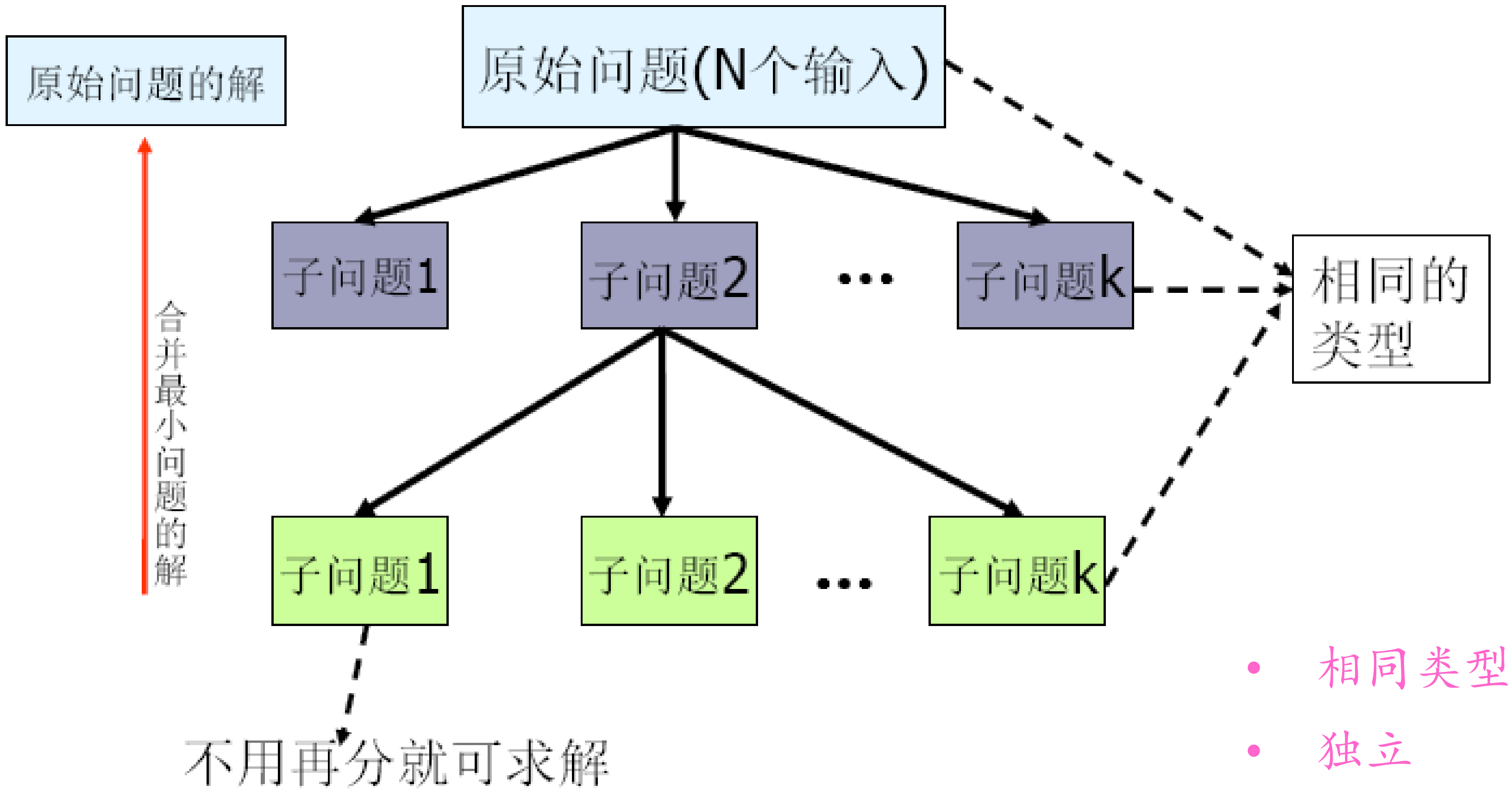
Analysis and Design of Algorithms

Chapter 5: Divide and Conquer

School of Software Engineering © Yanling Xu

Divide and Conquer Approach



- 相同类型
- 独立
- 规模均衡

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*

Divide and Conquer Approach

Algorithm

```
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。

Divide and Conquer Approach

Algorithm analysis—general divide-and-conquer recurrence

- *A problem's instance of size n is divided into a instances of size n/b (assuming n is a power of b)*
- *a of the problems needs of be solved*
- *f(n) is a function that counts for the time spent on dividing the problem into smaller ones and on combing their solutions*

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

Divide and Conquer Approach

Master Theorem

$T(n) = aT(n/b) + f(n)$, where $f(n) \in \Theta(n^k)$, $k>=0$

1. $a < b^k$ $T(n) \in \Theta(n^k)$
2. $a = b^k$ $T(n) \in \Theta(n^k \log n)$
3. $a > b^k$ $T(n) \in \Theta(n^{\log_b a})$

The same results hold with O instead of Θ

Multiplication of Large Integers

Idea of Multiplication of Large Integers

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*

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

- Efficiency: $\boldsymbol{n^2}$ one-digit multiplications

Multiplication of Large Integers

First Divide-and-Conquer Algorithm

***X*=**	A	B
***Y*=**	C	D

- 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

  $$X * Y = AC \cdot 10^{n} + (AD + BC) \cdot 10^{n/2} + BD$$

- If $n=2^k$, recurrence
- Could 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: ***$T(n) = O(n^2)$*** **×NO Promotion**

If $n=2^k$, then
$C(2^k)=4C(2^{k-1})=4[4C(2^{k-2})]=4^2C(2^{k-2})$
$=\ldots =4^kC(2^{k-k})=4^k$
$k=\log_2 n$
$C(2^k)=C(n)=4^{\log n}=2^{2\log n}=n^2$

For $n <> 2^k$ by Smoothness Rule, $O(n^2)$

如果在推导中不忽略O(n)，则
$C(2^k)=4C(2^{k-1}) + 2^k=4[4C(2^{k-2})+2^{k-1}]+2^k$
$=4^2C(2^{k-2})+4*2^{k-1}+2^k= 4^2C(2^{k-2})+2^{k+1}+2^k$
$= 4^kC(2^{k-k})+2^{k+k-1}+\ldots+2^k$
$= 2^k(2^k +2^{k-1} +\ldots +2^0)$
$= 2^k(1-2^{k+1})/(1-2)=2^{2k+1}-2^k$

Multiplication of Large Integers

Second Divide-and-Conquer Algorithm

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

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

  i.e., $(AD + BC) = (A + B) * (C + D) - AC - BD$

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

- *Analysis:*

  Recurrence for the number of multiplications $T(n)$:

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

If $n=2^k$, then

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

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

$k=\log_2 n$

$C(2^k)=C(n)=3^{\log n}=n^{\log 3}\approx n^{1.585}$

Solution: $T(n) = O(3^{\log_2 n}) = O(n^{\log_2 3}) \approx O(n^{1.585})$ ✓ **Promotion**

Multiplication of Large Integers

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

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m+1位的结果，使问题的规模变大，故不选择第2种方案。

☆ 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。

如果在推导中不忽略O(n)，则

$C(2^k)=3C(2^{k-1})+2^k=3[3C(2^{k-2})+2^{k-1}]+2^k=3^2C(2^{k-2})+3*2^{k-1}+2^k$

$=\ldots\ =3^kC(2^{k-k})+3^{k-1}*2^1+3^{k-2}*2^2+\ldots+3*2^{k-1}+2^k$

$=3^k*2^0+3^{k-1}*2^1+3^{k-2}*2^2+\ldots+3*2^{k-1}+3^0*2^k$ 其等比$q=2/3$首相$a_1=3^k*2^0$

$=a_1(1-q^{k+1})/(1-q)=3^{k+1}-2^{k+1}=3*3^{\log_2 n}-2*n=3*n^{\log_2 3}-2*n$ 前者是主项

Strassen's Matrix Multiplication

- ***Idea***
  - *brute-force alg.*: ***O(n³)***

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

For C_{ij}, , n multiples and n-1 *addings*

So for n 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,

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

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

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

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

Strassen's Matrix Multiplication

Idea

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

Strassen observed [1969] that the product of two matrices can be computed as follows

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

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

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

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

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

Analysis of Strassen's Matrix Multiplication

- *Number of additions:*

  $A(n) = 7A(n/2)+18(n/2)^2, \quad A(1) = 0$

  - Solution:

  $A(n) = n^{\log_2 7}$

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

Binary Search

问题描述：

已知一个按非降次序排列的元素表$a_0,a_1,\dots,a_{n-1}$，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。

Idea of Binary Search

Very efficient algorithm for searching in sorted array with non-decreasing order

K

vs

A[0] . . . A[m] . . . A[n-1]

选取一个下标m，可得到三个子问题：

$I_1=(m,a_0,\dots,a_{m-1},K)$

$I_2=(1,a_m,K)$

$I_3=(n-m-1,a_{m+1},\dots,a_{n-1},K)$

- If K = A[m], stop (successful search);
- otherwise, continue searching by the same method
  - in A[0..m-1] if K < A[m]
  - and in A[m+1..n-1] if K > A[m]

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。

如果对所求解的问题（或子问题）所选的下标 m 都是中间元素的下标，m= $\lfloor (l+r)/2 \rfloor$，则由此产生的算法就是二分检索算法。

Binary Search

- ***Example***

  - K=70

0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	85	93	98
l						m						r
							l		m			r
							l,m	r				

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 – an Iterative Algorithm

ALGORITHM BinarySearch(A[0..n-1], K)

$l \leftarrow 0$; $r \leftarrow n$-1

while $l \le r$ do // l and r crosses over→ can't find K

  $m \leftarrow \lfloor (l+r)/2 \rfloor$

  if K = A[m] return m //the key is found

  else if K < A[m] $r \leftarrow m$-1 //the key is on the left half of the array

  else $l \leftarrow m$+1 // the key is on the right half of the array

return -1

Binary Search

Binary Search – a Recursive Algorithm

```
ALGORITHM BinarySearchRecur(A[0..n-1], l, r, K)
if l > r
    return –1
else
    m ← ⌊(l + r) / 2⌋
    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*) *:*

  $$C_w(n) = C_w(\lfloor n/2 \rfloor) + 1 \,,$$
  $$C_w(1) = 1$$

  - Solution:

  $$C_w(n) = \Theta\,(\log n)$$

- *Best-case:*
  - successful $C_b(n) = 1$

    一次找到，即K＝A[n/2]

for $n = 2^k$,

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

$$C(2^0) = 1$$

backward substitutions:

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

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

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

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

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

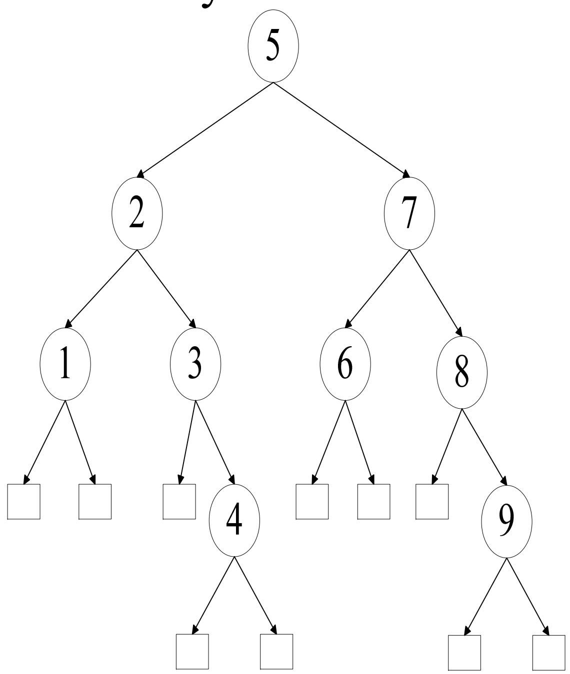
Binary Search

Analysis of Binary Search

Average-case:

- Consider the searching process as a binary tree. In looking at the binary tree, we see that there are i comparisons needed to search 2^{i-1} elements on level i of the tree.
- For a list with $n = 2^k$-1 elements, there are k levels in the binary tree.
- The average case for all successful search:

$$A(n) = \frac{1}{n}\sum_{i=1}^{k} i2^{i-1} \approx \log(n+1) - 1$$



Binary Search

Analysis of Binary Search

- 区分以下情况进行分析
  - 成功检索：指所检索的*K*恰好在A中出现
    由于A中共有n个元素，故成功检索恰好有n种可能的情况
  - 不成功检索：指 *K* 不出现在A中
    根据取值，不成功检索共有n+1种可能的情况 (取值区间)

    K <A(1) 或 A(i)< K <A(i+1), 1≤i<n-1 或 K >A(n)

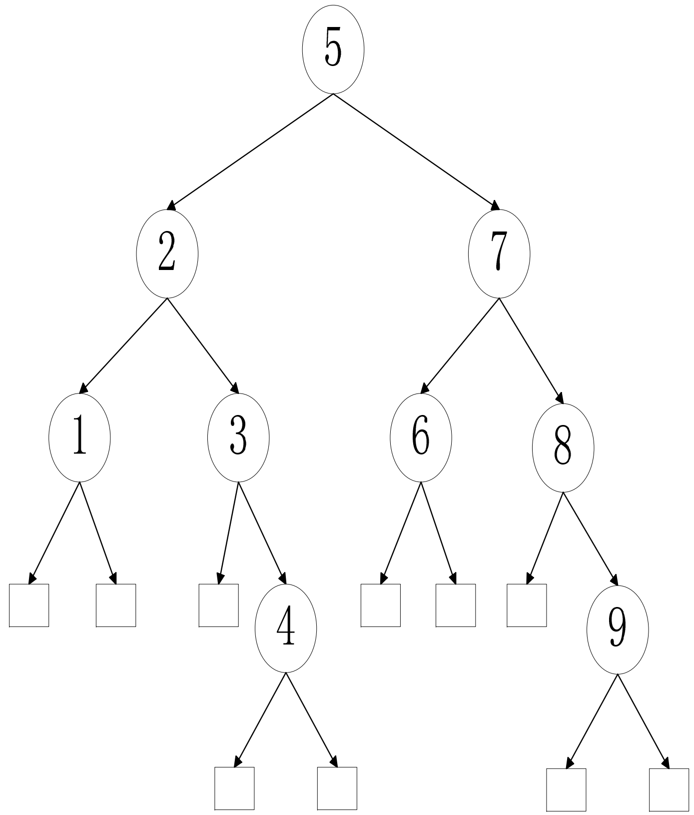
Binary Search

- 二元比较树

  算法执行过程的主体是k与一系列中间元素A(mid)比较。

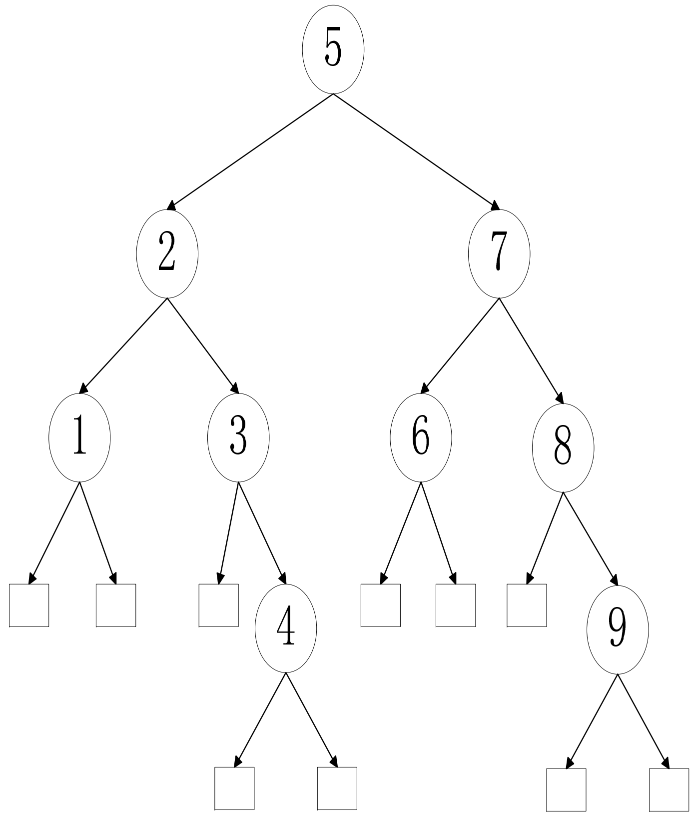
  用一棵二元树描述该过程，称为二元比较树

- 结点：
  - 内结点：
    - 代表一次元素比较
    - 用 圆形结点表示
    - 存放一个 mid值(下标)
    - 代表成功检索情况
  - 外结点：
    - 用方形结点表示，
    - 表示不成功检索情况
- 路径：代表检索中元素的比较序列



Binary Search

- 二元比较树的查找过程
  - 若*K*在A中出现，则算法的执行过程在一个圆形的内结点处结束
  - 若*K*不在A中出现，则算法的执行过程在一个方形的外结点处结束



注：外结点不代表元素的比较，因为比较过程在该外结点的上一级的内结点处结束。

Binary Search

Analysis of Binary Search

逐个查找法的复杂度AverageCase

$$T_{avg}(n) = \sum_{size(I)=n} p(I)T(I)$$

$$= \left(1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + 3 \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right) + n \cdot (1 - p)$$

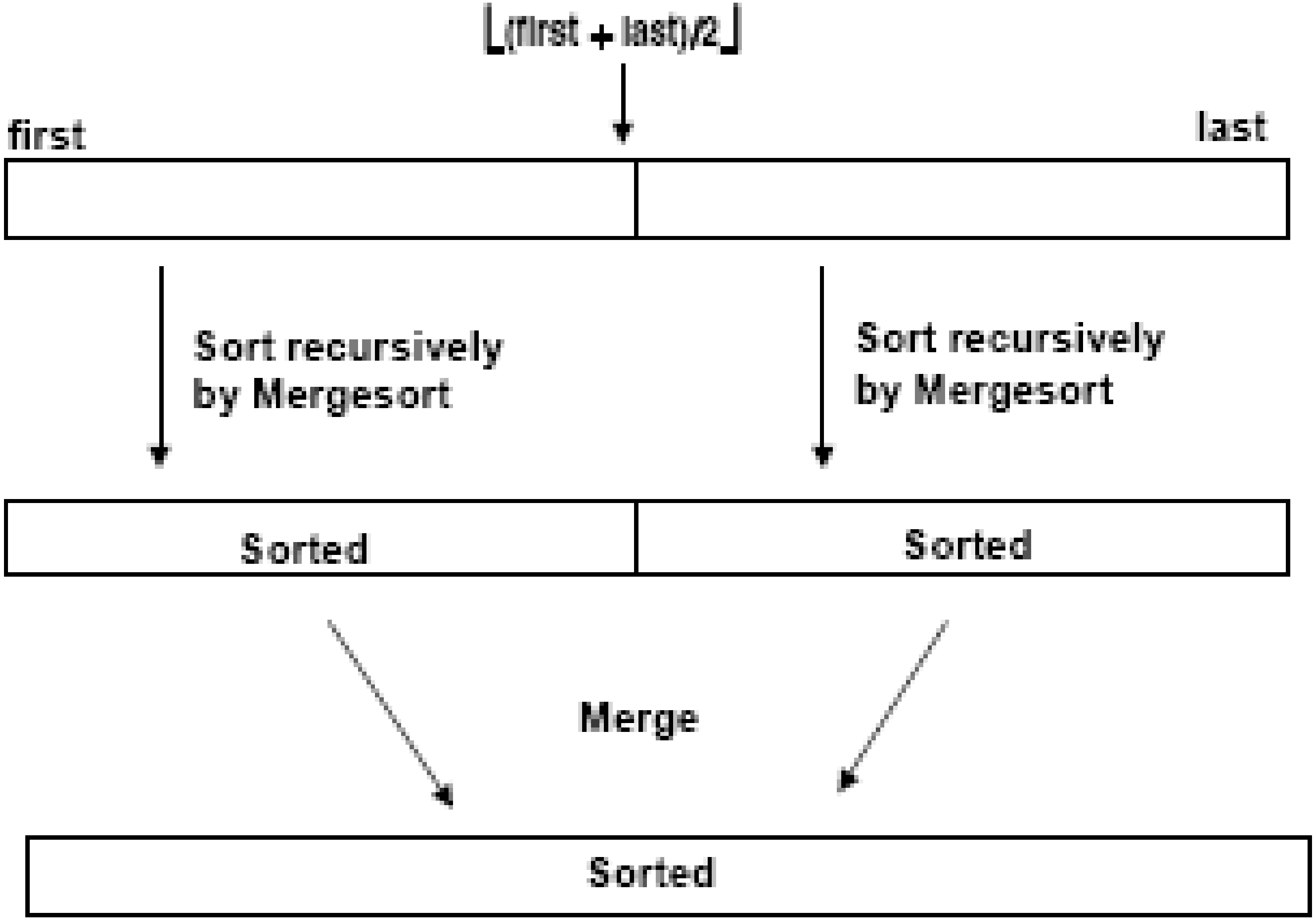
$$= \frac{p}{n}\sum_{i=1}^{n} i + n(1-p) = \frac{p(n+1)}{2} + n(1-p)$$

Mergesort

Idea of Mergesort

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

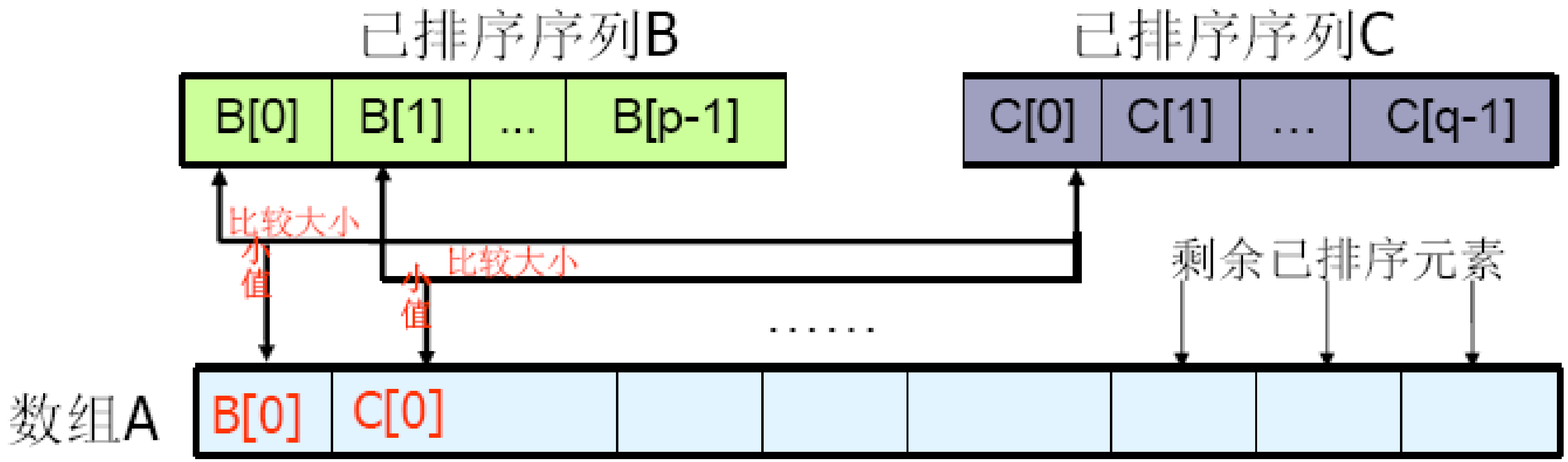
Mergesort



⌊(first + last)/2⌋
first
last
Sort recursively
by Mergesort
Sort recursively
by Mergesort
Sorted
Sorted
Merge
Sorted

Mergesort

MERGE



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 $A[0..\lfloor n/2\rfloor - 1]$ to $B[0..\lfloor n/2\rfloor - 1]$
  copy $A[\lfloor n/2\rfloor..n-1]$ to $C[0..\lceil n/2\rceil - 1]$
  Mergesort$(B[0..\lfloor n/2\rfloor - 1])$
  Mergesort$(C[0..\lceil n/2\rceil - 1])$
  Merge(B, C, A)

中分点 mid← $\lfloor(\text{low}+\text{high})/2\rfloor$

Mergesort

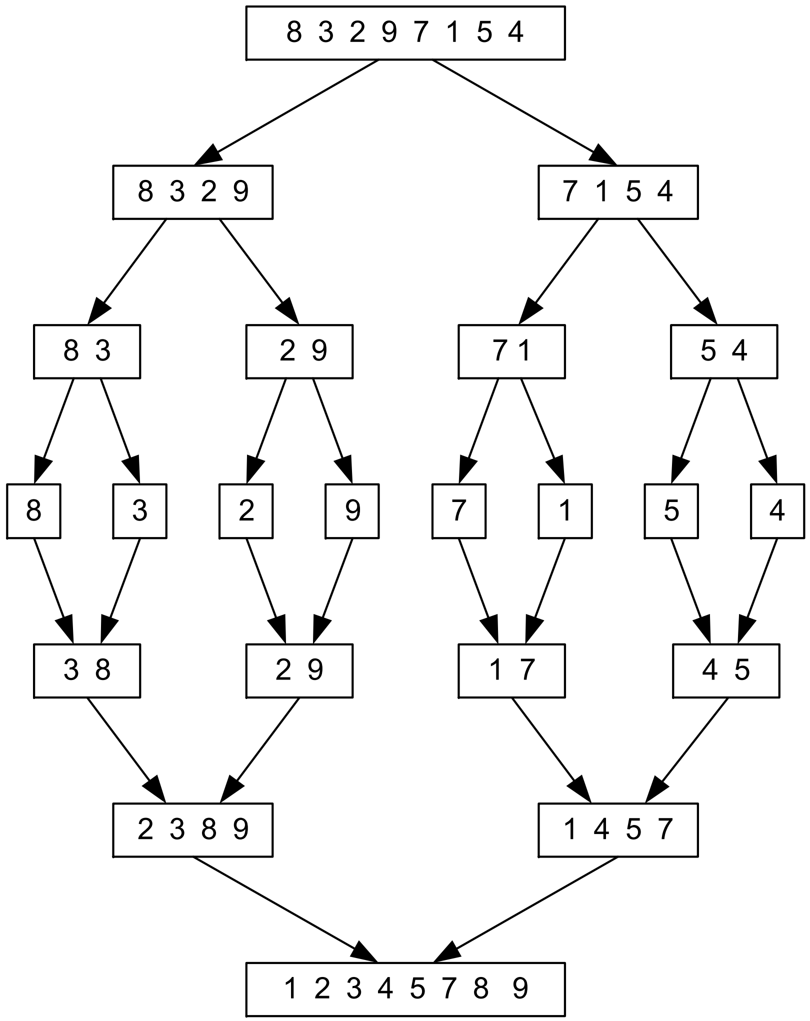
The Mergesort Algorithm ('cont)

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
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
    else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
    $k \leftarrow k + 1$
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

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



Mergesort

Analysis of Mergesort

- *Number of basic operations (key comparisons):*

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

$$C(1) = 0$$

Where $C_{merge}(n) = n-1$

- **$C(n) = \Theta(n \log n)$**

If $n=2^k$
devide：$D(n)=\Theta(1)$
conquer: $T(n)=2T(n/2)$, $T(1)=\Theta(1)$
merge:$T(n)= \Theta(n)$

$T(n)=2T(n/2)+\Theta(n)$
$=4T(n/4)+2\Theta(n/2)+\Theta(n)$
$=8T(n/8)+4\Theta(n/4)+2\Theta(n/2)+\Theta(n)$
$=2^{\wedge}\log n T(1)+\Theta(n)+\Theta(n)...+\Theta(n)$ [一共logn个$\Theta(n)$]
$= \Theta(n)+\log n\Theta(n) = \Theta(n\log n)$

If $2^k<n<2^{k+1}$,则有$T(n)\le T(2^{k+1})$。

Quicksort

Idea of Quicksort

------ Partition

- 快速分类是一种基于划分的分类方法；

- *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.*

- 快速分类：通过反复地对待排序集合进行划分达到分类目的的分类算法。

- *Conquer:*
  - *Sort the two subarrays A[l..s-1] and A[s+1..r] by recursive calls to quicksort*

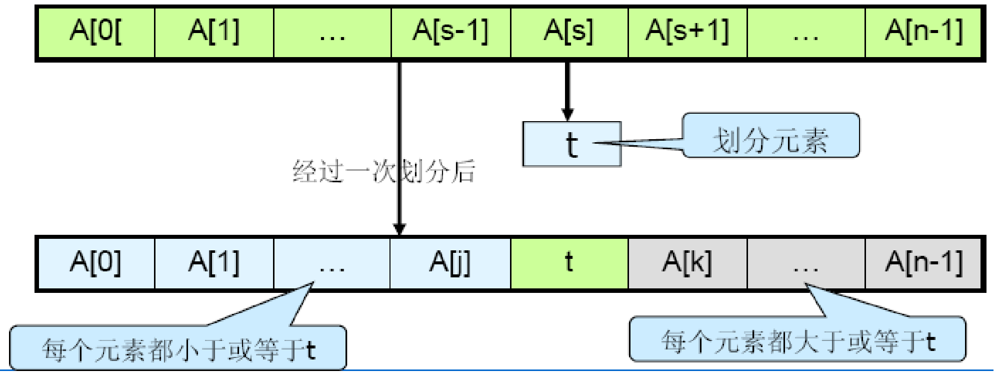
- A[l..s-1] 中所有元素小于等于 A[s+1..r] 中任何元素，所以这两个集合可独立进行划分

- *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 (‘cont)

- *Select a pivot w.r.t. whose value we are going to divide the list. (typically, p = A[l])*
- *Rearrange the list so that*
- *all elements in the first s positions are smaller than or equal to the pivot*
- *all elements in the remaining n-s positions are larger than or equal to p*



- *Exchange the pivot with the last element in the first sublist(i.e., $\leq$ sublist) – the pivot is now in its final position*
- *Sort the two sublists recursively using quicksort.*

Quicksort

Idea of Quicksort ('cont)

- *Strategy for pivot selection*
  - *Randomly selected*
  - *Simplest Strategy: selecting the array's first element* $A[l]$

Quicksort

Idea of Quicksort ('cont)

- *Procedure for rearranging elements in a partition*

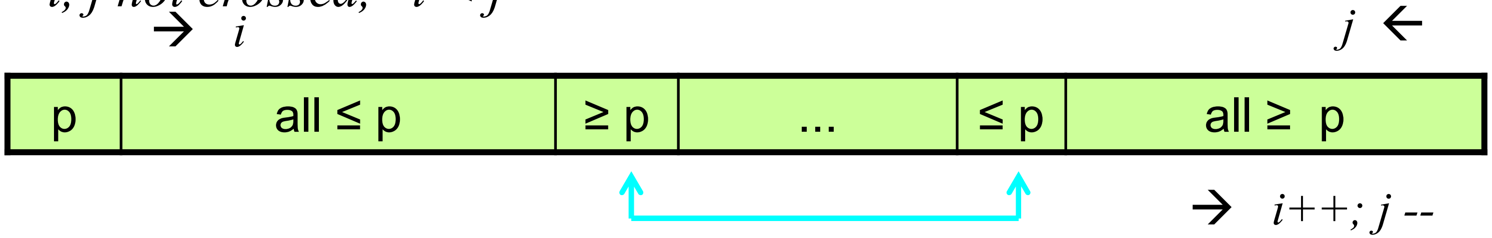
  ------ based on two-scans of the subarray

  - *Left-to-right scan: index i, starts with the second element,*
    - *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*
  - *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*

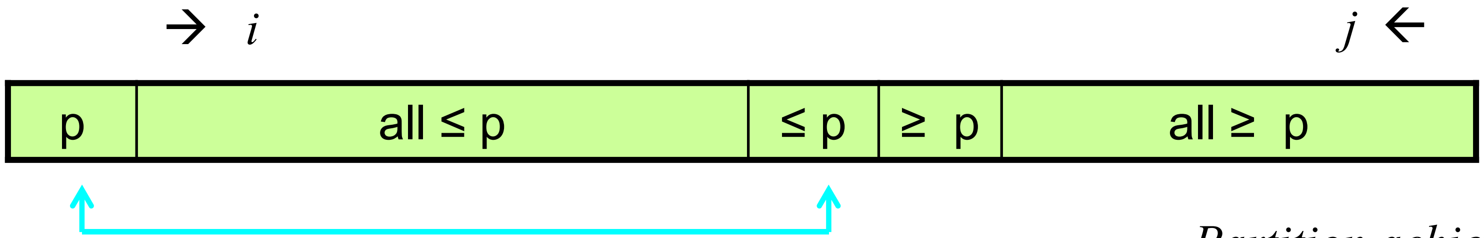
Quicksort

- *three cases for scan stopping*

i, j not crossed, i < j

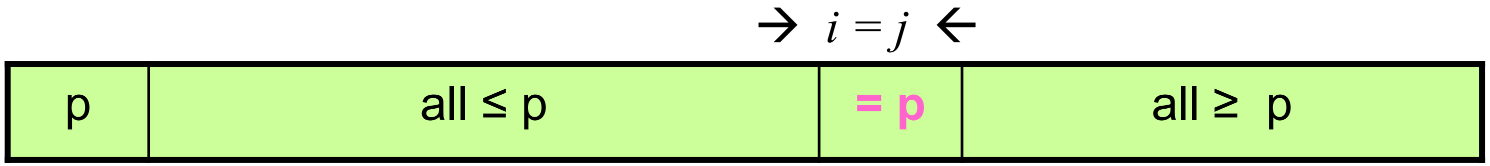


i, j crossed, i > j



Partition achieved

pointing to the same, i = j



Partition achieved, s = i = j

Quicksort

The Quicksort Algorithm

```
ALGORITHM Quicksort(A[l..r])
//Sorts a subarray by quicksort
//Input: A subarray A[l..r] of A[0..n-1],defined by its left and right
     indices l and r
//Output: The 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

The Quicksort Algorithm - Partitioning

```
template<class Type>
int Partition (Type a[], int l, int r)
{
        int i = l, j = r + 1;
        Type x=a[l];
        // 将< x的元素交换到左边区域
       // 将> x的元素交换到右边区域
        while (true) {
           while (a[++i] <x);
           while (a[- -j] >x);
           if (i >= j) break;
           Swap(a[i], a[j]);
           }
 //将x交换到它在排序序列中应在的位置上
       a[l] = a[j];
       a[j] = x;
return j;
}
```

Number of comparisons:

$n + 1$ (if indices i, j, cross over)

n (if indices i, j, coincide)

Quicksort

- ***Example***

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	j
A:	65	70	75	80	85	60	55	50	45	+∞	2	9
A:	65	45	75	80	85	60	55	50	70	+∞	3	8
A:	65	45	50	80	85	60	55	75	70	+∞	4	7
A:	65	45	50	55	85	60	80	75	70	+∞	5	6
A:	65	45	50	55	60	85	80	75	70	+∞	6	5
					j	......*i*						
A:	60	45	50	55	65	85	80	75	70	+∞		

交换划分元素

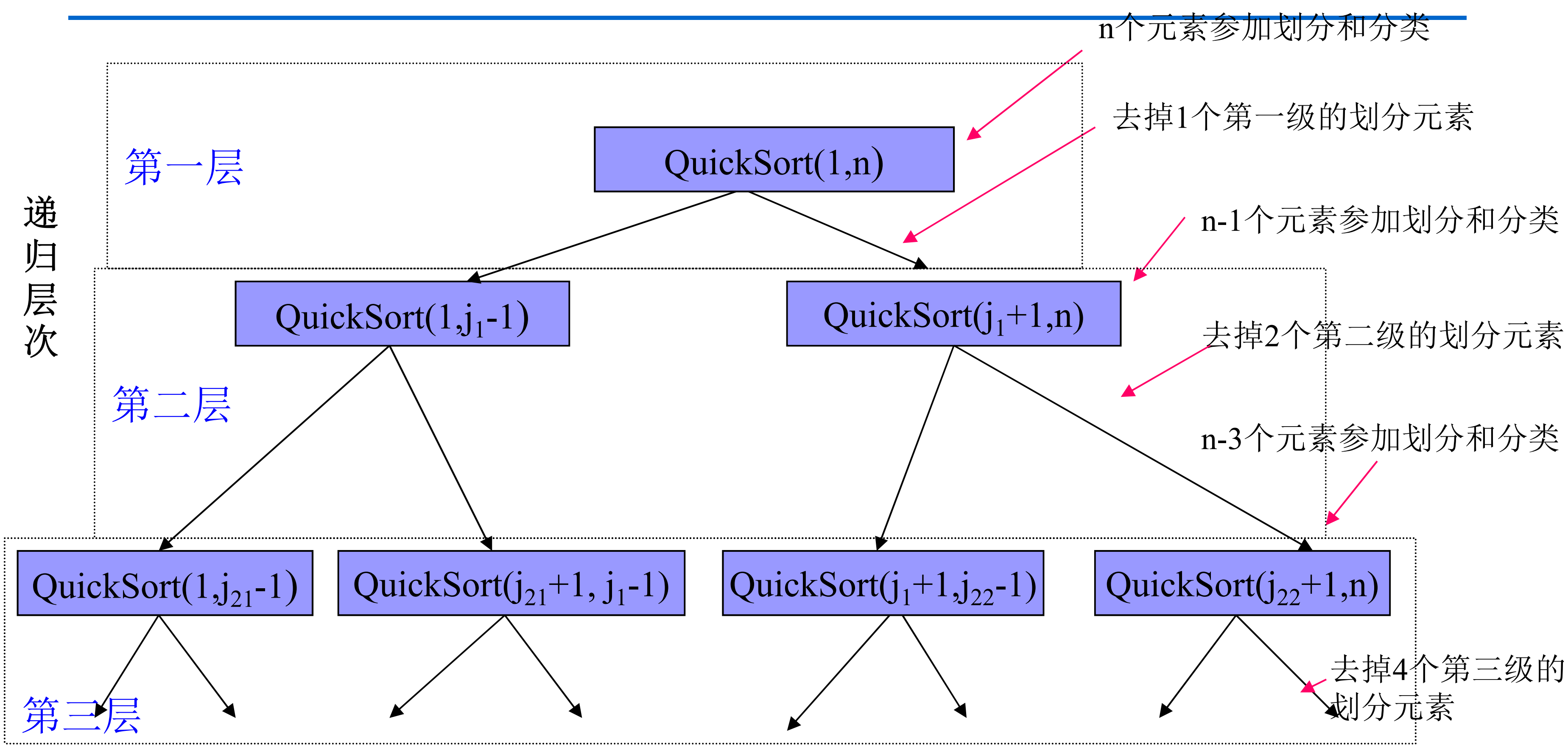
划分元素定位于此

- 经过一次“划分”，实现了对集合元素的调整：以划分元素为界，左边子集合的所有元素均小于等于右边子集合的所有元素。
- 按同样的策略对两个子集合进行分类处理。当子集合分类完毕后，整个集合的分类也完成了。这一过程避免了子集合的归并操作。

Quicksort

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

Quicksort



设在任一级递归调用上，调用PARTITION处理的所有元素总数为r，则，初始时r=n，以后的每级递归上，由于删去了上一级的划分元素，故r比上一级至少1：

理想情况，第一级少1，第二级少2，第三级少4， …；

最坏情况，每次仅减少1（每次选取的划分元素刚好是当前集合中最小或最大者）

Quicksort

Analysis of Quicksort

Number of comparisons for a partition:

$n + 1$ (if indices i, j, cross over)

n (if indices i, j, coincide)

- Best case: *split in the middle* — $\Theta(n \log n)$

$C_b(n) = 2C_b(n/2) + \Theta(n)$ //2 subproblems of size n/2 each

$C_b(1) = 0$

for $n = 2^k$, backward substitutions, could get it

Quicksort

Analysis of Quicksort

- *Worst case:* *sorted array!* — $\Theta(n^2)$

$C_w(n) = C_w(n-1) + \Theta(n)$ //2 subproblems of size 0 and n-1

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],

$\rightarrow i$ $j \leftarrow$

A [0]	A [1]	…	A [n-1]

n+1 Comparisons

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

Quicksort

Analysis of Quicksort

- *Average case*: *random arrays* — ***O(n log n)***

Partition slit in each position $s \in [0, n\text{-}1]$, with the same probability $1/n$

$$
\begin{aligned}
C_{avg}(n) &= \frac{1}{n}\sum_{s=0}^{n-1}\left[(n+1)+C_{avg}(s)+C_{avg}(n-1-s)\right] \\
C_{avg}(0) &= 0 \\
C_{avg}(1) &= 0
\end{aligned}
$$

Solution

$$C_{avg}(n) \approx 2n\ln n \approx 1.38 n\log_2 n$$

☆ quicksort makes only 38% more comparisons than in the best case

☆ its innermost loop is so efficient that it runs faster than mergesort on randomly ordered arrays

Quicksort

Improvements

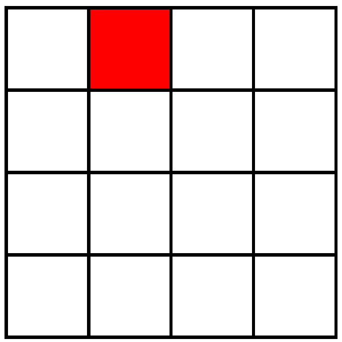
- 快速排序算法的性能取决于划分的对称性。通过修改算法partition，可以设计出采用随机选择策略的快速排序算法。
  - 随机选取划分元素

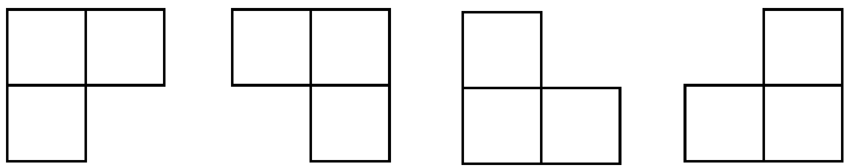
    在快速排序算法的每一步中，当数组还没有被划分时，可以在a[p:r]中随机选出一个元素作为划分基准，这样可以使划分基准的选择是随机的，从而可以期望划分是较对称的。

```
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);
}
```

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。

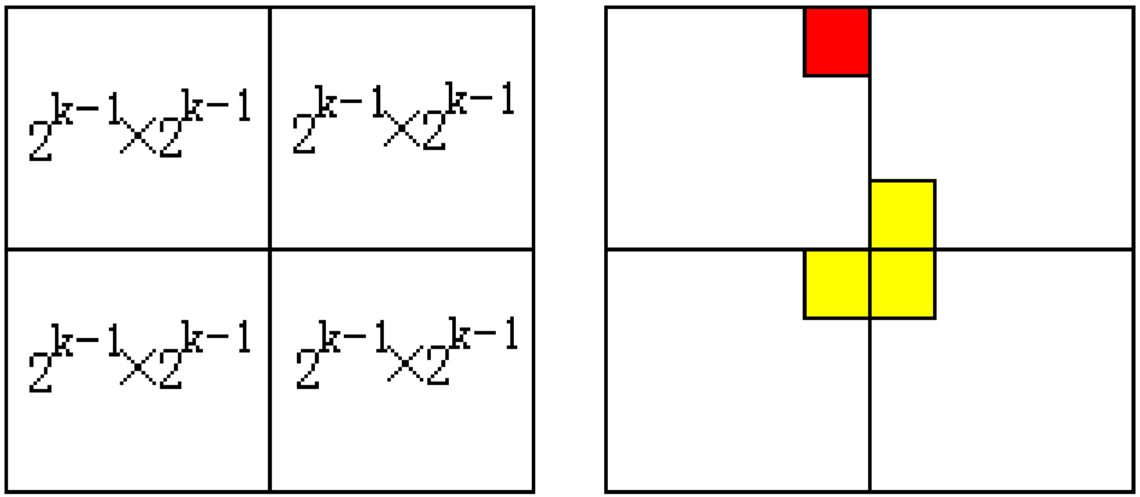




棋盘覆盖

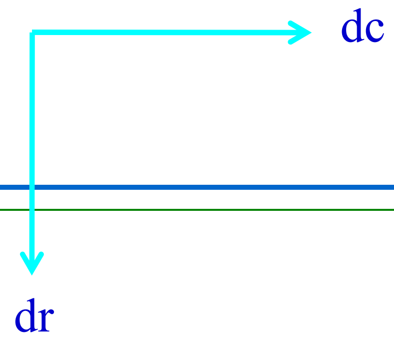
分治策略：

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。



(a) (b)

棋盘覆盖




```
void chessBoard(int tr, int tc, int dr, int dc, int size)
  {
     if (size == 1) return;
     int t = tile++,  // L型骨牌号
       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,则用 t 号L型骨牌覆盖该子棋盘的右下角
        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)       // 特殊方格在此棋盘中
        chessBoard(tr, tc+s, dr, dc, s);
     else {    // 此棋盘中无特殊方格,则用 t 号L型骨牌覆盖左下角
```

棋盘覆盖

```
    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)        // 特殊方格在此棋盘中
    chessBoard(tr+s, tc, dr, dc, s);
 else {      //此棋盘中无特殊方格,则用 t 号L型骨牌覆盖右上角
    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)       // 特殊方格在此棋盘中
    chessBoard(tr+s, tc+s, dr, dc, s);
 else {    //此棋盘中无特殊方格,则用 t 号L型骨牌覆盖左上角
    board[tr + s][tc + s] = t;
    chessBoard(tr+s, tc+s, tr+s, tc+s, s);}
}
```

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

覆盖2^k*2^k的棋盘所需L型骨牌个数为$(4^k-1)/3$

Concluding

- 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(最好用有相同的规模)，对这些小的问题求解，然后合并这些解，得到原始问题的解。
- 分治法的时间效率满足：$T(n)=aT(n/b)+f(n)$
- 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$\Theta(n\log n)$，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。
- 快速排序也是一种分治排序算法，具有出众的时间效率$n\log n$，最差效率是平方级的。
- 折半查找是一种对有序数组进行查找的算法，效率为$\log n$
- n位大整数乘法的分治算法，大约需要做$n^{1.585}$次乘法。

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。