算法设计与分析

主讲人：曹自强

Email: *zqcao@suda.edu.cn*

苏州大学 计算机学院







第五讲 排序算法

内容提要：

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

排序问题

- **问题描述：**

  输入：n个数的序列 $<a_1,a_2,\ldots,a_n>$

  输出：输入序列的一个重排 $<a_1',a_2',\ldots,a_n'>$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 有些应用程序本身就需对信息进行排序；
  ② 应用广泛，是许多算法的关键步骤；
  ③ 已有很多成熟算法，它们采用各种技术，具有历史意义；
  ④ 可以证明其非平凡下界，是渐近最优的；
  ⑤ 可通过排序过程的下界来证明其他一些问题的下界；
  ⑥ 在实现过程中经常伴随着许多工程问题出现（主机存储器层次结构、软件环境等）

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。







第四讲排序算法

内容提要：

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

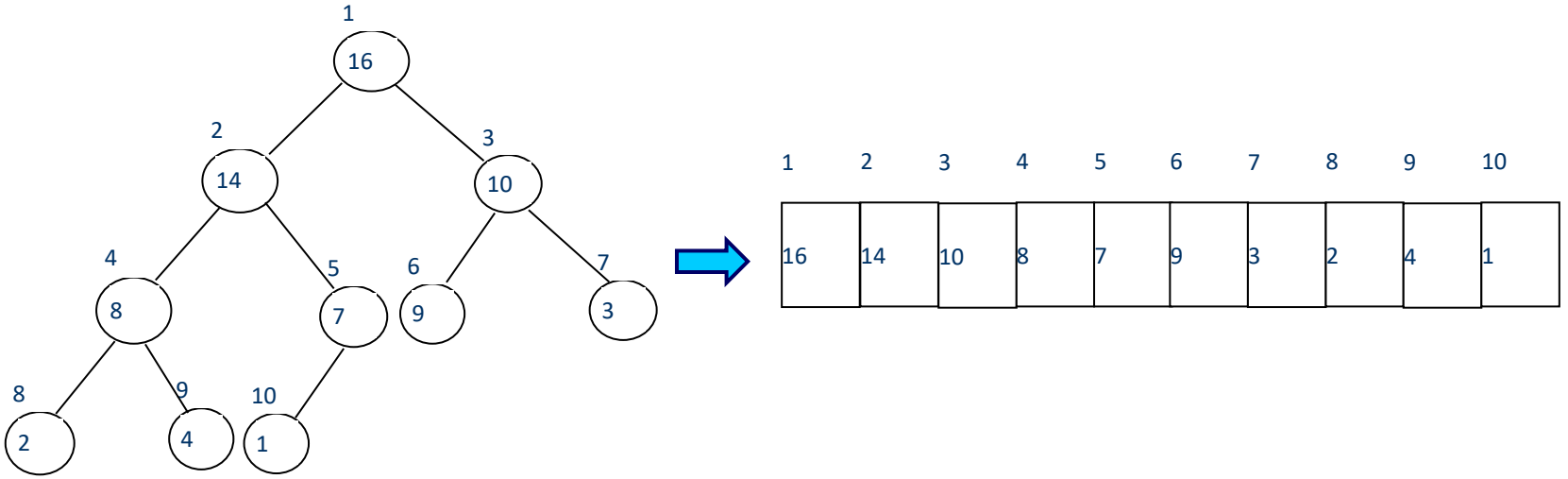
堆排序

- 堆排序（heapsort）结合了插入排序与归并排序的优点。
  - ✓运行时间与归并排序一致：$O(n \log n)$
  - ✓和插入排序都是一种原址排序算法：在排序输入数组时，只有常数个元素存储在输入数组之外。

- 堆排序引入了一种算法设计技术：利用了一种重要的数据结构——“堆”——来管理算法执行中的信息。

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。树的每一层都是填满的，最后一层可能除外（从一个结点的左子树开始填）



- 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； *heap-size*[A]: 是存放在A中的堆的元素个数；

② *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，右儿子为RIGHT(i) = 2 i + 1;

- 堆的分类：

➢ **最大堆**：除根节点之外的每个节点i，有 $A[PARENT(i)] \geq A[i]$

即某结点的值不大于其父结点的值，故堆中的最大元素存放在根结点中。

➢ **最小堆**：除根节点之外的每个节点i，有 $A[PARENT(i)] \leq A[i]$

即某结点的值不小于其父结点的值，故堆中的最小元素存放在根结点中。

- 在堆排序算法中，我们使用大根堆，堆中最大元素位于树根；
- 最小堆通常在构造优先队列时使用。

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$\Theta(\log n)$；
- 堆结构的基本操作：
  - ✓ MAX-HEAPIFY，运行时间为$O(\log n)$，保持最大堆性质；
  - ✓ BUILD-MAX-HEAP，以线性时间运行，可以在无序的输入数组基础上构造出最大堆；
  - ✓ HEAPSORT，运行时间为$O(n \log n)$，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$O(\log n)$, 可以让堆结构作为优先队列使用；

保持堆的性质

- MAX-HEAPIFY 函数的输入为一个数组A和下标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”，使以i为根的子树成为最大堆。

保持堆的性质

- **基本思想：**

1）找出A[*i*], A[LEFT(*i*)]和A[RIGHT(*i*)]三个元素中最大者，将其下标存在*largest*；

2）如果A[i]是最大的，则以i 为根的子树已是最大堆，程序结束。

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

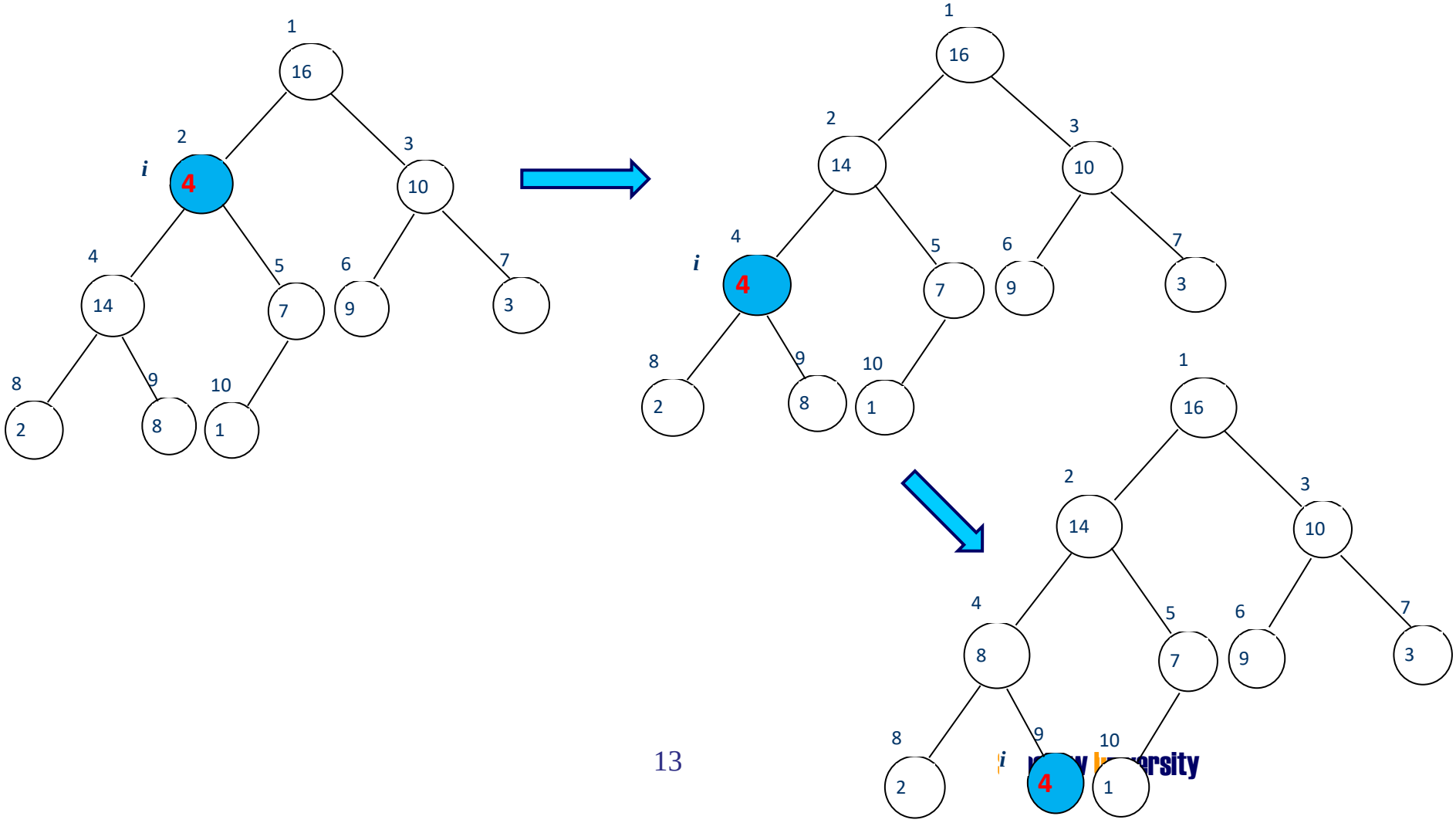
4）下标为*largest*的结点在交换后的值是A[*i*]，以该结点为根的子树有可能违反最大堆性质，对该子树递归调用MAX-HEAPIFY;

保持堆的性质

- MAX-HEAPIFY函数的输入为一个数组A和下标 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 )
```

保持堆的性质


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

- **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$T(n) = O(\log n)$

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

- 最坏情况发生在树的最底层恰好半满的时候
  - 第i层上的结点数目为 2^i (i=0, 1, 2,h)
  - 最后一层之上（不包含最后一层）包含 2^h -1 个结点(1);
  - 最底层半满时，包含$2^h/2$个结点 (2);
  - (1)+(2)=n
    - 2^h -1+ $2^h/2$=n
    - 2^h=2/3*n+ 2/3
  - 根节点的左子树个数有
    - $\frac{(1)-1}{2}+(2)$= 2^{h-1} -1+ $2^h/2$

      = 2^h-1= 2/3*n-1/3

      ≤ 2/3*n

?

递归式刻画运行时间：

$$T(n) \le T\left(\frac{2n}{3}\right)+\Theta(1)$$

根据主定理：

$n^{\log_b a} = n^{\log_{3/2} 1} = 1$

f(x)= $\Theta(1)$

满足情况2，

$T(n) = n^{\log_b a}\mathrm{O}(\log n)$

$= \mathrm{O}(\log n)$

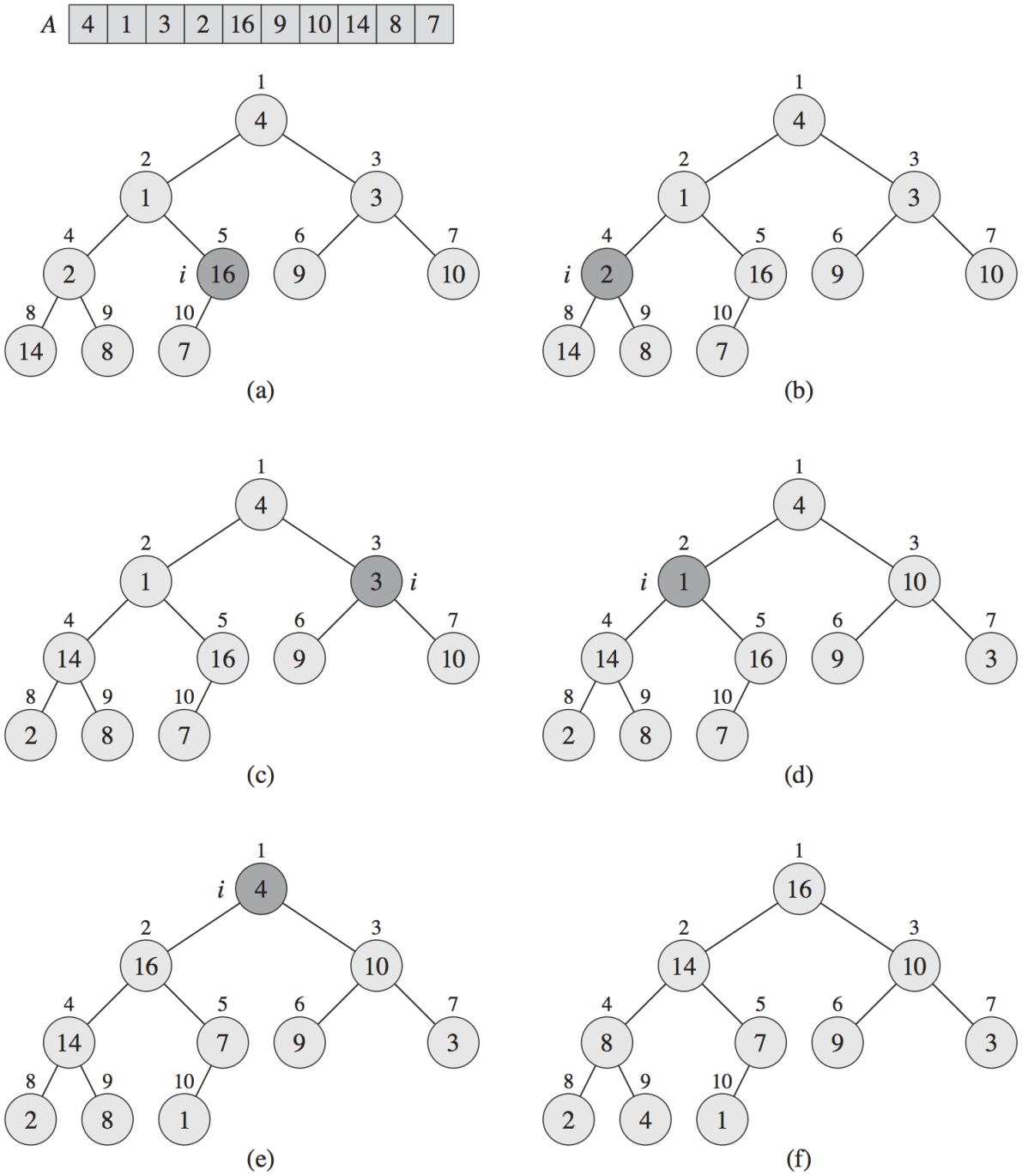
因为最大堆是一个类似完全二叉树的数组，左右子树的深度最多相差1，所以在子树的深度较深的一边时执行的次数可能越多。递归的时候，二分成左子树和右子树，当树的最底层半满时，左子树最大程度的占据树的比例，所以此时是最坏情况。

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（自底向上），伪代码如下：

```
BUILD-MAX-HEAP ( A )
1  heap-size[A] ← length[A];
2  for 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。


A 4 1 3 2 16 9 10 14 8 7
(a)
(b)
(c)
(d)
(e)
(f)

为了证明 BUILD-MAX-HEAP 的正确性，我们使用如下的循环不变量：

在第 2～3 行中每一次 **for** 循环的开始，结点 $i+1$，$i+2$，…，n 都是一个最大堆的根结点。

我们需要证明这一不变量在第一次循环前为真，并且每次循环迭代都维持不变。当循环结束时，这一不变量可以用于证明正确性。

初始化：在第一次循环迭代之前，$i=\lfloor n/2 \rfloor$，而$\lfloor n/2 \rfloor+1$，$\lfloor n/2 \rfloor+2$，…，n 都是叶结点，因而是平凡最大堆的根结点。

保持：为了看到每次迭代都维护这个循环不变量，注意到结点 i 的孩子结点的下标均比 i 大。所以根据循环不变量，它们都是最大堆的根。这也是调用 MAX-HEAPIFY(A, i)使结点 i 成为一个最大堆的根的先决条件。而且，MAX-HEAPIFY 维护了结点 $i+1$，$i+2$，…，n 都是一个最大堆的根结点的性质。在 **for** 循环中递减 i 的值，为下一次循环重新建立循环不变量。

终止：过程终止时，$i=0$。根据循环不变量，每个结点 1，2，…，n 都是一个最大堆的根。特别需要指出的是，结点 1 就是最大的那个堆的根结点。

建堆操作

- **时间复杂度分析：**

  每次MAX-HEAPIFY的时间为$O(\log n)$，**总共进行了O(n)次MAX-HEAPIFY,** 时间复杂度为$O(n \log n)$

不是渐近紧确的！

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，
    - 包含n个结点元素的堆的高度为$\lfloor \log n \rfloor$；
    - 堆最多包含$\lceil n/2^{h+1} \rceil$个高度为h的结点；
    - 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：

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

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

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 .$$

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：

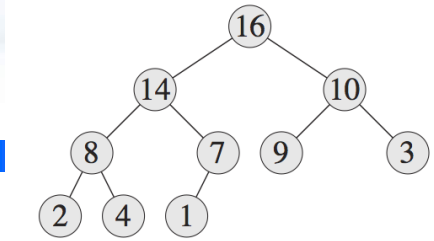
① 调用BUILD-MAX-HEAP将输入数组A[1…n]构建成一个最大堆；

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；

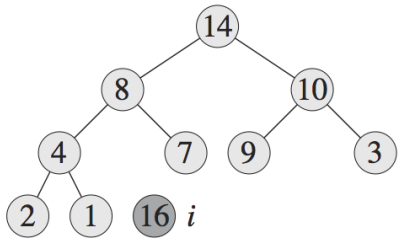
③ 减小堆的规模；

④ 重新调整堆，保持最大堆性质。

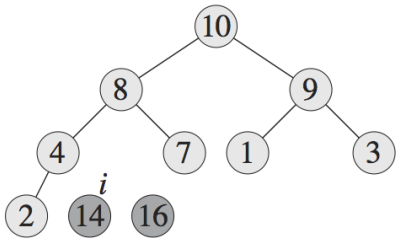
```
HEAPSORT( A )
1   BUILD-MAX-HEAP(A)
2   for  i ← length[A] downto 2
3      do  exchange A[1] ⇔ A[i]
4          heap-size[A] ← heap-size[A] -1
5          MAX-HEAPIFY( A, 1)
```



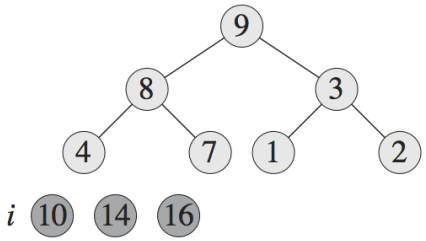
(a)



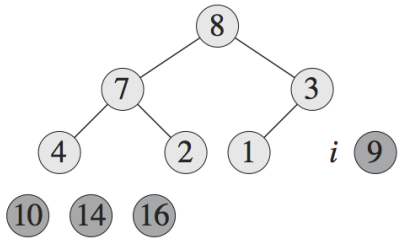
(b)



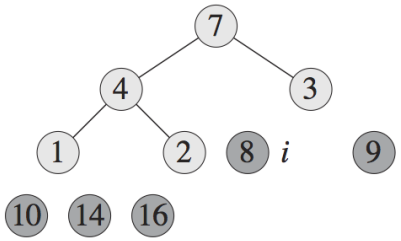
(c)



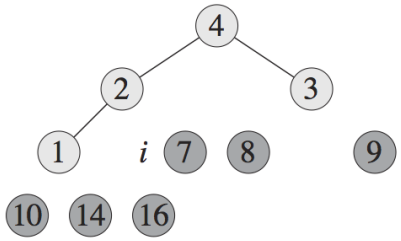
(d)



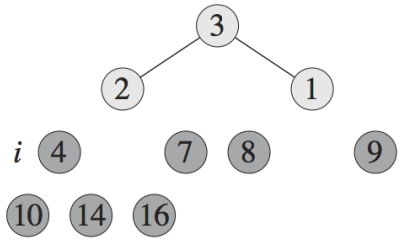
(e)



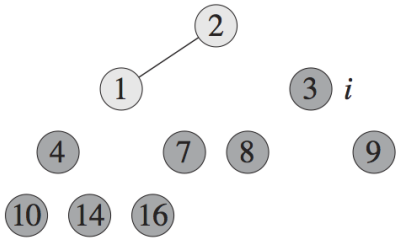
(f)



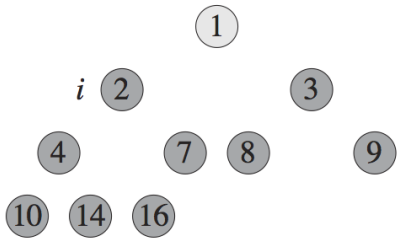
(g)



(h)



(i)



(j)

(k)

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$O(\log n)$。

  HEAPSORT过程的总时间代价为： $O(\mathrm{nlog}\, n)$ 。

- 是一种**原地排序算法**

应用：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**的一个应用是在一台**分时计算机上进行作业调度

优先级队列

- 最大优先级队列经常被用于分时计算机上的作业调度，对要执行的各作业及他们之间的相对优先关系加以记录。
  - ✓ 一个作业做完或被中断时，可用EXTRACT-MAX操作从所有等待的作业中，选择出具有最高优先级的作业。
  - ✓ 任何时候，一个新作业都可用INSERT加入到队列中去。
- 最小优先级队列支持的操作： INSERT， MINMUM，EXTRACT-MIN， DECREASE-KEY
  - ✓ 可被用在基于事件驱动的模拟器中。
  - ✓ 事件模拟要按照各事件发生时间的顺序进行
  - ✓ 每一步都使用EXTRACT-MIN选择下一个模拟的事件
  - ✓ 一个新事件产生时，使用INSERT将其放入队列中。

优先级队列（最大优先队列的操作）

- MAXIMUN

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

- EXTRACT-MAX，运行时间为$O(\log n)$

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

优先级队列（最大优先队列的操作）

- INCREASE-KEY：运行时间为$O(\log n)$

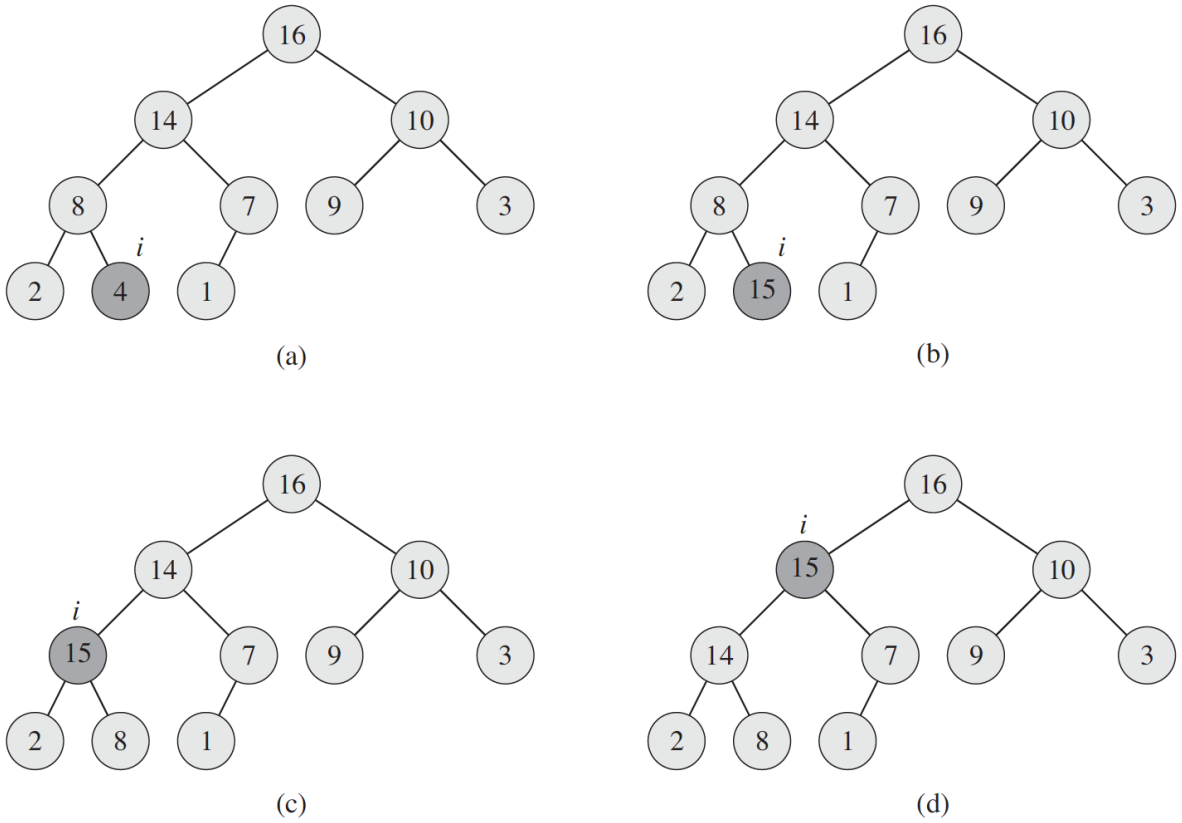
```
HEAP-INCREASE-KEY(A, i, key )
1  if  key < A[i]
2      then error "new key is smaller than current key"
3  A[ i ] ← key
4  while  i > 1 and A[PARANT(i)] < A[i]
5      do exchange A[i ] ↔ A[PARANT(i)]
6          i ← PARANT(i)
```

- INSERT: 运行时间为$O(\log n)$

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

- 一个堆可以在$O(\log n)$时间内，支持大小为n的集合上的任意优先队列操作。

Heap-Increase-key



排序算法

内容提要：

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

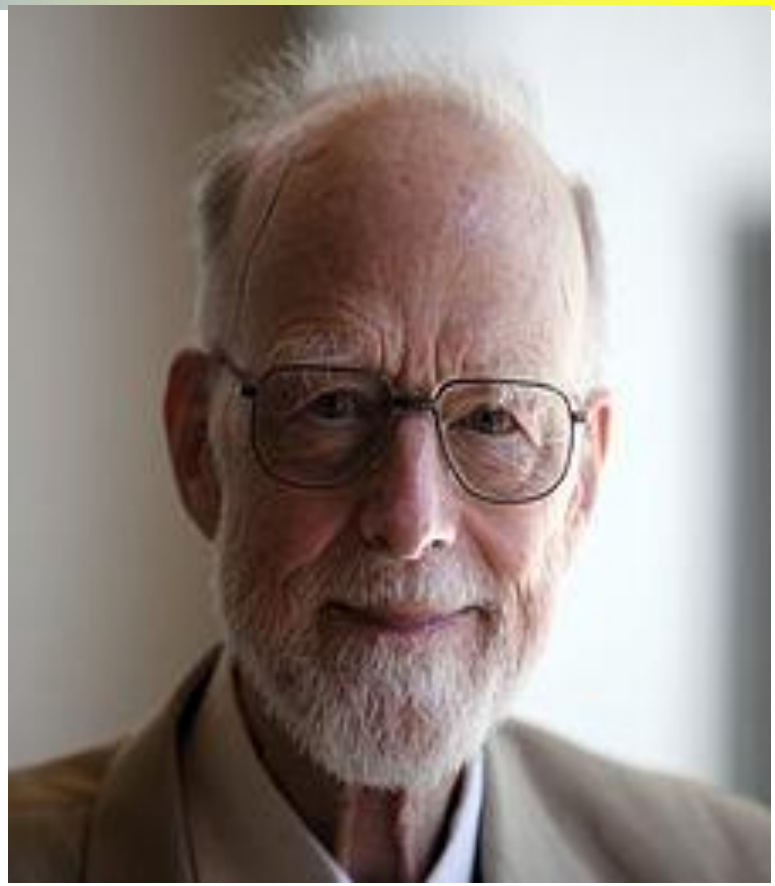
C.A.R.Hoare

（1934- ）英国计算机科学家

牛津大学古罗马文学学士

莫斯科国立大学博士

1980 获图林奖



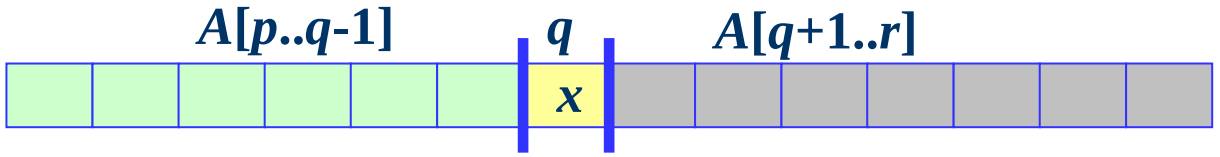
快速排序算法

- C.R.A.Hoare于1962 年提出
- 分治策略
- **原址**排序(sort “in place”)：在原来的数据区域内进行重排
- 节省内存
- 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$\Theta(n\log n)$且常数因子较小

快速排序算法

- 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：** 选一个主元 x，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]，而 A[q+1..r]中的元素都大于A[q] 。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 )
1 if   p < r
2    then  q ← PARTITION(A, p, r )
3          QUICKSORT( A, p, q-1 )
4          QUICKSORT( A, q+1, r )
```

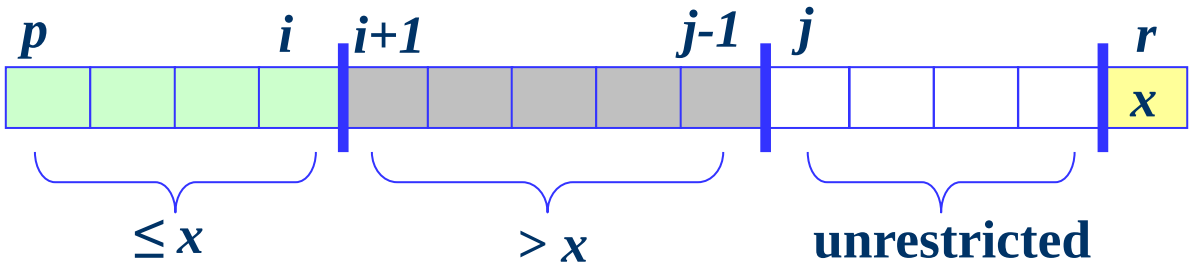
* 为排序一个完整数组，最初调用 QUICKSORT(A, 1, *length*[A])。

- 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。
- 划分所需时间：Θ(n)

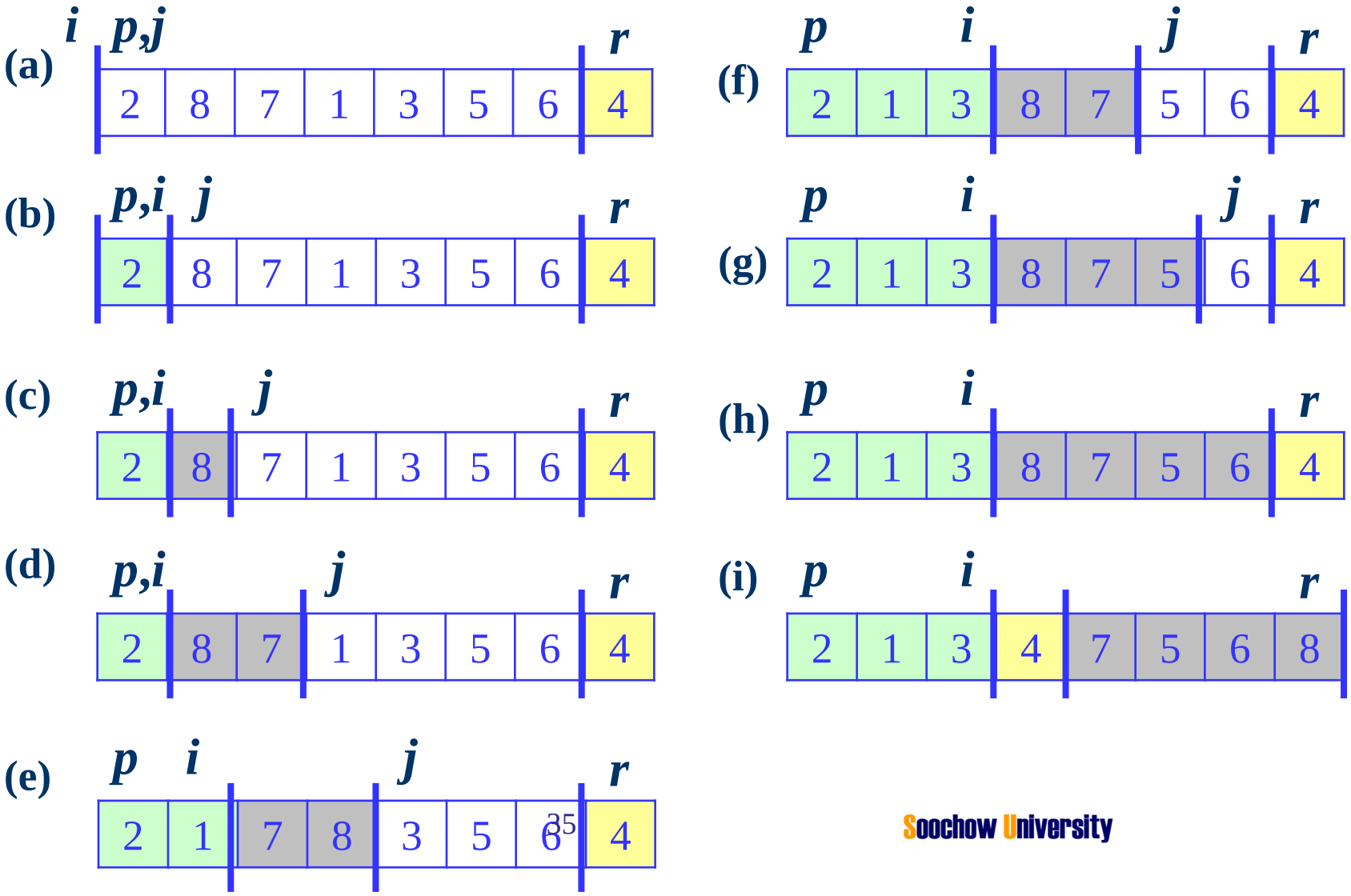
快速排序算法

- 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       if A[ j ] ≤ x
5            i ← i + 1
6            exchange A[i]  A[j]
7  exchange A[i + 1]  A[r]
8  return i + 1
```

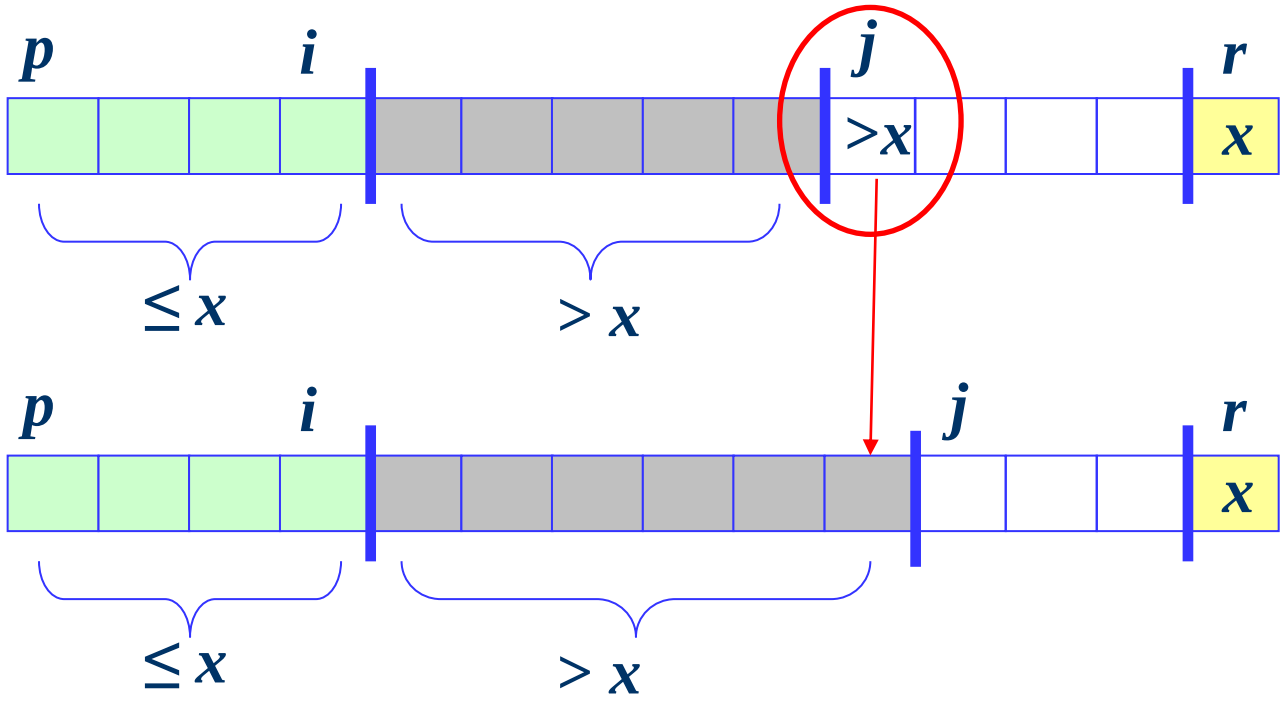


范例: (Partition, $x=A[r]=4$)



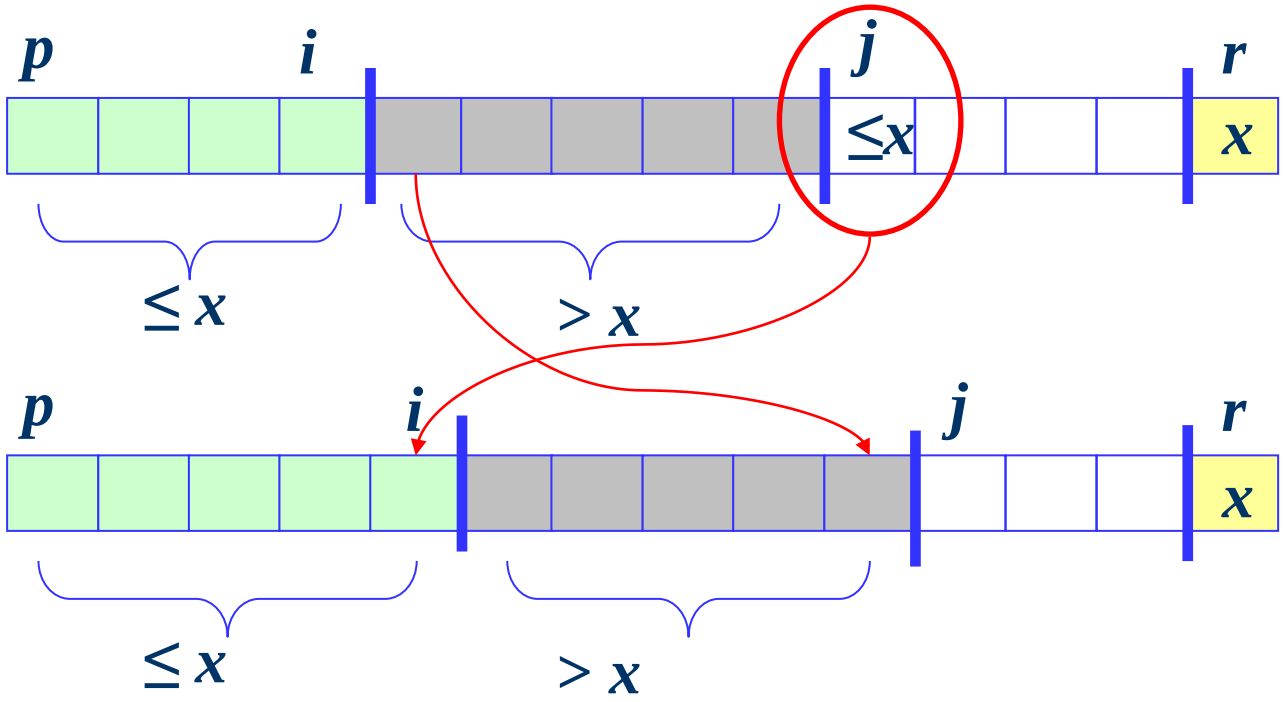
快速排序算法

i 和 *j* 如何改变：



快速排序算法

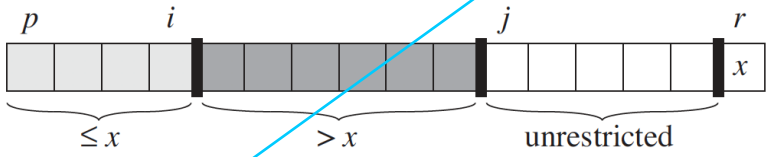
i 和 *j* 如何改变:

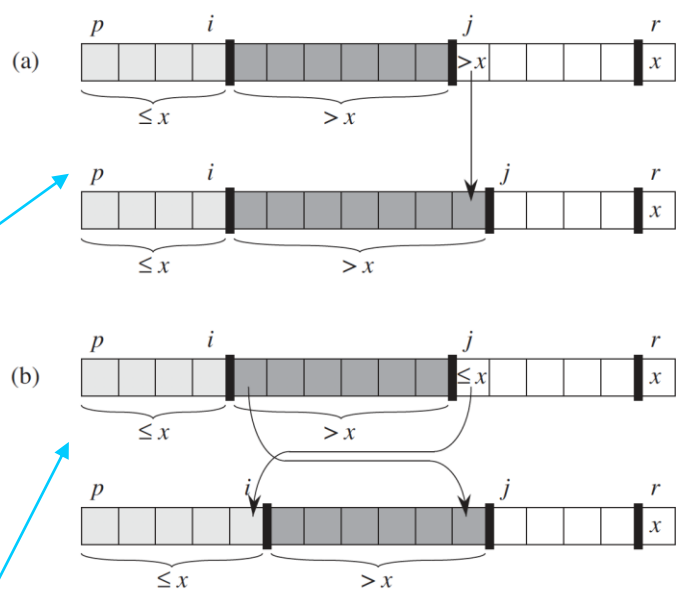


Quicksort

循环不变量

- $A[p..i] \leq x$
- $A[i+1..j-1] \geq x$
- $A[r] = x$





初始化：在循环的第一轮迭代开始之前，$i=p-1$ 和 $j=p$。因为在 p 和 i 之间、$i+1$ 和 $j-1$之间都不存在值，所以循环不变量的前两个条件显然都满足。第 1 行中的赋值操作满足了第三个条件。

保持：如图 7-3 所示，根据第 4 行中条件判断的不同结果，我们需要考虑两种情况。图 7-3(a)显示当 $A[j]>x$ 时的情况：循环体的唯一操作是 j 的值加 1。在 j 值增加后，对 $A[j-1]$，条件 2 成立，且所有其他项都保持不变。图 7-3(b)显示当 $A[j]\leqslant x$ 时的情况：将 i 值加 1，交换$A[i]$和 $A[j]$，再将 j 值加 1。因为进行了交换，现在有 $A[i]\leqslant x$，所以条件 1 得到满足。类似地，我们也能得到 $A[j-1]>x$。因为根据循环不变量，被交换进 $A[j-1]$的值总是大于 x 的。

终止：当终止时，$j=r$。于是，数组中的每个元素都必然属于循环不变量所描述的三个集合的一个，也就是说，我们已经将数组中的所有元素划分成了三个集合：包含了所有小于等于 x 的元素的集合、包含了所有大于 x 的元素的集合和只有一个元素 x 的集合。

快速排序算法

- 最坏情况：$\Theta(n^2)$ (对已排序好的輸入)

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

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

猜测: $T(n) \le c\,n^2 = O(n^2)$

Substituting:

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

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

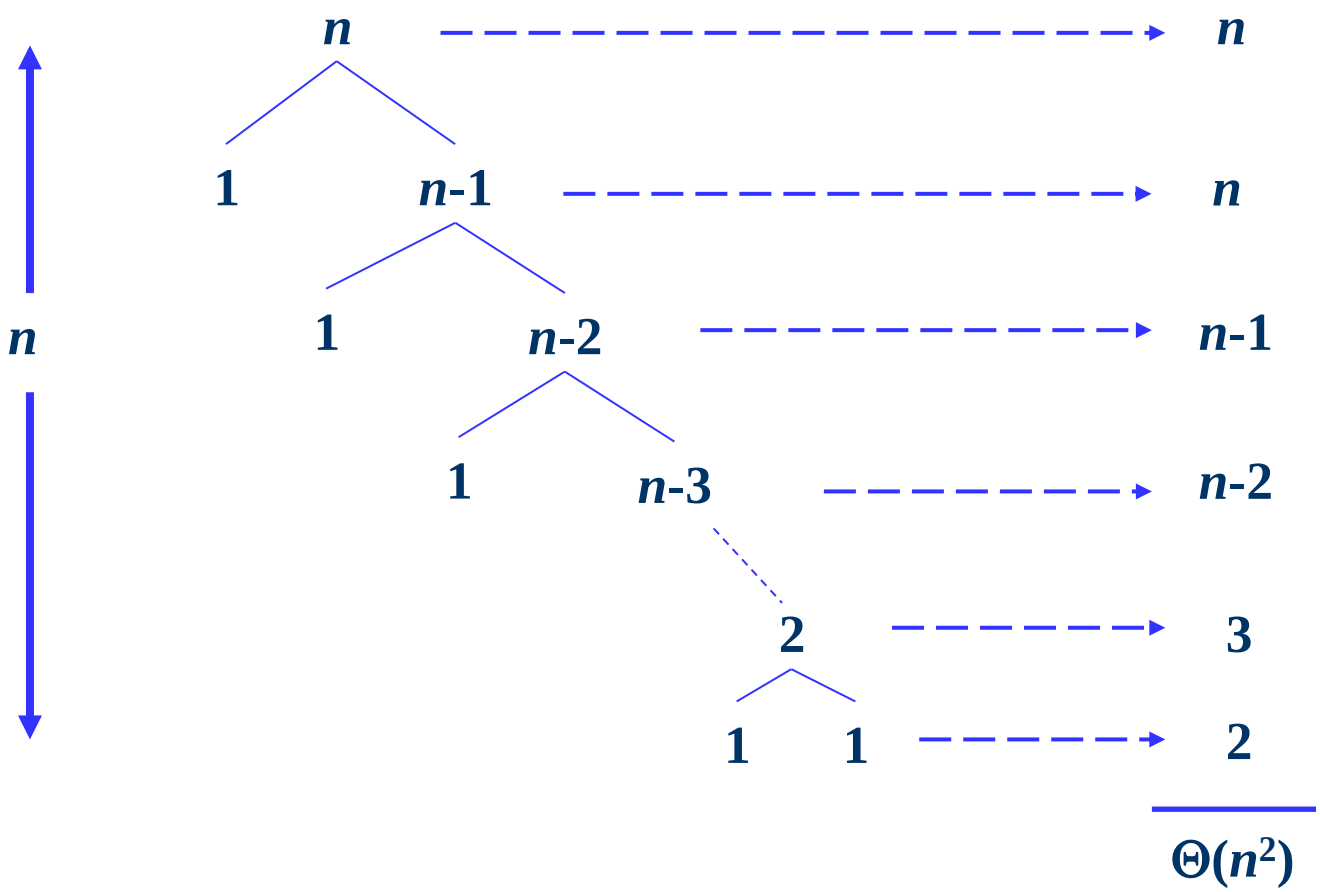
$$\le c(n\text{-}1)^2+\Theta(n)$$

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

$$\le cn^2$$ (挑选够大的 c 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： $\Theta(n \log n)$

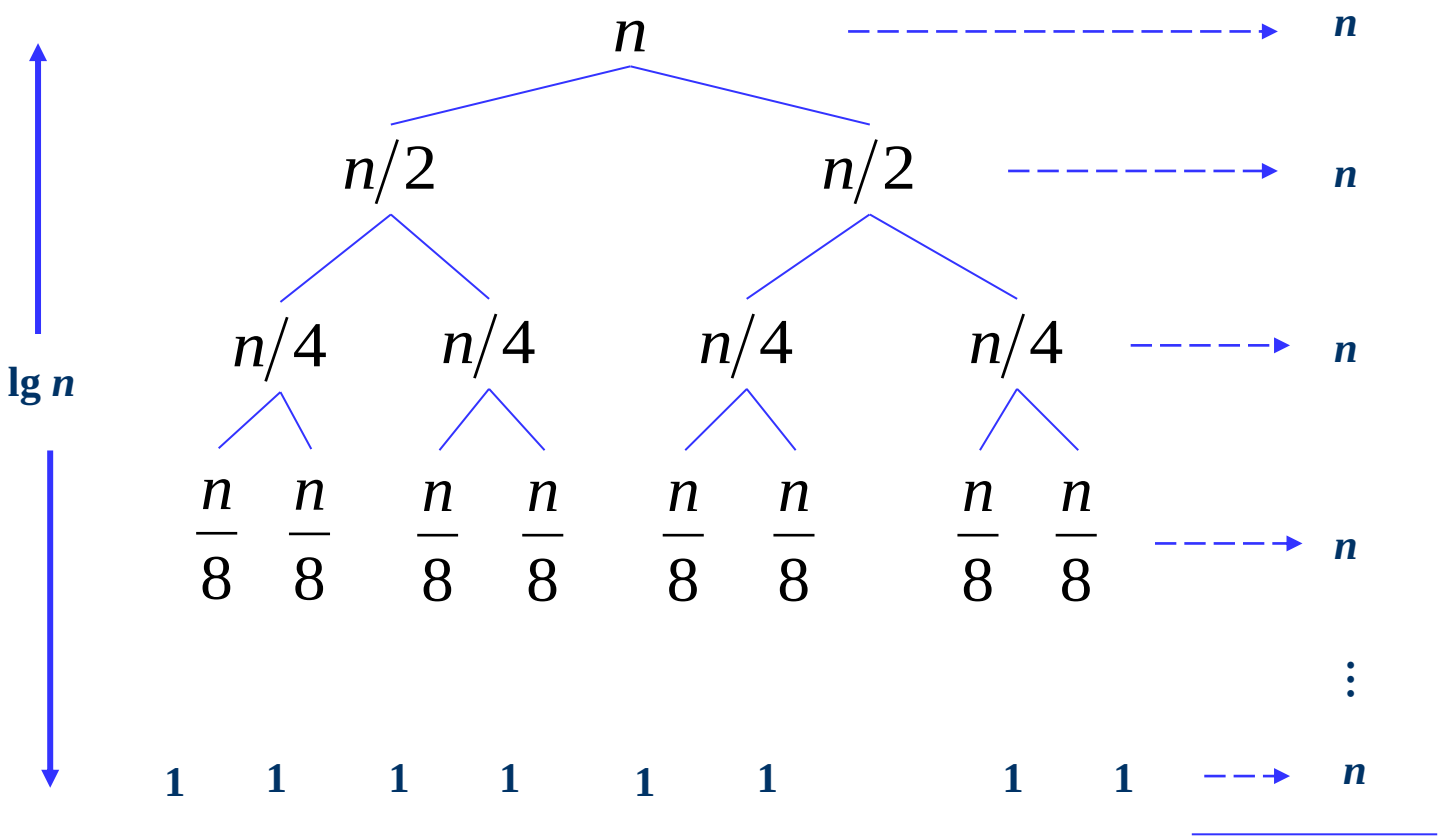
  此时得到的两个子问题的大小都不可能大于*n/2*, 运行时间的递归表达式为：

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

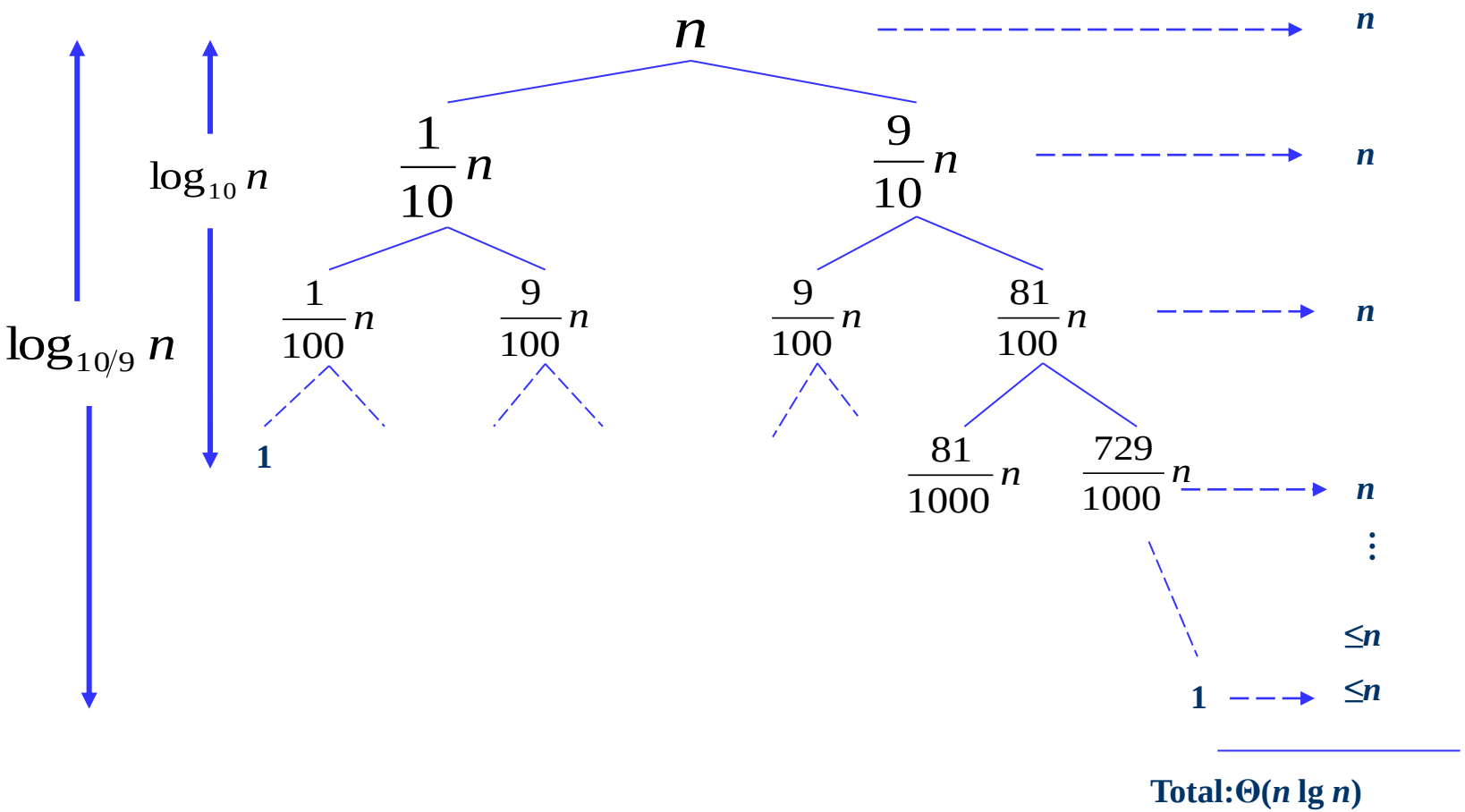
- 根据主定理，该递归式的解为： $T(n) = \Theta(n \log n)$

- 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$O(n\log n)$， 下界是$\Omega(n \log n)$， 故运行时间： $\Theta(n \log n)$

快速排序算法(最好情况划分)



快速排序算法（平衡的划分）



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$A[p...r]$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           RANDOMIZED-QUICKSORT( A, p, q-1 )
4           RANDOMIZED-QUICKSORT( A, q+1, r )
```

快速排序算法

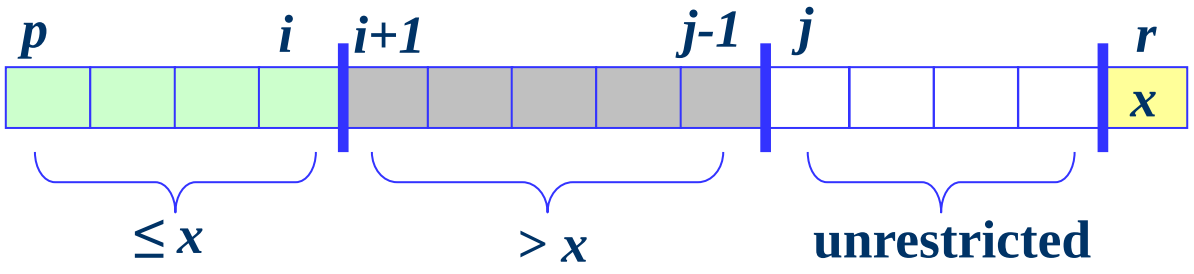
- 平均情况划分： $\Theta(n \lg n)$

- 引理7.1：
  - 当在一个包含n个元素的数组上运行QUICKSORT时，假设在PARTITION的第4行中所做的次数为X，那么QUICKSORT的运行时间为$O(n+X)$。
- 证明：
  - 算法最多对PARTITION调用n次
  - X？

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

- 首先注意到每一对元素至多比较一次。为什么呢？
  - 因为各个元素只跟只与主元素进行比较，并且在某一次Partion调用结束之后，该次调用到的主元就再也不会与任何其他元素比较了

我们的分析要用到指示器随机变量(见 5.2 节)。定义

$$X_{ij} = \mathrm{I}\{z_i \text{ 与 } z_j \text{ 进行比较}\}$$

其中我们考虑的是比较操作是否在算法执行过程中任意时间发生，而不是局限在循环的一次迭代或对 PARTITION 的一次调用中是否发生。因为每一对元素至多被比较一次，所以我们可以很容易地刻画出算法的总比较次数：

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

对上式两边取期望，再利用期望值的线性特性和引理 5.1，可以得到：

$$\mathrm{E}(X) = \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ 与 } z_j \text{ 进行比较}\} \qquad (7.2)$$

上式中的 $\Pr\{z_i$ 与 z_j 进行比较$\}$还需要进一步计算。在我们的分析中，假设 RANDOMIZED-PARTITION 随机且独立地选择主元。

快速排序算法

- 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.

- 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。

快速排序算法

- 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。

- **Case1:** 一旦一个满足$z_i < x < z_j$的主元x被选择后，z_i和z_j以后再也不会被比较 。
- **Case2:**z_i : z_j 会进行比较：当且仅当第一个从 Z_{ij} 选出来的 主元是 z_i 或 z_j。

$$\begin{aligned}\Pr\{z_i \text{ 与 } z_j \text{ 进行比较}\} &= \Pr\{z_i \text{ 或 } z_j \text{ 是集合 } Z_{ij} \text{ 中选出的第一个主元}\} \\ &= \Pr\{z_i \text{ 是集合 } Z_{ij} \text{ 中选出的第一个主元}\} \\ &\quad + \Pr\{z_j \text{ 是集合 } Z_{ij} \text{ 中选出的第一个主元}\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}\end{aligned}$$

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，

$$
\begin{aligned}
X &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \qquad \text{(套用 Harmonic Series)} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n)
\end{aligned}
$$

排序算法

内容提要：

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

排序算法

- 本节探讨排序所耗用的时间复杂度下限。
- 排序算法能有多快
- 已知排序算法

排序算法时间的下界

- 比较排序：排序结果中，各元素的次序基于输入元素间的比较，这类算法成为比较排序。
- 任何比较排序算法，排序n个元素时至少耗用$\Omega(n\lg n)$次比较，其时间复杂度至少为$\Omega(n\lg n)$
- 归并排序和堆排序是渐近最优的
- 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。

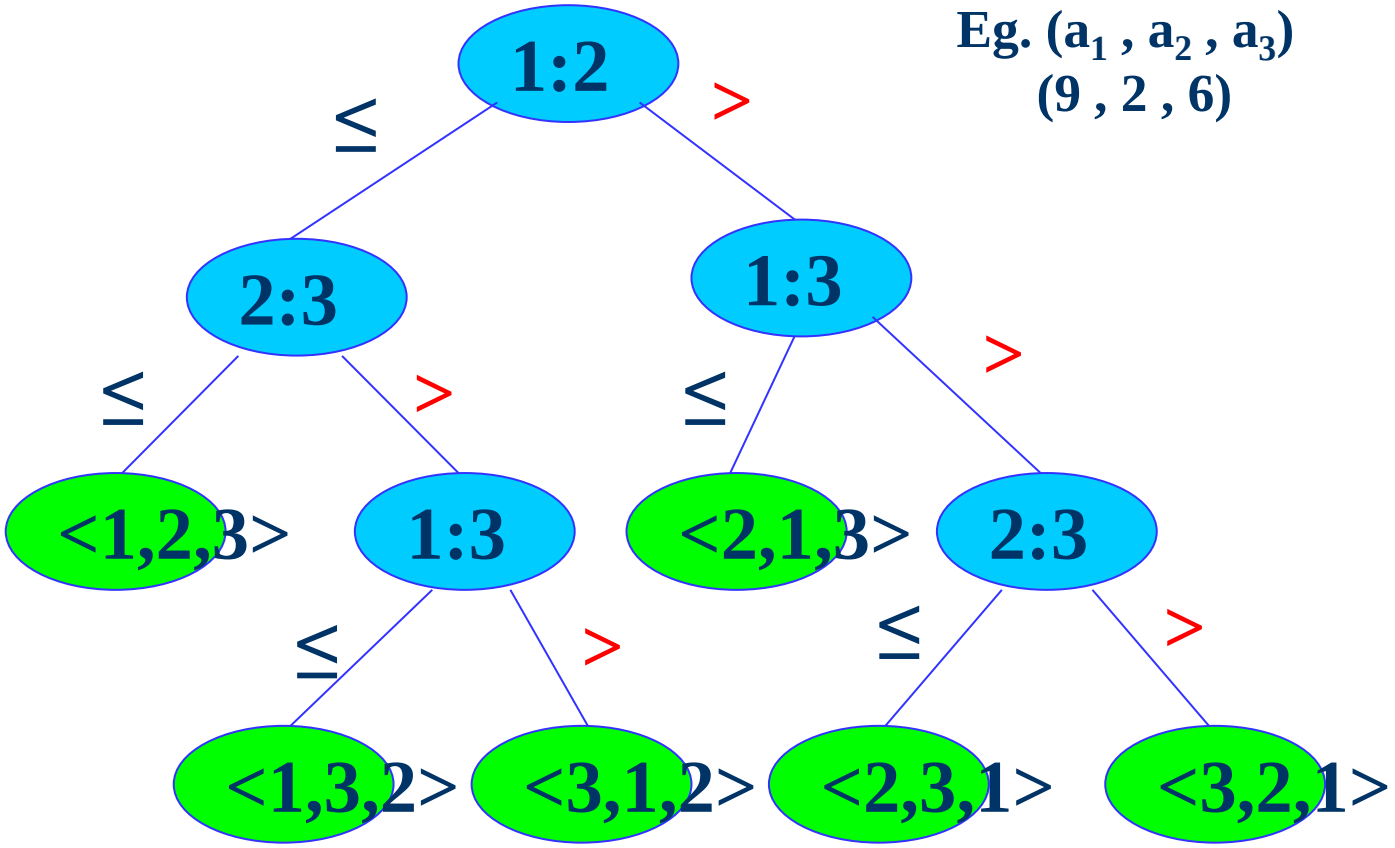
排序算法时间的下界

- 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）。
- 决策树是一棵满二叉树，表示某排序算法作用于给定输入所做的所有比较，忽略控制结构、数据移动等。

决策树模型：

- 每个内结点注明i : j (1≤i, j≤n),每个叶结点注明排列 $(\pi(1), \pi(2), \cdots, \pi(n))$
- 排序算法的执行对应于遍历一条从树的根到叶节点的路径。
- 在每个内结点处做比较 $a_i \le a_j$,内结点的左子树决定着 $a_i \le a_j$ 以后的比较，而右子树决定着 $a_i > a_j$ 以后的比较
- 到达一个叶结点时，排序算法就已确定了顺序 $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$
- 每一个从根节点到叶子结点的路径对应于比较排序算法的一次实际执行过程。
- 排序算法正确工作的必要条件：n个元素的 $n!$ 中排列都要作为决策树的一个叶子。

排序算法时间的下界



排序算法时间的下界

- 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Ω(nlogn)。
  - 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$h \geq \log(n!) \geq \Theta(n\log n)$。(后者由斯特林公式(3.18)得证, **Exercise 3.2-3**)
- 比较排序算法的最坏情况比较次数与其决策树的高度相等
- **定理1** 任意比较排序算法在最坏情况下，都需要做Ω(nlogn)次比较。
- 堆排序和合并排序是渐近最优的排序算法 （因为它们的运行时间上界O(nlogn)与 定理1给出的最坏情况下界Ω(nlogn)一致。）
- 快速排序不是渐近最优的比较排序算法。但是，快速排序其执行效率平均而言较堆排序和合并排序还好。

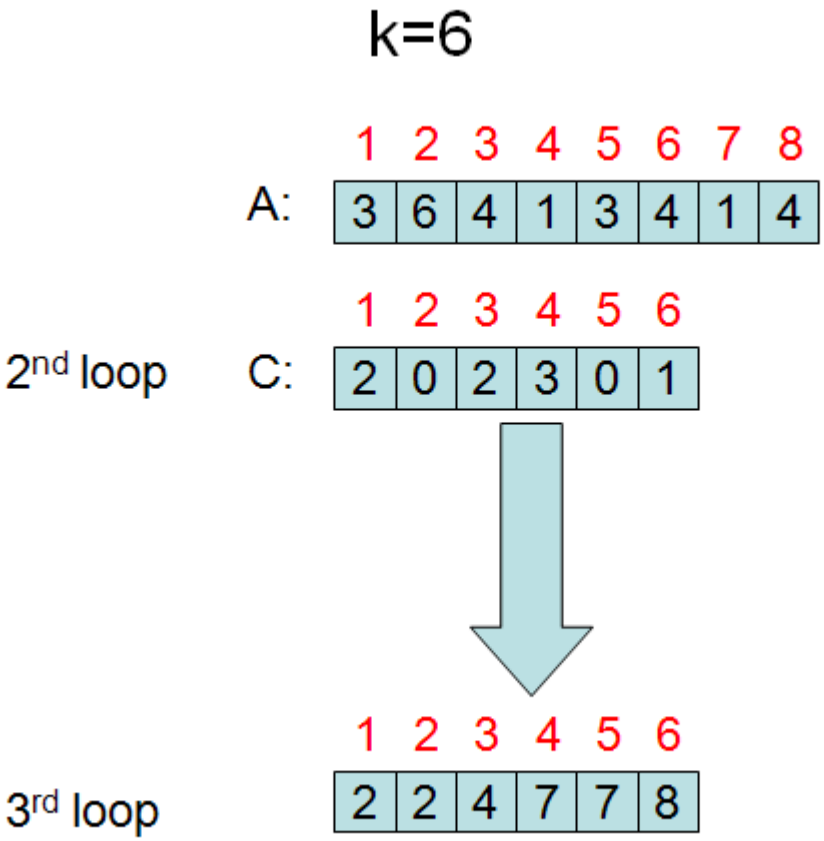
计数排序

- **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：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。
- 在计数排序算法的代码中，需三个数组：
  - 输入数组A[1…n]
  - 存放排序结果的数组B[1…n]
  - 提供临时存储区的数组C[0…k]

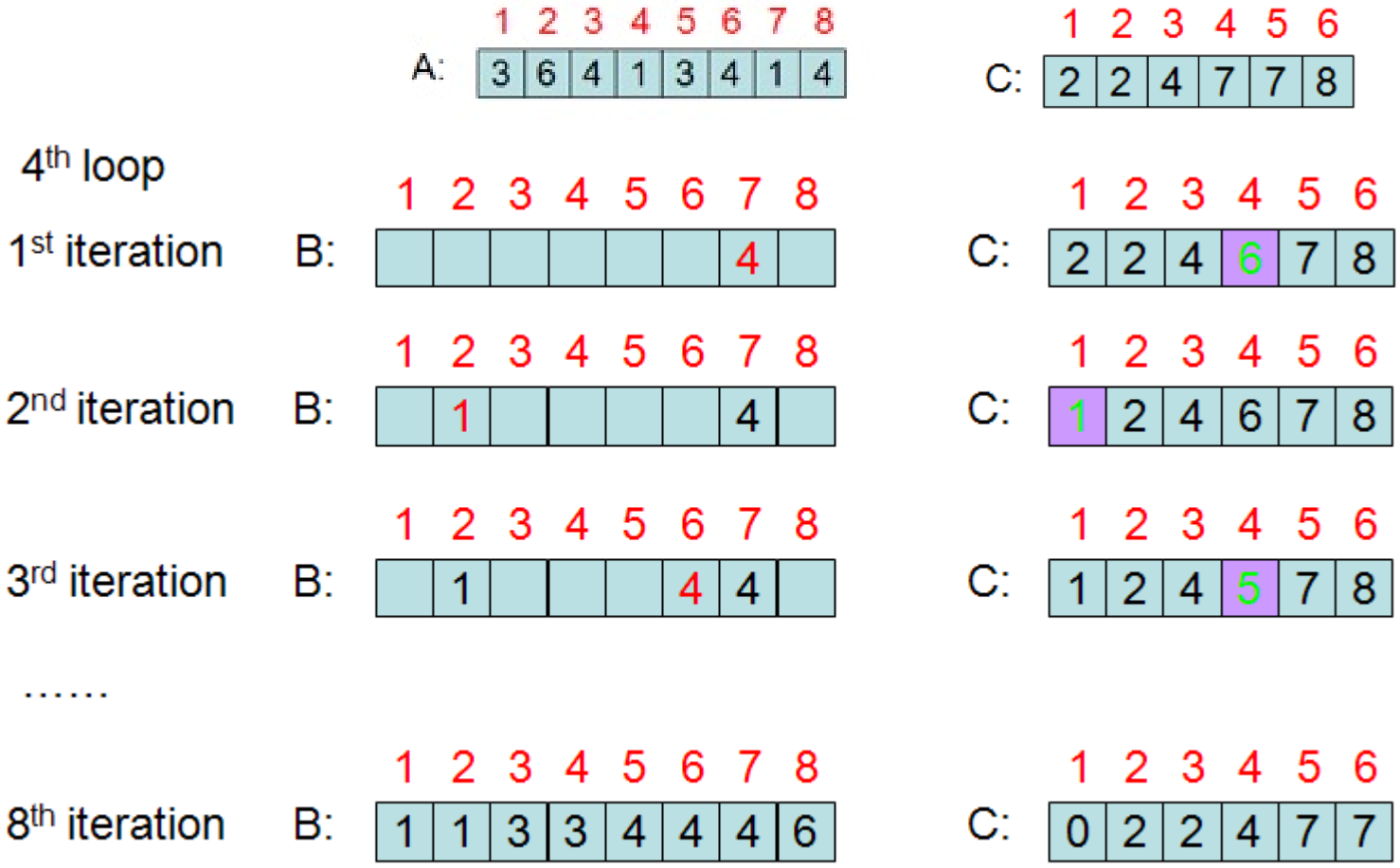
计数排序

```
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

计数排序


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

计数排序

•排序算法是稳定的：具有相同值的元素在输出数组中的相对次序 与它们在输入数组中的次序相同。

思考: 还有哪些排序算法是稳定的?

•经常被当做基数排序算法的一个子过程 。

基数排序（Radix Sort）

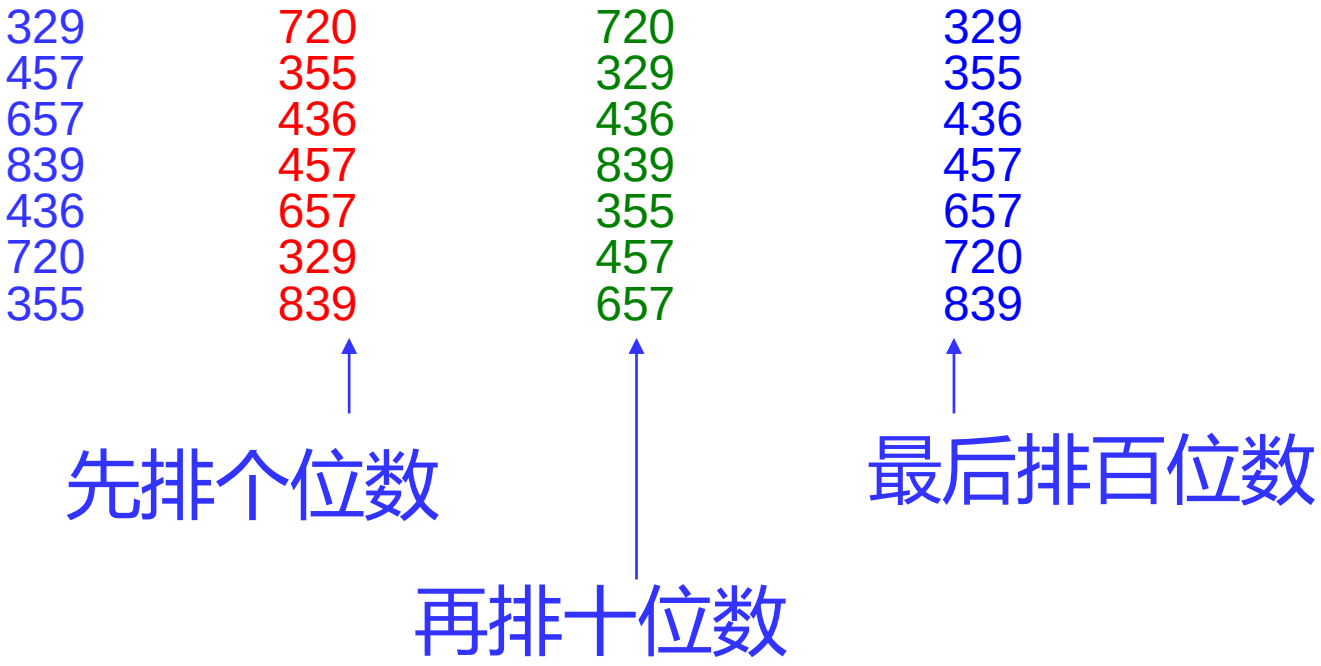
➢**Radix Sort(基数排序)：**假设所有待排序元素均为整数，至多*d*位。先按最低有效位进行排序，再按次低有效位排序，重复这个过程，直到对所有的d位数字都进行了排序。

➢基数排序关键是按位排序要稳定。

➢算法

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

基数排序


329
457
657
839
436
720
355
720
355
436
457
657
329
839
720
329
436
839
355
457
657
329
355
436
457
657
720
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$\Theta(n+k)$的时间（假设每位数可以取k种可能的值）。

- 因此总共花费$\Theta(d(n+k))$的时间。

- 如果d是常数，$k=O(n)$，则基数排序是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Θ(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,如果RADIX-SORT使用稳定排序方法耗时Θ (n+k),那么它就可以在 $\Theta((b/r)(n+2^r))$ 时间内正确地对这些数进行排序。

- **32位的字看做是4个8位的数，于是b=32, r=8, k=2^8-1=255, d=b/r=4**
- **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< $\lfloor \lg n \rfloor$，选择r=b, 此时为Θ(n)；如果b≥ $\lfloor \lg n \rfloor$， 选择r= $\lfloor \lg n \rfloor$ ，此时Θ(bn/lgn)**

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。
- 桶排序假设输入由一个随机过程产生，该过程将元素均匀的分布在区间[0，1)上。

- 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）。
  - 将n个输入数分布到各个桶中去。
  - 先对各桶中元素进行排序，然后依次把各桶中的元素列出来即可。

桶排序

假定要排序的n个元素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$\lceil nx \rceil$个桶 。

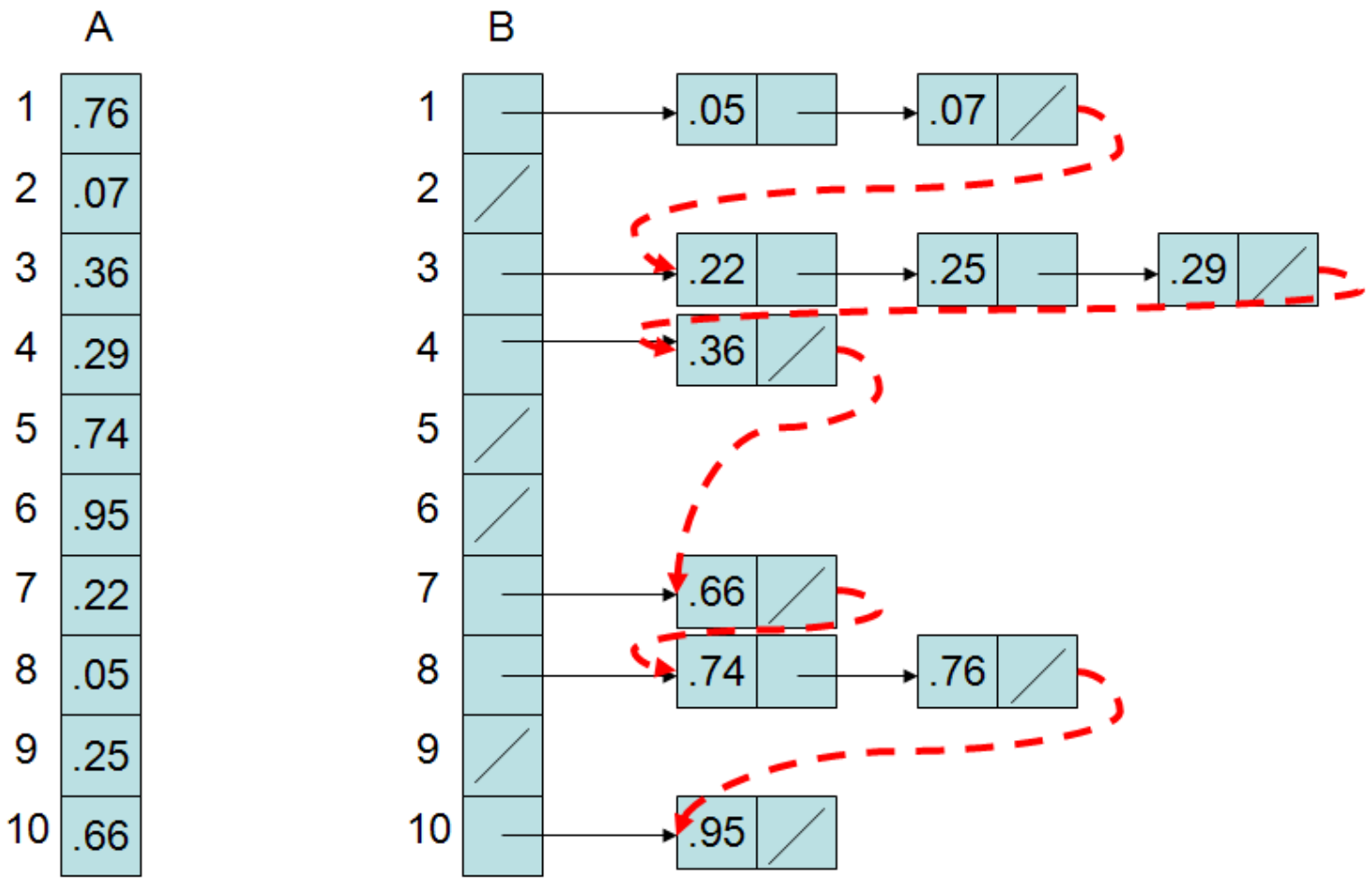
2）元素放进桶时，使用链表来存储，并利用插入排序法排序。

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

- 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


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！

排序算法

内容提要：

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

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
  - ✓ 如插入排序或选择排序等
  - ✓ 由于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
  - ✓ 如插入排序或冒泡排序等

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)