HITWH
SE

第三章

排序与分治算法

HITWH
SE

参考材料

《Introduction to Algorithm》

Chapter 6, 7, 8, 9

《Introduction to the Design and Analysis of Algorithm》

Chapter 4

提要

3.1 分治算法的原理
3.2 基于分治思想的排序算法
3.3 线性时间排序算法
3.4 Medians and Order Statistics
3.5 最邻近点对
3.6 凸包问题
3.7 FFT
3.8 整数乘法

HITWH
SE

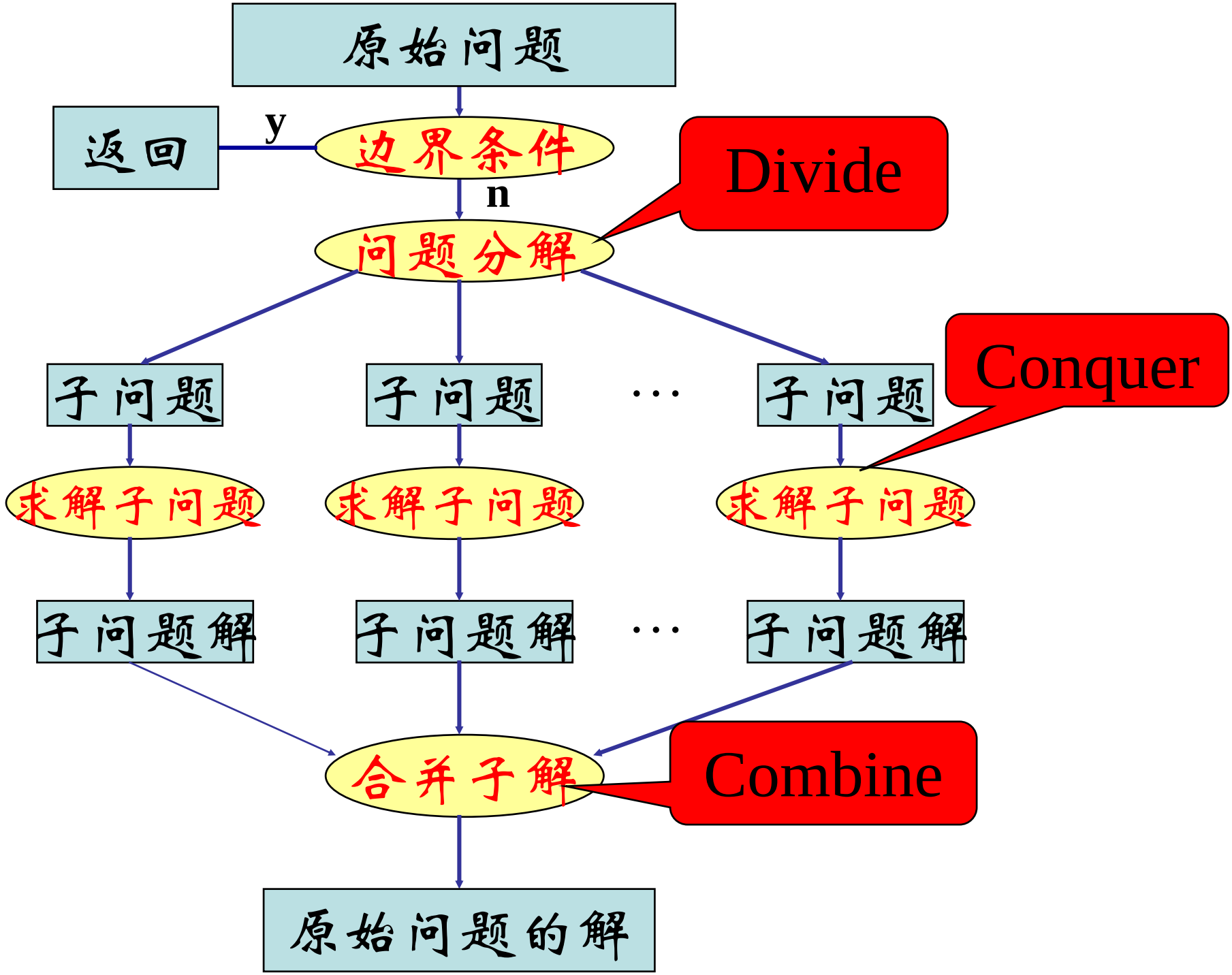
3.1 Divide-and-Conquer原理

- Divide-and-Conquer算法的设计
- Divide-and-Conquer算法的分析

HITWH
SE

Divide-and-Conquer算法的设计

- 设计过程分为三个阶段
  - Divide： 整个问题划分为多个子问题
    - 注意：分解的这组子问题$p_1,p_2,...p_m$未必一定是相同的子问题，即p_i和p_j可以是分别完成不同任务的子问题
  - Conquer：求解各子问题(递归调用正设计的算法)
  - Combine：合并子问题的解，形成原始问题的解


原始问题
返回
y
边界条件
n
Divide
问题分解
子问题
子问题
…
子问题
Conquer
求解子问题
求解子问题
求解子问题
子问题解
子问题解
…
子问题解
合并子解
Combine
原始问题的解

HITWH
SE

Divide-and-Conquer算法的分析

- 分析过程
  - 建立递归方程
  - 求解
- 递归方程的建立方法
  - 设输入大小为n, $T(n)$为时间复杂性
  - 当$n<c$, $T(n)=\theta(1)$

– Divide阶段的时间复杂性
  - 划分问题为a个子问题。
  - 每个子问题大小为n/b。
  - 划分时间可直接得到=$D(n)$
– Conquer阶段的时间复杂性
  - 递归调用
  - Conquer时间 = $aT(n/b)$
– Combine阶段的时间复杂性
  - 时间可以直接得到=$C(n)$

最后得到递归方程：

- $T(n)=\theta(1)$ if $n\le c$
- $T(n)=aT(n/b)+D(n)+C(n)$ if $n>c$

HITWH
SE

举例：最大最小值问题

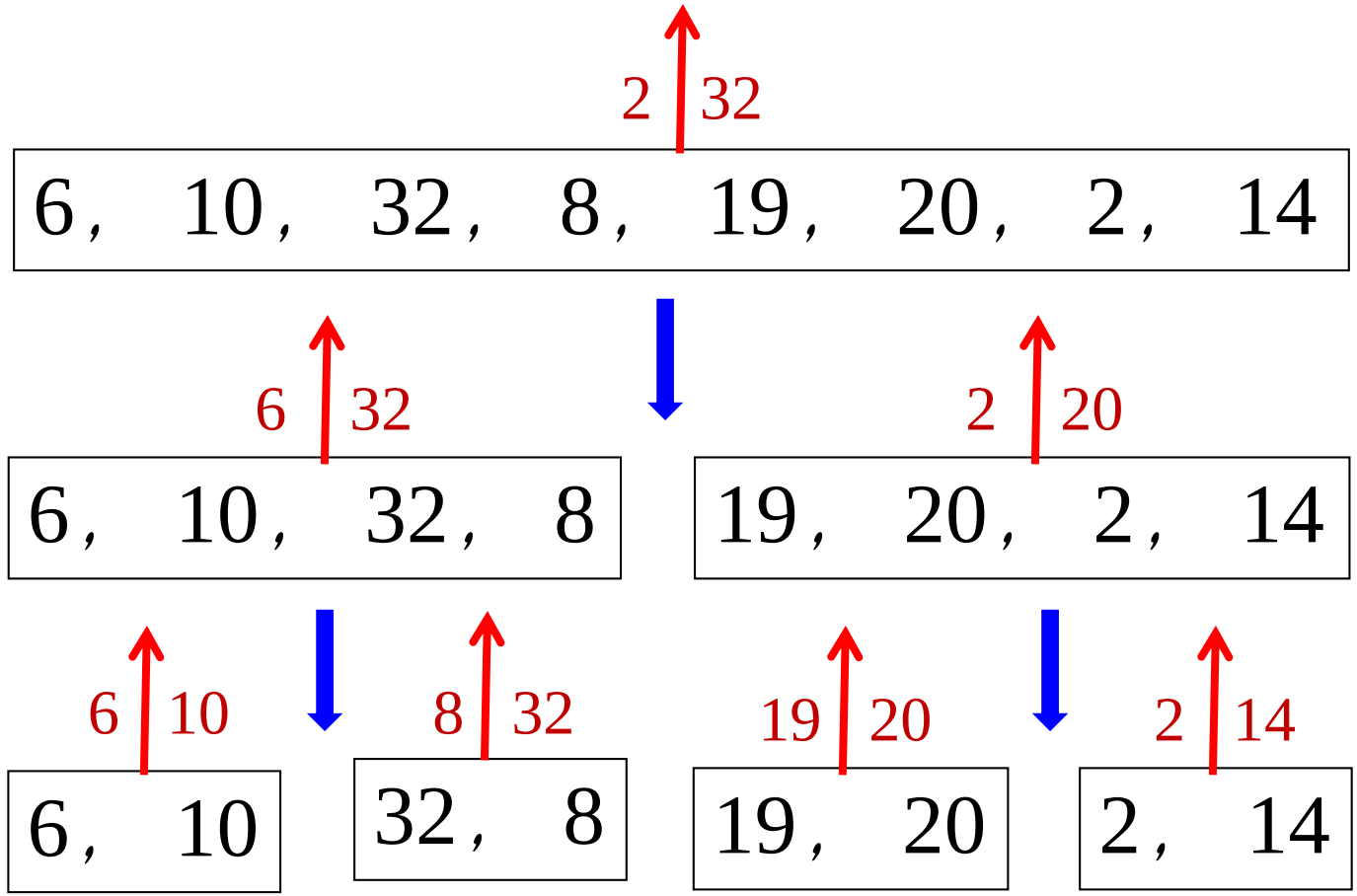
输入：数组$A[1,\ldots,n]$

输出：A中的max和min

通常，直接扫描需要$2n$-2次比较操作

我们给出一个仅需$3n/2$-2次比较操作的算法

基本思想


2 32
6, 10, 32, 8, 19, 20, 2, 14
6 32
2 20
6, 10, 32, 8
19, 20, 2, 14
6 10
8 32
19 20
2 14
6, 10
32, 8
19, 20
2, 14

算法

算法MaxMin(A)
输入: 数组$A[i,\ldots,j]$
输出:数组$A[i,\ldots,j]$中的max和min

1. If j-i+1 =1 Then 输出$A[i]$,$A[i]$,算法结束
2. If j-i+1 =2 Then
3. If $A[i]< A[j]$ Then 输出$A[i]$,$A[j]$; else 输出$A[j]$,$A[i]$;
   算法结束
4. $k\leftarrow(j+i)/2$
5. $m_1,M_1 \leftarrow$MaxMin($A[i{:}k]$);
6. $m_2,M_2 \leftarrow$MaxMin($A[k+1{:}j]$);
7. $m \leftarrow\min(m_1,m_2)$;
8. $M \leftarrow\max(M_1,M_2)$;
9. 输出m,M

算法复杂性分析

$T(1)=0$

$T(2)=1$

$T(n)=2T(n/2)+2$

$= 2(2T(n/2^2)+2)+2 = 2^2T(n/2^2)+2^2+2$

$= \ldots$

$=2^{k-1}T(2)+2^{k-1}+2^{k-2}+\ldots+2^2+2$

$=2^{k-1}+ 2^k-2$

$=n/2+ n -2$

$=3n/2-2$

$n=2^k$

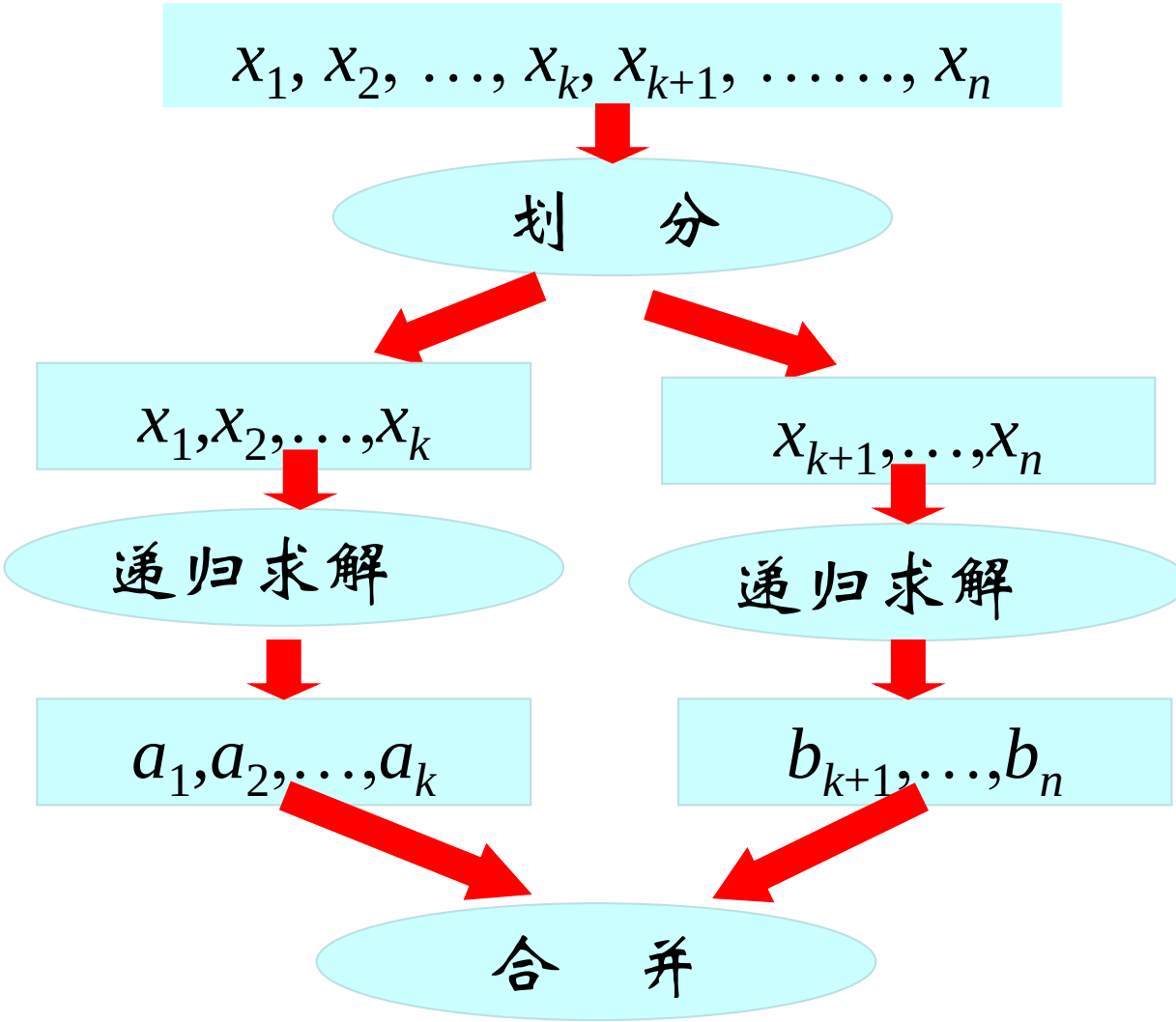
与Naïve算法相比，虽然同阶，但系数有所改进

HITWH
SE

3.2 基于分治的排序算法

- *Quicksort* Algorithm
- 排序问题的下界

基于分治思想的排序算法



划分的策略

根据某一策略将数据集合划分成两个部分

Mergesort：中间点

Quicksort：任选一个划分点x，利用x的值将数据划分成两部分

合并策略

不同的划分策略对应不同的合并策略

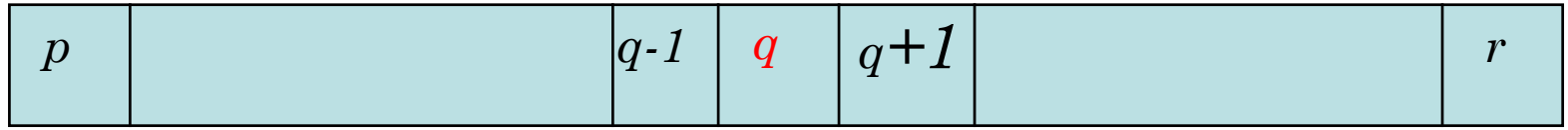
HITWHSE

3.2.1 *Quicksort*

- Idea of *Quicksort*
- *Quicksort* Algorithm
- Correctness Proof
- Performance Analysis
- Randomized Quicksort Algorithms

Idea of *Quicksort*

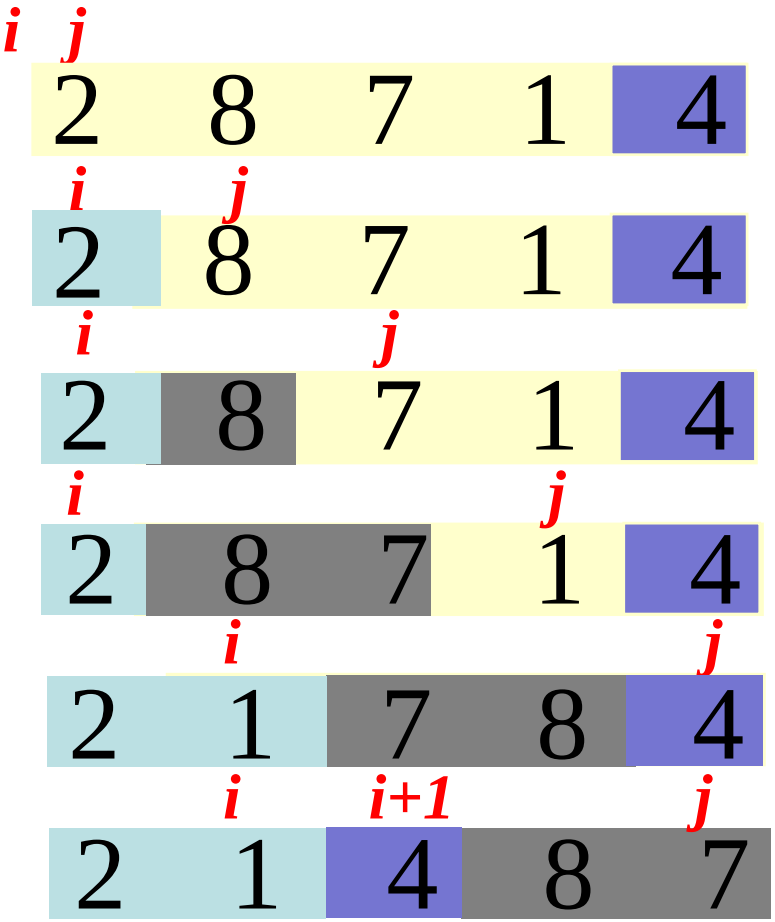
- Divide-and-Conquer
  - Divide:
    - Partition *A[p..r]* into *A[p..q]* and *A[q+1..r]*.



    - $\forall x \in A[p...q], x \le A[q], \ \forall y \in A[q+1...r], y > A[q]$.
    - *q* is generated by partition algorithm.
  - Conquer：
    - Sort *A[p...q-1]* and *A[q+1...r]* using quicksort recursively
  - Combine：
    - Since *A[p...q-1]* and *A[q+1...r]* have been sorted, nothing to do

- 划分 $A[p..r]$
  - 选择元素x作为划分点，$x=A[r]$
  - x逐一与其它元素作比较

算法执行过程中，
A 被分成4个区域



HITWH
SE

Partition(A, p, r)

$x \leftarrow A[r]$;

$i \leftarrow p-1$;

for $j \leftarrow p$ to $r-1$

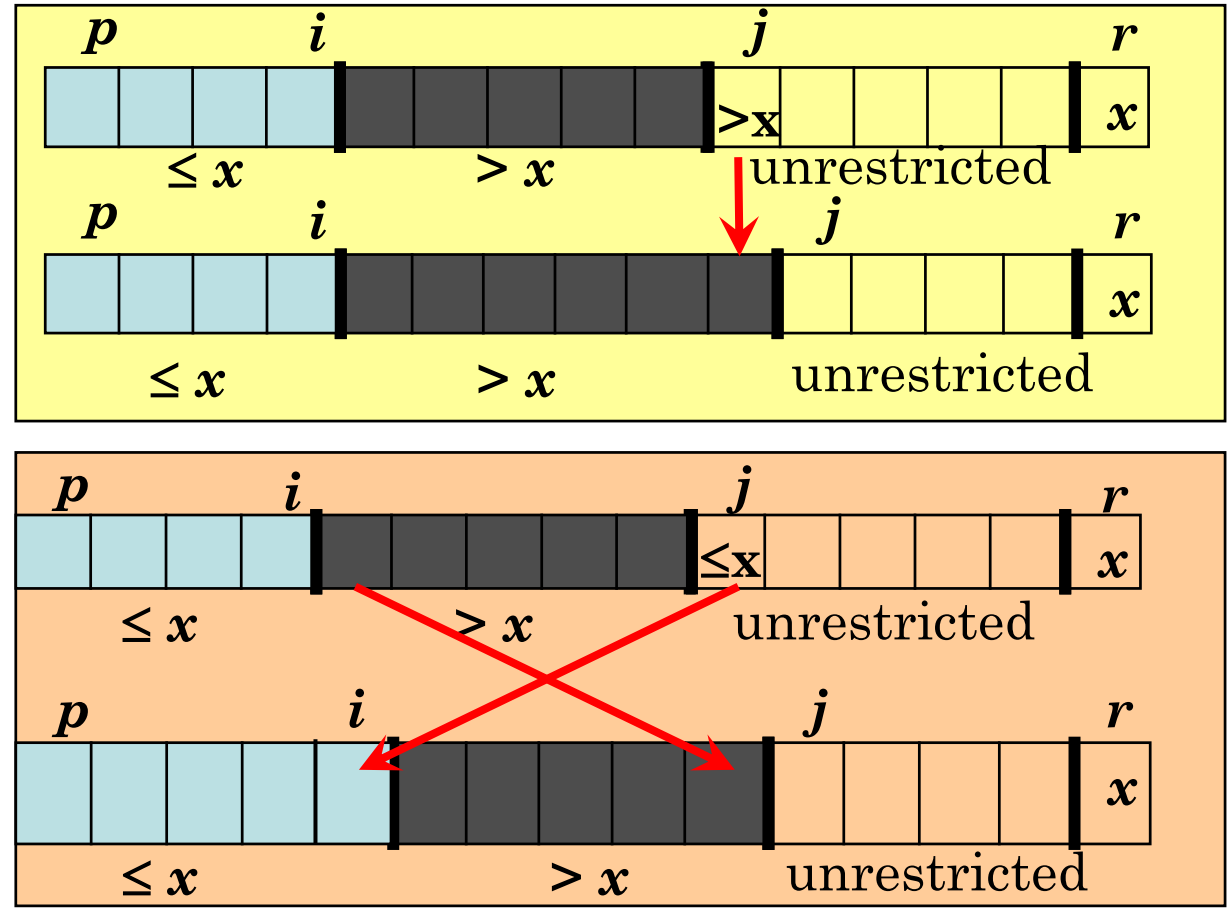
do if $A[j] \leq x$

$i \leftarrow i+1$;

exchange $A[i] \leftrightarrow A[j]$;

exchange $A[i+1] \leftrightarrow A[r]$;

return $i+1$;



Running time: $\Theta(n)$

Quicksort Algorithm

Quicksort(*A*,*p*,*r*)

If *p*<*r*

Then *q*=Partition(*A*, *p*, *r*);

Quicksort (*A*,*p*,*q*-*1*);

Quicksort (*A*,*q*+*1*,*r*);

Correctness Proof

- Loop Invariant(循环不变量方法)

证明主要结构是循环结构的算法的正确性

循环不变量：数据或数据结构的关键性质

依赖于具体的算法和算法特点

证明分三个阶段

(1) 初始阶段：循环开始前循环不变量成立

(2) 循环阶段：循环体每执行一次,循环不变量成立

(3) 终止阶段：算法结束后，循环不变量保证算法正确

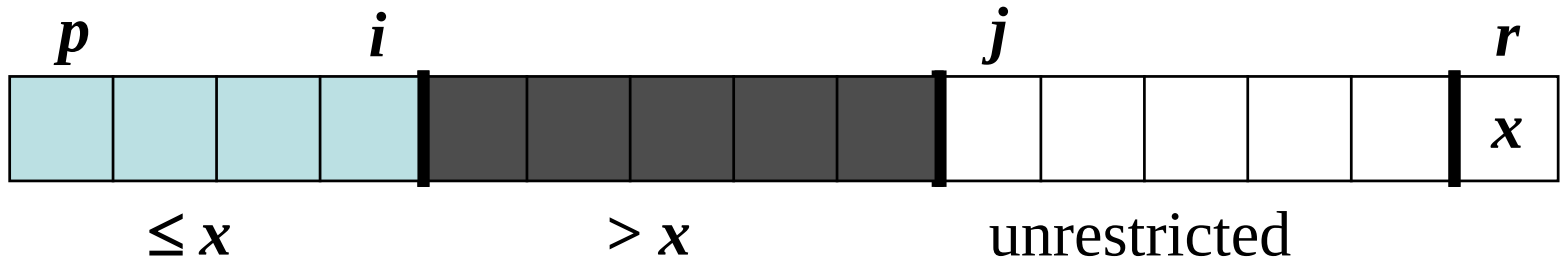
```
Partition(A, p, r)
    x ← A[r];
    i ← p − 1;
(3) for j ← p to r − 1
(4)     do if A[j] ≤ x
(5)           i ← i + 1;
(6)           exchange A[i] ↔ A[j];
    exchange A[i + 1] ↔ A[r];
    return i + 1;
```

- Correctness Proof

定义循环不变量：

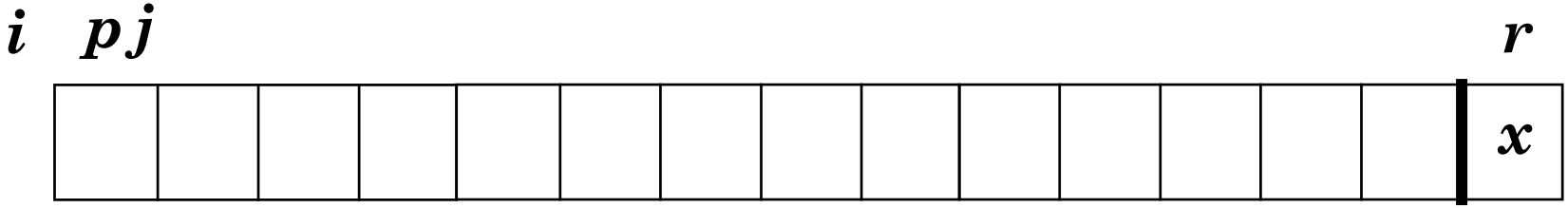
At the start of the loop of lines 3-6, for any *k*

1. *if* $p \le k \le i$, *then* $A[k] \le x$.
2. *if* $i+1 \le k \le j-1$, *then* $A[k] > x$.
3. *if* $k=r$, *then* $A[k]=x$.



– 初始阶段：$j=p$

算法迭代前：$i=p-1$, $j=p$, 条件 1 和 2 为真. 算法第1行使得条件 3 为真.

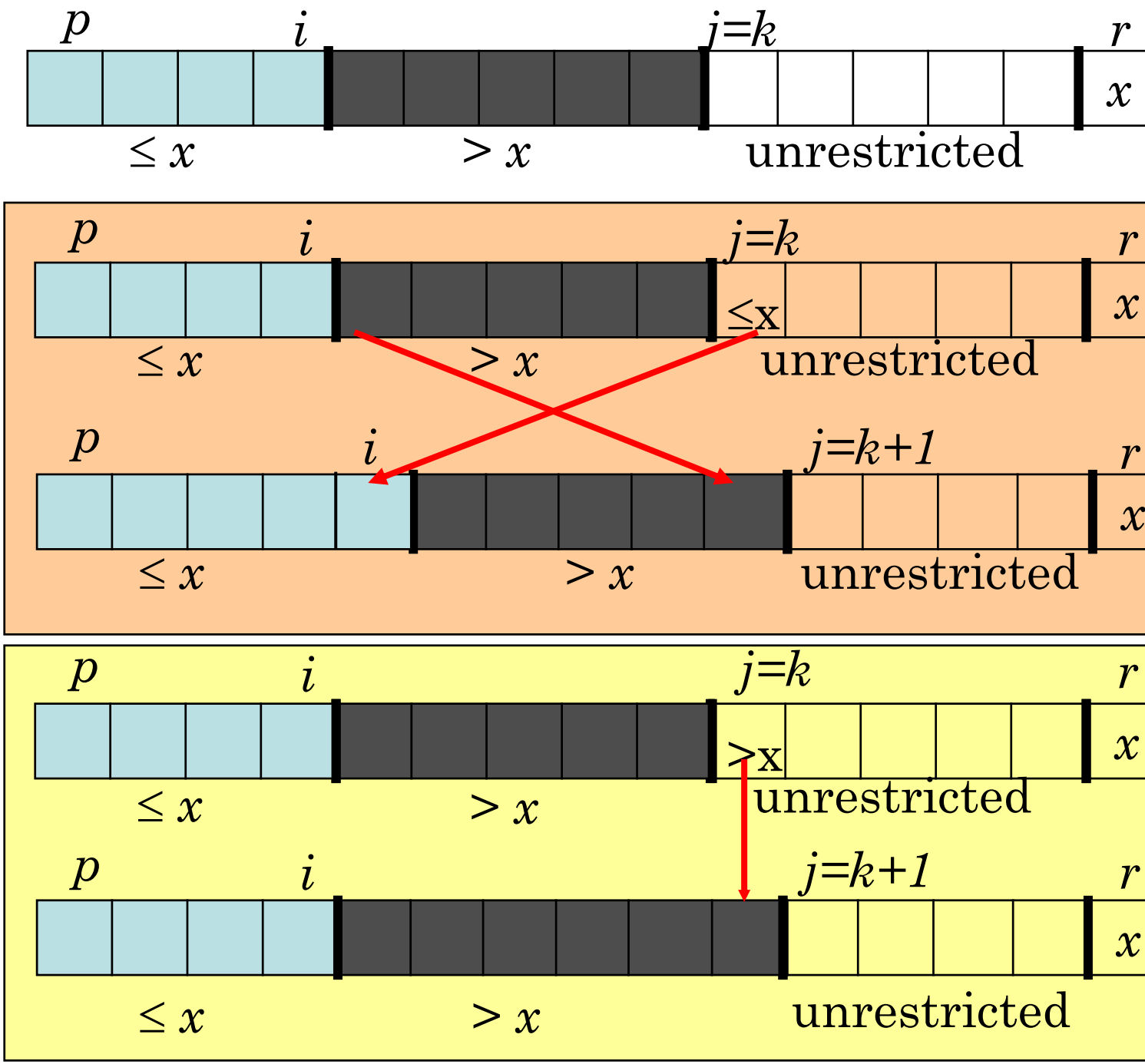


一保持阶段

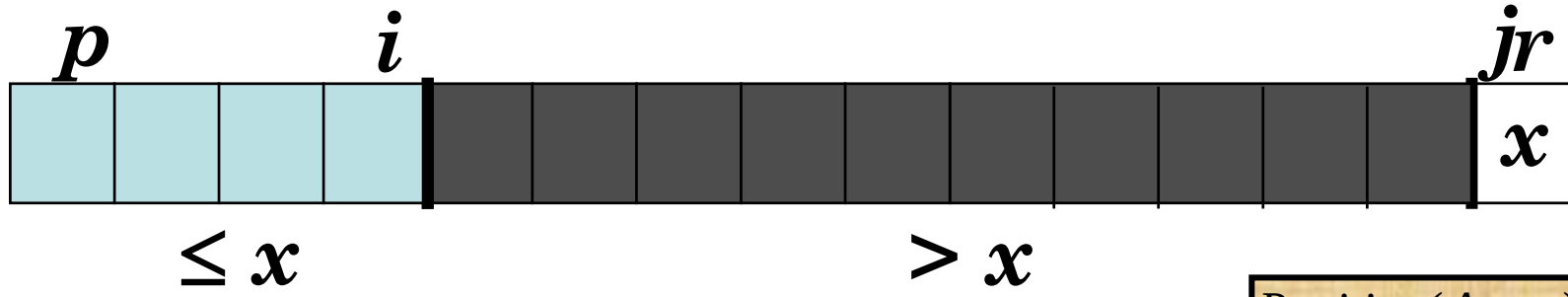
设$j=k$时循环不变量成立.

往证$j=k+1$时不变量成立.

```
Partition(A, p, r)
  x ← A[r];
  i ← p − 1;
  for j ← p to r − 1
     do if A[ j] ≤ x
           i ← i + 1;
           exchange A[i] ↔ A[ j];
  exchange A[i + 1] ↔ A[r];
  return i + 1;
```



– 终止阶段



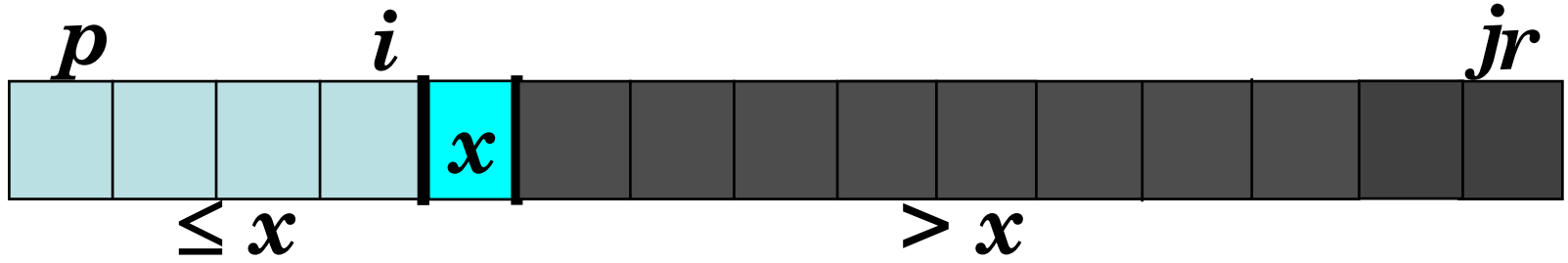
循环结束时, $j=r$, 产生三个集合:

1. 所有小于等于x的元素构成的集合.
2. 所有大于x的元素构成的集合.
3. 由元素x构成的集合.

算法结束时

最后一个步骤将A[r]与A[$i+1$]互换.

```
Partition(A, p, r)
  x ← A[r];
  i ← p − 1;
  for j ← p to r − 1
     do if A[j] ≤ x
           i ← i + 1;
           exchange A[i] ↔ A[j];
  exchange A[i + 1] ↔ A[r];
  return i + 1;
```



Performance Analysis

- Time complexity of PARTITION: $\theta(n)$
- Best case time complexity of *Quicksort*
  - Array in partition into 2 equal sets
  - $T(n)=2T(n/2)+\theta(n)$
  - $T(n)=\theta(nlogn)$

Performance Analysis

- Worst case time complexity of Quicksort
  - Worst Case
    - $|A[p..q-1]|=0$， $|A[q+1..r]|=n-1$

    - The worst case happens in call to Partition Algorithm
  - Worst case time complexity
    - $T(n)= T(0)+T(n-1)+\theta(n)=T(n-1)+\theta(n)=\theta(n^2)$

Performance Analysis

What is the average time complexity?

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

Why?

Performance Analysis

- 假如第一次划分后产生两个子序列，一个子序列包含s个元素，另一个子序列包含n-1-s个元素
- 一共有n种可能的划分，即$0 \le s \le n-1$，每种可能划分产生的概率为$1/n$
- 平均复杂性$T(n) = \frac{1}{n}\sum_{s=0}^{n-1}\big(T(s)+T(n-1-s)\big)+cn$

$$\frac{1}{n}\sum_{s=0}^{n-1}\big(T(s)+T(n-1-s)\big)=\frac{1}{n}\big(T(0)+T(n-1)+T(1)+T(n-2)+\cdots+T(n-1)+T(0)\big)$$

由于T(0)=0，有：$T(n)=\frac{1}{n}\big(2T(1)+2T(2)+\cdots+2T(n-1)\big)+cn$

$$nT(n)=2T(1)+2T(2)+\cdots+2T(n-1)+cn^2$$

Performance Analysis

$$nT(n) = 2T(1) + 2T(2) + \cdots + 2T(n-1) + cn^2$$

用n替换为n-1代入上式，有：

$$(n-1)T(n-1) = 2T(1) + 2T(2) + \cdots + 2T(n-2) + c(n-1)^2$$

两式相减：$nT(n) - (n-1)T(n-1) = 2T(n-1) + c(2n-1)$

$$nT(n) - (n+1)T(n-1) = c(2n-1)$$

$$\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = c\frac{(2n-1)}{(n+1)n} = c\frac{\left(3n-(n+1)\right)}{(n+1)n} = c\left(\frac{3}{n+1} - \frac{1}{n}\right)$$

递归地：$\frac{T(n-1)}{n} - \frac{T(n-2)}{n-1} = c\left(\frac{3}{n} - \frac{1}{n-1}\right)$

…

$$\frac{T(2)}{3} - \frac{T(1)}{2} = c\left(\frac{3}{3} - \frac{1}{2}\right)$$

Performance Analysis

我们得到：

$$\frac{T(n)}{n+1}-\frac{T(n-1)}{n}=\mathrm{c}\left(\frac{3}{n+1}-\frac{1}{n}\right)$$

$$\frac{T(n-1)}{n}-\frac{T(n-2)}{n-1}=\mathrm{c}\left(\frac{3}{n}-\frac{1}{n-1}\right)$$

…

$$\frac{T(2)}{3}-\frac{T(1)}{2}=\mathrm{c}\left(\frac{3}{3}-\frac{1}{2}\right)$$

$$\frac{T(n)}{n+1}=3\mathrm{c}\left(\frac{1}{n+1}+\frac{1}{n}+\cdots+\frac{1}{3}\right)-c\left(\frac{1}{n}+\frac{1}{n-1}+\cdots+\frac{1}{2}\right)+\frac{T(1)}{2}$$

$$=2\mathrm{c}\left(\frac{1}{n}+\frac{1}{n-1}+\cdots+\frac{1}{3}+\frac{1}{2}+1\right)-\frac{7}{2}c+\frac{T(1)}{2}+\frac{3c}{n+1}$$

$$=2cH_n-\frac{7}{2}c+\frac{T(1)}{2}+\frac{3c}{n+1}$$

$$T(n)=2cnH_n+2cH_n+\left(\frac{T(1)}{2}-\frac{7}{2}c\right)n+\left(\frac{T(1)}{2}-\frac{7}{2}c+3c\right)=O(n\log n)$$

Randomized Quicksort Algorithms

- **Randomized-Partition(*A*, *p*, *r*)**

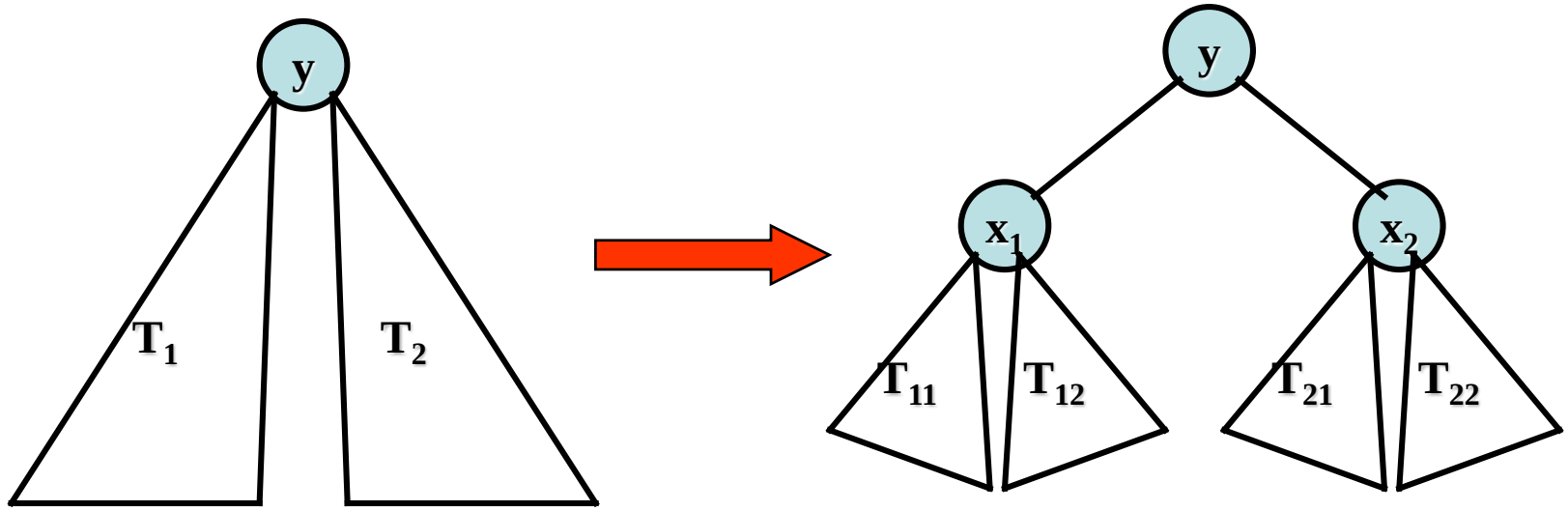
  1. i := Random(p, r)
  2. $A[r] \leftrightarrow A[i]$;
  3. Return Partition(A, p, r)

- **Randomized-Quicksort(*A*, *p*, *r*)**

  1. **If** $p < r$
  2. **Then** q := Randomized-Partition(A, p, r);
  3. Randomized-Quicksort(A, p, q-1);
  4. Randomized-Quicksort(A, q+1, r).

随机快速排序复杂性分析

- 我们可以用树表示算法的计算过程



- 我们可以观察到如下事实:
  - 一个子树的根必须与其子树的所有节点比较
  - 不同子树中的节点不可能比较
  - 任意两个节点至多比较一次

随机快速排序复杂性分析

- 基本概念
  - $x_{(i)}$表示A中Rank为i的元素(第i小元素)

    例如，$x_{(1)}$和$x_{(n)}$分别是最小和最大元素
  - 随机变量X_{ij}定义如下:

    X_{ij}=1如果$x_{(i)}$和$x_{(j)}$在运行中被比较，否则为0

    X_{ij}是$x_{(i)}$和$x_{(j)}$的比较次数
  - 算法的比较次数为 $\sum_{i=1}^{n}\sum_{j>i} X_{ij}$
  - 算法的复杂性为T(n)= $E[\sum_{i=1}^{n}\sum_{j>i} X_{ij}]=\sum_{i=1}^{n}\sum_{j>i} E[X_{ij}]$

随机快速排序复杂性分析

$$T(n)=E[\sum_{i=1}^{n}\sum_{j>i}X_{ij}]=\sum_{i=1}^{n}\sum_{j>i}E[X_{ij}]$$

- 计算$E[X_{ij}]$
  - 设p_{ij}为$x_{(i)}$和$x_{(j)}$在运行中被比较的概率，则

$$E[X_{ij}]=p_{ij}\times 1+(1-p_{ij})\times 0=p_{ij}$$

关键问题成为求解p_{ij}

随机快速排序复杂性分析

- 求解p_{ij}
  - $Z_{ij}=\{x_{(i)}, x_{(i+1)}, \cdots, x_{(j)}\}$是$x_{(i)}$和$x_{(j)}$之间元素集合，$Z_{ij}$在同一棵子树时，$x_{(i)}$和$x_{(j)}$才可能比较.
  - $x_{(i)}$和$x_{(j)}$在执行中被比较，需满足下列条件：
    - $x_{(i)}$是Z_{ij}中第一个被选择的子树根节点，或者
    - $x_{(j)}$是Z_{ij}中第一个被选择的子树根节点
  - 一棵子树所有点等可能地被选为划分点，所以$x_{(i)}$或$x_{(j)}$被选为划分点的概率 $= 2/|Z_{ij}|=2/(j-i+1)$.
  - $x_{(i)}$和$x_{(j)}$被进行比较的概率：

$$p_{ij} = 2/(j-i+1)$$

随机快速排序复杂性分析

- 现在我们有

$$\sum_{i=1}^{n}\sum_{j>i}E[X_{ij}]=\sum_{i=1}^{n}\sum_{j>i}p_{ij}=\sum_{i=1}^{n}\sum_{j>i}\frac{2}{j-i+1}$$

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

$$\le 2\sum_{i=1}^{n}\sum_{k=1}^{n}\frac{1}{k}=2nH_n=O(n\log n)$$

定理. 随机排序算法的期望时间复杂性为$O(n\log n)$

3.2.2 排序问题的下界

问题的下界

- 问题的下界(lower bound of a problem)
  - 是用于解决该问题的任意算法所需要的最小时间复杂度
  - 问题难度的一种度量
    - 如果问题可由一个具有较低时间复杂性的算法解决，则该问题是简单的；否则是困难的
  - 通常指：最坏情况下界
- 问题的下界是不唯一的
  - 例如. $\Omega(1)$, $\Omega(n)$, $\Omega(n\log n)$ 都是排序的下界
  - 只有$\Omega(n\log n)$是有意义的
  - 下界应尽可能地高，达到上限
  - 下界的分析都是经过严格理论分析和证明，而非纯粹猜测

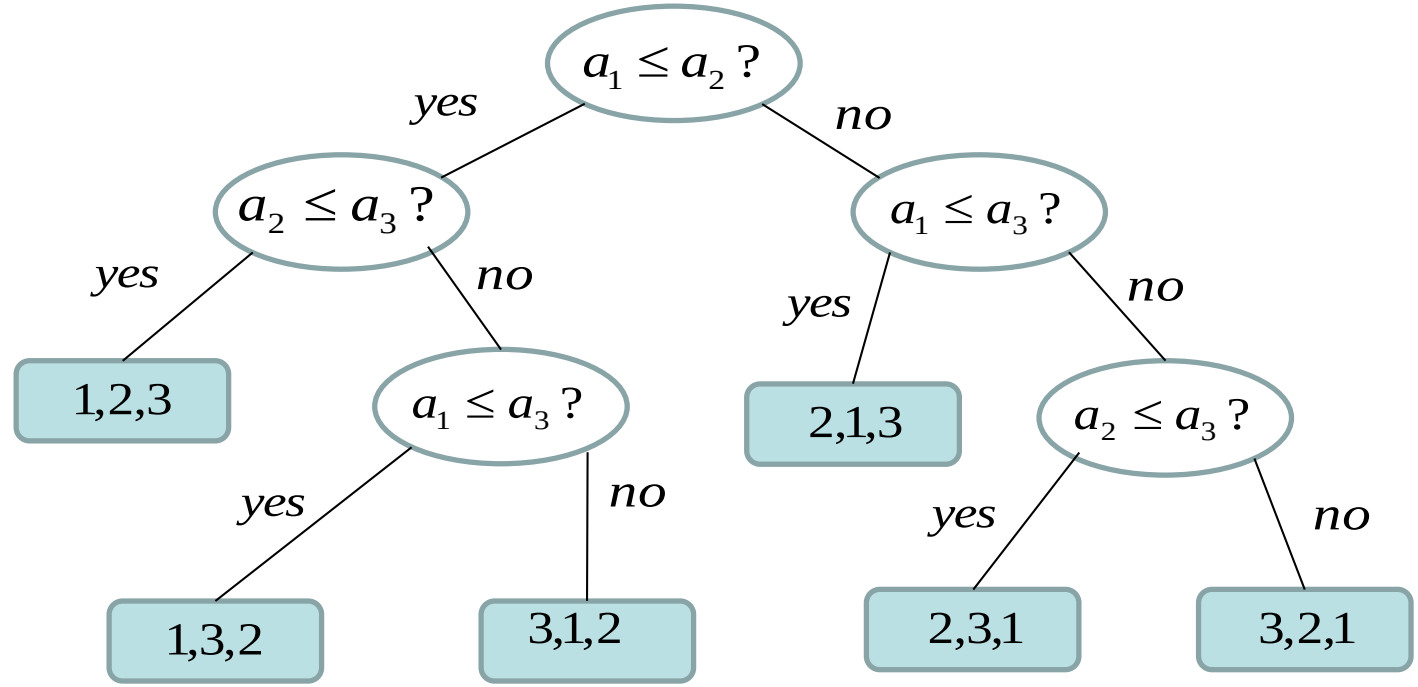
问题的下界的意义

- 如果一个问题的最高下界是$\Omega(n\log n)$而当前最好算法的时间复杂性是$O(n^2)$.
  - 我们可以寻找一个更高的下界.
  - 我们可以设计更好的算法.
  - 下界和算法都是可能改进的.
- 如果一个问题的下界是$\Omega(n\log n)$且算法的时间复杂性是$O(n\log n)$, 那么这个算法是最优的

排序的下界

- 通常，基本操作是比较和交换的排序算法可以用一个二叉决策树描述
  - 通过忽略比较以外的细节来抽象表示比较排序算法
  - 每个内节点表示一个比较操作$a_i \leq a_j$；
  - 所有被排序元素的全排列是树的叶节点；

对于特定输入数据集的排序过程，对应于从树的根结点到叶子节点的一条路径



排序的下界

- n个元素有$n!$种不同排列
- 其排序过程对应于一个高度为h，具有$n!$个叶子节点的二叉决策树.
- 由于高度为h的二叉树至多有2^h个叶子节点
- 则有$2^h \geq n!$

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

即： $h \geq \lg(n!) = \Omega(n\lg n)$

排序的下界是: $\Omega(n\log n)$

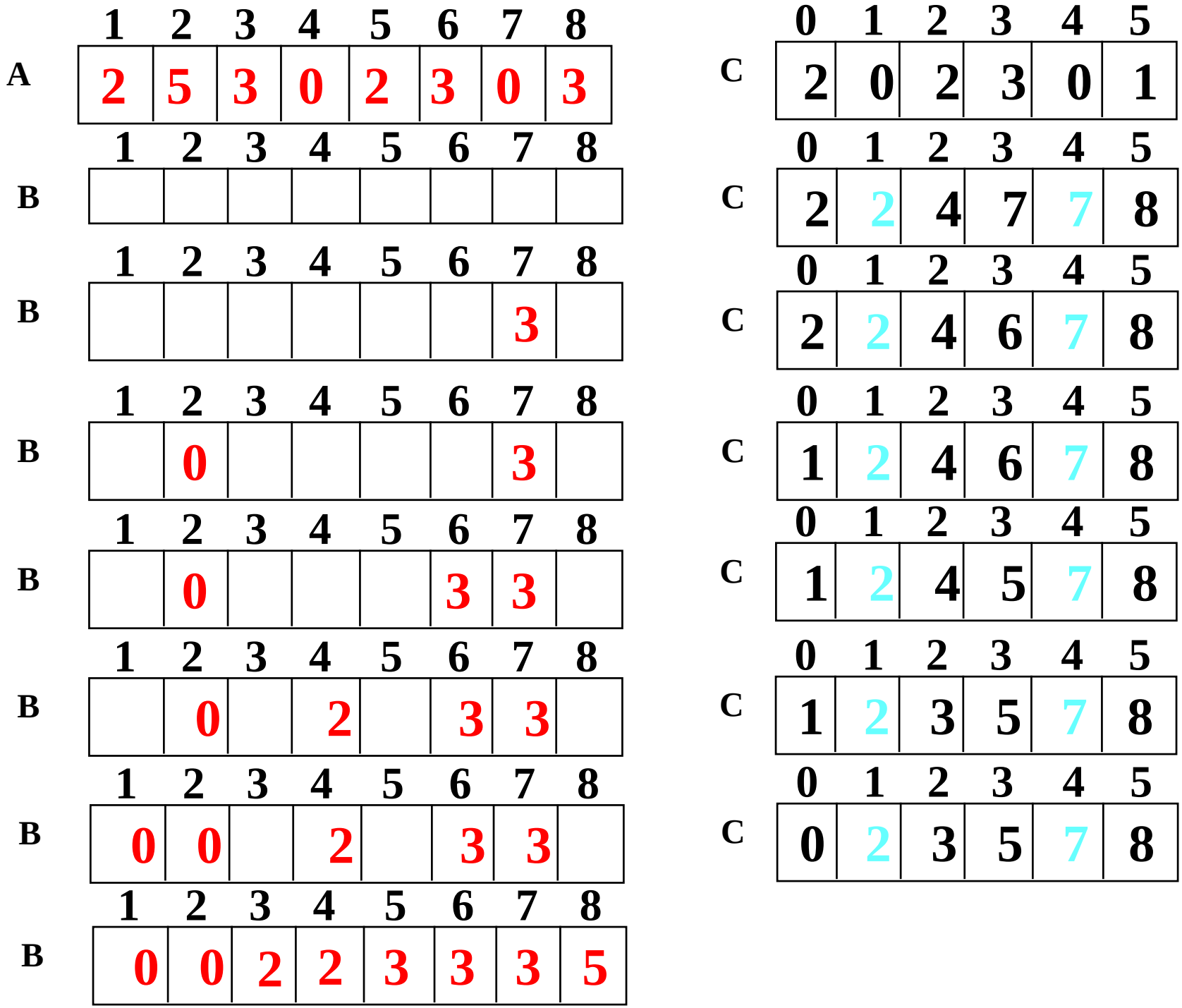
HITWH
SE

3.3 Sorting in Linear Time

- Counting Sort Algorithm
- Radix Sort Algorithm
- Bucket Sort Algorithm

Counting Sort

- Input: $A[1..n]$, $0 \le A[i] \le k$ for $1 \le i \le n$
- Output: $B[1..n]$=sorted $A[1..n]$
- Idea
  - Use $C[0..k]$ to compute the position of each $A[i]$
  - Put each $A[i]$ for $i=n$ *to* 1 into $B[C[A[i]]]$


A
1 2 3 4 5 6 7 8
2 5 3 0 2 3 0 3
C
0 1 2 3 4 5
2 0 2 3 0 1
B
1 2 3 4 5 6 7 8
C
0 1 2 3 4 5
2 2 4 7 7 8
B
1 2 3 4 5 6 7 8
3
C
0 1 2 3 4 5
2 2 4 6 7 8
B
1 2 3 4 5 6 7 8
0 3
C
0 1 2 3 4 5
1 2 4 6 7 8
B
1 2 3 4 5 6 7 8
0 3 3
C
0 1 2 3 4 5
1 2 4 5 7 8
B
1 2 3 4 5 6 7 8
0 2 3 3
C
0 1 2 3 4 5
1 2 3 5 7 8
B
1 2 3 4 5 6 7 8
0 0 2 3 3
C
0 1 2 3 4 5
0 2 3 5 7 8
B
1 2 3 4 5 6 7 8
0 0 2 2 3 3 3 5

- Algorithm and Time complexity

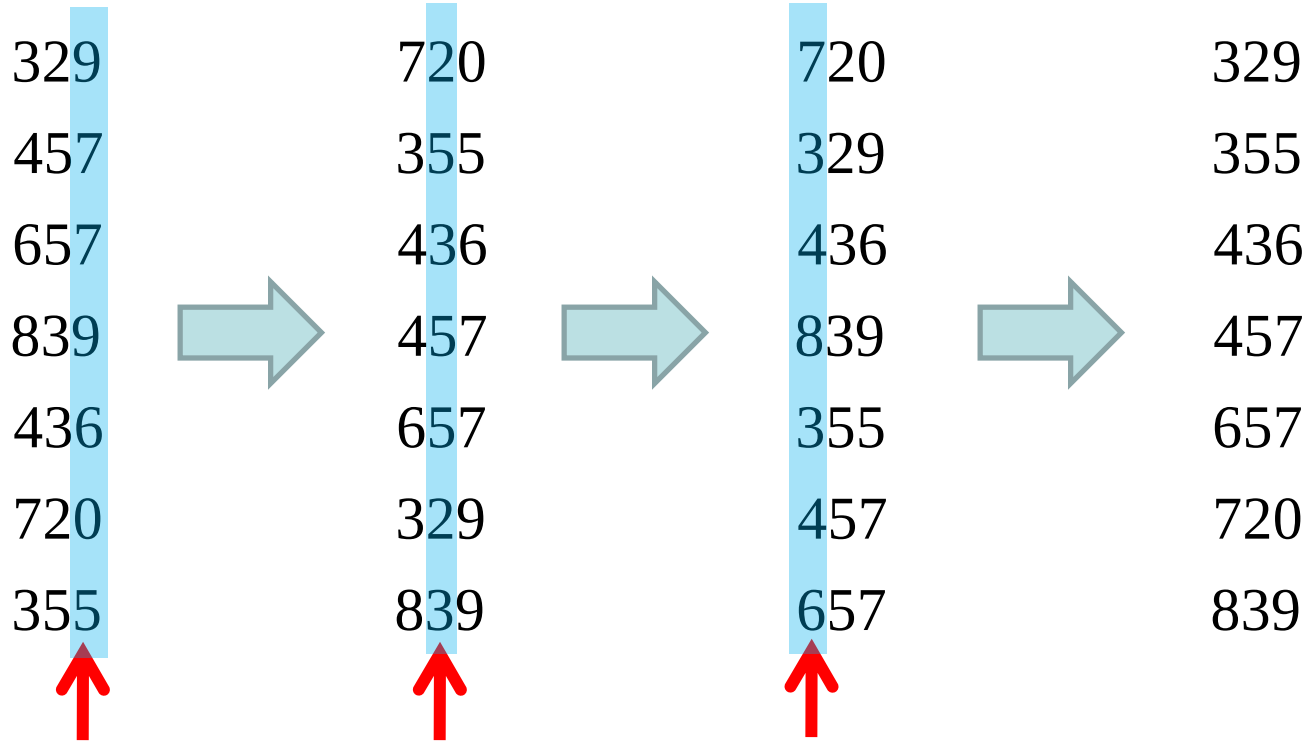
```
for i ← 0 to k                                   O(k)
    do  C[i] ← 0;
for j ← 1 to length[A]                           O(n)
    do  C[A[j]] ← C[A[j]] + 1;
for i ← 1  to k                                  O(k)
    do  C[i] ← C[i] + C[i − 1];
for j ← length[A]  downto 1                      O(n)
    do  begin
        B[C[A[j]]] ← A[j];
        C[A[j]] ← C[A[j]] − 1;
```

Time Complexity=O(n+k)

- Counting Sort算法性质
  - Counting sort doesn't sort in place
  - Counting sort is stable
    - That is, the same values appear in the output array in the same order as they do in the input array.
  - Problems
    - *A[i]* must be integer.
    - *k* should be small

Radix Sort Algorithm

- Idea of Radix sort algorithm
  - Use stable sort algorithm
  - Sort the *n* *d*-digit elements from the lowest digit to the highest digit



- Radix sort algorithm

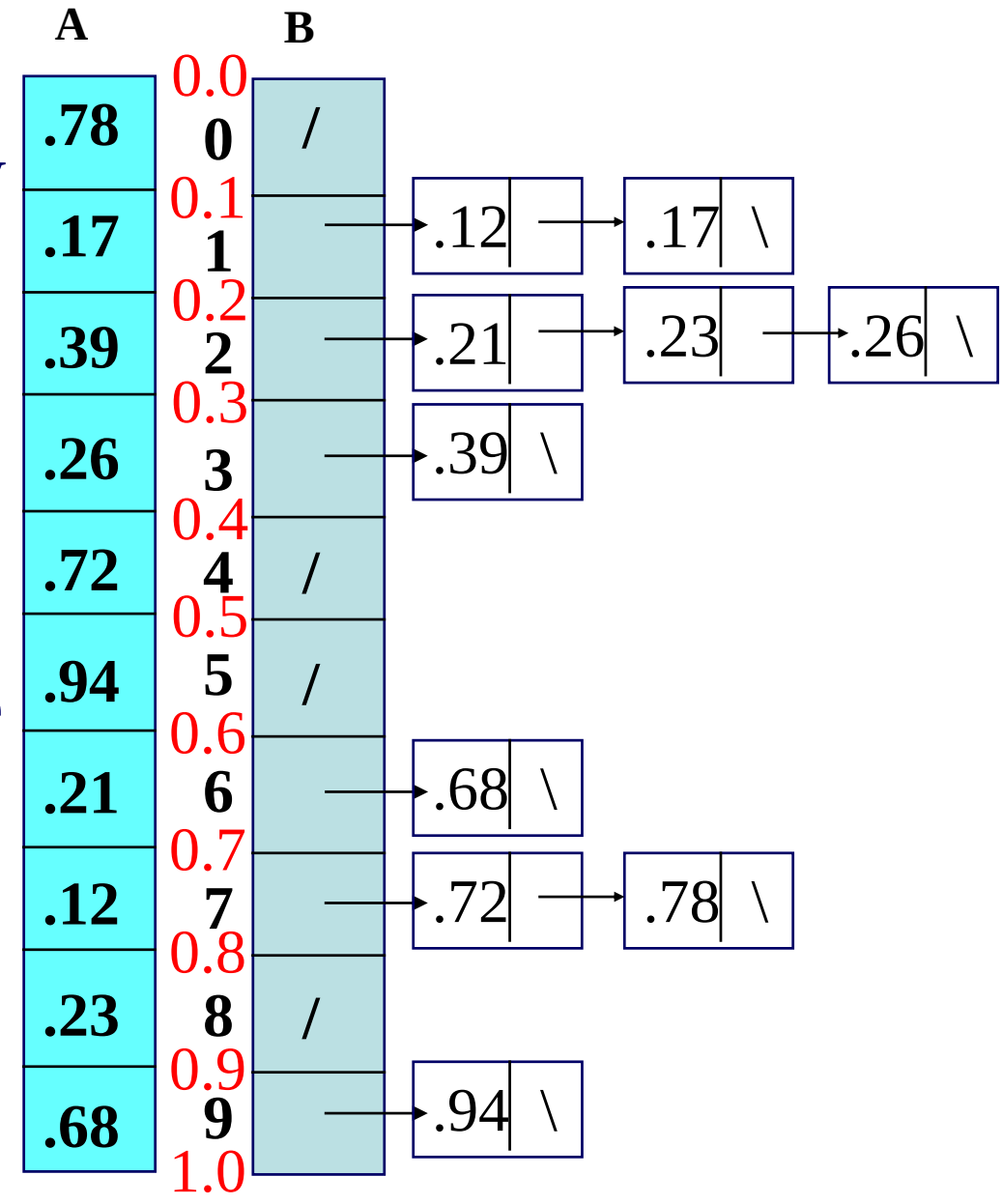
Input: Array A, each element is a number of d digit.
Radix−Sort(A, d)
for $i \leftarrow 1$ to d do
    use a stable sort to sort array A on digit i;

- Time complexity of Radix sort algorithm
  - Using Counting sort algorithm, $0 \leq A[i] \leq k$
  - The time complexity is $O(d(n+k))$
- Problems

- Extension of Radix sort
  - Input: n b-binary-digit number, any $r \leq b$
  - Radix sort can sort these numbers in $\Theta((b/r)(n+2^r))$
  - Why
    - View each number as $d = \lceil b/r \rceil$ digits of r bits each.
    - Each digit is an integer in the range 0 to $2^r - 1$
    - Use counting sort with $k = 2^r - 1$
  - How about b=500, r=100?

Bucket Sort

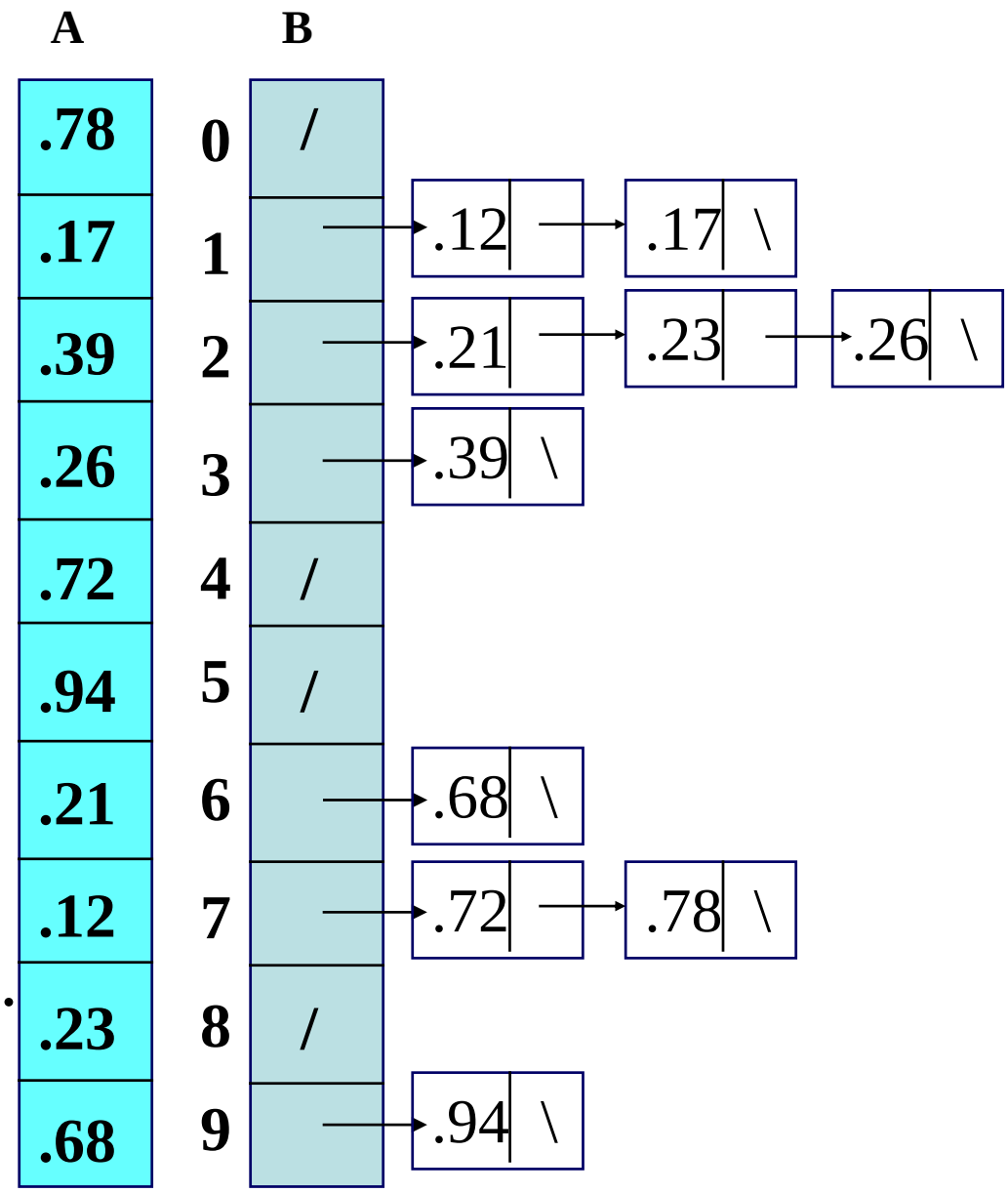
- **Assumption of Bucket Sort**
  - Input is elements uniformly distributed in [0, 1) independently
- **Idea of Bucket Sort**
  - Divide [0, 1) into *n* equal-sized bucket
  - Distribute the input into the *n* bucket
  - Sort the numbers in each bucket
  - List all the sorted numbers in each bucket in order



- **Bucket Sort Algorithm**

Bucket-Sort(*A*)

1. *n*=*length*[*A*];
2. For *i*=*1* To *n* Do
3. Insert *A*[*i*] into list *B*[$\lfloor nA[i] \rfloor$];
4. For i=0 To n-1 Do
5. Sort list *B*[*i*] with insert sort;
6. concatenate lists *B*[*0*], …, *B*[*n-1*].



- **Time complexity**
  - Let the random variable $n_i=|B[i]|$
  - The time complexity:

$$T(n)=\Theta(n)+\sum_{i=0}^{n-1}O(n_i^{\,2})$$

  - Since $E[n_i^2]=2-1/n$

$$E[T(n)]=\Theta(n)+\sum_{i=0}^{n-1}E[O(n_i^{\,2})]$$

$$=\Theta(n)+O(n(2-1/n))=\Theta(n)$$