Analysis and Design of Algorithms

Chapter 6: Divide and Conquer

School of Software Engineering © Yaping Zhu

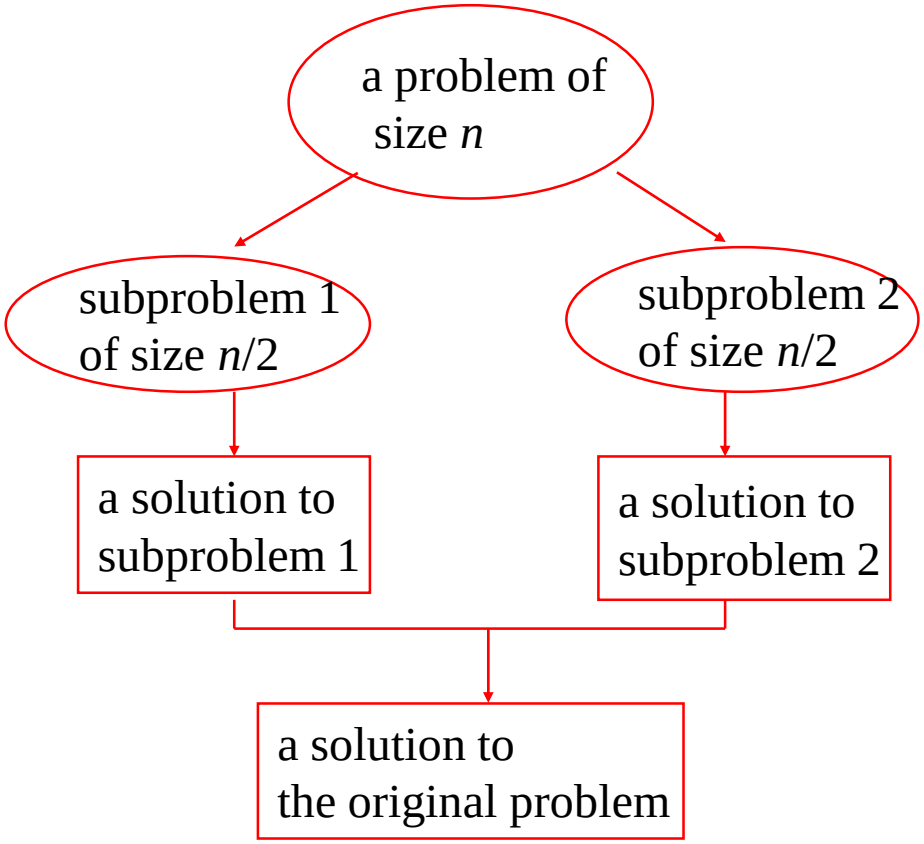
Decrease and Conquer

- *回顾*

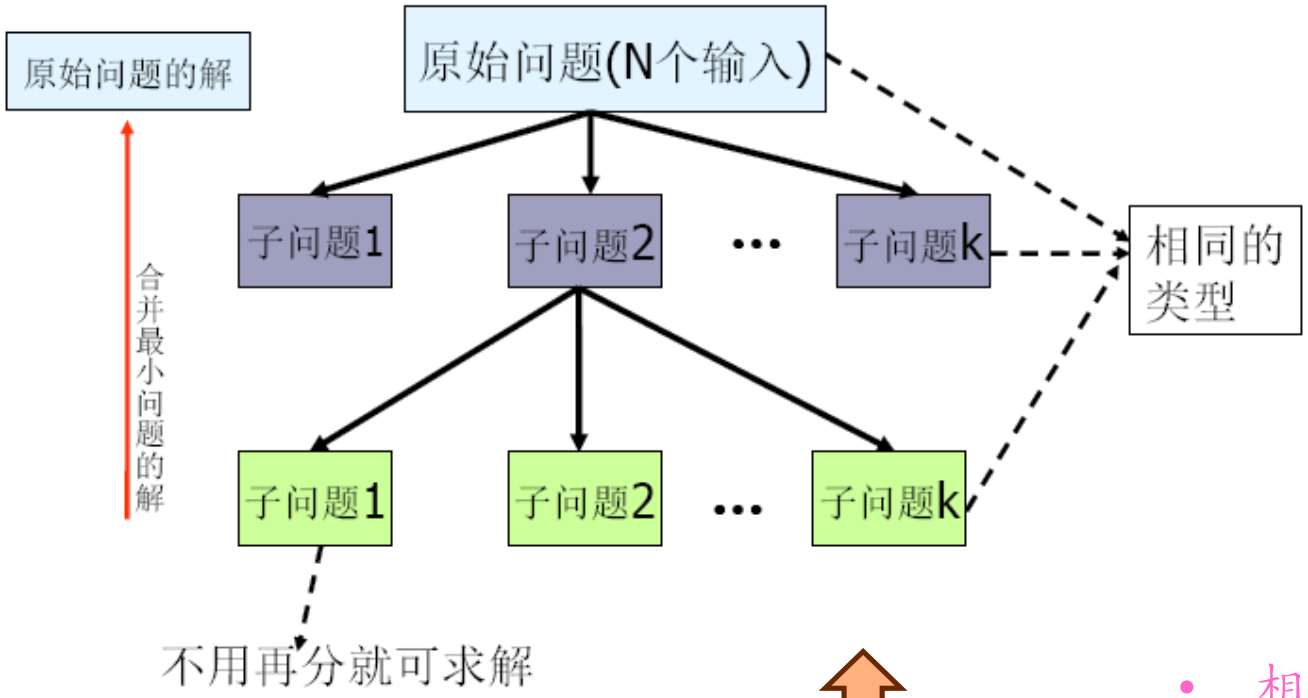
For comparison: Divide and Conquer (later)

Computing a^n

$$a^n = \begin{cases} a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil} & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases}$$



Divide and Conquer Approach



- 相同类型
- 独立
- 规模均衡

Divide and Conquer Approach

战国时期纵横之术就使得秦国通过**合纵连横**的分治策略而统一了当时的六国（具体措施：**笼络燕齐，稳住魏楚，消灭韩赵；远交近攻，逐个击破。**）秦国，就是在与六国连横的过程中，一方面击破了合纵，另一方面不断深挖本国的潜能，使国家不断的富强，最终取得了统一霸业的伟大胜利。

各个击破
分而治之

Divide and Conquer Approach

- ***Three Steps of the Divide and Conquer Approach***

  - *Divide the problem into two or more smaller subproblems;*
  - *Conquer the subproblems by solving them recursively;*
  - *Combine the solutions to the subproblems into the solutions to the original problem.*

Mergesort

Idea of Mergesort (合并排序)

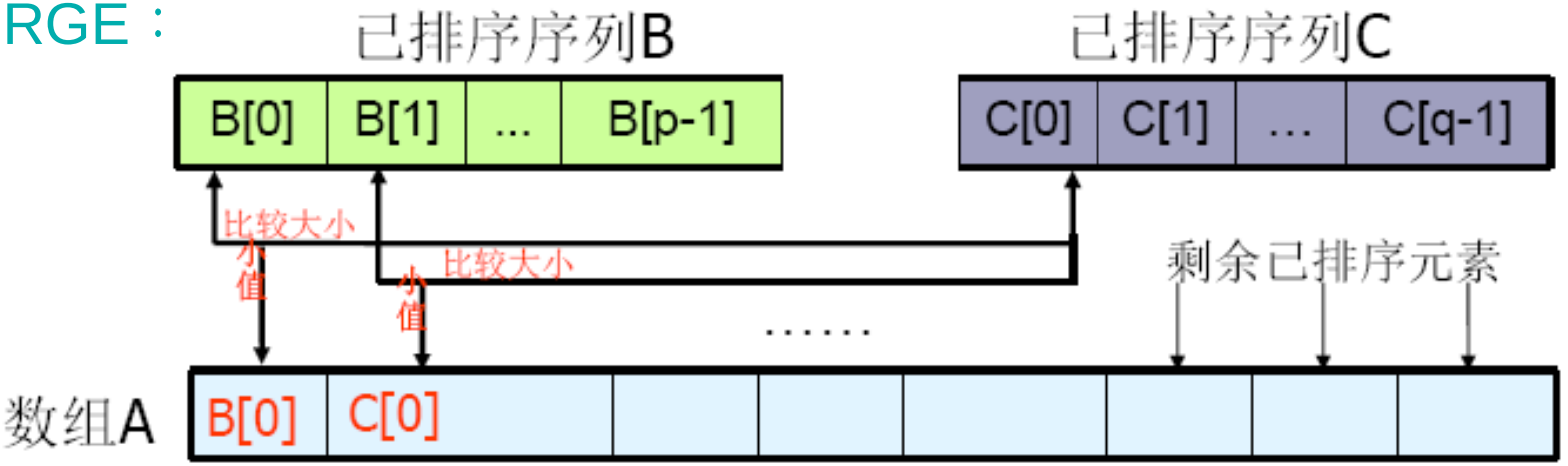
- *Step 1 Divide:* *divide array A[0..n-1] in two about equal halves and make copies of each half in arrays B and C*
- *Step 2 Conquer:*
  - *If number of elements in B and C is 1, directly solve it (go to step 3)*
  - *Sort arrays B and C recursively (go to step 1)*
- *Step 3 Combine:* *Merge sorted arrays B and C into a single sorted A*

Mergesort

Idea of Mergesort (合并排序)

- *Repeat the following until no elements remain in one of the arrays:*
  - *compare the first elements in the remaining unprocessed portions of the arrays B and C*
  - *copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array*
- *Once all elements in one of the arrays are processed, the remaining unprocessed elements from the other array are copied into the end of A.*

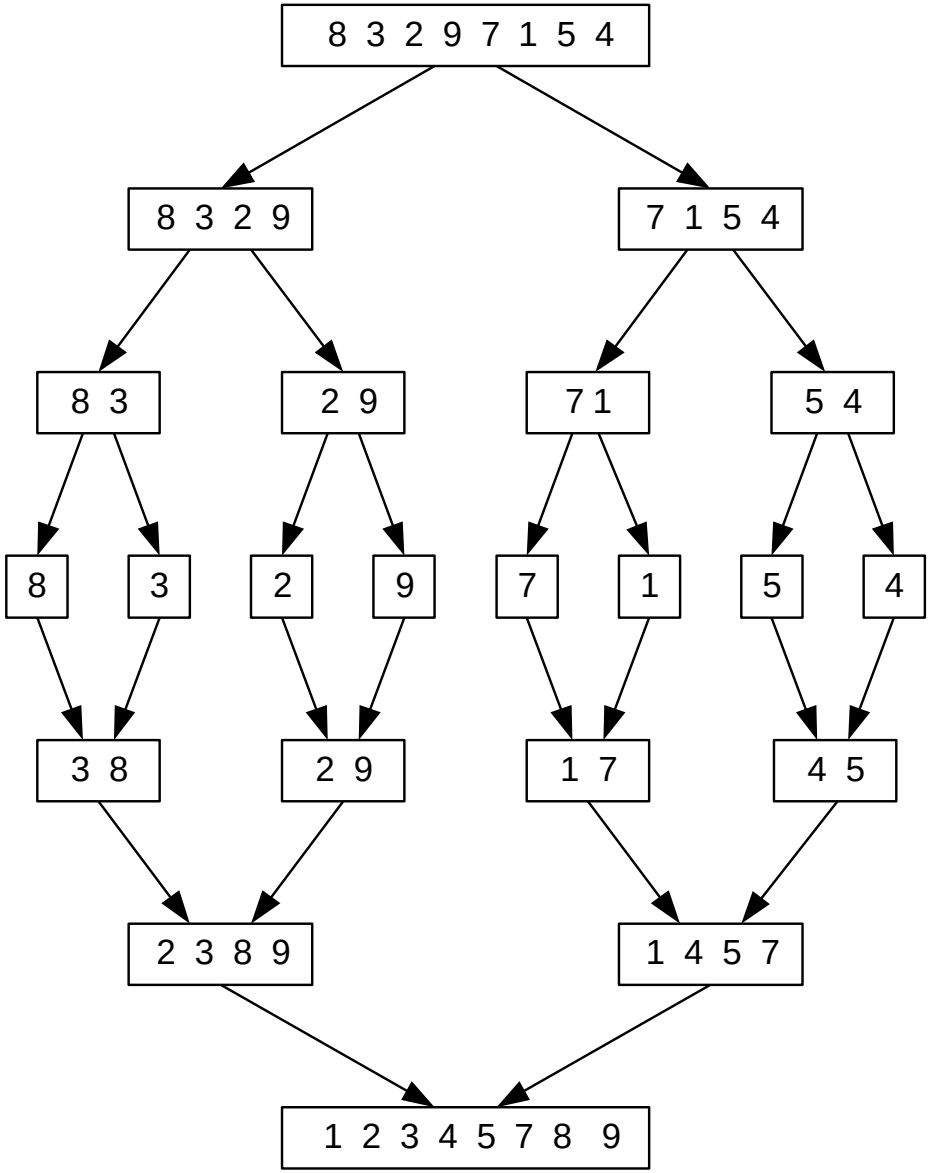
MERGE：



Mergesort

- ***Example:***
  - 8 3 2 9 7 1 5 4

在合并排序算法中，将问题划分成两个子问题是很快的，算法的主要工作在于合并子问题的解。



Mergesort

The Mergesort Algorithm

ALGORITHM *Mergesort*$(A[0..n-1])$

```
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌊n/2⌋ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)
```

中分点 mid←⌊(low+high)/2⌋

Mergesort

The Mergesort Algorithm ('cont)

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

Mergesort

Analysis of Mergesort

- *Input size: n*
- *Basic operation: key comparisons*
- *假设 n 是 2 的乘方，键值比较次数的 C(n)的递推关系式为*

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n>1$$
$$C(1) = 0$$

- *最坏情况下，* $C_{merge}(n) = n-1$
- *因此，得到最坏情况下的递推关系式：*

$$C(n) = 2C(n/2) + n-1 \quad \text{for } n>1$$
$$C(1) = 0$$

- *求解得到：* $\boldsymbol{C_{worst}(n) = \Theta(n \log n)}$

Divide and Conquer Approach

- ***Algorithm analysis—general divide-and-conquer recurrence（通用递推关系式）***

- *An instance of size **n** can be divided into **a** instances of size **n/b** (**assuming n is a power of b**), where, **a** is the number of the problems need to be solved;*
- *We have the following recurrence for the running time T(n):*

$$T(n) = aT(n/b) + f(n)$$

注意，这里只是递推关系，还未标明初始状态。

*Where, **f(n)** is a function that accounts for the time spent on dividing the problem into smaller ones and on combing their solutions.*

Mergesort

Master Theorem (主定理)

- *general divide-and-conquer recurrence*

$$T(n) = aT(n/b) + f(n)$$

- *得到通用递推关系式后，可利用主定理直接求增长次数*（仅仅得到增长次数，乘法常量未知；利用初始条件，解递推关系可得精确解）。

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in the recurrence, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the O and Ω notations, too.

Mergesort

$$T(n) = aT(n/b) + f(n)$$

- *主定理应用举例：*
  - **Mergesort** *的递推关系式：* $C(n) = 2C(n/2) + n-1$ for $n>1$
  - *利用主定理：*

$f(n)=n-1 \in \Theta(n)$, hence, $d=1$; since $a=2$, $b=2$, hence, $a=b^d$

$C(n) \in \Theta(n^d \log n) = \Theta(n \log n)$

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in the recurrence, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the O and Ω notations, too.

Binary Search

Idea of Binary Search

问题描述：

已知一个按非降次序排列的元素表A[n]=[a_0,a_1,…,a_{n-1}]，判定某个给定元素K是否在该表中出现。若是，则找出该元素在表中的位置并返回其所在位置的下标j；否则，返回值-1。

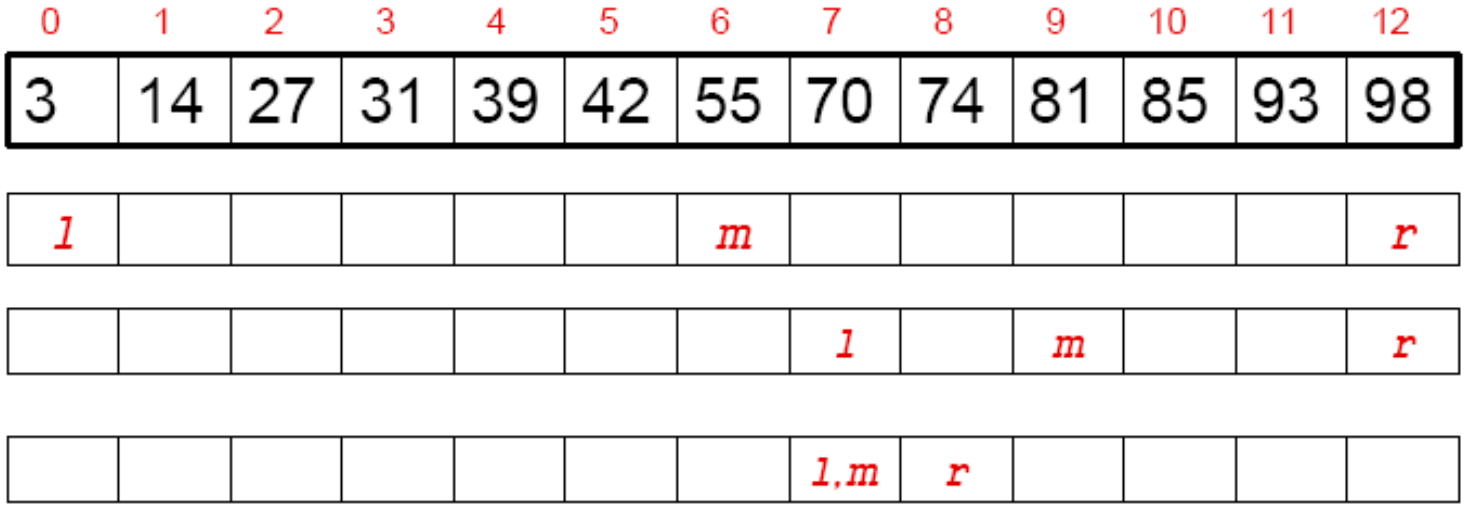
- Compare a search key K with the array's middle element $A[m]$
- If K = A[m], stop (successful search);
- otherwise, continue searching by the same method

  if K < A[m] in A[0..m-1]
  if K > A[m] in A[m+1..n-1]

Binary Search

- *Example*

  - K=70



Binary Search

- ***Example***

If A(1:9) = (-15，-6，0，7，9，23，54，82，101)

search in A: *k* = 101，-14，82。

Searching process:

K = 101			K = -14			K = 82		
low	high	mid	low	high	mid	low	high	mid
1	9	5	1	9	5	1	9	5
6	9	7	1	4	2	6	9	7
8	9	8	1	1	1	8	9	8
9	9	9	2	1				
		找到			找不到			找到
successful search			unsuccessful search			successful search		

Binary Search

Binary Search – a Recursive Algorithm

ALGORITHM BinarySearchRecur($A[0..n-1]$, l, r, K)
if $l > r$
    return –1
else
    $m \leftarrow \lfloor (l + r) / 2 \rfloor$
    if $K = A[m]$
        return m
    else if $K < A[m]$
        return BinarySearchRecur($A[0..n-1]$, l, $m-1$, K)
    else
        return BinarySearchRecur($A[0..n-1]$, $m+1$, r, K)

Basic operation:
while循环中k与A 中元素的比较运算
three-way comparison

Binary Search

Analysis of Binary Search

- Basic operation: *key comparison* (*three-way comparison*)
- *Worst-case* *(successful or fail)* :

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1,$$
$$C_{worst}(1) = 1$$

- Solution:

$$C_{worst}(n) = \Theta(\log n)$$

利用主定理求解？

for $n = 2^k$,

$C(2^k) = C(2^{k-1}) + 1$ for $k>0$

$C(2^0) = 1$

backward substitutions:

$C(2^k) = C(2^{k-1}) + 1$

$= [C(2^{k-2}) + 1] + 1$

$= C(2^{k-2}) + 2 = \ldots = C(2^{k-i}) + i \ \ldots$

$= C(2^{k-k}) + k = 1 + k$

then, $C(n) = \log_2 n + 1$

Quicksort

Idea of Quicksort

- *Divide:* *Partition array A[l..r] into 2 subarrays, A[l..s-1] and A[s+1..r] such that each element of the first array is ≤A[s] and each element of the second array is ≥ A[s]. (computing the index of s is part of partition.)*
- *Implication: A[s] will be in its final position in the sorted array.*
- *Conquer:* *Sort the two subarrays A[l..s-1] and A[s+1..r] by recursive calls to quicksort.*
- *Combine:* *No work is needed, because A[s] is already in its correct place after the partition is done, and the two subarrays have been sorted.*

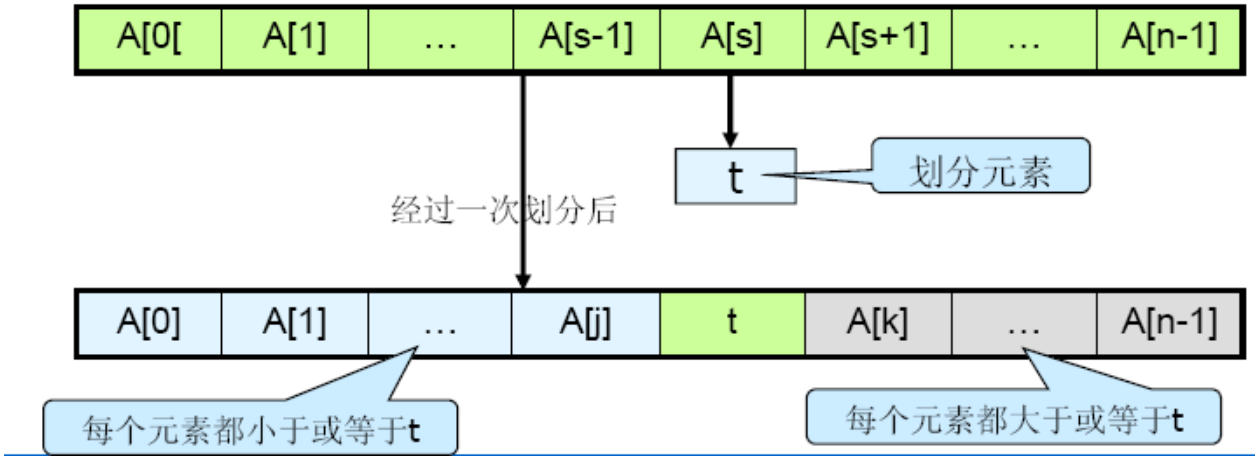
Quicksort

Idea of Quicksort

1. *Select a pivot whose value we are going to divide the list.* (Strategy for selecting pivot: 1. Randomly selected；2. Simplest Strategy: selecting the array's first element A[*l*])

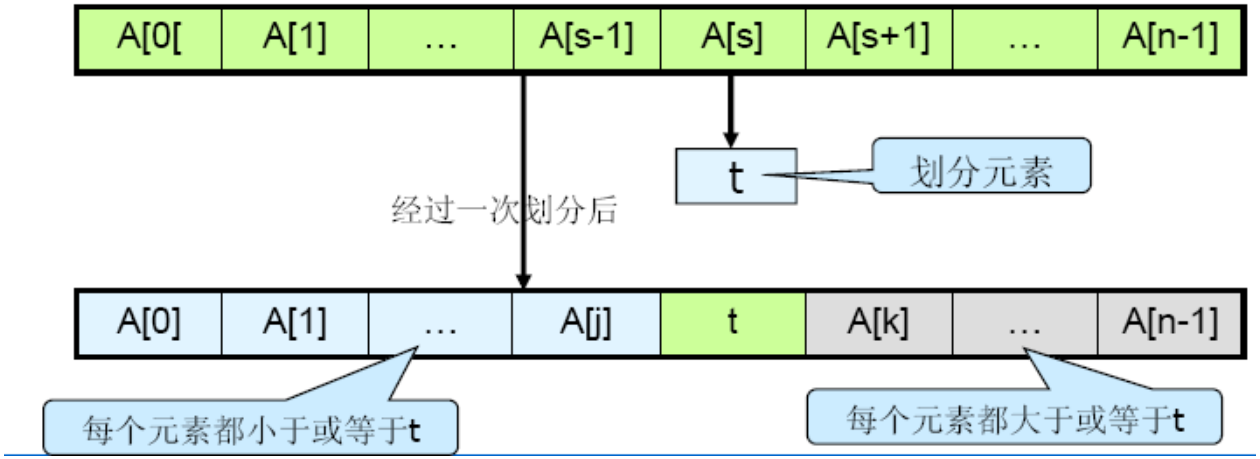
2. *Rearrange the list so that*

- *all elements in the first s positions are smaller than or equal to* ***p;***
- *all elements in the remaining n-s positions are larger than or equal to* ***p.***



Quicksort

Idea of Quicksort



3. *After exchanging* ***p*** *with the last element in the first sublist (i.e.,* ≤ *sublist) , the pivot* ***p*** *is now in its final position.*

4. *Sort the two sublists recursively using quicksort (i.e.* ***go to step 1****).*

Quicksort

- ***The Quicksort Algorithm --Pseudo Code of the Algorithm***

ALGORITHM *Quicksort*$(A[l..r])$

```
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right
//         indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l < r
    s ←Partition(A[l..r])  //s is a split position
    Quicksort(A[l..s − 1])
    Quicksort(A[s + 1..r])
```

Quicksort

Idea of Quicksort

- 快速排序是一种基于**划分（按照元素的值）**的排序方法。（合并排序是按照元素的位置进行划分。）
- 通过反复地对待排序集合进行划分达到分类目的的排序算法。
- $A[l..s\text{-}1]$ 中所有元素小于等于 $A[s\text{+}1..r]$ 中任何元素，所以这两个集合可独立进行划分。

$$\underbrace{A[0]\ldots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\ldots A[n-1]}_{\text{all are} \ge A[s]}$$

这两拨只是在数值上都分别小于（或大于）A[s]，但内部元素的顺序尚未确定，只有A[s]的位置是已经可以确定的。

Quicksort

Idea of Quicksort ('cont)

Procedure for rearranging elements in a partition

------ based on two-scans of the subarray

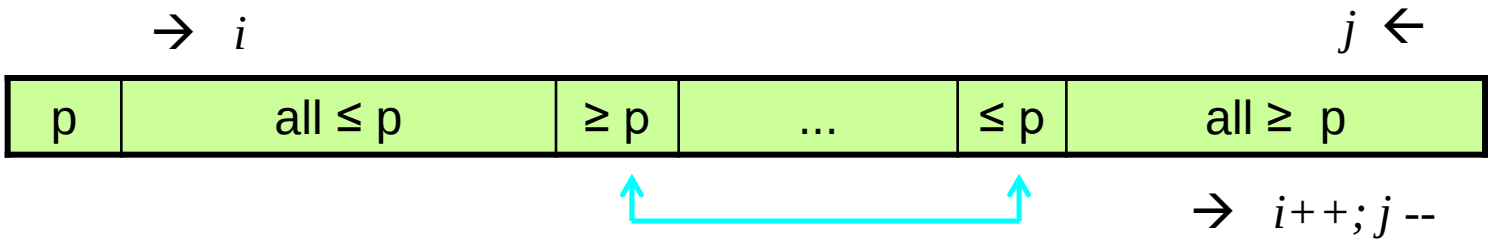
1. *Left-to-right scan: index i, starts with the second element*（选取第一个数为pivot）,
   - *Wants elements smaller than the pivot to be in the first part*
   - *Skip over elements that are smaller than the pivot*
   - *Stop on encountering the first element greater than or equal to pivot*

2. *Right-to-left scan: index j, starts with the last element,*
   - *Wants elements larger than the pivot to be in the second part of the subarray*
   - *Skip over elements that are larger than the pivot*
   - *Stop on encountering the first element smaller than or equal to pivot*

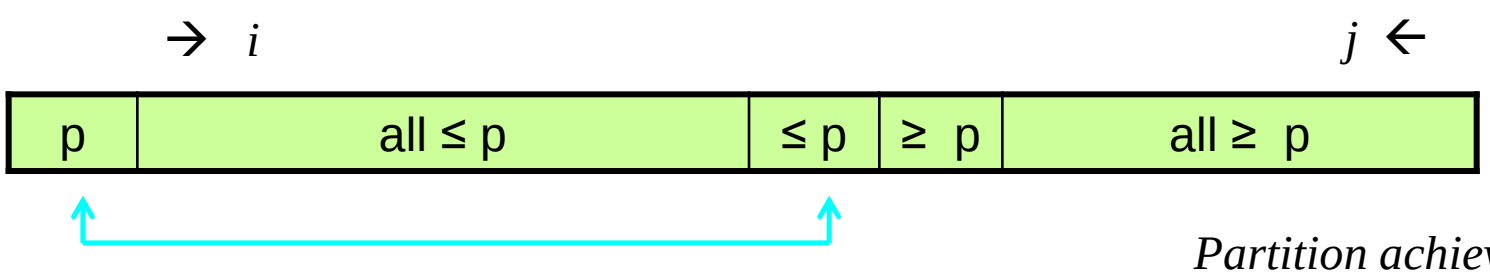
Quicksort

- *three cases for scan stopping*

1. *i, j not crossed, i < j*

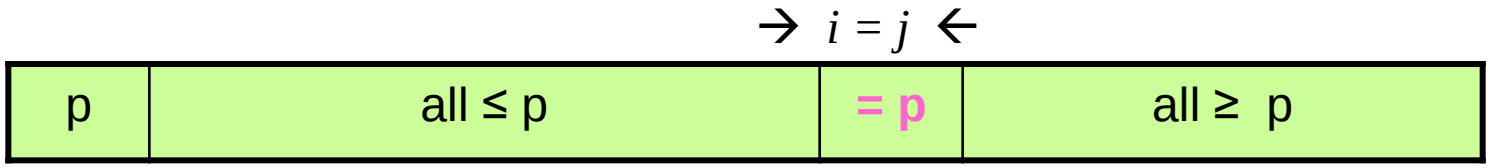


2. *i, j crossed, i > j*



Partition achieved

3. *pointing to the same, i = j*



Partition achieved, s = i = j

Quicksort

- *Example* A : **65** 70 75 80 85 60 55 50 45

Quicksort

The Quicksort Algorithm --Pseudo for partitioning procedure

ALGORITHM *HoarePartition*$(A[l..r])$

```
//Partitions a subarray by Hoare's algorithm, using the first element
//          as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and right
//          indices l and r (l < r)
//Output: Partition of A[l..r], with the split position returned as
//          this function's value
p ← A[l]
i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])    //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

Quicksort

- ***Analysis of Quicksort***
  - Basic operation: *key comparison*
  - *Based on whether the partitioning is balanced.*

Quicksort

Analysis of Quicksort

Number of comparisons for a partition:

$n + 1$ (指针交叉)

n (指针相等)

- **Best case**: *split in the middle* （所有分裂点位于子数组的中点）

最优情况下，键值比较次数的递推关系式：

$C_{best}(n) = 2C_{best}(n/2) + n$ //2 subproblems of size n/2 each

$C_{best}(1) = 0$

利用主定理，可得：$C_{best}(n) \in \Theta(n \log n)$

（也可以对 $n = 2^k$ 使用 backward substitutions, 再利用平滑法则得到。）

Quicksort

Analysis of Quicksort

- ***Worst case**: sorted array!* （所有分裂点趋于极端：两个子数组一个为空，另一个仅仅比被划分的数组少一个元素）

n+1 Comparisons

For example: A [0…*n*-1] is a strictly increasing array, and A [0] is used as pivot, the left-to-right scan stops on A [1], right-to-left scan goes all the way to A [0].

最坏情况下，键值比较的次数：

$$C_w = (n+1) + n + \ldots + 3 = (n+1)(n+2)/2 - 3 \in \Theta(n^2)$$

Quicksort

Analysis of Quicksort

- ***Average case:*** *random array*（分裂点出现在任意位置，左右数组大小分别为s和n-1-s，分裂点位于每个位置的概率为1/n：

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

平均情况下，键值比较的次数：

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log n \qquad \text{— } O(n \log n)$$

快速排序在平均情况下，比较操作只比最优情况多39% 。而且，其最内层循环效率高，在处理随机数组时速度比合并排序快。

Quicksort

Improvements

- 快速排序算法的性能取决于划分的对称性。通过修改 partition 算法，可以设计出采用随机选择策略的快速排序算法。
- 随机选取划分元素

在快速排序算法的每一步中，当数组还没有被划分时，可以在$a[p:r]$中随机选出一个元素作为划分基准，这样可以使划分基准的选择是随机的，从而可以期望划分是较对称的。

```
template<class Type>
int RandomizedPartition (Type a[], int p, int r)
{
        int i = Random(p,r);
        Swap(a[i], a[p]);
        return Partition (a, p, r);
}
```

Quicksort

Improvements

- 当子数组足够小时，改用插入排序方法，或者根本就不再对小数组进行排序，而是在快速排序结束后再使用插入排序的方法对整个近似有序的数组进行排序；
- 一些划分方法的改进：例如三路划分，将数组分成三段，每段的元素分别小于、等于、大于中轴元素（书上习题第9题）

Multiplication of Large Integers（大整数乘法）

- ***Idea of Multiplication of Large Integers :*** karatsuba

  Consider the problem of multiplying two (large) *n*-digit integers represented by arrays of their digits such as:

  A = 12345678901357986429; B = 87654321284820912836

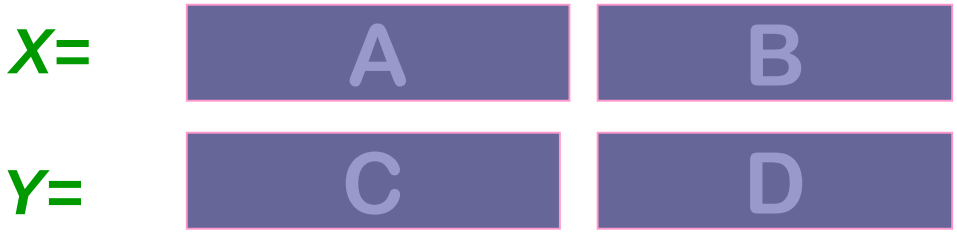
  - *brute-force algorithm*

$$
\begin{array}{r}
a_1\ a_2 \ldots\ a_n \\
\times\ b_1\ b_2 \ldots\ b_n \\
(d_{10})\, d_{11} d_{12} \ldots d_{1n} \\
(d_{20})\, d_{21} d_{22} \ldots d_{2n} \\
\ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots \\
(d_{n0})\, d_{n1} d_{n2} \ldots d_{nn}
\end{array}
$$

    - Efficiency: n^2 one-digit multiplications

Multiplication of Large Integers

- ***First Divide-and-Conquer Algorithm***



- if $X = A\,10^{n/2} + B$, and $Y = C\,10^{n/2} + D$ -- *divide X,Y into two parts*
  where X and Y are n-digit, A, B, C, D are $n/2$-digit numbers

$$X * Y = A * C \cdot 10^{n} + (A * D + B * C) \cdot 10^{n/2} + B * D$$

Idea of the algorithm:

- If $n = 2^k$, recurrence;
- Otherwise, stop, when $n = 1$ or n is small enough to multiply the numbers of that size directly.

Multiplication of Large Integers

First Divide-and-Conquer Algorithm

- *Analysis:*
  - Basic operation: one-digit multiplication

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

Solution: ***T(n) = O(n²)*** ×**NO Promotion**

① If $n = 2^k$, then

$C(2^k) = 4C(2^{k-1}) = 4[4C(2^{k-2})] = 4^2C(2^{k-2})$

$= \ldots = 4^kC(2^{k-k}) = 4^k$

Given $\mathbf{n = 2^k}$, we have $\mathbf{k = \log_2 n;}$

Then, $C(2^k) = 4^{\log n} = 2^{2\log n} = n^2 = C(n)$

② If $n <$ or $> 2^k$, by Smoothness Rule, $C(n) \in O(n^2)$

Multiplication of Large Integers

Second Divide-and-Conquer Algorithm

$$X * Y = A * C \cdot 10^{n} + (A * D + B * C) \cdot 10^{n/2} + B * D$$

- The idea is to decrease the number of multiplications from 4 to 3:

Since we have: $(A + B) * (C + D) = A * C + (A * D + B * C) + B * D$

i.e., $(A * D + B * C) = (A + B) * (C + D) - A * C - B * D$

$$X * Y = A * C \cdot 10^{n} + [(A + B) * (C + D) - A * C - B * D] \cdot 10^{n/2} + B * D$$

which requires only 3 multiplications at the expense of (4-1) extra addition/subtraction.

Multiplication of Large Integers

Second Divide-and-Conquer Algorithm

- The idea is to decrease the number of multiplications from 4 to 3:

$$(A + B) * (C + D) = AC + (AD + BC) + BD$$

$$\text{i.e., } (AD + BC) = (A + B) * (C + D) - A*C - B*D$$

which requires only 3 multiplications at the expense of (4-1) extra add/sub.

- *Analysis:*

n位数的乘法需要对 n/2 位数做3次乘法运算，因此，乘法次数的递推关系式为：

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1$$

Multiplication of Large Integers

Second Divide-and-Conquer Algorithm

- *Analysis:*

  n位数的乘法需要对n/2位数做3次乘法运算，因此，乘法次数的递推关系式为：

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1$$

使用反向替换法:

If $n = 2^k$, then

$M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2M(2^{k-2})$

$= \ldots = 3^kC(2^{k-k}) = 3^k$

$K = \log_2 n$ $\quad M(2^k) = M(n) = 3^{\log n} = n^{\log 3} \approx n^{1.585}$

Solution:

$$M(n) = O(3^{\log 2^n}) = O(n^{\log 2^3}) \approx O(n^{1.585})$$ ✓Promotion

Strassen's Matrix Multiplication

Idea

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] * B[k, j]
return C
```

- *Brute-force* :

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

*For $C_{ij,}$, **n** multiplications and **n-1** summations*

So for n^2 elements in C, $T(n)= O(n^3)$

Strassen's Matrix Multiplication

Idea

- *Divide and Conquer— idea1*

  Divide A, B, and C into 4 equal-size sub-matrix,

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

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

- *Number of multiplications:*

  M(n) = 8M(n/2),

  ***M*(1) = 1**

  M(n) = O(n³)

Strassen's Matrix Multiplication

Idea

- *Divide and Conquer — idea2* *to reduce the times for multiply* (Strassen)

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

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

其中，

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

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

Strassen's Matrix Multiplication

Analysis of Strassen's Matrix Multiplication

- *Number of multiplications:*

  $M(n) = 7M(n/2)$,
  $M(1) = 1$

  If $n = 2^k$, then

  $M(n) = 7M(n/2) = 7^2M(n/2^2) \ldots\ldots$
  $= 7^kM(1) = 7^k$

  - Solution:

$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$ **vs.** **n^3 of brute-force alg.**

- If n is not a power of 2, matrices can be padded with zeros.

☆ Practical implementation of Strassen's alg. usually switch to brute-force method after matrix sizes become smaller than some "crossover point" .

Strassen's Matrix Multiplication

- *Standard vs Strassen: Practical:*

	N	Multiplications	Additions
Standard alg.	100	1,000,000	990,000
Strassen's alg.	100	411,822	2,470,334
Standard alg.	1000	1,000,000,000	999,000,000
Strassen's alg.	1000	264,280,285	1,579,681,709
Standard alg.	10,000	10^{12}	$9.99*10^{11}$
Strassen's alg.	10,000	$0.169*10^{12}$	10^{12}

Strassen's Matrix Multiplication

- *More algorithms for matrix Multiplication:*

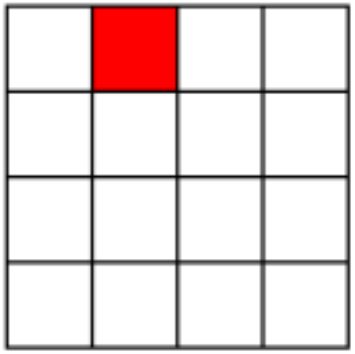
- Algorithms with better asymptotic efficiency are known but they are even more complex.

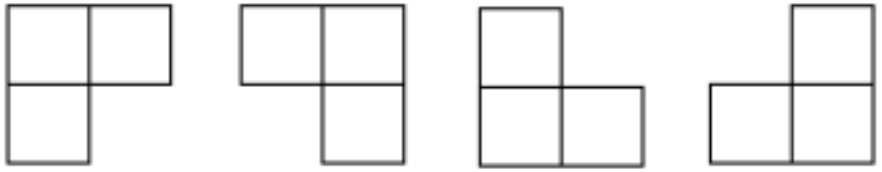
时间	复杂度	作者
<1969	O(n^3)	
1969	O(n^2.81)	Strassen
1978	O(n^2.79)	Pan
1979	O(n^2.7799)	Bini,Lotti etc.
1981	O(n^2.55)	Schonhage
1984	O(n^2.52)	Victor Pan
1987	O(n^2.48)	Strassen
1987	O(n^2.376)	Coppersmith and Winograd

DeepMind团队开发的人工智能系统Alpha Tensor在超过70种大小各异的矩阵上击败了现有的最佳算法。如，9*9矩阵从511降到498，11*11矩阵从919减至896。

棋盘覆盖（Chessboard Cover）问题

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。

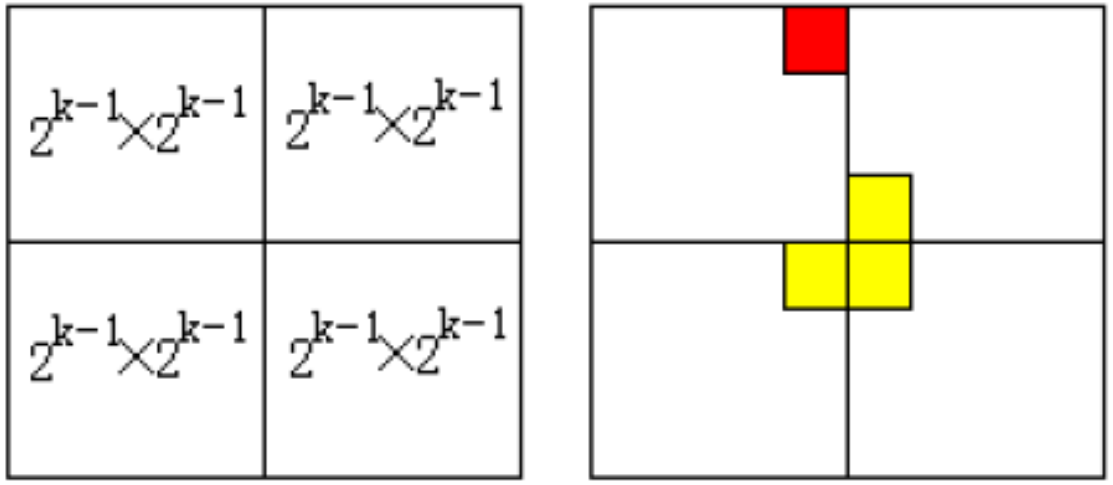




棋盘覆盖问题

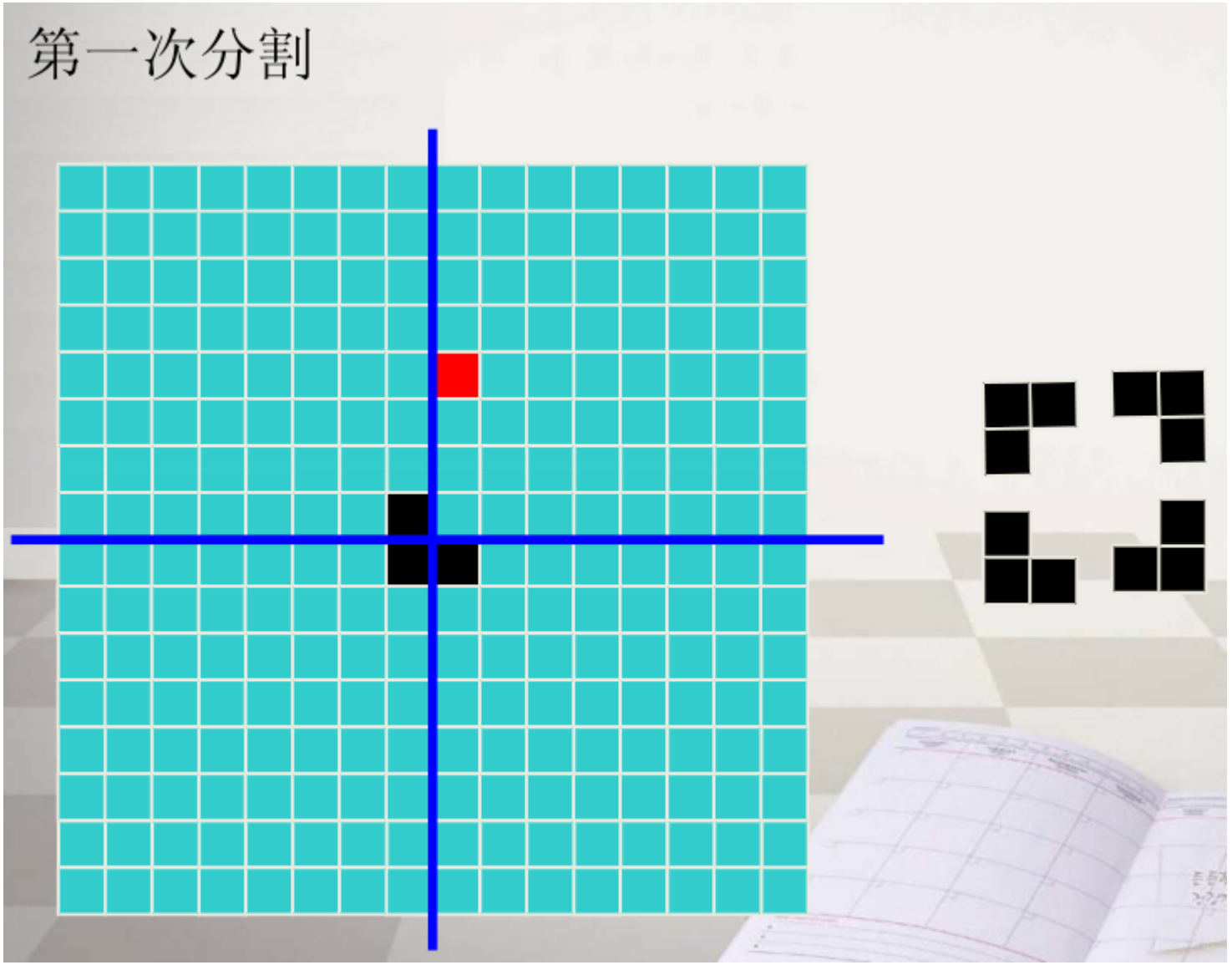
分治策略：

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，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。

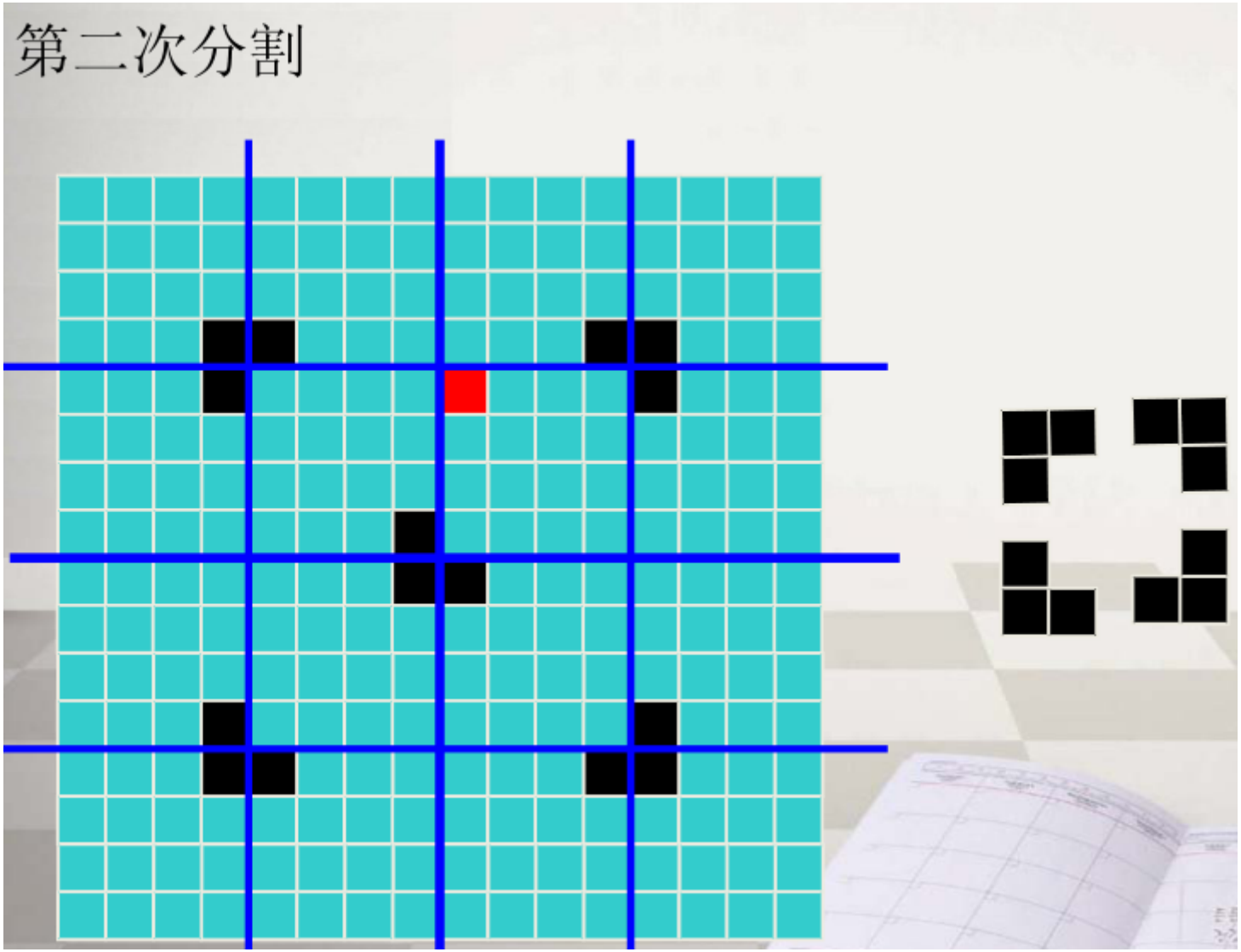


52

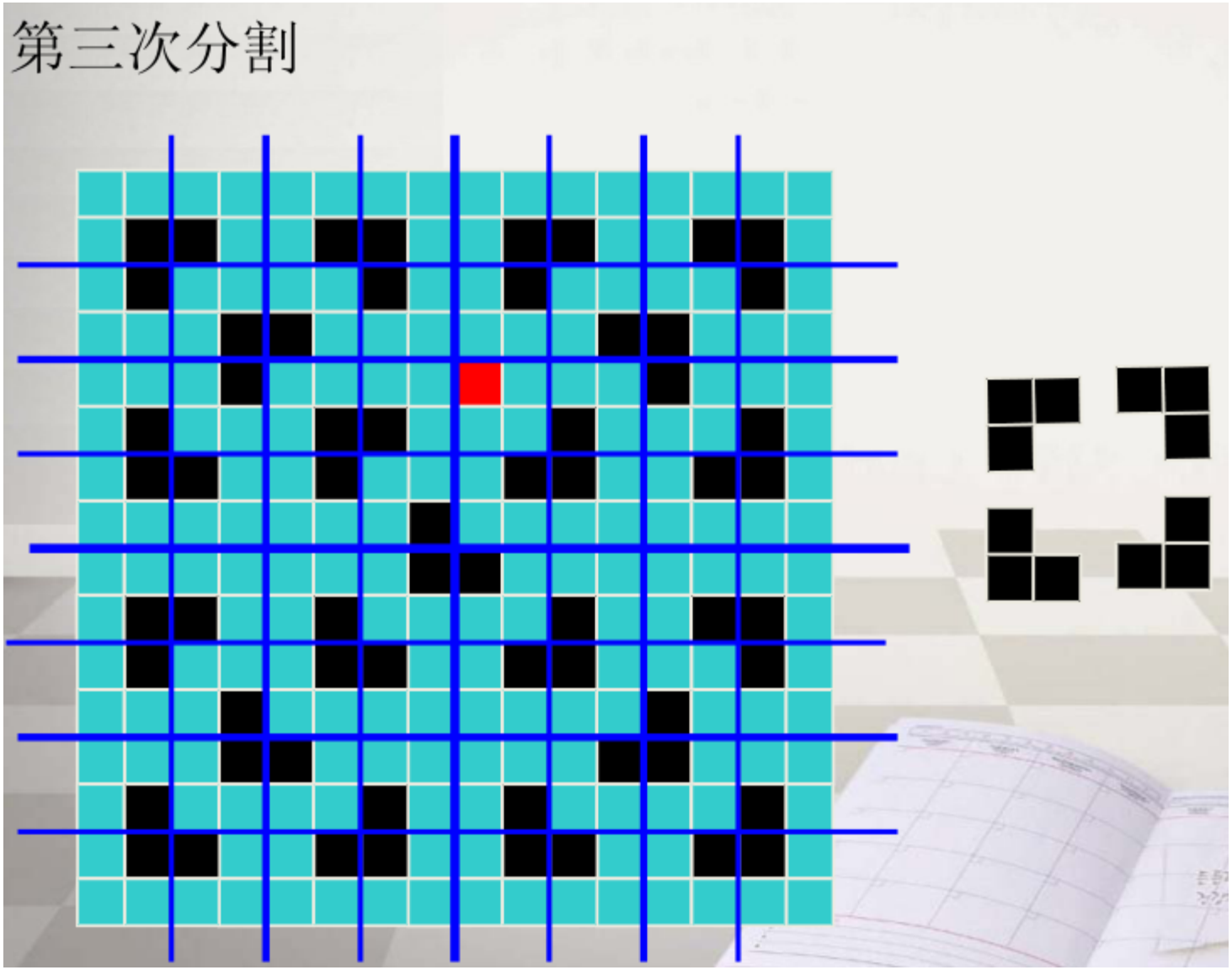
棋盘覆盖问题



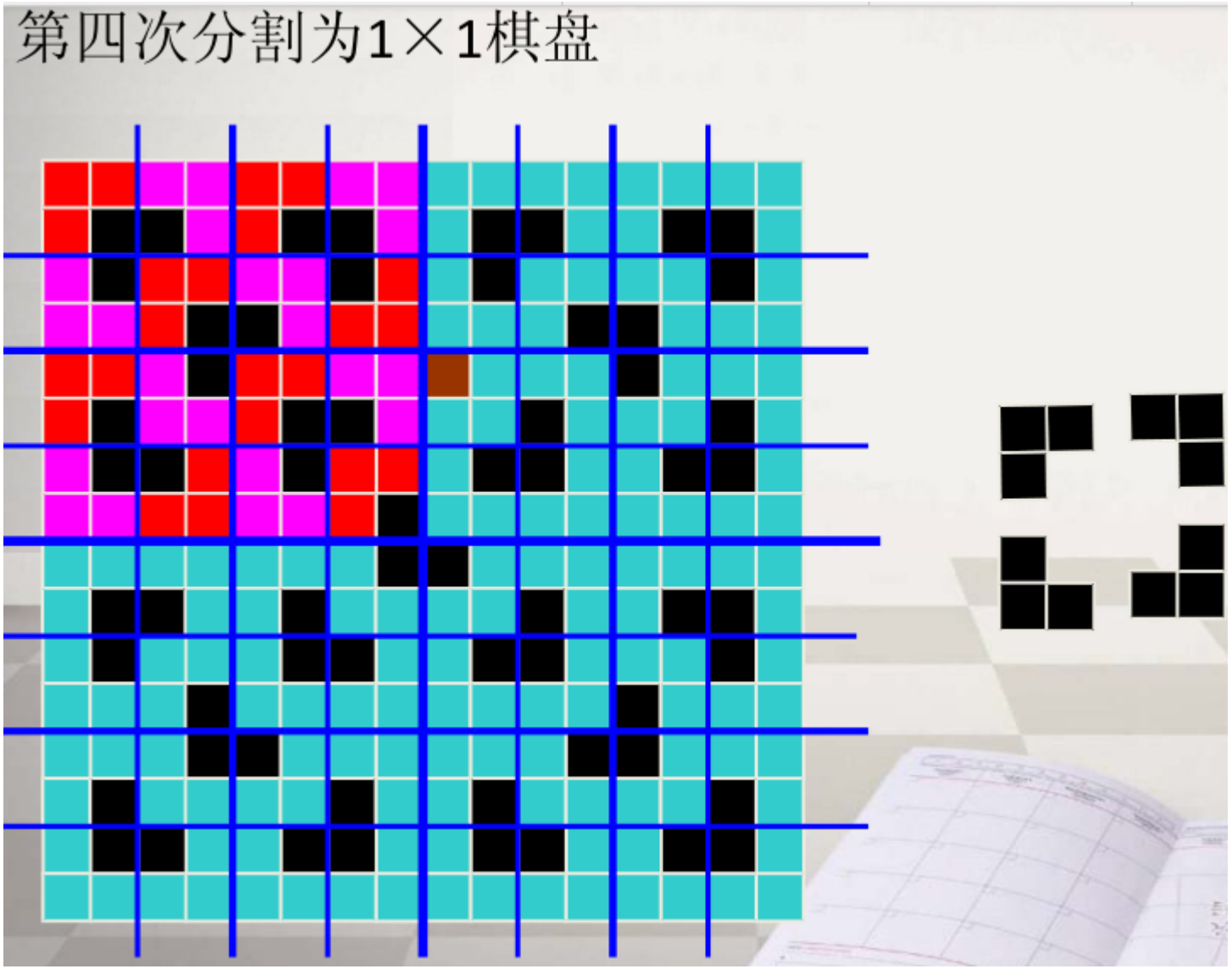
棋盘覆盖问题



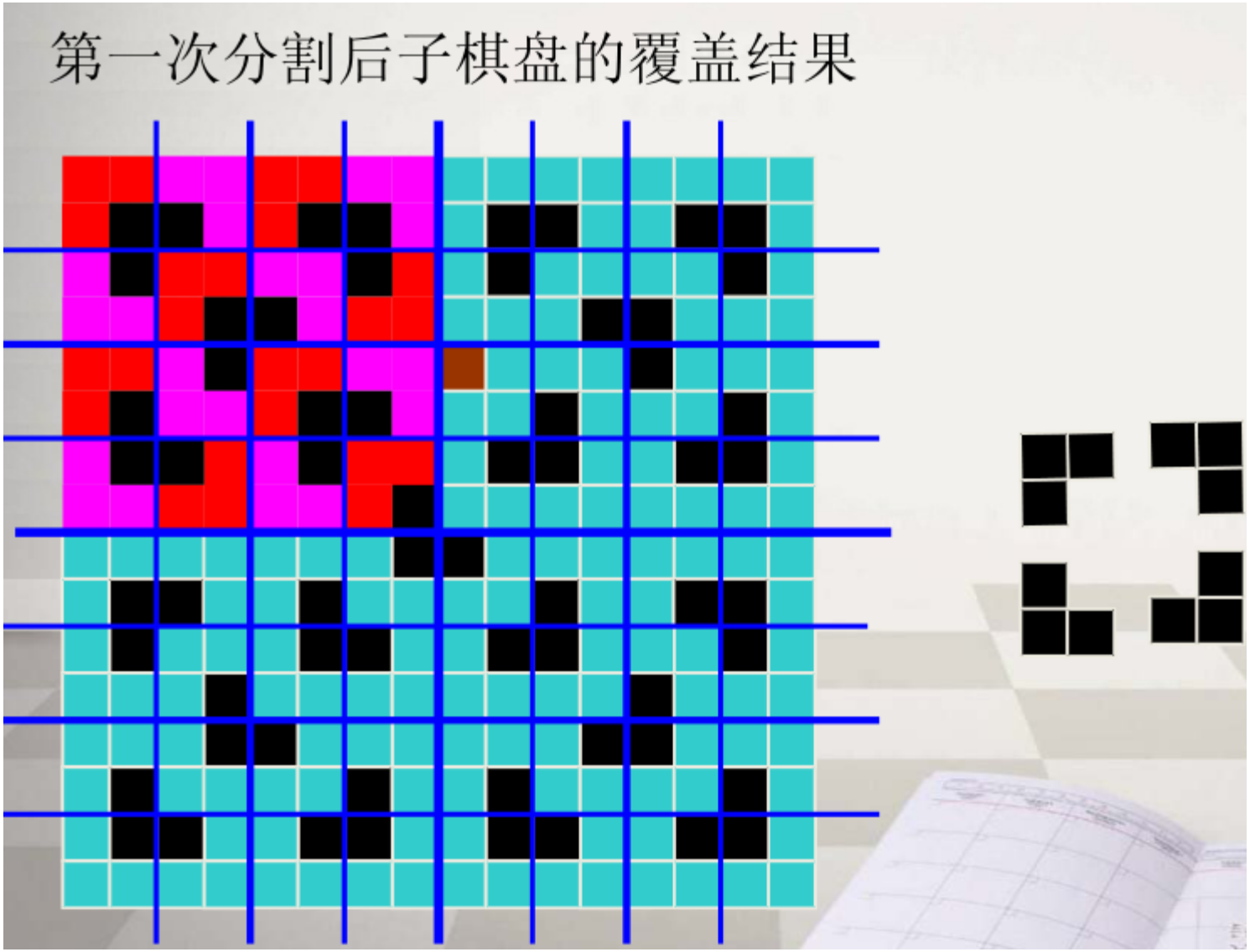
棋盘覆盖问题



棋盘覆盖问题



棋盘覆盖问题



棋盘覆盖问题

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

如何推导出所需要的骨牌个数呢？

(4^k -1)/3

Summary

1. 分治法是一种一般性的算法设计技术，他将问题的实例划分为若干个较小的实例(最好拥有相同的规模)，对这些小的问题求解，然后合并这些解，得到原始问题的解。
2. 分治法的时间效率满足：T(n)=aT(n/b)+f(n)
3. 合并排序是一种分治排序算法，任何情况下，该算法的时间效率都是Θ(nlogn)，它的键值比较次数非常接近理论的最小值，缺点是需要大量的额外存储空间。
4. 快速排序也是一种分治排序算法，具有出众的时间效率nlogn，最差效率是平方级的。
5. 折半查找是一种对有序数组进行查找的算法，效率为logn
6. n位大整数乘法的分治算法，大约需要做$n^{1.585}$次乘法。
7. Stressan 算法也是分治算法。

思考题

1. 设$a[0:n-1]$是一个已排好序的数组。设计搜索算法，使得当搜索元素在数组中时，i和j相同，均为x在数组中的位置；搜索元素x不在数组中时，返回小于x的最大元素的位置i和大于x的最小元素位置j。

并对自己的程序进行复杂性分析。

2. 给定2个大整数u和v，分别有m位和n位数字，且$m<=n$。用通常的乘法求uv的值需要$O(mn)$时间。当m比n小得多时，试设计一个算法，在上述情况下用$O(nm^{\log(3/2)})$时间求出uv值。