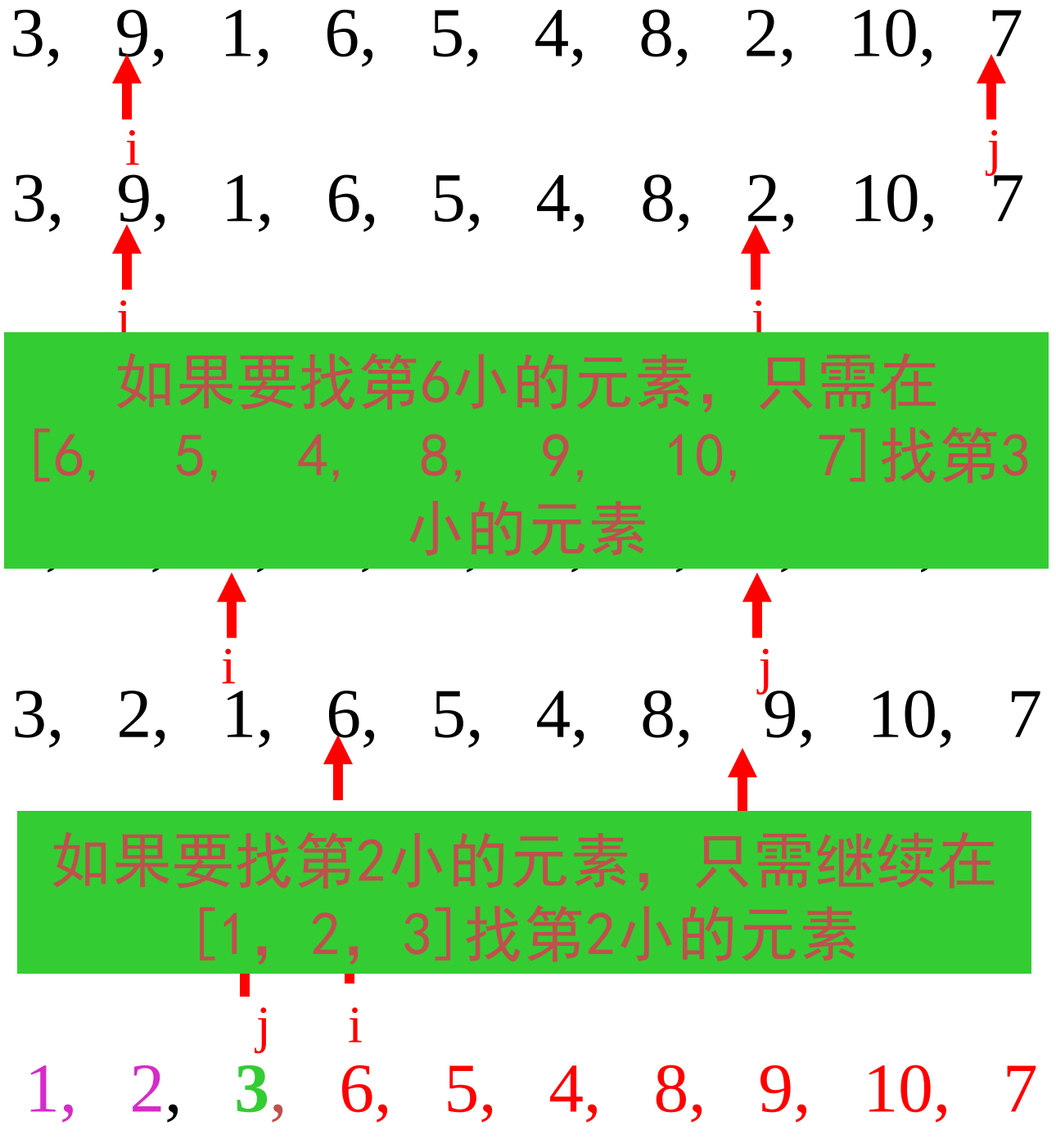
[算法思路] 模仿快速排序,对输入数组A进行二分,使子数组A1的元素≤ A2中的元素,分点J由随机数产生.

- 若K≤J,则K为A1的第K小元,
- 若K>J,则K为A2的第 K-J小元.

分治算法RandomizedSelect(a, p, r, k)




a
p
r
RandomizedPartition(a,p,r)
p
小于等于a[i]的元素
个数j=i-p+1
i
大于a[i]的元素
个数r-i
r
k>j
RandomizedSelect(a,i+1, r,k-j)


3, 9, 1, 6, 5, 4, 8, 2, 10, 7
i
j
3, 9, 1, 6, 5, 4, 8, 2, 10, 7
i
i
如果要找第6小的元素，只需在[6, 5, 4, 8, 9, 10, 7]找第3小的元素
i
j
3, 2, 1, 6, 5, 4, 8, 9, 10, 7
如果要找第2小的元素，只需继续在[1，2，3]找第2小的元素
j
i
1, 2, 3, 6, 5, 4, 8, 9, 10, 7
3
支点

```
template < class T >
T RandomizedSelect (a[ ], int p, int r, int k)
 {  if (p==r) return a[p];
    int  i = RandomizedPartition(a, p, r);
         j=i-p+1;   //支点元素是第j小个元素,左半部元素个数

    if  k==j   return A[i];
    if ( k < j )
        return RandomizedSelect(a, p, i-1, k);
    else
        return RandomizedSelect(a, i +1, r, k - j);
 }
```

p,r 数组范围
K:第k小个元素

与快速排序算法不同,它只对子数组之一进行递归处理。

找数组a[0:n-1]第k小元素调用RandomizedSelect(a,0,n-1,k)

① 执行RandomizedPartiton，数组a[p:r]划分成两个子数组a[p:i]和a[i+1:r]，其中a[p:i]元素≤a[i+1:r]元素

② 计算子数组a[p:i]中元素个数j

- ✓如果k<j,则a[p:i]中第k小元素落在子数组a[p:i]中;
- ✓如果k>j,则要找的第k小元素落在子数组a[i+1:r]中，要找的a[p:r]中第k小的元素是a[i+1:r]中的第k-j小的元素。
- ✓如果k=j,找到第k小个元素，返回a[i]

③递归循环。

[分析]

$$T_{min}(n)=\begin{cases} d \\ T(n)=T(n/2)+cn \end{cases}$$

得: $T_{min}(n)=\theta(n)$ (等分点)

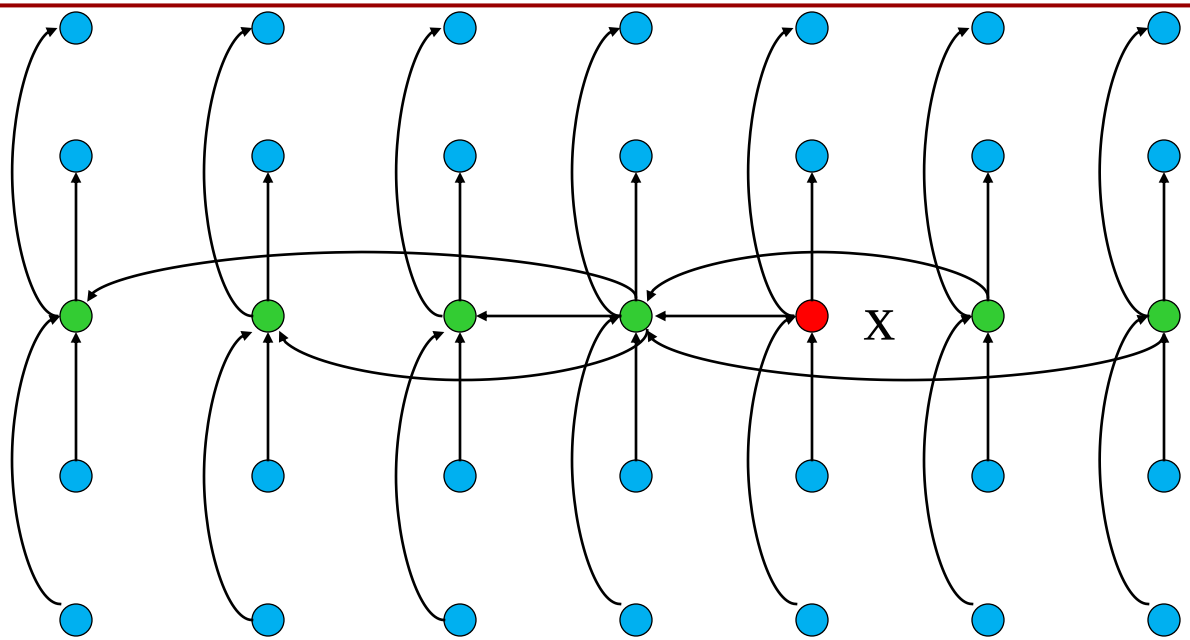
最坏情况: $T_{max}(n)=\Omega(n^2)$ (左半总为空)

最坏情况下的选择算法分析

- ✓算法RandomizedSelect需要$\Omega(n^2)$计算时间。
  - ✓例如在找最小元素时，总是在最大元素处划分。$(n-1)+(n-2)+\ldots+1=n^2$
- ✓算法RandomizedSelect不稳定的原因：
  - ✓由于随机划分函数使用了一个随机数产生器Random，产生p和r之间的一个随机整数，因此产生的划分基准是随机的。
- ✓可以证明，算法可以在$O(n)$平均时间内找出n个输入元素中的第k小元素。

[算法思路](中间的中间):

- 将A中元素分为n/r组,除最后一组外,每组元素为r个,用任意排序算法排序,取出每组的中位数,
- 对n/r个中位数递归调用,找到中位数,
- 以该元素作为划分基准.



- 5, 6, 4, 3, 7, 9, 10, 1, 12, 11, 13, 15, 2, 8, 14
- [5, 6, 4, 3, 7] ----------中位数为5
- [9, 10, 1, 12, 11]----------中位数为10
- [13, 15, 2, 8, 14]----------中位数为13
- 5, 10, 13的中位数为 10
- 所选的基准元素为10

- 5, 6, 4, 3, 7, 9, 10, 1, 12, 11, 13, 15, 2, 8, 14
- 10, 6, 4, 3, 7, 9, 5, 1, 12, 11, 13, 15, 2, 8, 14
  i j
- 10, 6, 4, 3, 7, 9, 5, 1, 12, 11, 13, 15, 2, 8, 14
  i j
- 10, 6, 4, 3, 7, 9, 5, 1, 8, 11, 13, 15, 2, 12, 14
  i j
- 10, 6, 4, 3, 7, 9, 5, 1, 8, 2, 13, 15, 11, 12, 14
  i j
- 2, 6, 4, 3, 7, 9, 5, 1, 8, 10, 13, 15, 11, 12, 14

$\varepsilon=10/15=2/3,\ 1-\varepsilon=1/3$

最坏情况下的选择算法:

定理:若r=5且数组中元素互不相同,则n>75时,
max{ |left|,|right| }<=3n/4

```
template<class T>
T Select(T a[], int p, int r, int k )
{  if  (r-p<75) {   用简单排序法排序
      return a[p+k-1] };

  for(int i=0 ;i<=(r-p-4)/5 ; i++)
     将a[p+5*i]至a[p+5*i-4]的第3小元素与a[p+i]交换位置;

  //找中位数的中位数
  T x=Select(a, p, p+(r-p-4)/5, (r-p-4)/10);
  int i=Partition(a,p,r,x), j=i-p+1;
  if (k<=j) return Select(a,p,i,k)
  else return Select(a,i+1,r,k-j)  }
```

[算法分析]

$$T(n)=\begin{cases} c_1 & n<75 \\ c_2 n+T(n/5)+T(3n/4) & \end{cases}$$

得: T(n)=O(n)

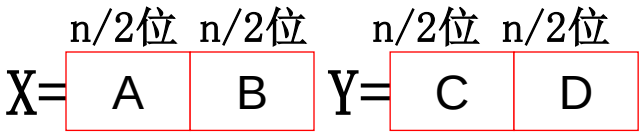
2.4 大整数的乘法

[问题描述] 设X,Y是两个n位二进制数,求X*Y.

- 一般方法： $O(n^2)$ ×效率太低!
- 分治法：$X = a \times 2^{n/2} + b$

$$Y = c \times 2^{n/2} + d$$

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



- 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

[算法改进]

- 为了降低时间复杂度，必须减少乘法的次数。
  为此，把XY写成另外的形式：

$$XY = ac2^n + ((a-b)(d-c)+ac+bd)\ 2^{n/2} + bd \text{ 或}$$

$$XY = ac2^n + ((a+b)(c+d)-ac-bd)\ 2^{n/2} + bd$$

[分析]这两个算式更复杂一些，但它们仅需要3次n/2位乘法。{ac、bd和(a±b)(d±c)}

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

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

大数相乘的分治递归算法

function MULT(X,Y,n)

{X,Y为2个小于2^n的二进制整数，S=XY}

```
{ S=SIGN(X)*SIGN(Y);          存放XY的符号
  X:=ABS(X);  Y:=ABS(Y);
  if n=l then
     if (X=1) and (Y=1)  then  return(S)
     else return(0)
  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);      } 存放三个乘积
         m3:=MULT(B，D，n/2);
         S:=S*(m1*2^n+(m1+m2+m3)*2^(n/2)+m3);
         return (S) }}
```

二进制大整数乘法同样可应用于十进制大整数的乘法.

2.5 Strassen矩阵相乘法

[常规算法] 设矩阵$A=(a_{ij})_{n\times n}$, $B=(b_{ij})_{n\times n}$, $C=A\times B=(c_{ij})_{n\times n}$

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

- 如果依此定义来计算矩阵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。

  计算C共需$n\times n^2$个乘法, $n^2(n-1)$个加法. $T(n)=O(n^3)$

[算法思路]

- 首先假定*n是2（$n=2^k$）的幂*，如果相乘的两矩阵A和B不是方阵，可以通过**适当添加全零行和全零列，使之成为行列数为2的幂的方阵**。
- 使用分治法，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，每个子矩阵都是n/2×n/2的方阵 。由此可将方程C=A×B重写为：

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

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

- 如果n=2，则两个2阶方阵的乘积可以直接计算出来，共需8次乘法和4次加法。
- 当子矩阵的阶大于2时，可以继续将**子矩阵分块**，直到子矩阵的阶降为2。这样，就产生了一个**分治降阶的递归算法**。依此算法，计算2个n阶方阵的乘积转化为计算8个n/2阶方阵的乘积和4个n/2阶方阵的加法。2个n/2×n/2矩阵的加法显然可以在$c*n^2/4(O(n^2))$ 时间内完成，这里c是一个常数。
- 上述分治法的计算时间耗费T(n)的递归方程满足：

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

$T(n)=O(n^3)$

✖没有改进，原因是没有减少矩阵乘法次数。

✓为了降低时间复杂度，必须减少乘法次数，其关键在于计算2个2阶方阵的乘积时所用乘法次数能否少于8次。为此，Strassen提出了一种只用**7次乘法运算**计算2阶方阵乘积的方法（但增加了加/减法次数）：

✓$M_1=A_{11}(B_{12}-B_{22})$

$M_2=(A_{11}+A_{12})B_{22}$

$M_3=(A_{21}+A_{22})B_{11}$

$M_4=A_{22}(B_{21}-B_{11})$

$M_5=(A_{11}+A_{22})(B_{11}+B_{22})$

$M_6=(A_{12}-A_{22})(B_{21}+B_{22})$

$M_7=(A_{11}-A_{21})(B_{11}+B_{12})$

$$\begin{bmatrix} M_5+M_4-M_2+M_6 & M_1+M_2 \\ M_3+M_4 & M_5+M_1-M_3-M_7 \end{bmatrix}$$

✓做了这7次乘法后，再做若干次加/减法就可以得到：

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

[分析]

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

得: $T(n)=O(n^{\log 7})$

$=O(n^{2.81})$

✓较大的改进

算法

矩阵相乘 Strassen 算法

procedure STRASSEN(n, A, B, C)；

{ if n=2 then MATRIX_MULTIPLY(A, B, C)

else { **将矩阵A和B分块；**

STRASSEN(n/2， A_{11}， $B_{12}-B_{22}$， M_1)；

STRASSEN(h/2， $A_{11}+A_{12}$， B_{22}， M_2)；

STRASSEN(n/2， $A_{21}+A_{22}$， B_{11}， M_3)；

STRASSEN(n/2， A_{22}， $B_{21}-B_{11}$， M_4)；

STRASSEN(n/2， $A_{11}+A_{22}$， $B_{11}+B_{22}$， M_5)

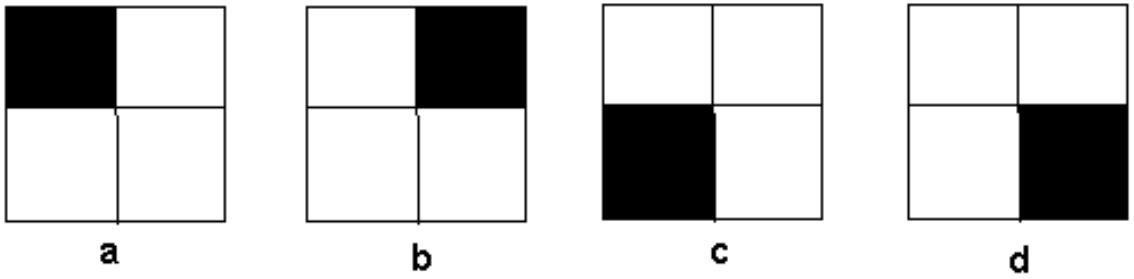
STRASSEN(n/2， $A_{12}-A_{22}$， $B_{21}+B_{22}$， M_6)；

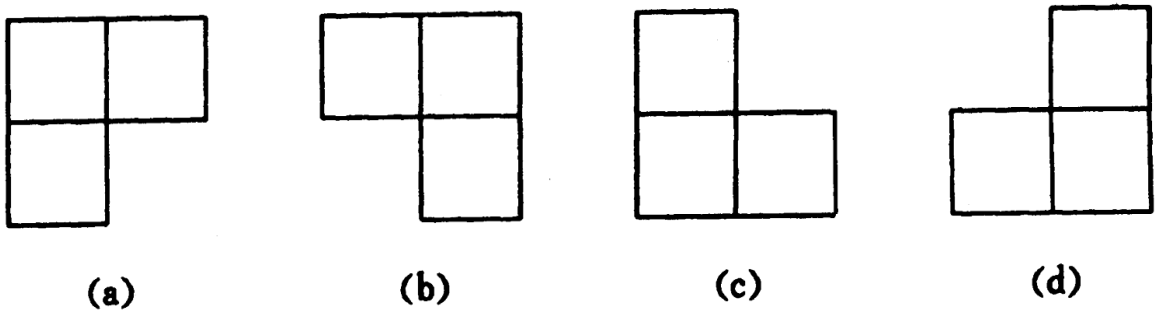
STRASSEN(n/2， $A_{11}+A_{21}$， $B_{11}+B_{12}$， M_7)

$$C:=\begin{bmatrix} M_5+M_4-M_2+M_6 & M_1+M_2 \\ M_3+M_4 & M_5+M_1-M_3-M_7 \end{bmatrix}$$

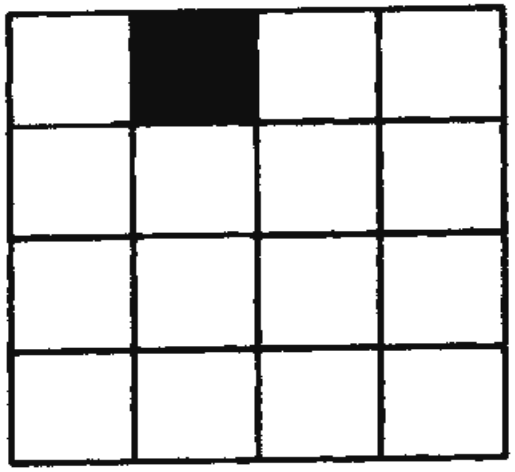
2.6 棋盘覆盖

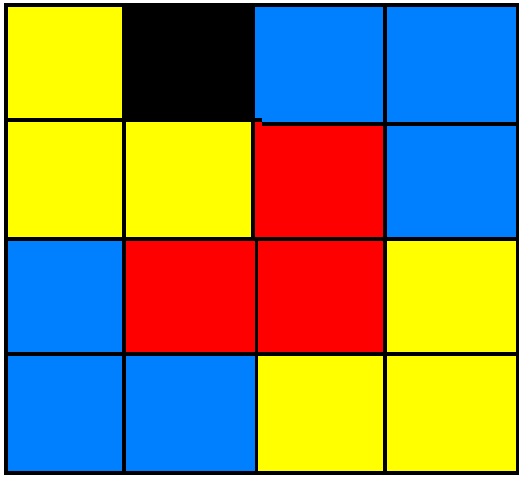
[问题陈述]在一个$2^k \times 2^k$个方格组成的棋盘中,恰有一方格残缺，残缺方格的位置有2^{2k}种。故对任何$k \geq 0$,残缺棋盘有2^{2k}种．要求用L型骨牌覆盖残缺棋盘上的所有方格且任何2个L型骨牌不得重叠覆盖。





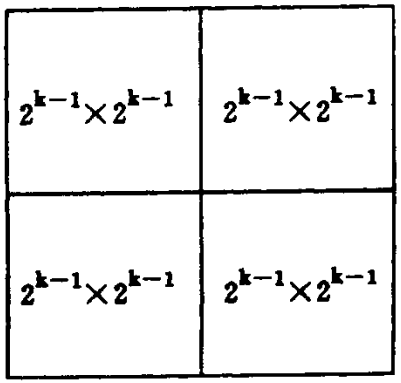
$2^k \times 2^k$ 的棋盘覆盖中，用到的骨牌数为$(4^k-1)/3$



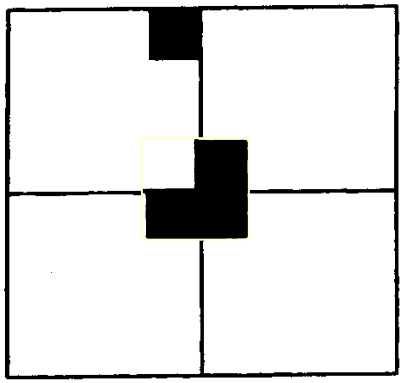


[算法思路] 当k>0时,将$2^k \times 2^k$棋盘分割为4个$2^{k-1} \times 2^{k-1}$子棋盘,残缺方格必位于4个子棋盘之一.其余3个子棋盘中无残缺方格. 为此将剩余3棋盘转化为残缺棋盘:

用一个L型骨牌覆盖这3个较小棋盘的结合处.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残缺方格,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.

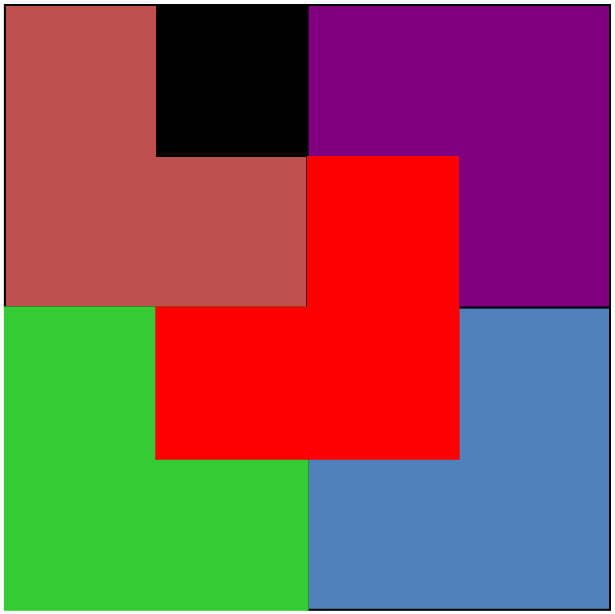


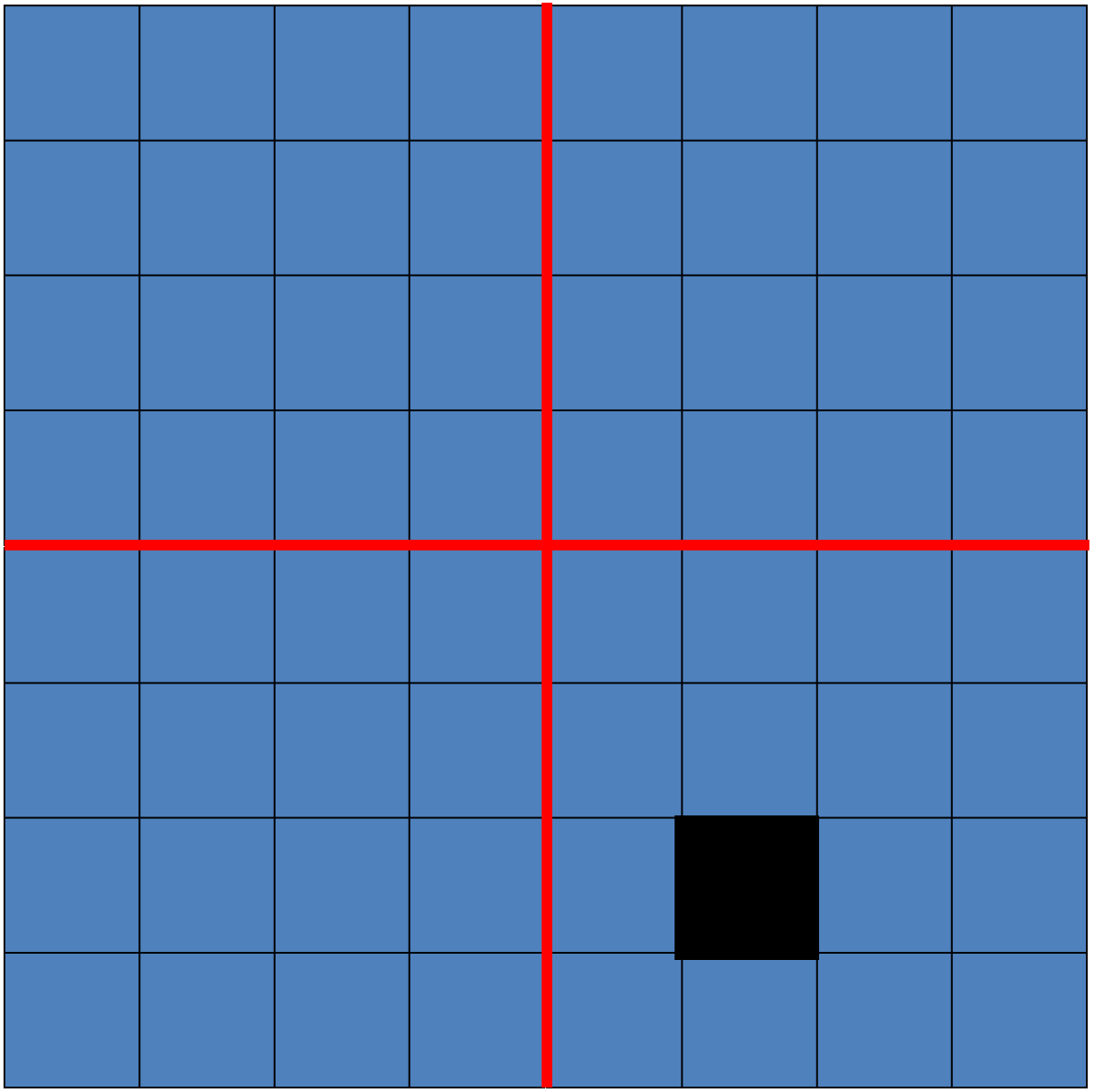
(a)

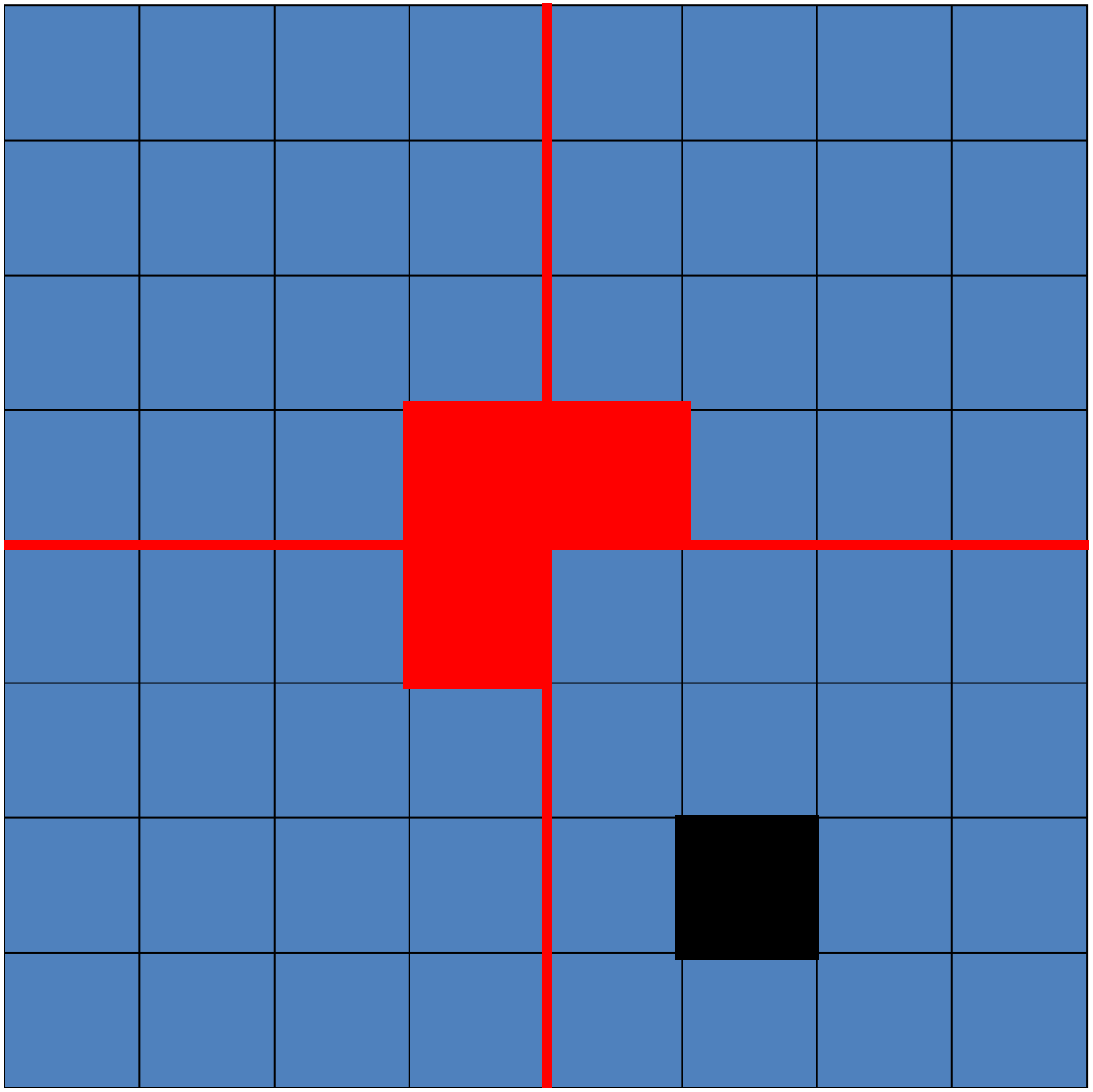


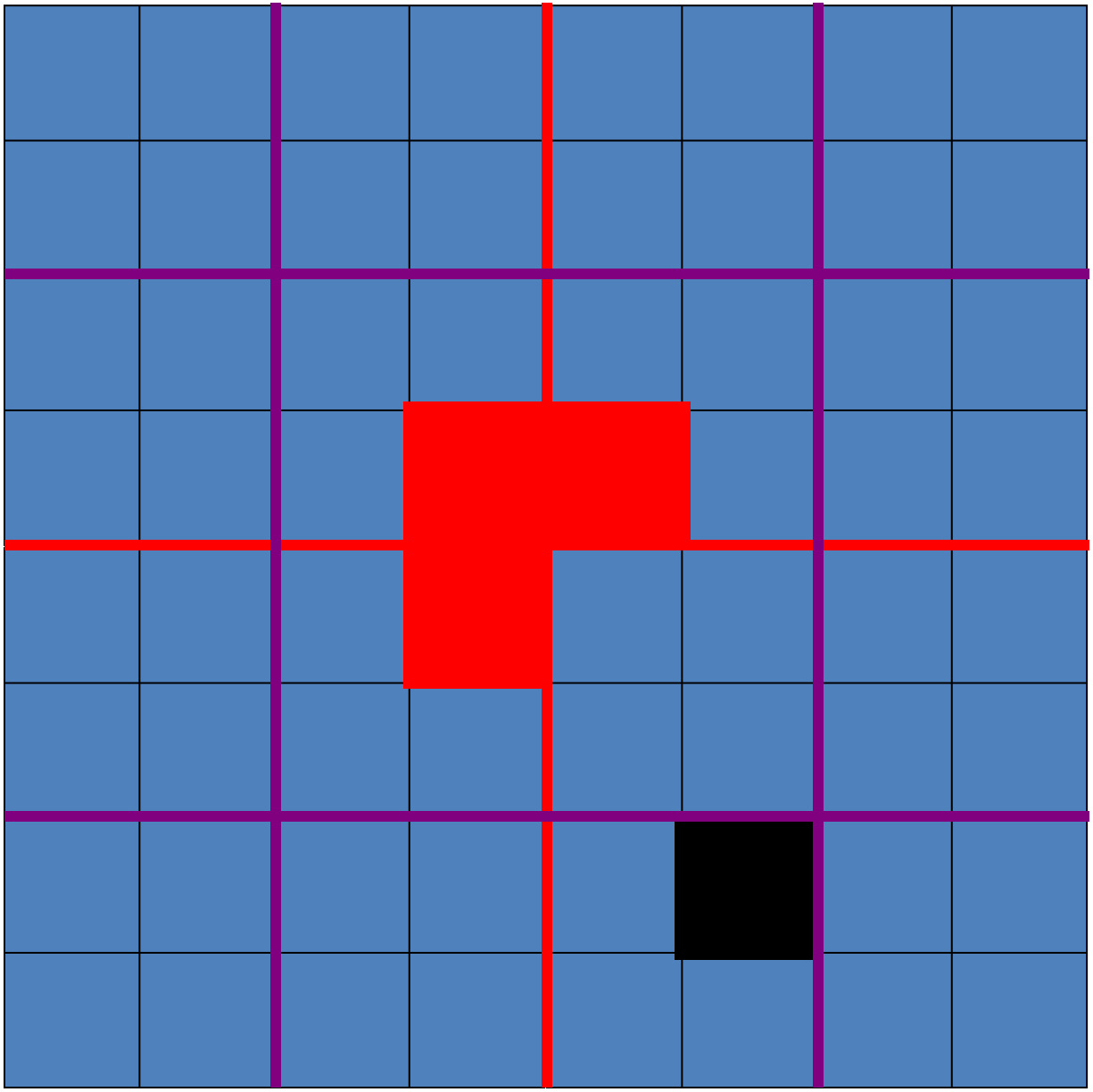
(b)

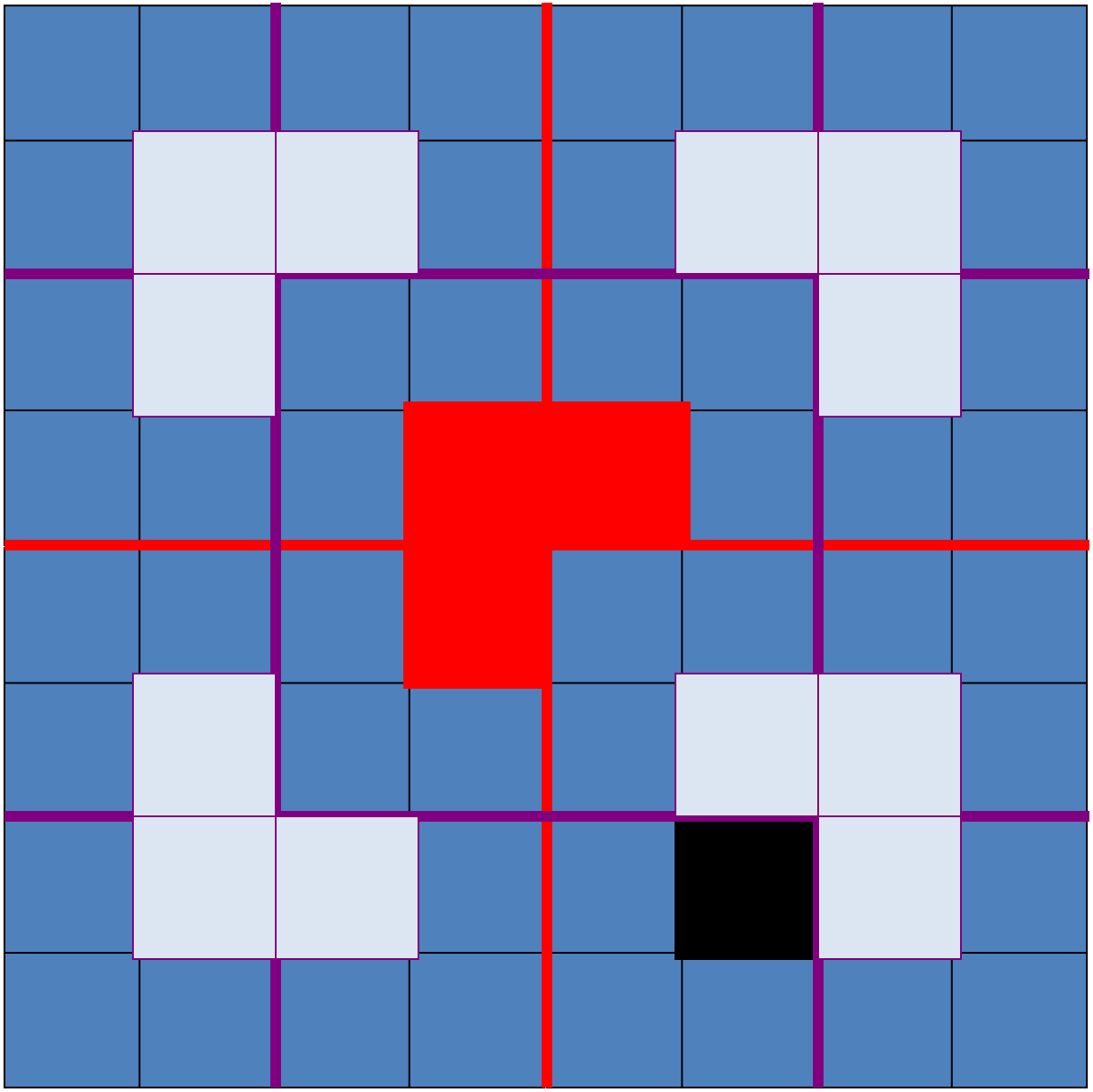
连续用这种方法直到把棋盘简化成2*2的棋盘。

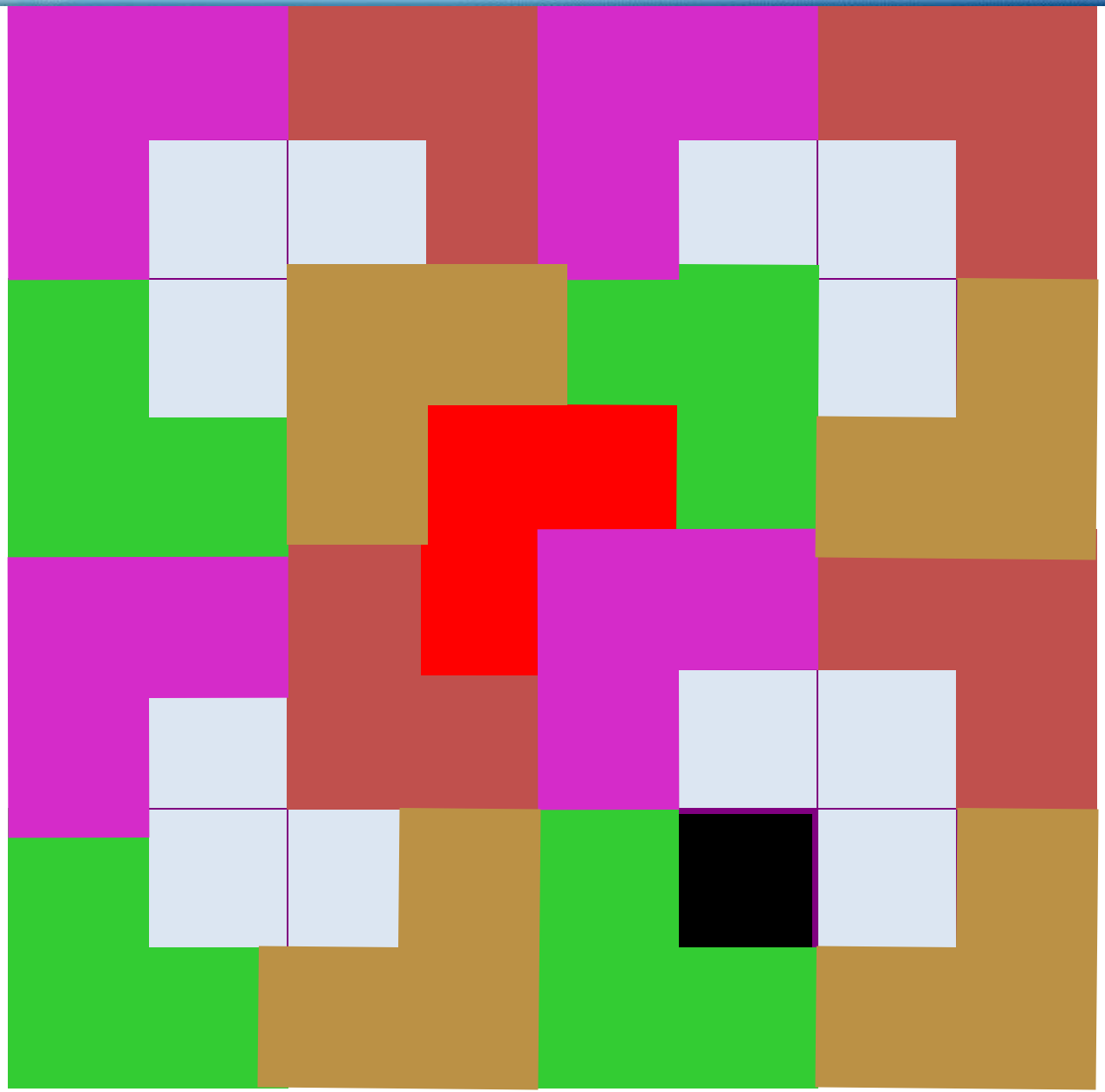












[算法分析] 设T(k)为覆盖$2^k \times 2^k$残缺棋盘的时间，当k=0时覆盖它需要常数时间O(1).

当k>0时,测试哪个子棋盘残缺以及形成3个残缺子棋盘需要O(1),覆盖4个残缺子棋盘需四次递归调用，共需时间4T(k-1) .

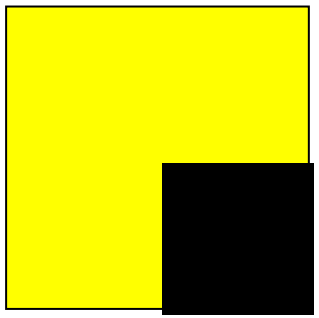
$$T(k)=\begin{cases} O(1) \\ 4T(k-1)+O(1) \end{cases}$$

与所需骨牌数同阶

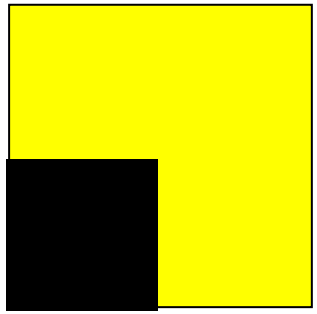
迭代法解得: $T(k)=\theta(4^k)$

```
void ChessBoard(int tr, int tc, int dr, int dc, int
size)
 { if (size==1) return;
    int t=tile++，// L型骨牌数
     s=size / 2;  // 分割棋盘
    //覆盖左上角子棋盘
     if (dr < tr +s && dc < tc +S)
          //特殊方格在此棋盘中
          ChessBoard(tr, tc, dr,dc,S);
     else{//此棋盘中无特殊方格
          //用t号L型骨牌覆盖右下角
          Board[tr+s-1][tc+s-1]=t;
          //覆盖其余方格
         Chessboard(tr，tc，tr+s-1，tc+s-l，s);}
```

tr:左上角方格行
tc:左上角方格列
dr:残缺方格行
dc:残缺方格列
size:棋盘行数
Board:为全局变量
二维数组表示棋盘



```
//覆盖右上角子棋盘
if (dr <= tr +s && dc >= tc +S)
   //特殊方格在此棋盘中
  ChessBoard (tr， tc +s ， dr， dc， S);
else{//此棋盘中无特殊方格
     //用t号骨牌覆盖左下角
     Board[tr+s-1][tc+s]=t;
     //覆盖其余方格
    Chessboard(tr， tc+s， tr+s-1， tc+s， s)； }
```



```
void outputBoard( int size)
 { for int i=0 ;i<size ;i++) {
     for int j=0 ;j<size ;j++);
       cout<<setw(5)<<board [i][j];
    cout<<endl ;}}
```

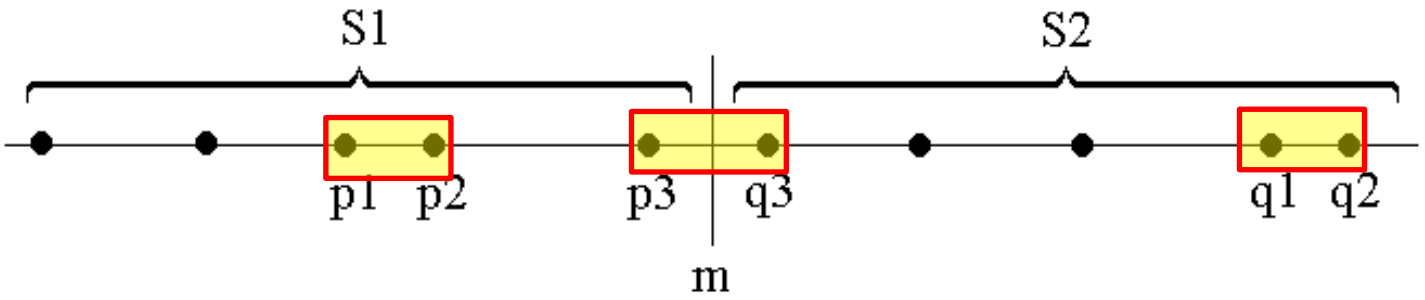
2.10 最接近点对问题

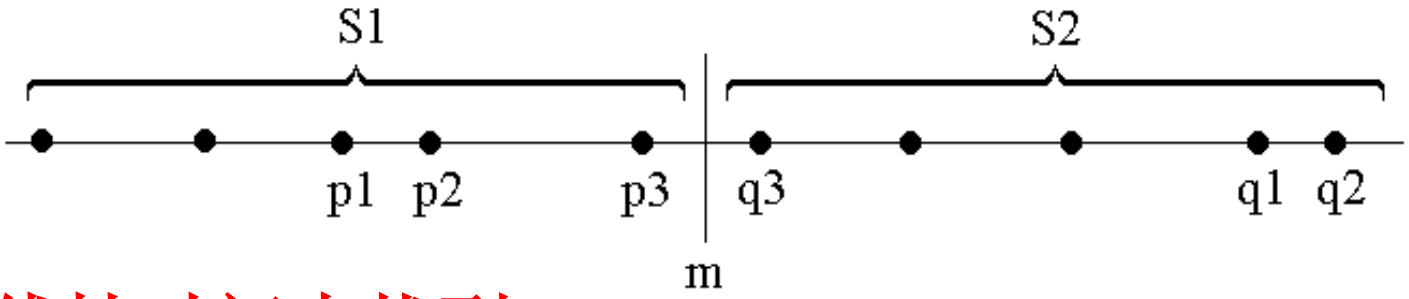
[问题描述]

给定平面S上n个点,找其中的一对点,使得在$n(n-1)/2$个点对中, 该点对的距离最小。

- 简化问题（一维问题）
  - 为了使问题易于理解和分析，先来考虑一维的情形。此时，S中的n个点退化为x轴上的n个实数 $x1,x2,...,xn$。最接近点对即为这n个实数中相差最小的2个实数。

- 假设我们用x轴上某个点m将S划分为2个子集S1和S2，基于**平衡子问题**的思想，用S中各点坐标的中位数来作分割点。
- 递归地在S1和S2上找出其最接近点对{p1,p2}和{q1,q2}，并设**d=min{|p1-p2|,|q1-q2|}**，S中的最接近点对或者是{p1,p2}，或者是{q1,q2}，或者是某个{p3,q3}，其中p3∈S1且q3∈S2。
- **能否在线性时间内找到p3,q3？**





能否在线性时间内找到p3,q3？

- 如果S的最接近点对是{p3,q3}，即|p3-q3|<d，则p3和q3两者与m的距离不超过d，即**p3∈(m-d,m]，q3∈(m,m+d]**。
- 由于在S1中，每个长度为d的半闭区间至多包含一个点（否则必有两点距离小于d），并且m是S1和S2的分割点，因此(m-d,m]中至多包含S中的一个点。由图可以看出，**如果(m-d,m]中有S中的点，则此点就是S1中最大点。**
- **同理,如果(m,m+d]中有S中的点，则此点就是S2中最小点。**
- 用线性时间就能找到区间(m-d,m]和(m,m+d]中所有点，即p3和q3。**用线性时间就可以将S1的解和S2的解合并成为S的解。**

```
Bool Cpairl(S,d)
{
    n=|S|;
    if (n<2) {d=MAX; return false;}
    m=S中各点坐标的中位数;

  Cpairl(S1,d1);      //找到S1中最短距离
  Cpairl(S2,d2);      //找到S2中最短距离
  p=max(S1);
  q=min(S2);          //找到S1和S2中相邻接点

  d=min(d1,d2,q-p); return t
}
```

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

得: $T(n)=O(n\log n)$

二维平面的情况：

[算法思路]

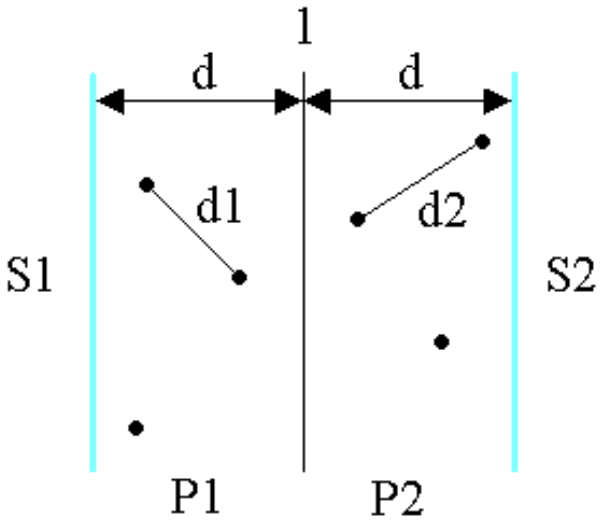
1) n较小时直接求 (n=2).

2) 将S上的n个点分成大致相等的2个子集S1和S2

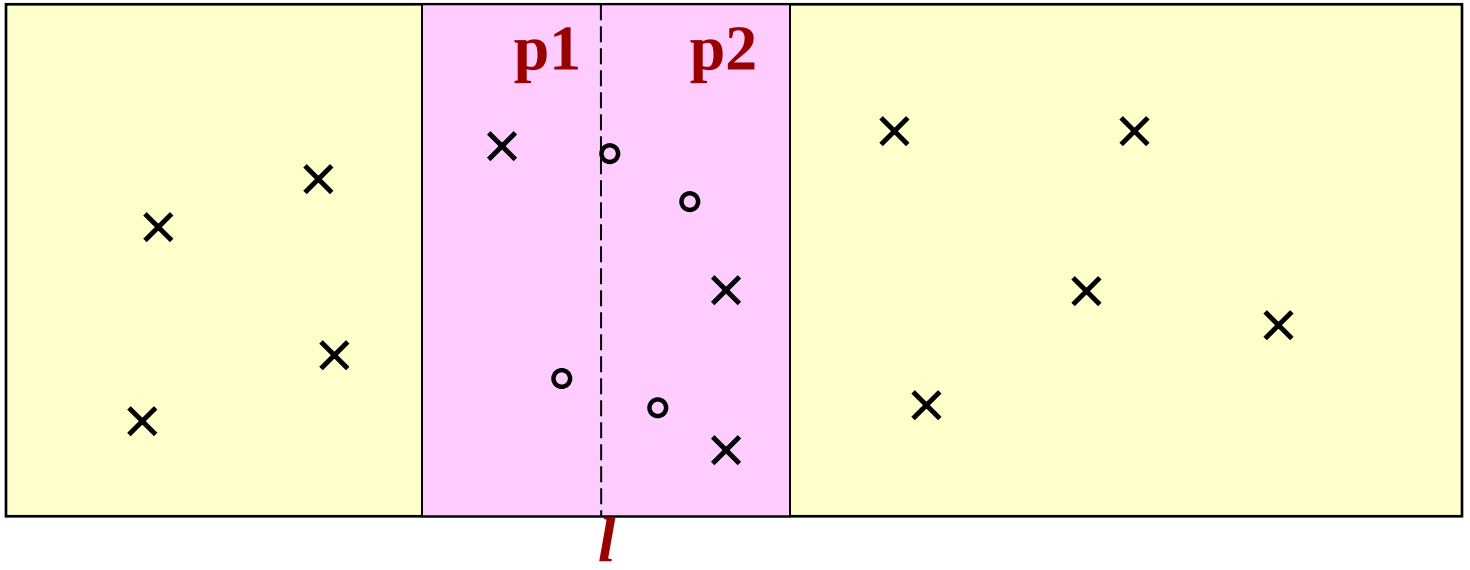
3) 分别求S1和S2中的最接近点对

4) 求一点在S1、另一点在S2中的最近点对

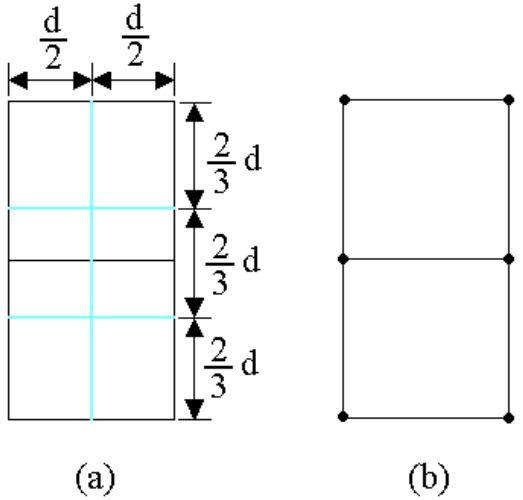
5) 从上述三对点中找距离最近的一对.

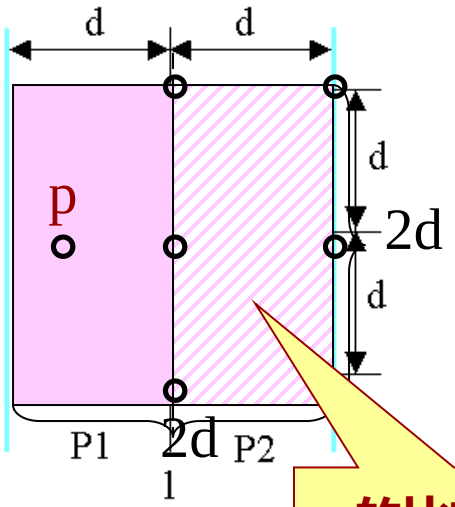


- 设 d1,d2分别为s1,s2中最近点对的距离, d=min{d1,d2},
- 设p1,p2分别为分割线*l*左右宽为d的垂直长条区域,若第三个最近点对<d 必在p1和p2中。
- 对∀p∈p1, 检查p2中 满足p.y-d<q.y<p.y+d 的点q, 判断是否q与p的距离小于d?



- 考虑P1中任意一点p，它若与P2中的点q构成最接近点对的候选者，则必有distance(p，q)＜d。**满足这个条件的P2中的点一定落在一个d×2d的矩形R中**
- 由d的意义可知，P2中任何2个S中的点的距离都不小于d。由此可以推出**矩形R中最多只有6个S中的点**。
- 因此，在分治法的合并步骤中**最多只需要检查6×n/2=3n个候选者**





p的比较区,最多含6个点

证明:将矩形R的长为2d的边3等分，将它的长为d的边2等分，由此导出6个(d/2)×(2d/3)的矩形。

若矩形R中有多于6个S中的点，则由鸽舍原理易知至少有一个(d/2)×(2d/3)的小矩形中有2个以上S中的点。设u，v是位于同一小矩形中的2个点，则

$$(x(u)-x(v))^2+(y(u)-y(v))^2 \le (d/2)^2+(2d/3)^2=\frac{25}{36}d^2$$

distance(u,v)≤5d/6<d。这与d的意义相矛盾。

- 为了确切地知道要检查哪6个点，可以将p和P2中所有S2的点投影到垂直线l上。
  - 由于能与p点一起构成最接近点对候选者的S2中点一定在矩形R中，所以它们在直线l上的投影点距p在l上投影点的距离小于d。由上面的分析可知，投影点最多只有6个。

- 将P1和P2中所有S中点按其y坐标排好序，则对P1中所有点，对排好序的点列作一次扫描可找出所有最接近点对的候选者。对P1中每一点最多只要检查P2中排好序的相继6个点。

```
public static double cpair2(S)
{
    n=|S|;
    if (n < 2) return ;
1. m=S中各点x轴坐标的中位数;
    构造S1和S2;
    // S1={p∈S|x(p)<=m},
    // S2={p∈S|x(p)>m}
2. d1=cpair2(S1);
   d2=cpair2(S
3. dm=min(d1,
```

4. 设P1是S1中距垂直分割线l的距离在dm之内的所有点组成的集合；

P2是S2中距分割线l的距离在dm之内所有点组成的集合；

将P1和P2中点依其y坐标值排序；

并设X和Y是相应的已排好序的点列；

5. 通过扫描X以及对于X中每个点检查Y中与其距离在dm之内的所有点(最多6个)可以完成合并；

当X中的扫描指针逐次向上移动时，Y中的扫描指针可在宽为2dm的区间内移动；

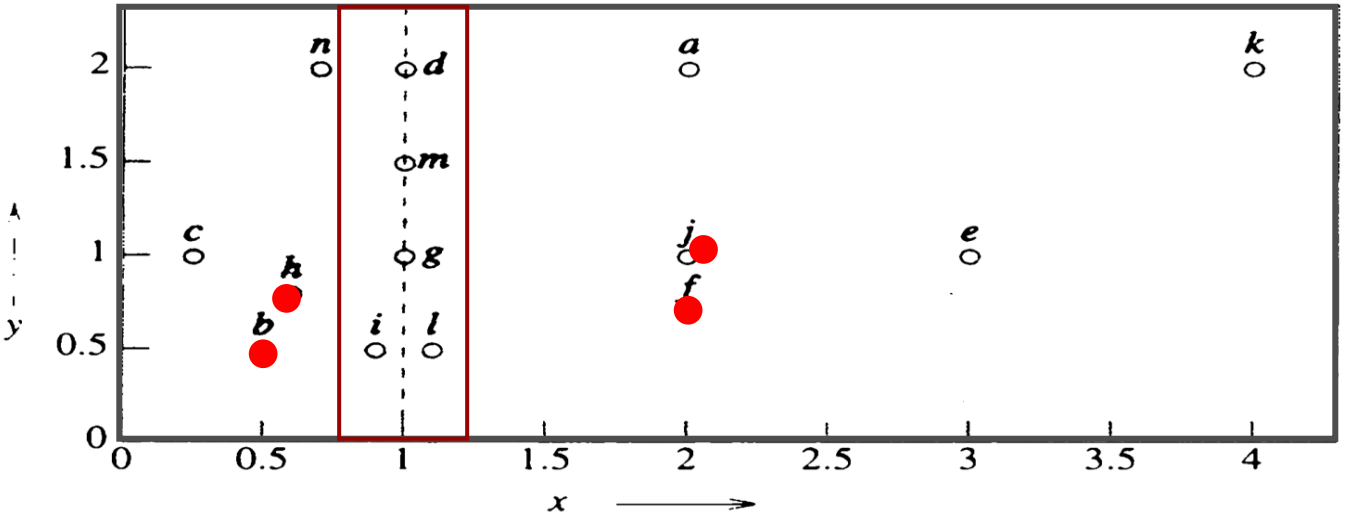
设d1是按这种扫描方式找到的点对间的

复杂度分析

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

T(n)=O(nlogn)

课堂练习



							g
x_i	2	0.5	0.25	1	3	2	1
y_i	2	0.5	1	2	1	0.7	1
点	*h*	*i*	*j*	*k*	*l*	*m*	*n*
x_i	0.6	0.9	2	4	1.1	1	0.7
y_i	0.8	0.5	1	2	0.5	1.5	2

图中14个点(d,m ∈s1,g∈s2),s1中的最近点对为 (b,h), 其距离为0.316. s2中最近点对(f, j), 其距离为0.3, 故d=0.3.考察是否存在第三类点，p1={d，i，m}，p2={g，l}，i的比较区中仅含点*l*, i和*l* 的距离小于d， 因此(i,l)是最近的点对。

2.11 循环赛日程表

[问题描述] **设有$n=2^k$个运动员要进行网球循环赛。现要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。

例如 4个选手的比赛的一个日程表

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

例如 8个选手的比赛的一个日程表

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

[算法思路]

1. **分解:** 将所有的选手分为两组,各n/2人，分别安排各组的日程表. 如果每组人数大于2,可递归地对选手进行分割，直到每组剩下2个选手时。

2 **求子问题的解:**直接安排这2个选手的比赛表。

3.**合并子问题成原问题:**自底向上,逐级将两组赛程对调，得到各组与另一组选手的赛程.

课后练习

P36 2-2,2-3,2-7

交作业时间:10.09

实验

杭电子网站：http://acm.hdu.edu.cn

选择竞赛 ECUST_Algorithm_2019_1

密码：123456