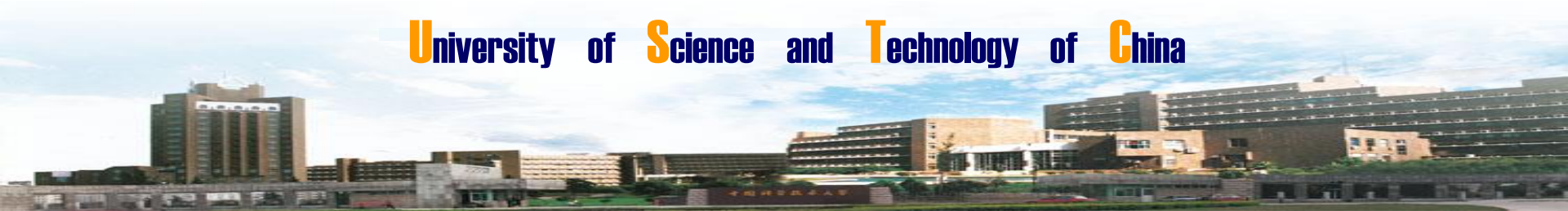
算法基础

主讲人： 庄连生

Email: { *lszhuang@ustc.edu.cn* }

***Spring 2018*，USTC**



第六讲 排序

内容提要：

- 排序问题
- 堆排序算法
- 快速排序算法
- 线性时间排序
- 排序算法比较

第六讲 排序

内容提要：

- 排序问题
- 堆排序算法
- 快速排序算法
- 线性时间排序
- 排序算法比较

排序问题

- **问题描述：**

  输入：n个数的序列 $< a_1, a_2, ..., a_n >$

  输出：输入序列的一个重排$< a_1', a_2', ..., a_n' >$，使得 $a_1' \le a_2' \le \cdots \le a_n'$

- 输入数据的结构可以各种各样，比如n元数组、链表等；
- 排序问题是计算机科学领域中的最基本问题：
  ① 应用广泛，是许多算法的关键步骤；
  ② 已有很多成熟算法；
  ③ 可以证明其非平凡下界，是渐进最优的；
  ④ 在实现过程中经常伴随着许多工程问题出现

排序问题

- 当待排序记录的关键字均不相同时，排序结果是惟一的，否则排序结果不唯一。
- **排序的稳定性：**
  - ① 在待排序的文件中，若存在多个关键字相同的记录，经过排序后这些具有相同关键字的记录之间的相对次序保持不变，该排序方法是**稳定的**；
  - ② 若具有相同关键字的记录之间的相对次序发生变化，则称这种排序方法是**不稳定的**。
- 排序算法的稳定性是针对所有输入实例而言的。即在所有可能的输入实例中，只要有一个实例使得算法不满足稳定性要求，则该排序算法就是不稳定的。

第六讲 排序

内容提要：

- 排序问题
- 堆排序算法
- 快速排序算法
- 线性时间排序
- 排序算法比较

二叉树

□ 二叉树：

① **完全二叉树**：深度为k，有n个结点的二叉树当且仅当其每一个结点都与深度为k的满二叉树中编号从1至n的结点一一对应时，称为完全二叉树。

特 点：叶子结点只可能在层次最大的两层上出现；对任一结点，若其右分支下子孙的最大层次为l，则其左分支下子孙的最大层次必为l或$l+1$

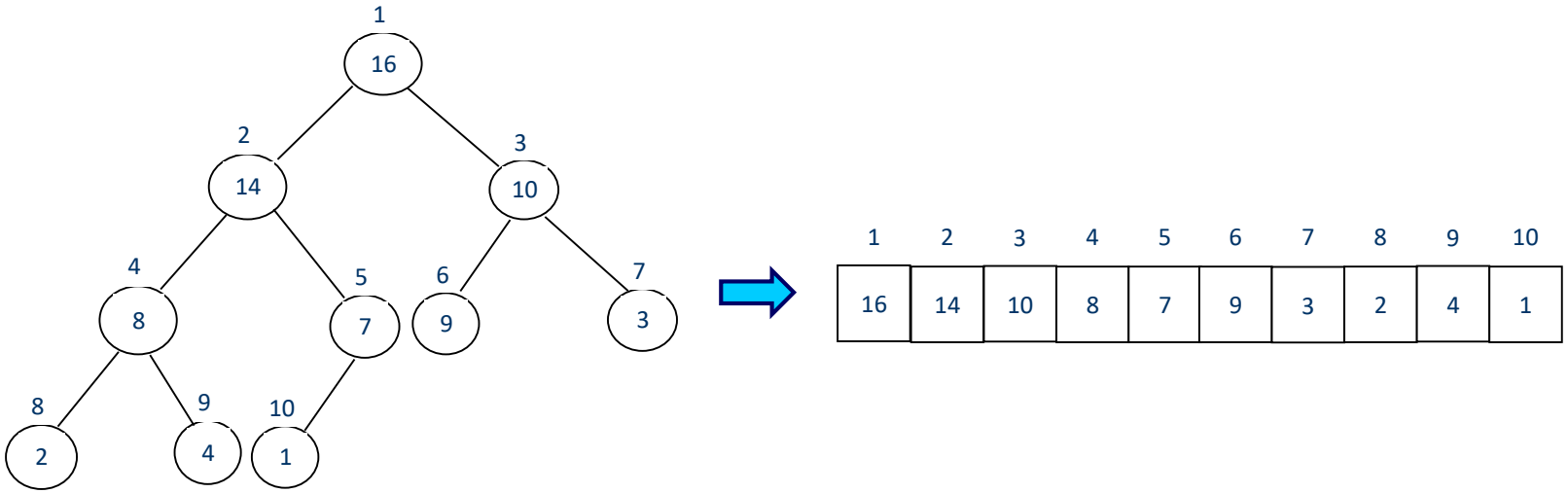
② **满二叉树**：一棵深度为k，且有(2^k-1) 个节点的二叉树。

特点：每一层上的结点数都是最大结点数。

③ **关系**：满二叉树必然是完全二叉树，反之不成立；

堆数据结构

- 堆数据结构是一种数组对象，可以被视为一棵完全二叉树。树中每个节点与数组中存放该结点值的那个元素对应。



- 表示堆的数组对象A具有两个性质：

① *length*[A]：是数组中的元素个数；

② *heap-size*[A]: 是存放在A中的堆的元素个数；

③ *heap-size*[A] ≤ *length*[A]

堆数据结构

- 作为数组对象的堆，给定某个结点的下标i，则：

① 父节点 PARENT(i) = floor(i /2),

② 左儿子为 LEFT(i) = 2 * i,

③ 右儿子为 RIGHT(i) = 2 * i + 1;

- 堆的分类：

① **大根堆**：除根节点之外的每个节点 i，有 $A[PARENT(i)] \geq A[i]$

② **小根堆**：除根节点之外的每个节点 i，有 $A[PARENT(i)] \leq A[i]$

- 在堆排序算法中，我们使用大根堆，堆中最大元素位于树根；

堆数据结构

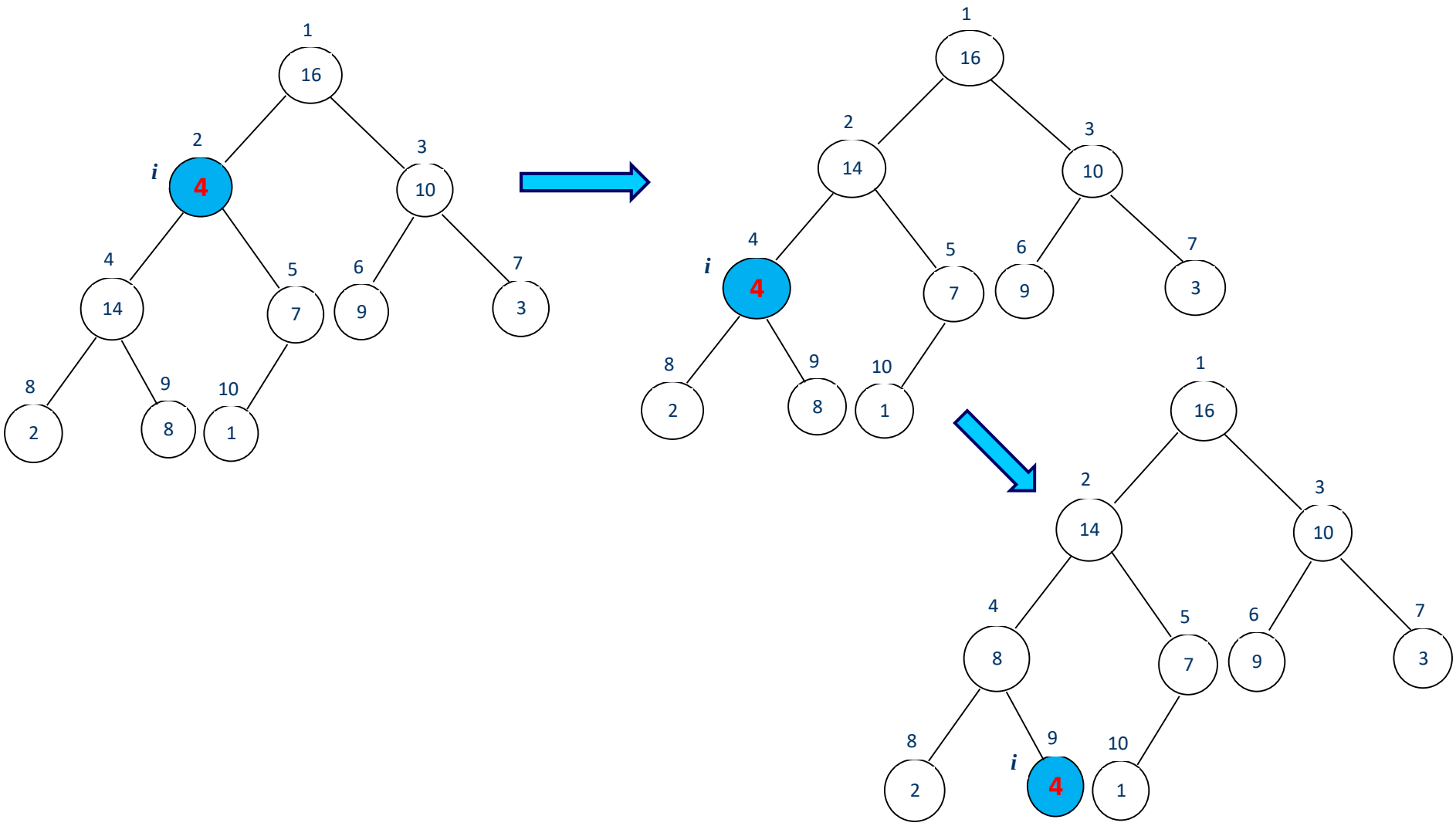
- 视为完全二叉树的堆：
  - ✓ 结点在堆中的高度定义为从本结点到叶子的最长简单下降路径上边的数目；
  - ✓ 定义堆的高度为树根的高度；
  - ✓ 具有n个元素的堆其高度为$\Theta(\lg n)$；
- 堆结构的基本操作时间至多与树的高度成正比：
  - ✓ MAX-HEAPIFY，运行时间为$O(\lg n)$，保持最大堆性质；
  - ✓ BUILD-MAX-HEAP，以线性时间运行，从无序的输入数组构造出最大堆；
  - ✓ HEAPSORT，运行时间为$O(n\lg n)$，对一个数组进行原地排序；
  - ✓ MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY 和 HEAP-MAXIMUM过程的运算时间为$O(\lg n)$, 可以让堆结构作为优先队列使用；

保持堆性质

- MAX-HEAPIFY函数的输入为一个数组A和小标*i*。假定以LEFT(*i*) 和RIGHT(*i*)为根的两棵二叉树都是最大堆，MAX-HEAPIFY让A[*i*]在最大堆中“下降”，使得以*i*为根的子树成为最大堆。

```
MAX-HEAPIFY( A, i )
1   l ← LEFT( i );
2   r ← RIGHT( i );
3   if  l ≤ heap-size[A] and A[ l ] > A[ i ]
4         then    largest ← l
5         else    largest ← i
6    if  r ≤ heap-size[ A] and A[ r ] > A[ largest ]
7         then    largest ← r
8    if   largest ≠ i
9          then   exchange A[ i ] ↔ A[ largest ]
10                MAX-HEAPIFY( A, largest )
```

保持堆性质


1
16
2
i
4
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1
1
16
2
14
3
10
4
i
4
5
7
6
9
7
3
8
2
9
8
10
1
1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
i
4
10
1

保持堆性质

- **基本思想：**

1) 找出A[*i*], A[LEFT(*i*)]和A[RIGHT(*i*)]中最大者，将其下标存在*largest*；

2) 交换A[*i*]和A[*largest*]使得结点*i*和其子女满足最大堆性质；

3) 下标为*largest*的结点在交换后的值是A[*i*]，以该结点为根的子树有可能违反最大堆性质，对该子树递归调用 MAX-HEAPIFY;

保持堆性质

- **时间复杂度分析：**

当MAX-HEAPIFY作用在一棵以结点i为根的、大小为n的子树上时，其运行时间为调整元素A[i]、A[LEFT(i)]和A[RIGHT(i)]的关系时所用时间Θ(1)，再加上对以i的某个子节点为根的子树递归调用MAX-HEAPIFY所需的时间。i结点的子树大小最多为$2n/3$（此时，最底层恰好半满），运行时间递归表达式为：

$$T(n) \le T(2n/3) + \theta(1)$$

根据主定理，该递归式的解为 $T(n) = O(\lg n)$

即：MAX-HEAPIFY作用于一个高度为h的结点所需的运行时间为$O(h)$

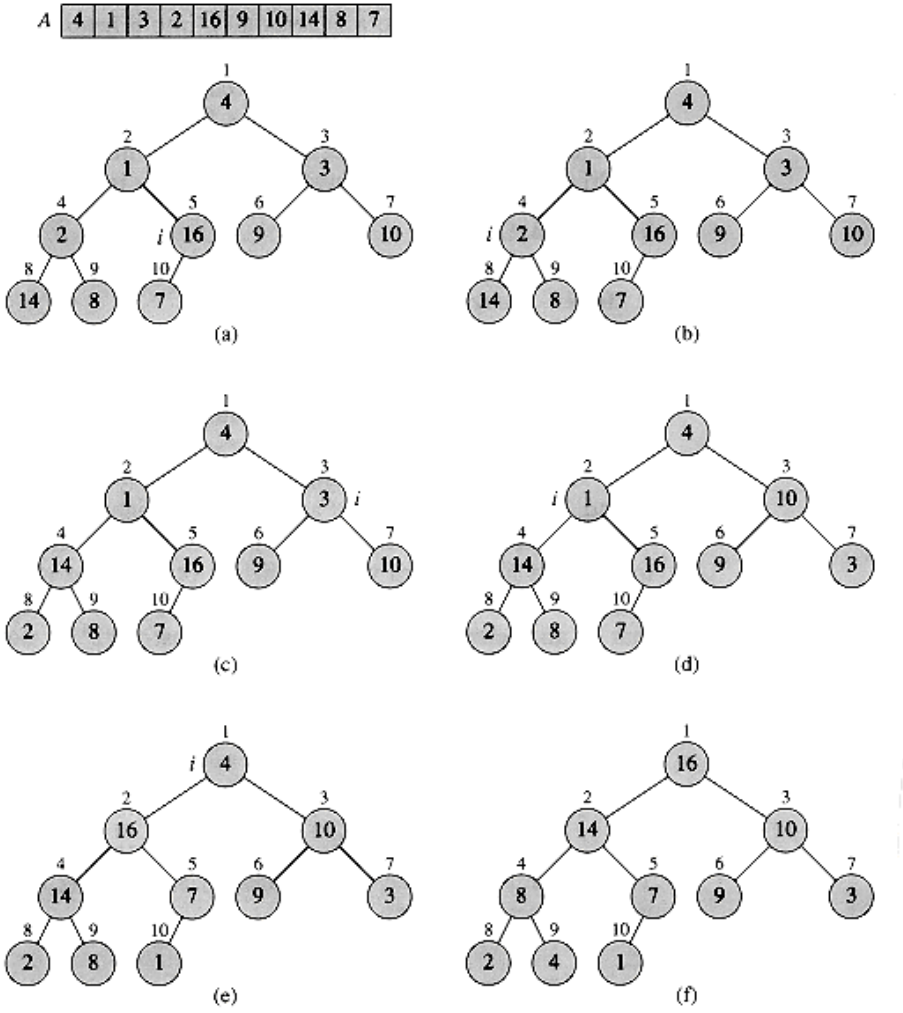
建堆操作

- 输入是一个无序数组A，BUILD-MAX-HEAP把数组A变成一个最大堆，伪代码如下：

```
BUILD-MAX-HEAP ( A )
1   heap-size[A] ← length[A];
2for i ← FLOOR( length[A]/2 )  downto 1
3       do MAX-HEAPIFY( A, i )
```

- **基本思想**：数组A[(*n*/2 +1) … *n*]中的元素都是树中的叶子结点，因此每个都可以看作是只含一个元素的堆。BUILD-MAX-HEAP对数组中每一个其它内结点从后往前都调用一次MAX-HEAPIFY。

建堆操作



建堆操作

- **时间复杂度分析：**

在树中不同高度的结点处运行MAX-HEAPIFY的时间不同，其作用在高度为h的结点上的运行时间为O(h)，故BUILD-MAX-HEAP时间代价为：

$$T(n) \le \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^{h+1}} \right)$$

$$\le O\left(n \sum_{h=0}^{\infty} \frac{h}{2^{h+1}} \right) = O(n)$$

这说明，BUILD-MAX-HEAP可以在线性时间内，将一个无序数组建成一个最大堆。

堆排序算法

基本思想：

① 调用BUILD-MAX-HEAP将输入数组A[1…n]构建成一个最大堆；

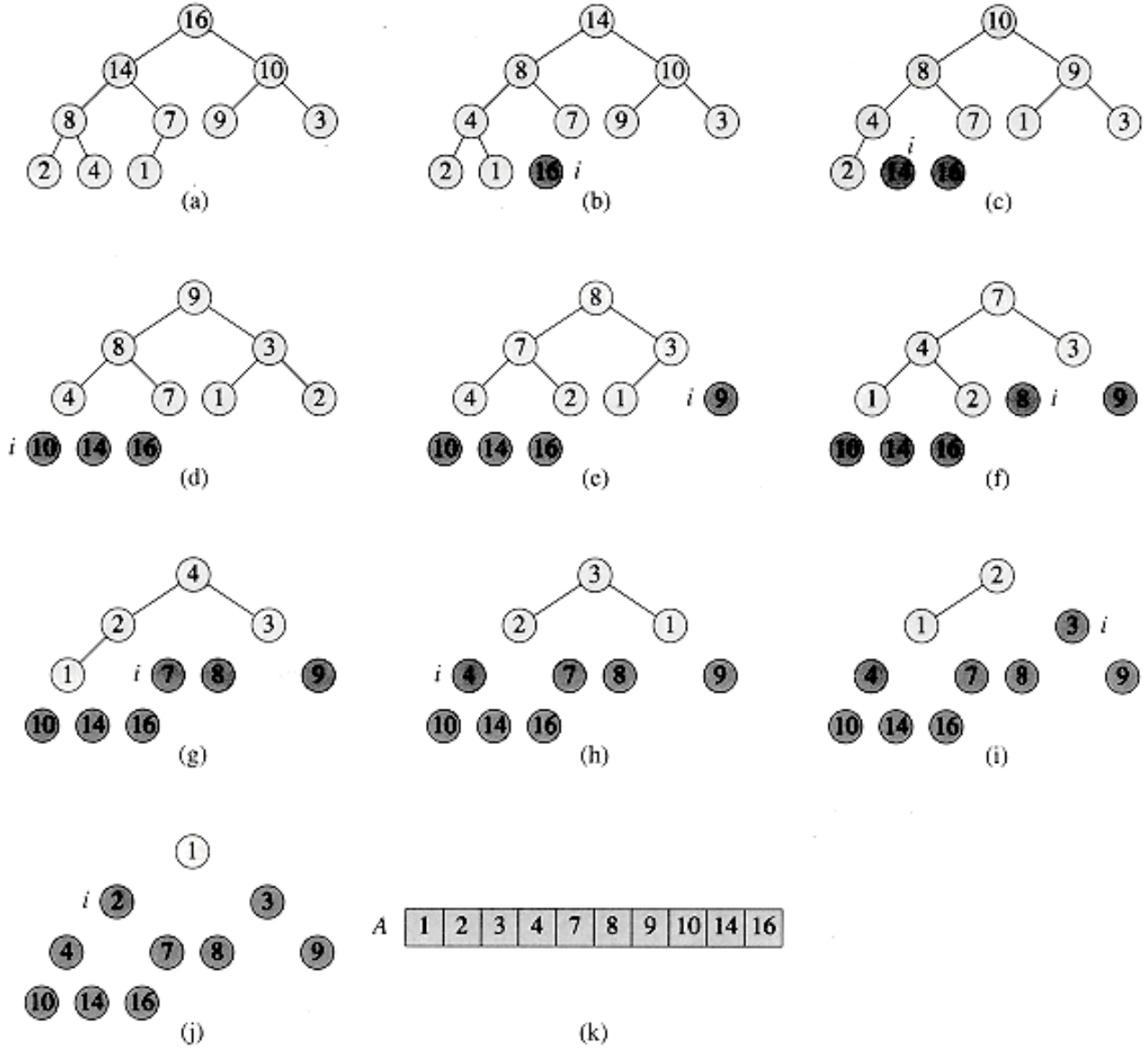
② 互置A[1]和A[n]位置，使得堆的最大值位于数组正确位置；

③ 减小堆的规模；

④ 重新调整堆，保持最大堆性质。

```
HEAPSORT( A )
1  BUILD-MAX-HEAP(A)
2  for  i ← length[A] downto 2
3     do  exchange A[1] ↔ A[i]
4         heap-size[A] ← heap-size[A] -1
5         MAX-HEAPIFY( A, 1)
```

堆排序算法



堆排序算法

- **时间复杂度分析：**

  调用BUILD-MAX-HEAP时间为$O(n)$, n-1次MAX-HAPIFY调用的每一次时间代价为$O(lgn)$。

  HEAPSORT过程的总时间代价为：$O(nlgn)$。

思考：堆排序算法与插入排序算法设计策略关系是否类似？（减治法）

优先级队列

- 优先级队列是一种用来维护由一组元素构成的集合S的数据结构，这一组元素中的每一个都有一个关键字Key。
- 一个最大优先级队列支持以下操作：
  ① INSERT(S, x)：把元素x插入集合S中；
  ② MAXIMUM(S)：返回S中具有最大关键字的元素；
  ③ EXTRACT-MAX(S)：去掉并返回S中的具有最大关键字的元素；
  ④ INCREASE-KEY(S, x, k)：将元素x的关键字的值增加到k，这里k值不能小于x的原始关键字的值。
- 最大优先级队列经常被用于分时计算机上的作业调度

优先级队列

- HEAP-MAXIMUN

```
HEAP-MAXIMUM( A )
1  return A[1]
```

- HEAP-EXTRACT-MAX，运行时间为O(*lgn*)

```
HEAP-EXTRACT-MAX( A )
1if  heap-size[A] < 1
2      then error “heap underflow”
3max ← A[1]
4A[1] ← A[heap-size[A]]
5heap-size[A] ← heap-size[A] – 1;
6MAX-HEAPIFY( A, 1 )
7return max
```

优先级队列

- HEAP-INCREASE-KEY：运行时间为O(*lg*n)

```
HEAP-INCREASE-KEY(A, i, key )
1if  key < A[i]
2      then error "new key is smaller than current key"
3A[ i ] ← key
4while  i > 1 and A[PARANT(i)] < A[i]
5      do exchange A[i ] ↔ A[PARANT(i)]
6           i ← PARANT(i)
```

- MAX-HEAP-INSERT: 运行时间为O(lgn)

```
MAX-HEAP-INSERT(A, key )
1heap-size[A] ←  heap-size[A] + 1
2A[heap-size[A]] ← -∞
3HEAP-INCREASE-KEY( A, heap-size[A], key )
```

第六讲 排序

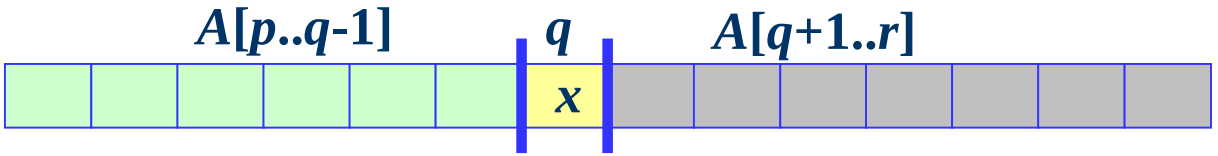
内容提要：

- 排序问题
- 堆排序算法
- 快速排序算法
- 线性时间排序
- 排序算法比较

快速排序算法

- 快速排序是C.R.A.Hoare于1962年提出的一种地排序算法。对于包含n个数的输入数组，最坏情况运行时间为$\Theta(n^2)$，期望运行时间为$\Theta(n\lg n)$且常数因子较小。
- 基本思想是采用了一种分治的策略把未排序数组分为两部分，然后分别递归调用自身进行排序：

① **分解**：数组A[p…r]被划分为两个（可能空）子数组A[p…q-1]和A[q+1..r]，使得A[p…q-1]中每个元素都小于或等于A[q]和A[q+1..r]中的元素。下标q在这个划分过程中进行计算；



② **解决**：递归调用快速排序，对子数组A[p...q-1]和A[q+1..r]排序；

③ **合并**：不需要任何操作。

快速排序算法

- 快速排序伪代码：

```
QUICKSORT(A, p, r )
1if   p < r
2     then  q ← PARTITION(A, p, r )
3           QUICKSORT( A, p, q-1 )
4           QUICKSORT( A, q+1, r )
```

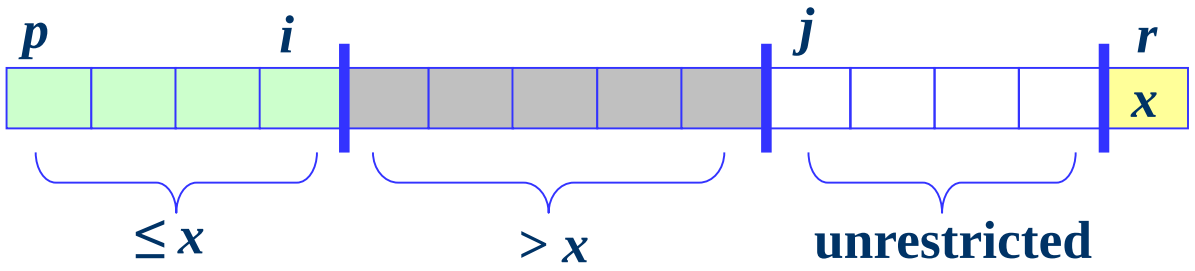
* 为排序一个完整数组，最初调用 QUICKSORT(A, 1, *length*[A])。

- 数组划分过程PARTITION是QUICKSORT算法的关键，它对子数组A[p..r]进行就地排序。

快速排序算法

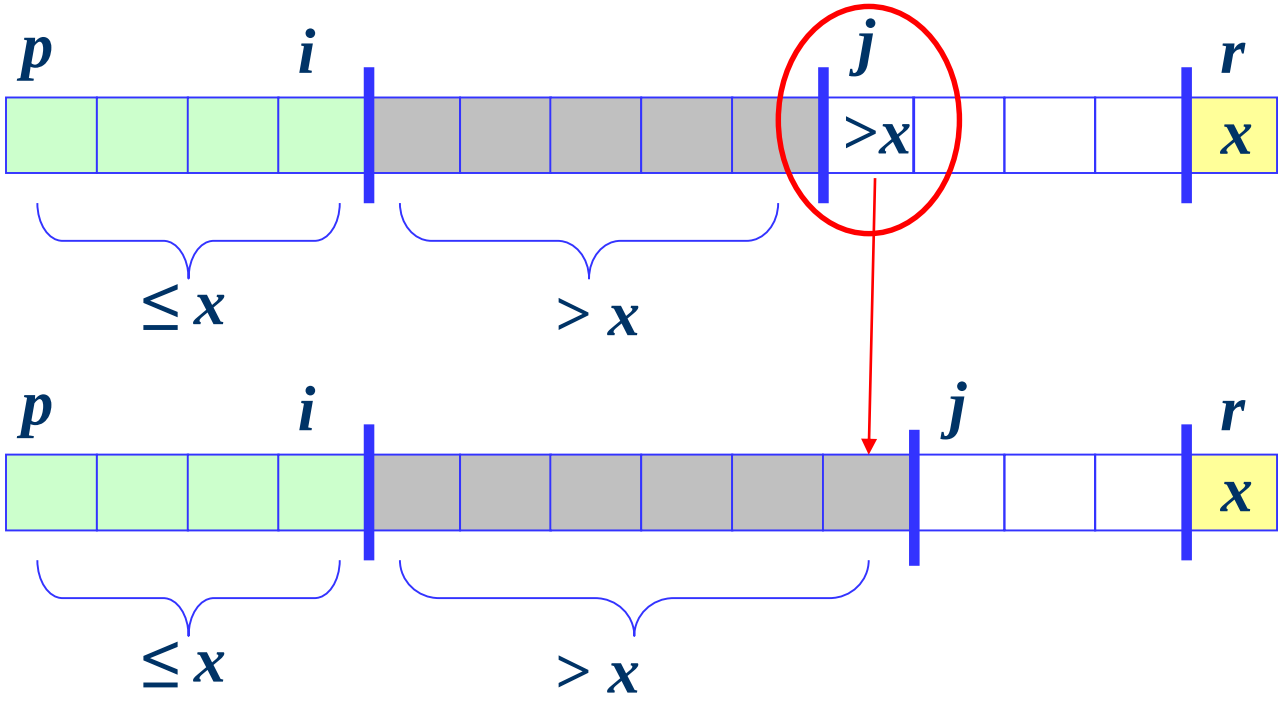
- 数组划分过程 PARTITION

```
PARTITION(A, p, r )
1  x ← A[ r ]          // x 为主元
2  i ← p - 1
3  for  j ← p to r – 1
4       do if  A[ j ] ≤ x
5             then  i ← i + 1
6                   exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```



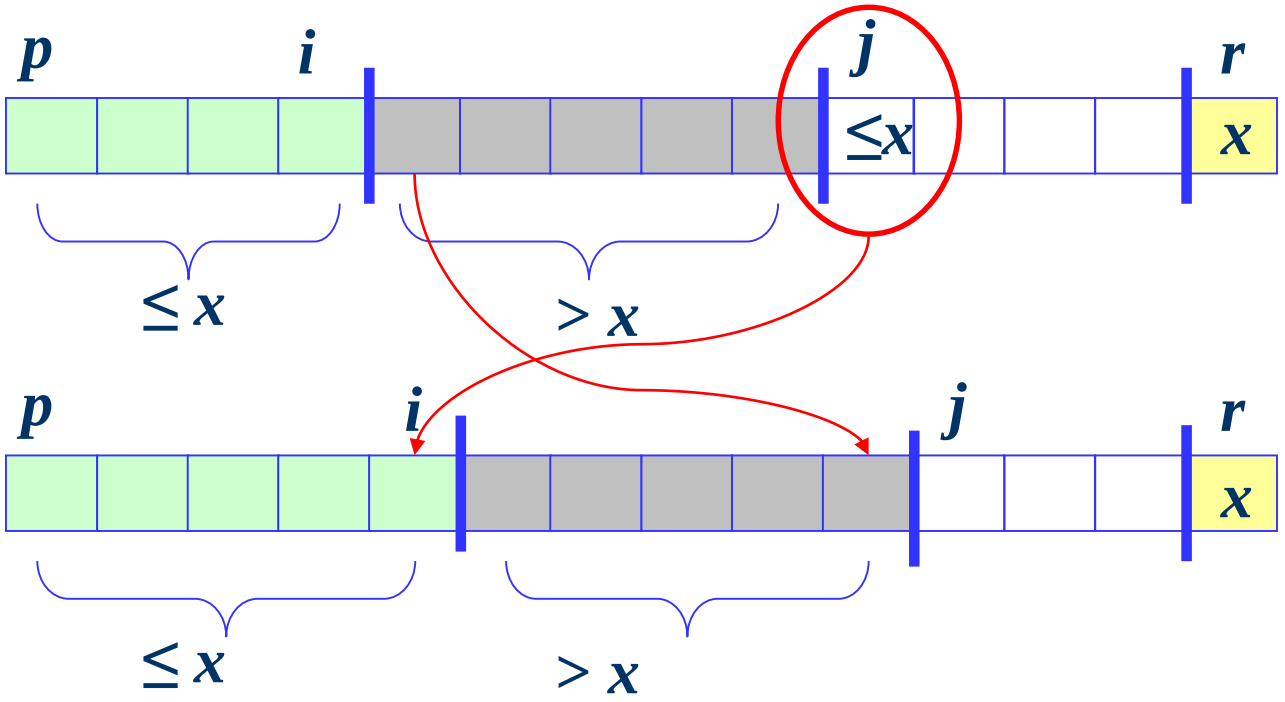
快速排序算法

i 和 *j* 如何改变：

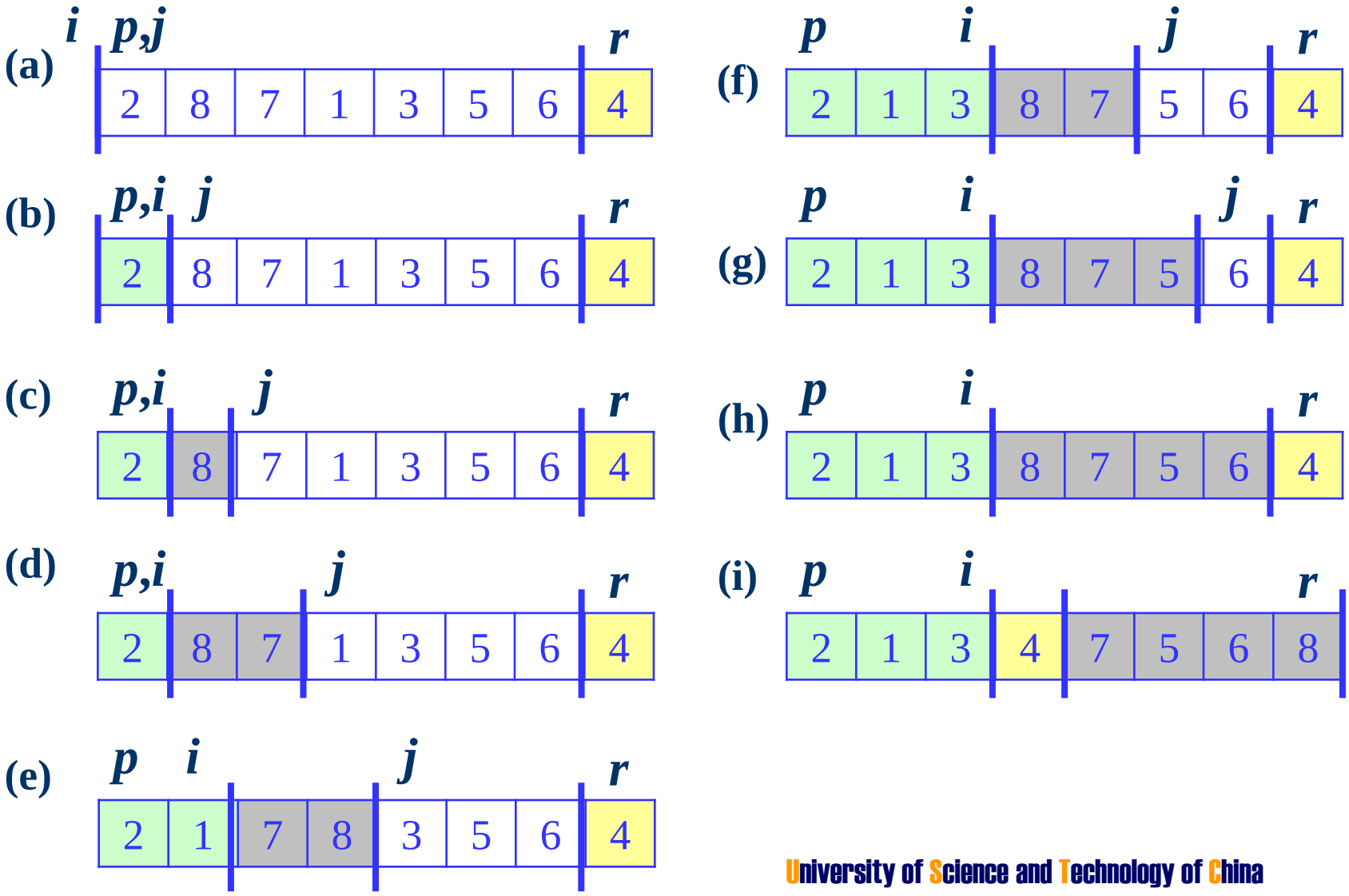


快速排序算法

i 和 *j* 如何改变:



范例: (Partition, $x=A[r]=4$)



快速排序算法

- 最坏情况：$\Theta(n^2)$ (對於已排序好的輸入)

$$T(n) = \max_{1 \le q \le n} \{T(q-1) + T(n-q)\} + \Theta(n)$$

$$= \max_{0 \le k \le n-1} \{T(k) + T(n-k-1)\} + \Theta(n)$$

猜测: $T(n) \le c\, n^2 = O(n^2)$

Substituting:

$$T(n) \le \max_{0 \le k \le n-1} \{ck^2 + c(n-k-1)^2\} + \Theta(n)$$

$$\le c \max_{0 \le k \le n-1} \{k^2 + (n-k-1)^2\} + \Theta(n)$$

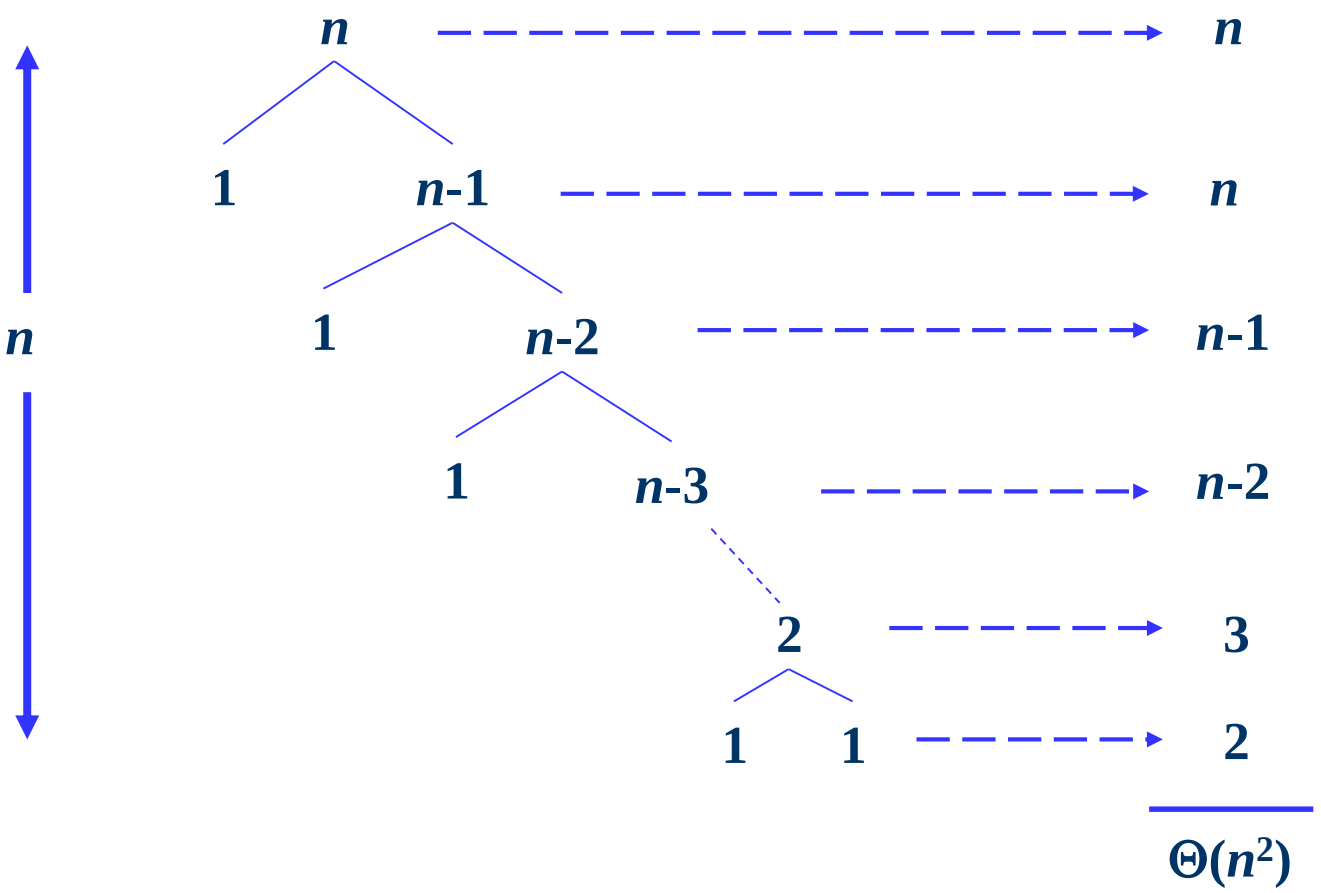
$$\le c(n\text{-}1)^2 + \Theta(n)$$

$$\le cn^2\text{-}c(2n\text{-}1) + \Theta(n)$$

$$\le cn^2 \text{ (挑选够大的 } c \text{ 即可)}$$

快速排序算法

- $T(n)=\Theta(n^2)$



快速排序算法

- 最佳情况划分： $O(n\lg n)$

  此时得到的两个子问题的大小都不可能大于$n/2$, 运行时间的递归表达式为：

$$T(n) \le 2\,T(\,n/2\,) + \Theta(\,n\,)$$

  根据主定理，该递归式的解为： $T(n) = O(\,n\lg n\,)$

- 如果以固定比例进行划分，即使该比例很不平衡（如100:1），则其运行时间仍然为$O(\,n\lg n\,)$。

快速排序算法

- 平均情况划分： $\Theta(n \lg n)$

假设所有元素都不相同，则$T(n)=O(n+X)$，X 是 Partition 中第四行的执行次数。

每次调用 Partition 的時候，如果 $A[i]<x<A[j]$ 或 $A[j]<x<A[i]$，$A[i]$ 和 $A[j]$ 将来就不会再相互比较.

快速排序算法

- 范例: 令 A={3，9，2，7，5}。 第一个回合之后，A={3，2，5，9，7}。 之后{3，2} 再也不会和 {9，7} 比较了。

- 将 A 的元素重新命名为 z_1，z_2，…，z_n，其中 z_i 是第 i 小的元素。且定义 Z_{ij}={z_i，z_{i+1}，…，z_j}为z_i与z_j之间的元素集合。

- 定义z_i : z_j ： 当且仅当第一个从 Z_{ij} 选出来的 pivot 是 z_i 或 z_j。

快速排序算法

- 对于任意的 i 和 j，發生 $z_i : z_j$ 的概率为$2/(j-i+1)$，因此，

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

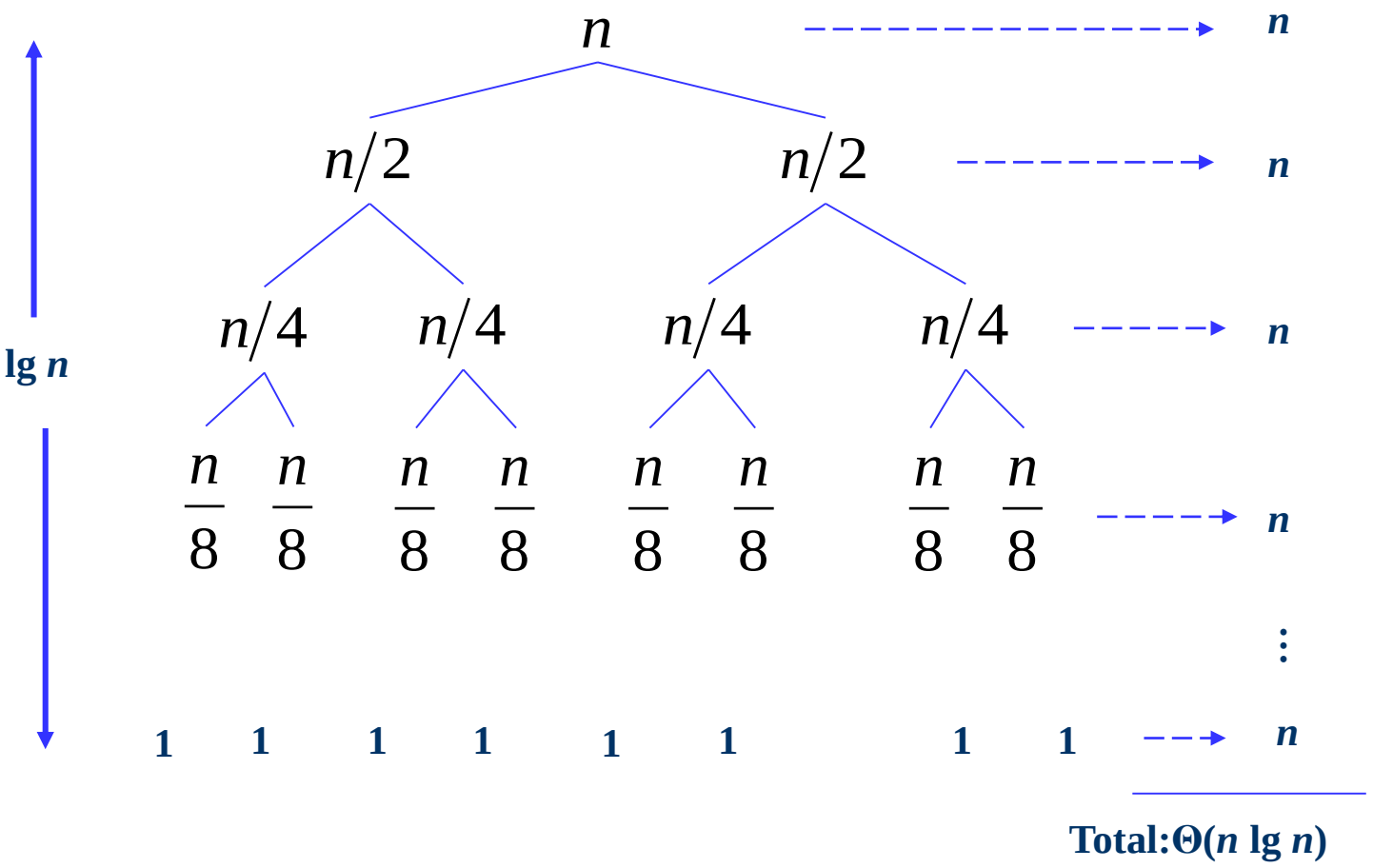
$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$ (套用 Harmonic Series)

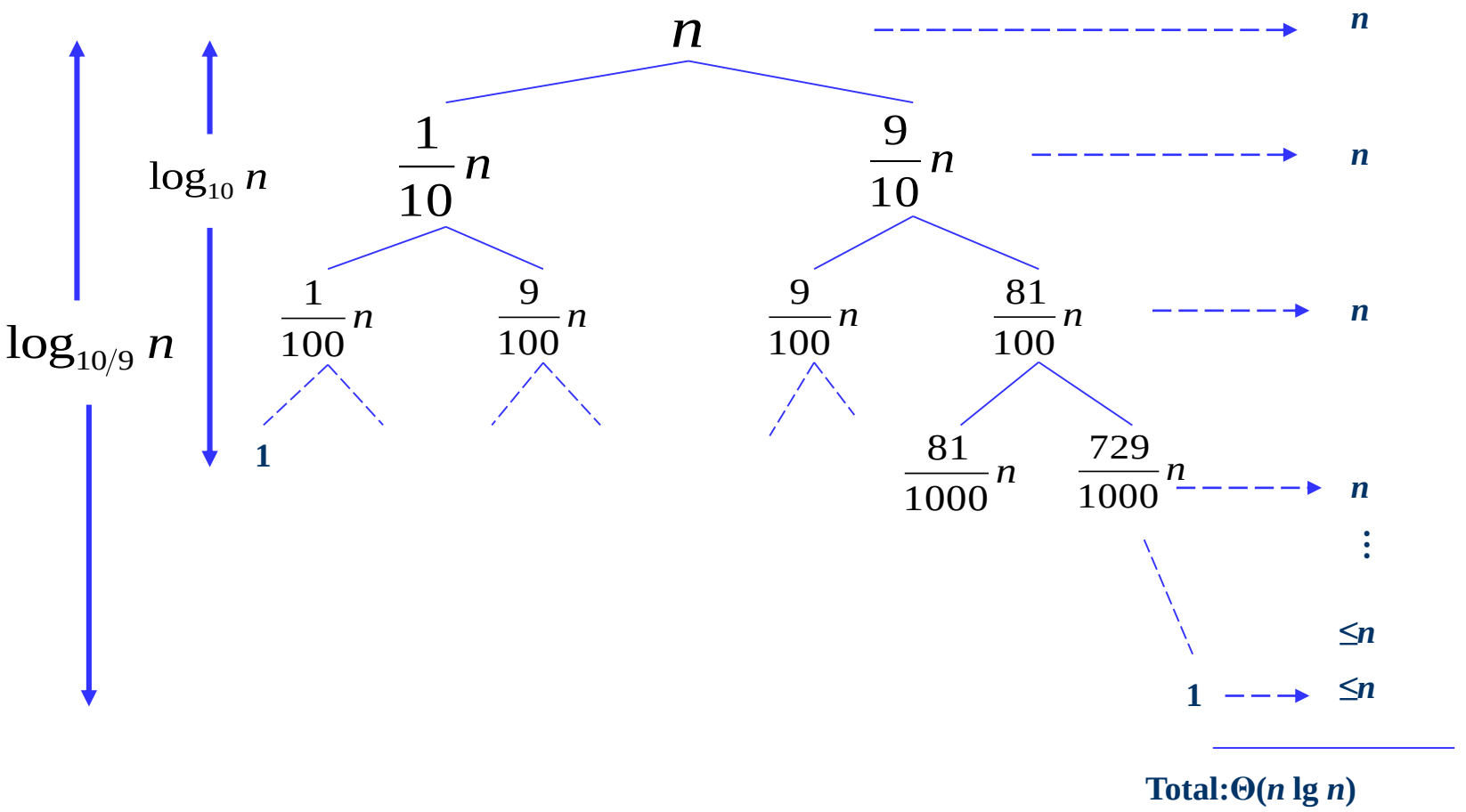
$$= \sum_{i=1}^{n-1} O(\lg n)$$

$$= O(n \lg n)$$

快速排序算法


n
n
n/2
n/2
n
n/4
n/4
n/4
n/4
n
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n
1
1
1
1
1
1
1
1
n
lg n
Total:Θ(n lg n)

快速排序算法


n
n
1/10 n
9/10 n
n
1/100 n
9/100 n
9/100 n
81/100 n
n
log10 n
log10/9 n
1
81/1000 n
729/1000 n
n
≤n
1
≤n
Total:Θ(n lg n)

快速排序算法

- 其他分析

$$E(n) = (n-1) + \frac{1}{n}\sum_{q=1}^{n}\{E(q-1)+E(n-q)\}$$

$$= (n-1) + \frac{2}{n}\sum_{k=1}^{n-1}E(k)$$

为了简单起见，假设：

$$E(n) = n+1 + \frac{2}{n}\sum_{k=1}^{n-1}E(k)$$

$$\Rightarrow nE(n) = n^2+n + 2\sum_{k=1}^{n-1}E(k) \quad \text{------(1)}$$

$$\Rightarrow (n\text{-}1)E(n\text{-}1) = (n-1)^2+(n-1)+2\sum_{k=1}^{n-2}E(k) \quad \text{------(2)}$$

(用 *n*-1 替换掉 (1) 裡面的 *n*)

快速排序算法

(1)-(2), 可得

$$nE(n) = (n+1)E(n-1)+2n$$

$$\Rightarrow E(n) = \frac{n+1}{n}E(n-1)+2 \quad \text{(套用 iteration method)}$$

$$= \frac{n+1}{n}\left\{\frac{n}{n-1}E(n-2)+2\right\}+2 = \frac{n+1}{n-1}E(n-2)+2\frac{n+1}{n}+2$$

$$= \frac{n+1}{n-2}E(n-3)+2\frac{n+1}{n-1}+2\frac{n+1}{n}+2 = \bullet \quad \bullet \quad \bullet \quad \bullet$$

$$= \frac{n+1}{2}E(1)+2(n+1)\left(\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n}\right)+2 = \Theta(n)+\Theta(n)\sum_{k=3}^{n}\frac{1}{k}+2$$

$$= \Theta(n)+\Theta(n)\times\Theta(\lg n)+2 \quad \text{(套用 Harmonic Series)}$$

$$= \Theta(n\lg n)$$

快速排序的随机化版本

- 如何防止出现最坏情况发生？

- **策略1**：显示地对输入进行排列使得快速排序算法随机化

```
RANDOMIZED-QUICKSORT(A, p, r )
1if   p < r
2     RANDOMIZE-IN-PLACE(A)
3     QUICKSORT( A )
```

- 可以达到目的，是否还有其它策略呢？

快速排序的随机化版本

- **策略2**：采用**随机取样（random sampling）**的随机化技术
- **做法**：从子数组$A[p\ldots r]$中随机选择一个元素作为主元，从而达到可以对输入数组的划分能够比较对称。

```
RANDOMIZED-PARTITION(A, p, r )
1  i ← RANDOM( p, r )
2  exchange A[r] ↔ A[i]
3   return PARTITION( A, p, r )
```

- 新排序算法调用RANDOMIZED-PARTITION

```
RANDOMIZED-QUICKSORT(A, p, r )
1if   p < r
2     then  q ← RANDOMIZED-PARTITION(A, p, r )
3           QUICKSORT( A, p, q-1 )
4           QUICKSORT( A, q+1, r )
```

第六讲 排序

内容提要：

- 排序问题
- 堆排序算法
- 快速排序算法
- 线性时间排序
- 排序算法比较

排序算法时间的下界

- 本节探讨排序所耗用的时间复杂度下限。
- 任何一个以比较为基础的排序算法，排序n个元素时至少耗用$\Omega(n\lg n)$次比较，时间复杂度至少为$\Omega(n\lg n)$；
- 但不使用比较为基础的排序算法，在某些情形下可以在$O(n)$的时间内执行完毕。

排序算法时间的下界

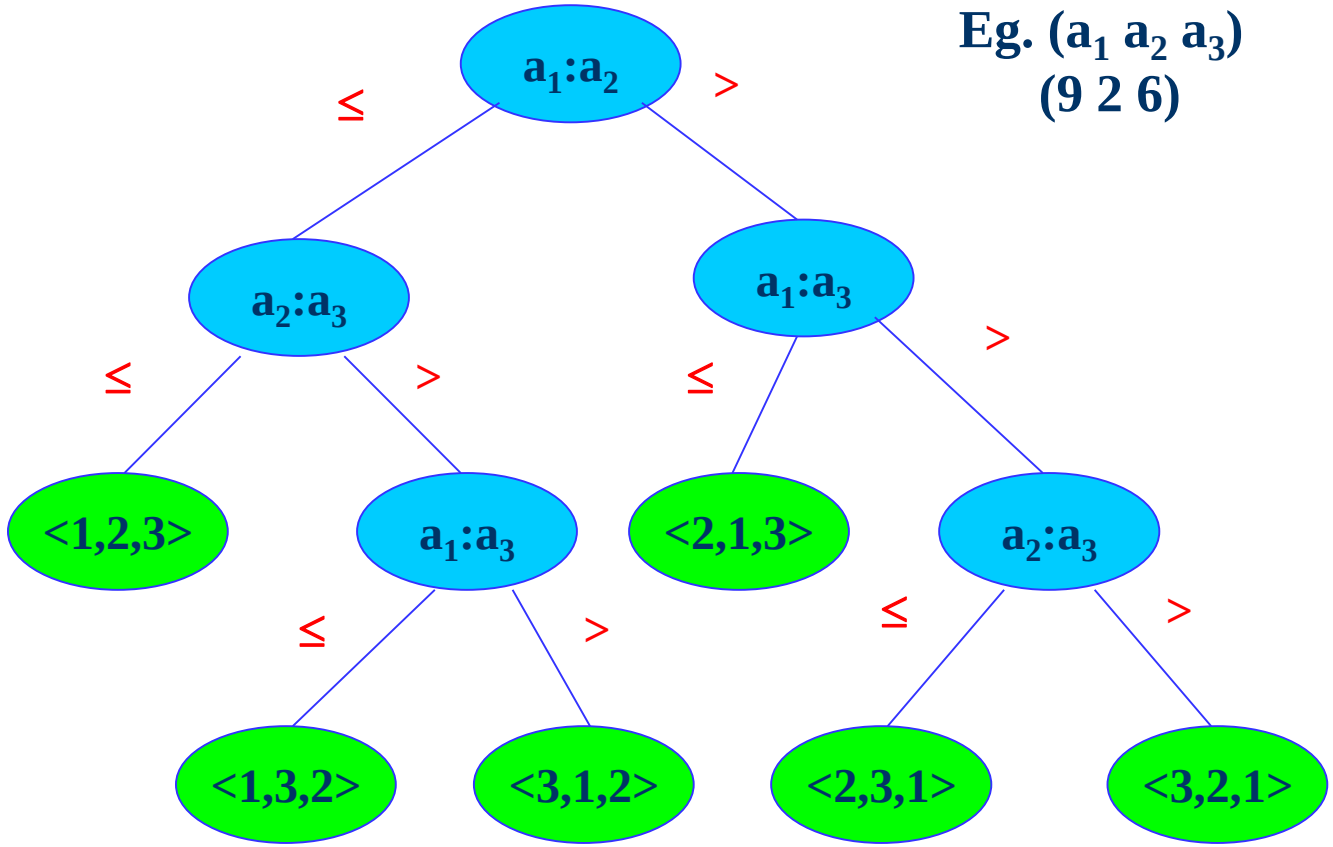
- 一个以元素比较为基础的排序算法可以按照比较的顺序建出一个决策树（Decision-Tree）。

- 决策树模型：

① 每一个从根节点到叶子结点的路径都代表一种排序结果。

② 任何一个以元素比较为基础排序n个元素的排序算法，所对应的决策树的高度至少有$\Omega(n\log n)$。

排序算法时间的下界



Eg. (a_1 a_2 a_3)
(9 2 6)

排序算法时间的下界

- 证明：因为可能有$n!$种可能的排序结果，故对应的Decision tree至少有$n!$个叶子结点。而高度为h的二叉树最多有2^h个叶子结点。因此$h \geq \log_2(n!) \geq \Theta(n\log n)$。(后者由斯特林公式得证：$n!>(n/e)^n$)

- 理论上分析，Heapsort与Mergesort是渐进最优的排序算法，而Quicksort并不是渐进最有的排序算法。但是，Quicksort其执行效率平均而言较Heapsort和Mergesort还好。

计数排序

- Counting Sort (计数排序) 不需要藉由比较來做排序。但是，必须依赖于一些对于待排序集合中元素性质的假设：
- 如果所有待排序元素均为整数，介于1到k之间。则当$k=O(n)$, 时间复杂度： $O(n+k)$
- 基本思想是：对每一个输入元素x,统计出小于x的元素的个数。然后，根据这一信息直接把元素x放到它在最终输出数组中的位置上。

计数排序

```
Input: A[1..n], where A[j]∈{1,2,…,k}
Output:B[1..n], sorted
CountingSort(A,B,k)
{  for i = 0 to k
       do C[i]←0
   for j = 1 to length[A]
       do C[A[j]]←C[A[j]]+1
    //C[i]包含等于i的元素个数
   for i = 1 to k
       do C[i]←C[i]+C[i-1]
    //C[i]包含小于或等于i的元素个数
   for j = length[A] downto 1
       do B[C[A[j]]]←A[j]
          C[A[j]]←C[A[j]]-1
}
```

计数排序

k=6

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

		1	2	3	4	5	6
2nd loop	C:	2	0	2	3	0	1

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

A:

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

计数排序

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

4th loop

1st iteration B:

1	2	3	4	5	6	7	8
						4	

C:

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

2nd iteration B:

1	2	3	4	5	6	7	8
	1					4	

C:

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

3rd iteration B:

1	2	3	4	5	6	7	8
	1				4	4	

C:

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

……

8th iteration B:

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

C:

1	2	3	4	5	6
0	2	2	4	7	7

排序算法是稳定的，经常被当做基数排序算法的一个子过程 。

基数排序（Radix Sort）

- Radix Sort(基数排序)无需利用元素间的比较排序，但是必须依赖一些对于待排序集合中元素性质的假设（通常假设所有待排序元素均为整数，至多*d*位)。

```
Radix-Sort(A,d)
{ for i = 1 to d
    do use stable sort to sort A on digit i
}
```

基数排序

- 关键想法：利用计数排序法由低位数排到高位数。

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

先排个位数

再排十位数

最后排百位数

基数排序

```
Radix-Sort(A,d)
{ for i = 1 to d
      do use stable sort to sort A on digit i
}
```

- 此处必须使用的稳定的排序算法，如果使用Counting Sort则每次迭代只需花$\Theta(n+10)$的时间。
- 因此总共花费$O(d(n+10))$的时间。
- 如果d是常数，则Radix Sort为一个可以在线性时间完成的排序算法。

基数排序

- 引理8.3：给定n个d位数，每一个数位可以取k种可能的值。基数排序算法能以$\Theta(d(n+k))$的时间正确地对这些数进行排序。

- 引理8.4: 给定n个b位数和任何正整数$r<b$, RADIX-SORT能在$\Theta((b/r)(n+2^r))$时间内正确地对这些数进行排序。
- 对于给定的n值和b值，我们希望所选择的r值能够最小化表达式$(b/r)(n+2^r)$。如果$b<floor(\lg n)$，选择r=b, 此时为$\Theta(n)$；如果$b >= floor(\lg n)$, 选择$r= floor(\lg n)$，此时$\Theta(bn/\lg n)$

桶排序（Bucket Sort）

- 当元素均匀分布在某个区间时，Bucket sort平均能在O(n)的时间完成排序。

- 基本思想：把区间[0,1)划分成n个相同大小的子区间（称为桶）。然后，将n个输入数分布到各个桶中去。先对桶中元素进行排序，然后依次把各桶中的元素列出来即可。

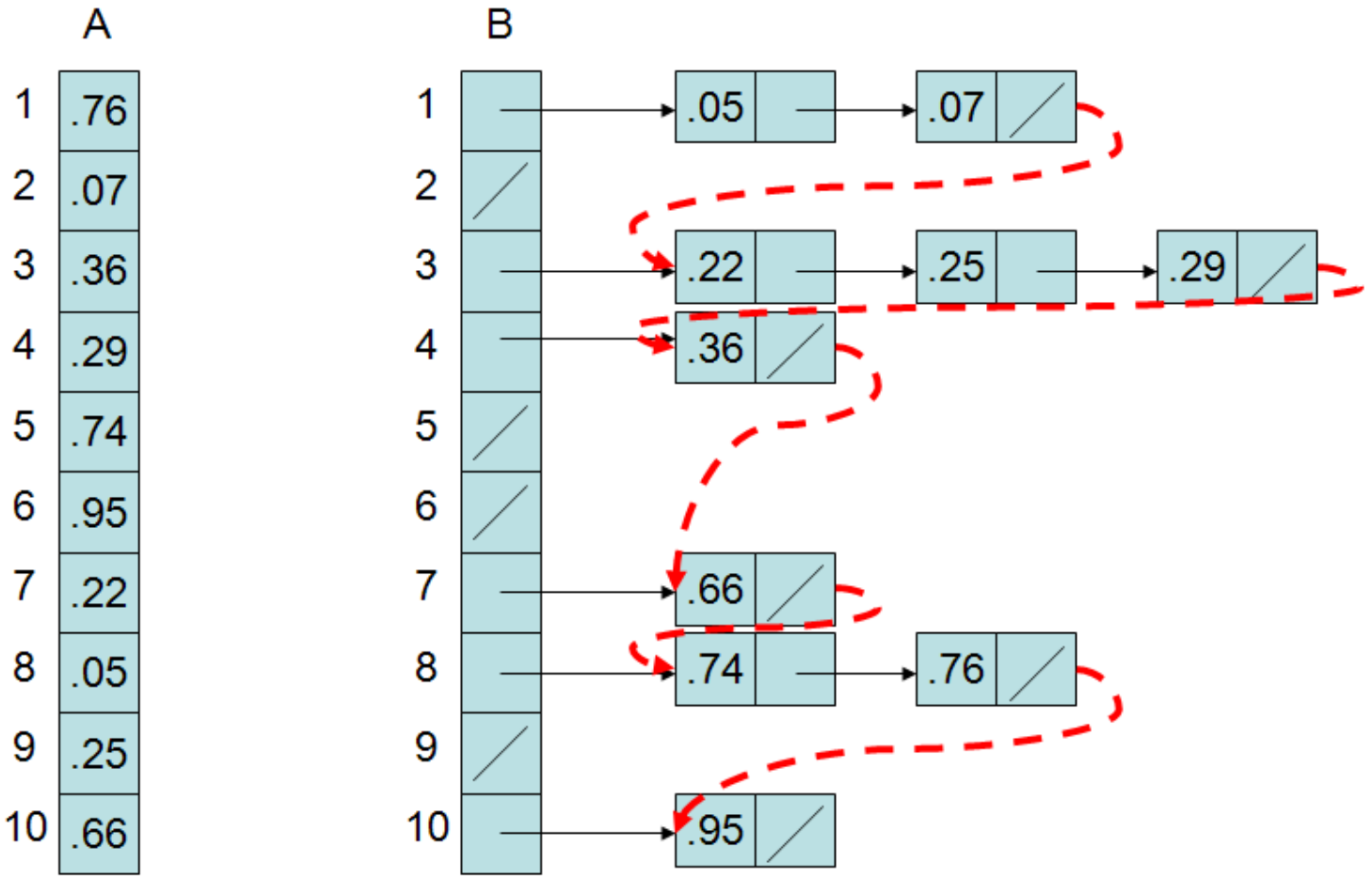
桶排序

- BUCKET-SORT(A)
- 1 n ← length[A]
- 2 for i ← 1 to n
- 3 do insert A[i] into list B[floor(nA(i))]
- 4 for i ← 0 to n-1
- 5 do sort list B[i] with insertion sort
- 6 concatenate the lists B[0], B[1], …, B[n-1] together in order

桶排序

- 假定要排序n個元素A[1..n]均是介于[0,1]之間的数值，桶排序步骤如下：
- 1）准备n个桶(bucket)，B[1..n]，将元素x依照x所在的区间放进对应的桶中：即第$\lceil xn \rceil$个桶 。
- 2）元素放进桶时，使用链表来存储，并利用插入排序法排序。
- 3）只要依序将链表串接起來，即得到已排序的n个元素。

桶排序


A
1 .76
2 .07
3 .36
4 .29
5 .74
6 .95
7 .22
8 .05
9 .25
10 .66
B
1 .05 .07
2
3 .22 .25 .29
4 .36
5
6
7 .66
8 .74 .76
9
10 .95

桶排序

时间复杂度分析：

- 假定分到第i个桶的元素个数是n_i。
- 最差情形：$T(n)=O(n)+\sum_{1\le i\le n}O(n_i^2)$
  $=O(n^2)$.

- 平均情形：$T(n)=O(n)+\sum_{1\le i\le n}O(E[n_i^2])$
  $=O(n)+\sum_{1\le i\le n}O(1)$
  $=O(n)$
- $E[n_i^2]=\Theta(1)$ 的证明请参考课本！

第六讲 排序

内容提要：

- 排序问题
- 堆排序算法
- 快速排序算法
- 线性时间排序
- 排序算法比较

二分法插入排序

- 特点：在直接插入排序的基础上减少比较的次数，即在插入R_i时改用二分法比较找插入位置，便得到二分法插入排序
- 限制：必须采用顺序存储方式。

二分法插入排序

例：有6个记录，前5个已排序的基础上，对第6个记录排序。

[15 27 36 53 69] 42

↑low ↑mid ↑high

(42>36)

[15 27 36 53 69] 42

↑low ↑high

↑mid

(42<53)

[15 27 36 53 69] 42

↑high ↑low

[15 27 36 42 53 69]

(high<low，查找结束，插入位置为low或high+1)

二分法插入排序

```
void  binSort(SortObject * pvector)
{
    int i, j, left, mid, right;
    RecordNode temp;
    for( i = 1; i < pvector->n; i++ )
        {
            temp = pvector- >record[i];
            left = 0;  right = i – 1;
            while (left <= right)
            {
               mid = (left+right)/2;
               if (temp.key < vector->record[mid].key)
                   right = mid-1;
            else
                   left = mid+1;
        }//while
        for(j=i-1; j>=left;  j--)
              pvector->record[j+1] =
                    pvector->record[j];
              if(left != i)
         pvector->record[left] = temp;
    }// for
}  // binSort
```

二分法插入排序

比较次数：

- 二分插入排序的比较次数与待排序记录的初始状态无关，仅依赖于记录的个数，插入第i个记录时，如果 $i = 2^j (0 \le j \le \lfloor \log_2 n \rfloor$ ，则无论排序码的大小，都恰好经过 $j = \log_2 i$ 次比较才能确定插入位置，如果，$2^j < i \le 2^{j+1}$ 则比较次数为j+1，因此，将n($n=2^k$)个记录排序的总比较次数为

$$\sum_{i=1}^{n} \lceil \log_2 i \rceil = 0 + 1 + 2 + 2 + \ldots + k + k + \ldots + k$$

$$= n \log_2 n - n + 1 \approx n \log_2 n$$

二分法插入排序

性能分析：

- 当n较大时，比直接插入排序的最大比较次数少得多。但大于直接插入排序的最小比较次数
- 算法的移动次数与直接插入排序算法的相同
  - 最坏的情况为$n^2/2$
  - 最好的情况为n
  - 平均移动次数为$O(n^2)$
  - 二分法插入排序算法的平均时间复杂度为$T(n)= O(n^2)$
- 二分插入排序法是稳定的排序算法,在检索时采用left>right结束，left、right的修改原则是：temp.key < pvector->record[mid].key，保证排序是稳定的。

二分法插入排序

结论：

- 移动次数与直接插入排序相同，最坏的情况为$n^2/2$，最好的情况为n，平均移动次数为$O(n^2)$
- 二分法插入排序算法的平均时间复杂度为

  $T(n)= O(n^2)$
- 二分法插入排序是稳定的

表插入排序

- 表插入排序是在直接插入排序的基础上减少移动的次数。
- 基本思想:
  - 在记录中设置一个指针字段，记录用链表连接
  - 插入记录R_i时，记录R_0至R_{i-1}已经排序，先将记录R_i脱链
  - 再采用顺序比较的方法找到R_i应插入的位置，将R_i插入链表。

表插入排序

记录的数据结构：

```
struct Node;        /* 单链表结点类型 */
typedef struct Node ListNode;
struct Node
{   KeyType key;    /* 排序码字段 */
    DataType info; /* 记录的其它字段 */
    ListNode *next;       /* 记录的指针字段 */
};
typedef ListNode * LinkList;
```

表插入排序

算法性能分析：

第i趟排序：最多比较次数i次，最少比较次数1次。

n-1趟总的比较次数：

最多：$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

最少：n-1

记录移动次数：0

时间效率：$0(n^2)$

辅助空间：0(n)　[指针]

稳定性：p->key <= now->key保证稳定的排序。

冒泡排序

- 方法
  - ① 先将序列中的第一个记录R_0与第二个记录R_1比较，若前者大于后者，则两个记录交换位置，否则不交换
  - ② 然后对新的第二个记录R_1与第三个记录R_2作同样的处理
  - ③ 依次类推，直到处理完第n-1个记录和第n个记录
  - 从(R_0，R_1)到(R_{n-2}，R_{n-1})的n-1次比较和交换过程称为一次起泡
  - 经过这次起泡，n个记录中最大者被安置在第n个位置上

冒泡排序

④ 此后，再对前n-1个记录进行同样处理，使n-1个记录的最大者被安置在整个序列的第n-1个位置上。

⑤ 然后再对前n-2个记录重复上述过程.....，这样最多做n-1次起泡就能完成排序

- 可以设置一个标志noswap表示本次起泡是否有记录交换，如果没有交换则表示整个排序过程完成
- 起泡排序是通过相邻记录之间的比较与交换，使值较大的记录逐步从前(上)向后(下)移，值较小的记录逐步从后(下)向前(上)移，就像水底的气泡一样向上冒，故称为起泡排序

冒泡排序

算法评价：

- 若文件初状为正序，则一趟起泡就可完成排序，排序码的比较次数为n-1，且没有记录移动，时间复杂度是O(n)
- 若文件初态为逆序，则需要n-1趟起泡，每趟进行n-i次排序码的比较，且每次比较都移动三次，比较和移动次数均达到最大值：

$$C_{\max} = \sum_{i=1}^{n-1}(n-i) = n(n-1)/2 = O(n^2)$$

$$M_{\max} = \sum_{i=1}^{n-1}3(n-i) = 3n(n-1)/2 = O(n^2)$$

冒泡排序

算法评价：

- 起泡排序最好时间复杂度是O(n)
- 起泡排序最坏时间复杂度为$O(n^2)$
- 起泡排序平均时间复杂度为$O(n^2)$
- 起泡排序算法中增加一个辅助空间temp，辅助空间为S(n)=O(1)
- 起泡排序是稳定的

各种排序算法评价

- 排序算法之间的比较主要考虑以下几个方面：
  - ✓ 算法的时间复杂度
  - ✓ 算法的辅助空间
  - ✓ 排序的稳定性
  - ✓ 算法结构的复杂性
  - ✓ 参加排序的数据的规模
  - ✓ 排序码的初始状态

各种排序算法评价

- 当数据规模n较小时，n^2和$n\log_2 n$的差别不大，则采用简单的排序方法比较合适
  - ✓ 如直接插入排序或直接选择排序等
  - ✓ 由于直接插入排序法所需记录的移动较多，当对空间的要求不多时，可以采用表插入排序法减少记录的移动
- 当文件的初态已基本有序时，可选择简单的排序方法
  - ✓ 如直接插入排序或起泡排序等

各种排序算法评价

- 当数据规模n较大时，应选用速度快的排序算法
  - 快速排序法最快，被认为是目前基于比较的排序方法中最好的方法
  - 当待排序的记录是随机分布时，快速排序的平均时间最短。但快速排序有可能出现最坏情况，则快速排序算法的时间复杂度为$O(n^2)$，且递归深度为n，即所需栈空间为O(n)

谢谢！

Q & A