排序

湖南大学信息科学与工程学院

Menu

- Sorting!
  - Insertion Sort
  - Merge Sort
- Solving Recurrences

The problem of sorting

Input: array $A[1 \ldots n]$ of numbers.

Output: permutation $B[1 \ldots n]$ of A such that $B[1] \leq B[2] \leq \cdots \leq B[n]$.

e.g. $A = [7, 2, 5, 5, 9.6] \rightarrow B = [2, 5, 5, 7, 9.6]$

How can we do it efficiently ?

Why Sorting?

- Obvious applications
  - Organize an MP3 library
  - Maintain a telephone directory
- Problems that become easy once items are in sorted order
  - Find a median, or find closest pairs
  - Binary search, identify statistical outliers
- Non-obvious applications
  - Data compression: sorting finds duplicates
  - Computer graphics: rendering scenes front to back

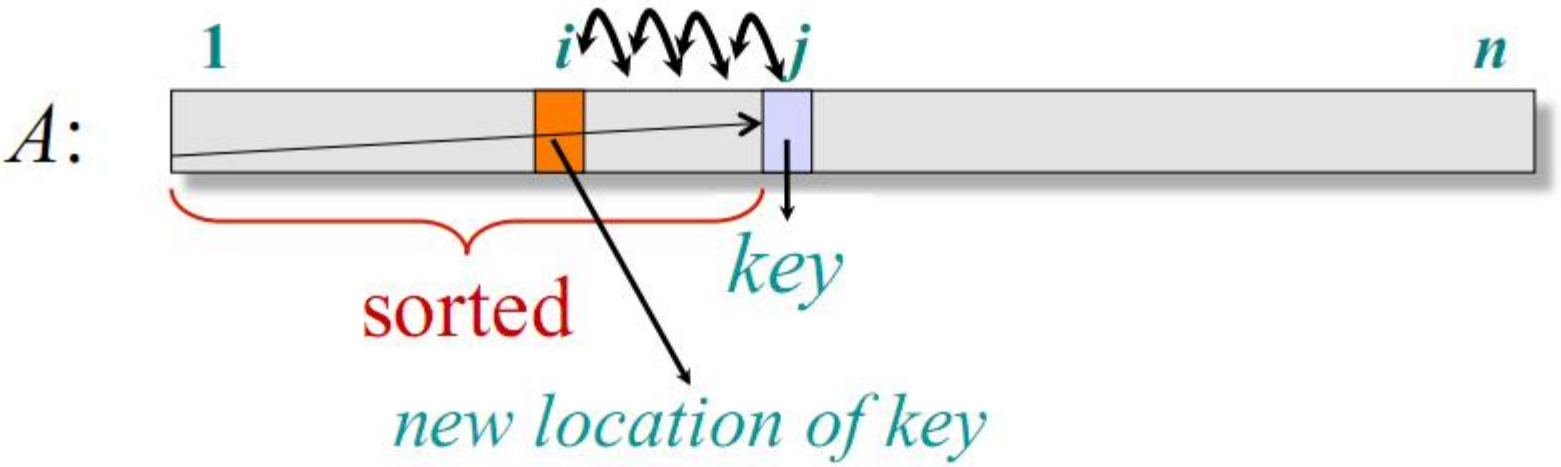
Insertion sort

INSERTION-SORT (A, n) $\triangleright$ $A[1\ ..\ n]$

for $j \leftarrow 2$ **to** n

insert key $A[j]$ **into the (already sorted) sub-array** $A[1\ ..\ j\text{-}1]$**.**
by pairwise key-swaps down to its right position

Illustration of iteration *j*

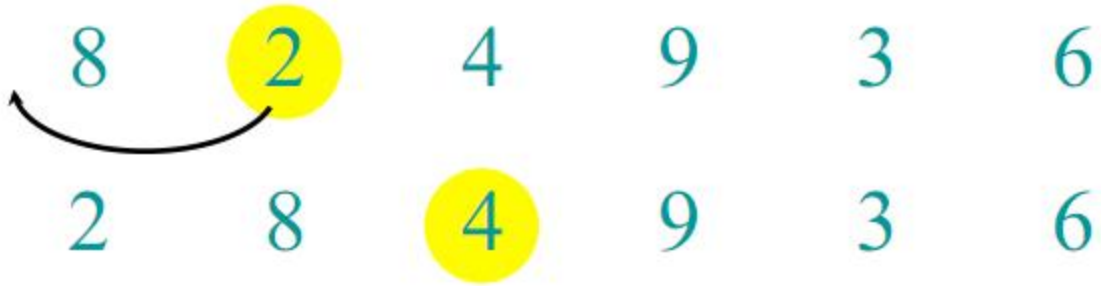


Example of insertion sort

8 2 4 9 3 6

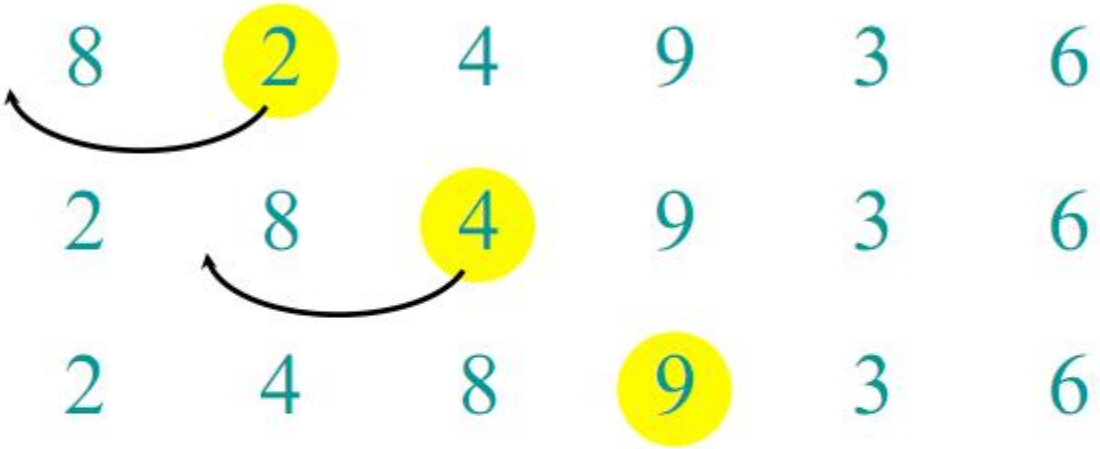
Example of insertion sort

Example of insertion sort

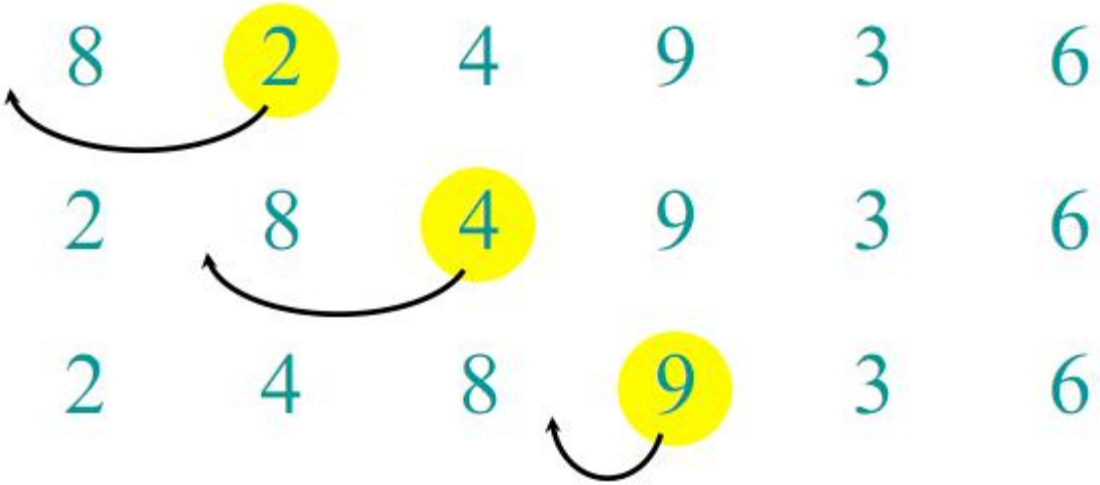


Example of insertion sort

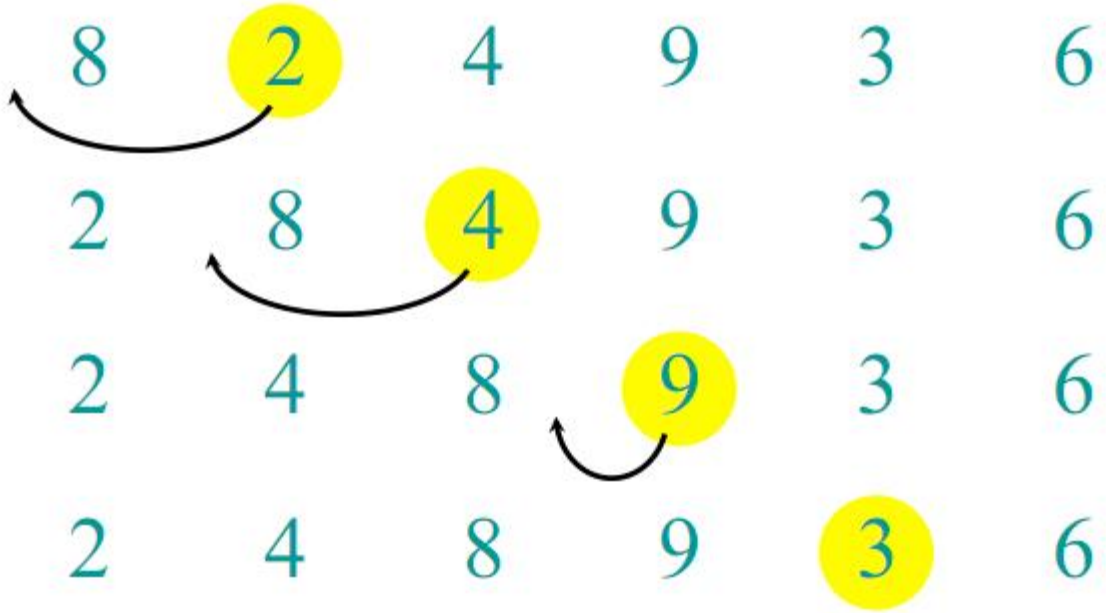
Example of insertion sort



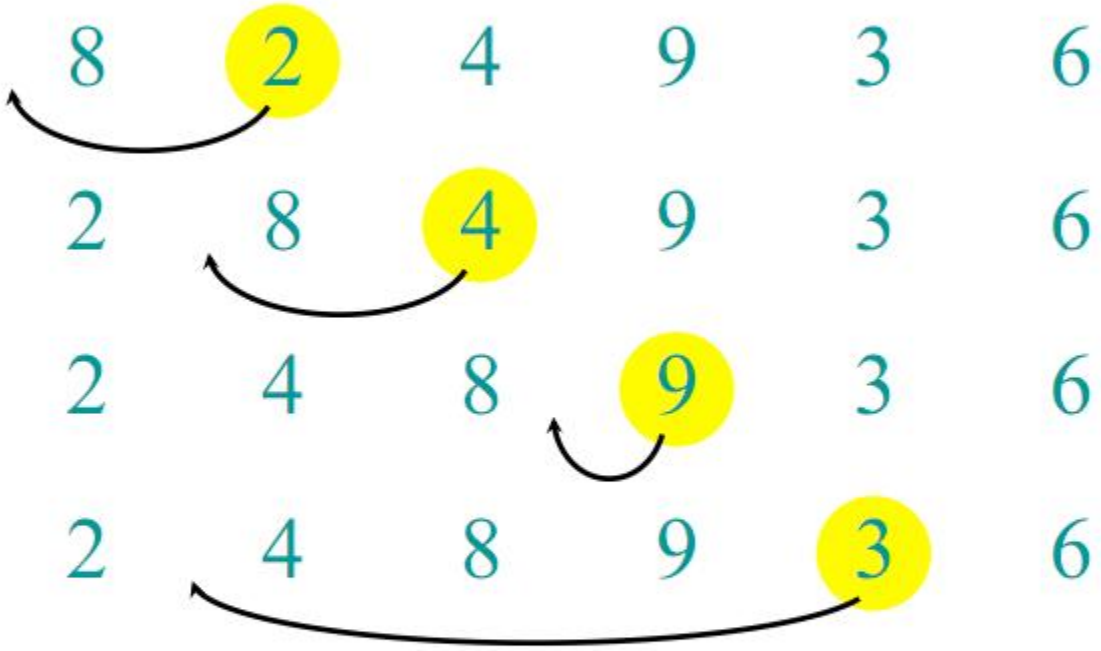
Example of insertion sort



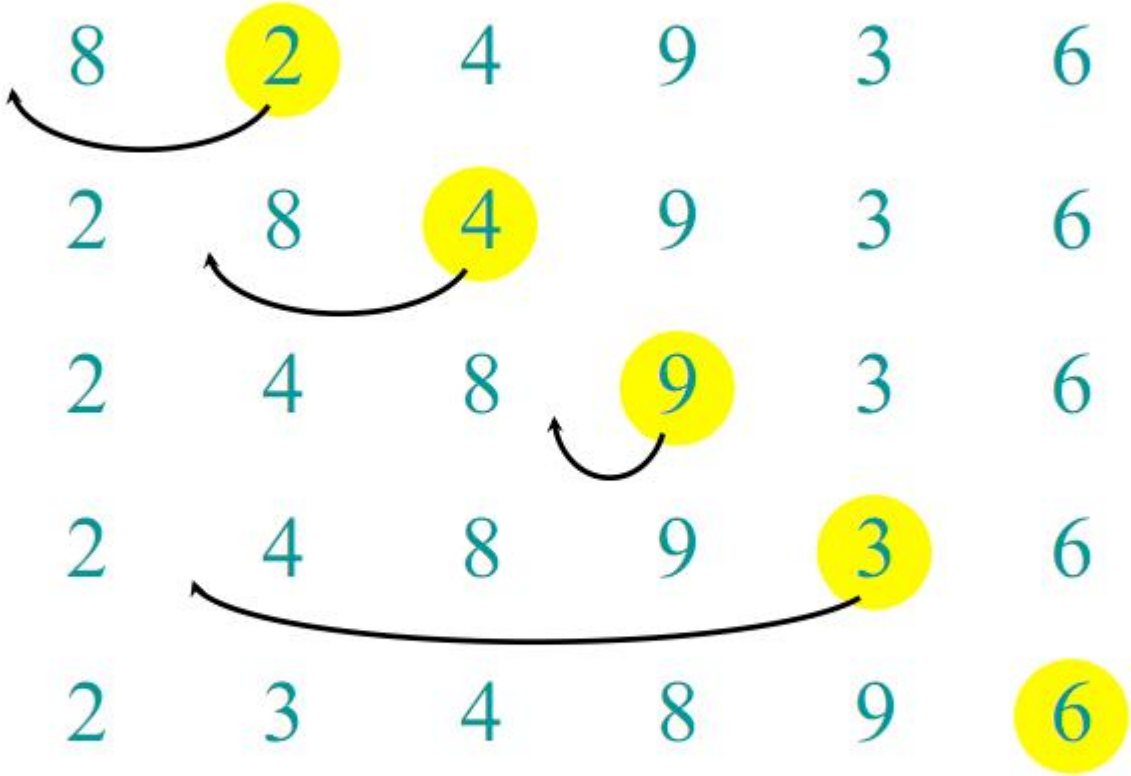
Example of insertion sort



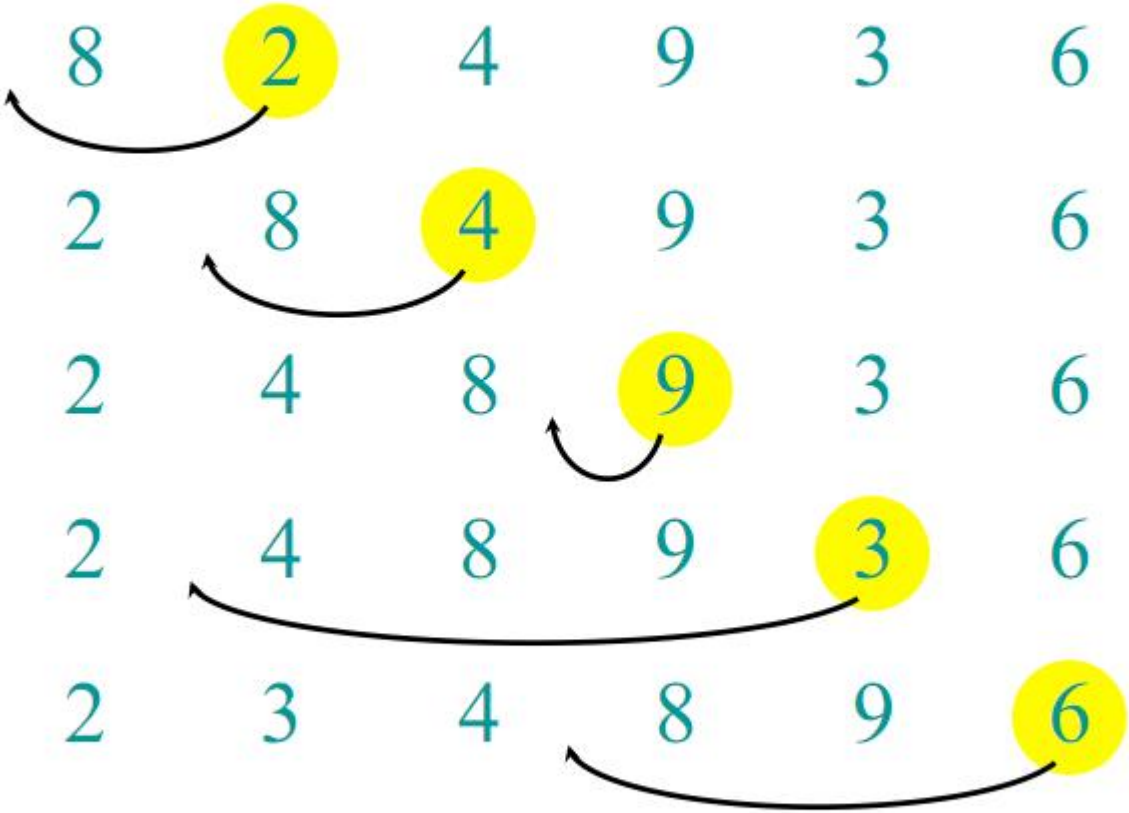
Example of insertion sort



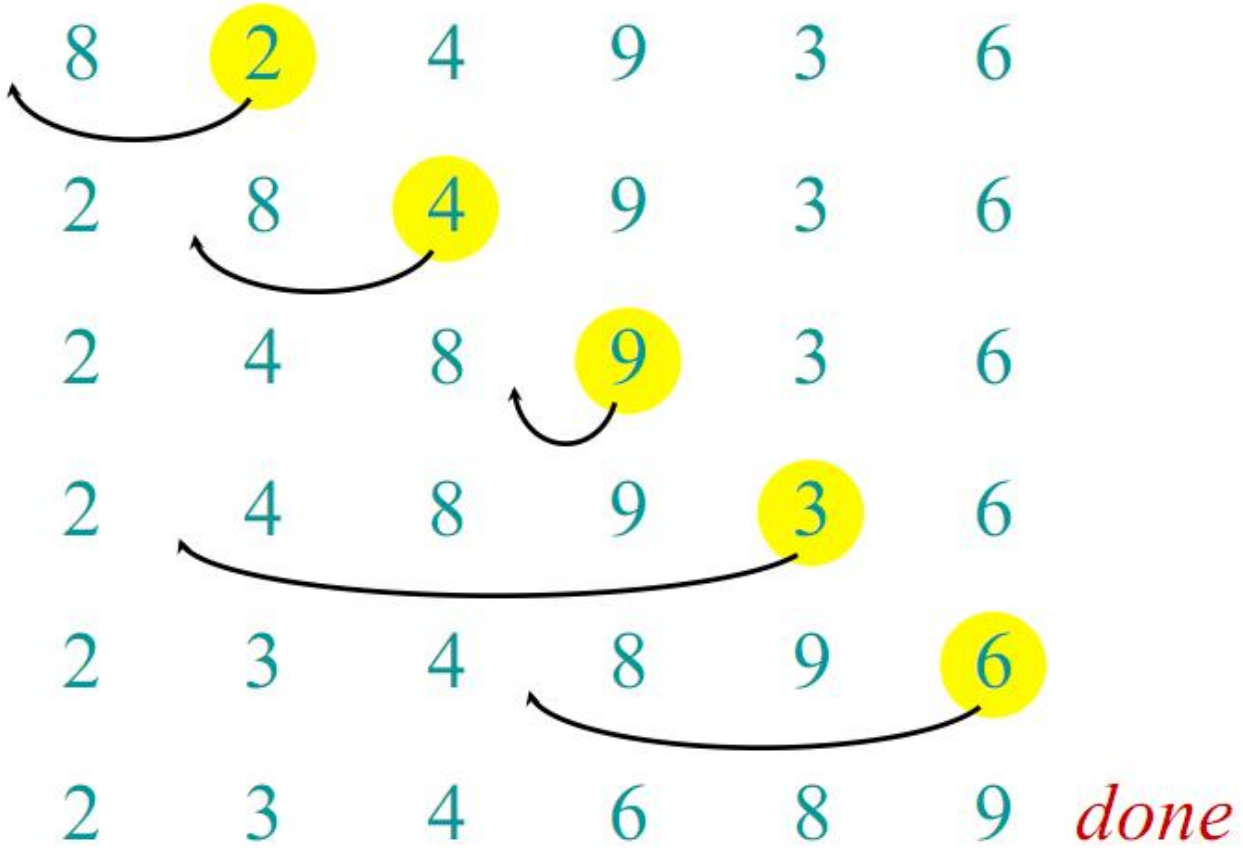
Example of insertion sort



Example of insertion sort



Example of insertion sort



Running time? $\Theta(n^2)$ because $\Theta(n^2)$ compares and $\Theta(n^2)$ swaps

e.g. when input is $A = [n, n-1, n-2, \ldots, 2, 1]$

Binary Insertion sort

BINARY-INSERTION-SORT (A, n) $\triangleright$ $A[1 \,..\, n]$

for $j \leftarrow 2$ **to** n

insert key $A[j]$ **into the (already sorted) sub-array** $A[1 \,..\, j\text{-}1]$**.**
Use binary search to find the right position

Binary search with take $\Theta(\log n)$ time.
However, shifting the elements after insertion will still take $\Theta(n)$ time.

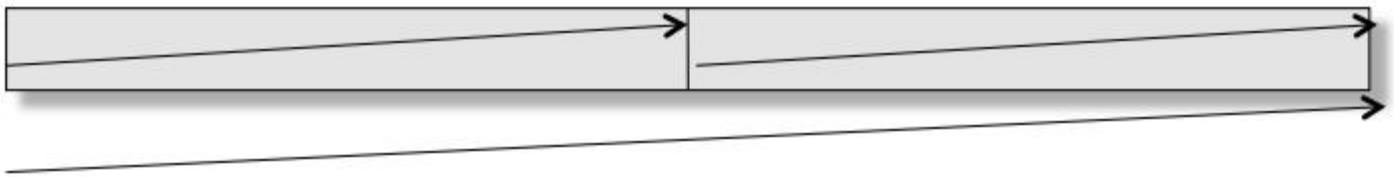
Complexity: $\Theta(n \log n)$ comparisons
$\Theta(n^2)$ swaps

Meet Merge Sort

divide and conquer

MERGE-SORT $A[1 \ldots n]$

1. If $n = 1$, done (nothing to sort).
2. Otherwise, recursively sort $A[\,1 \ldots n/2\,]$ and $A[\,n/2+1 \ldots n\,]$.
3. "***Merge***" the two sorted sub-arrays.

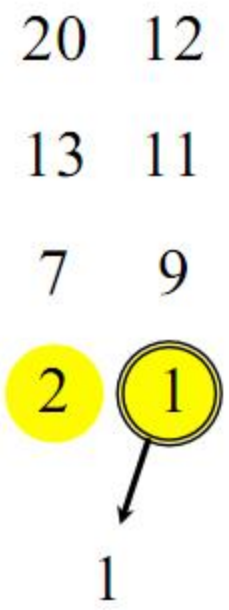


Key subroutine: **MERGE**

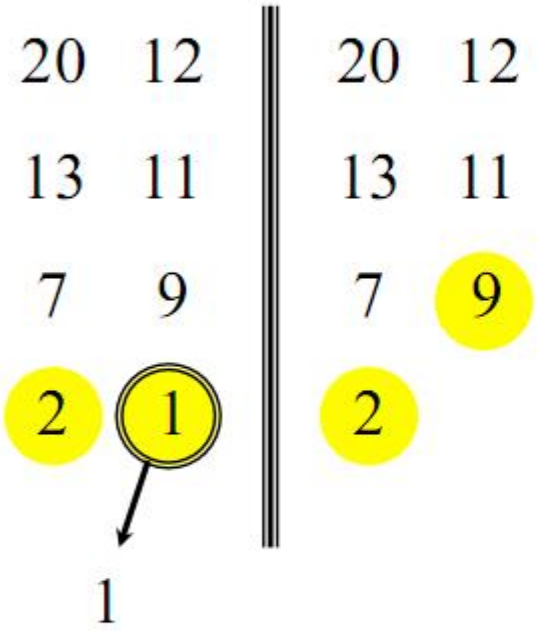
Merging two sorted arrays

20	12
13	11
7	9
2	1

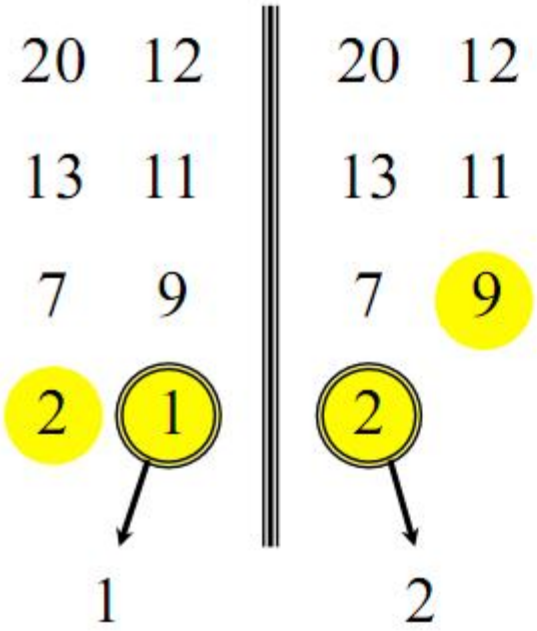
Merging two sorted arrays



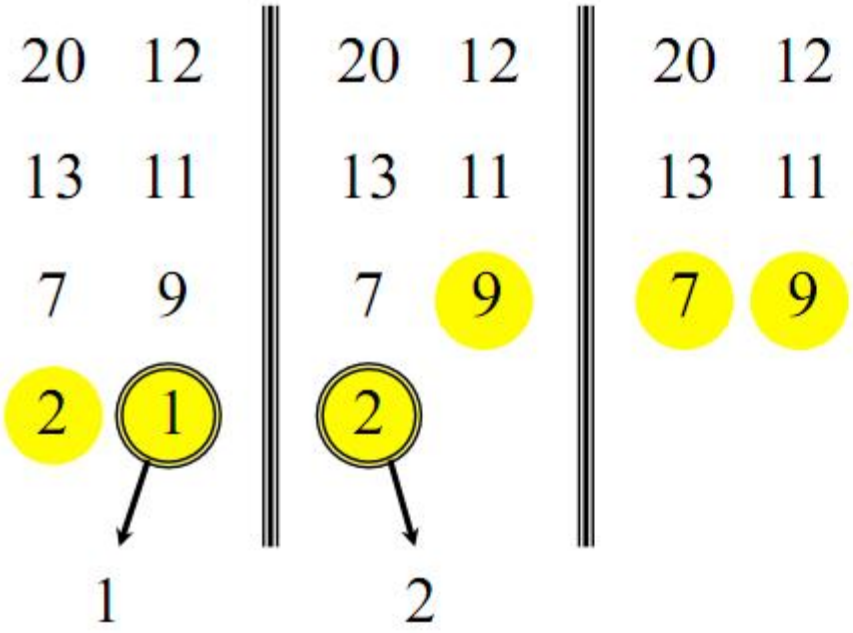
Merging two sorted arrays



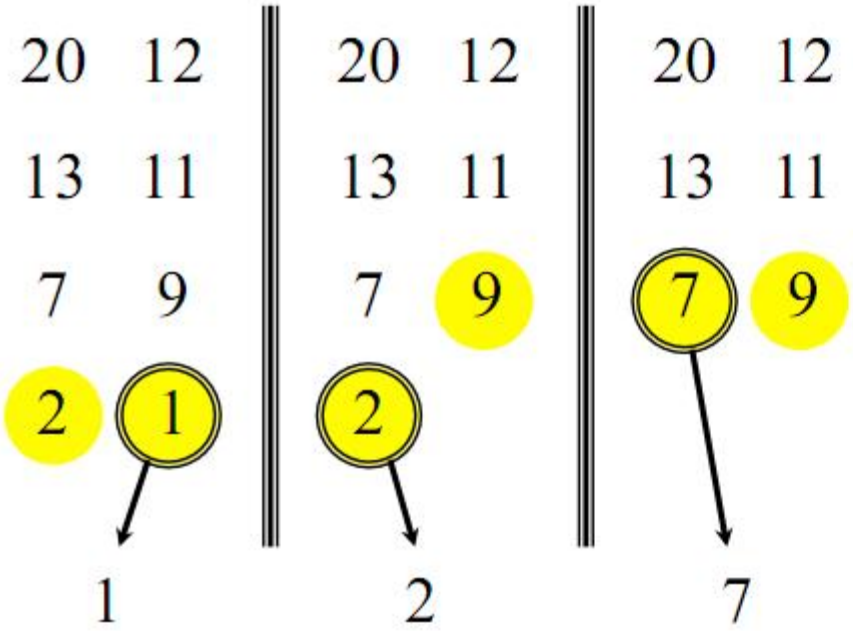
Merging two sorted arrays



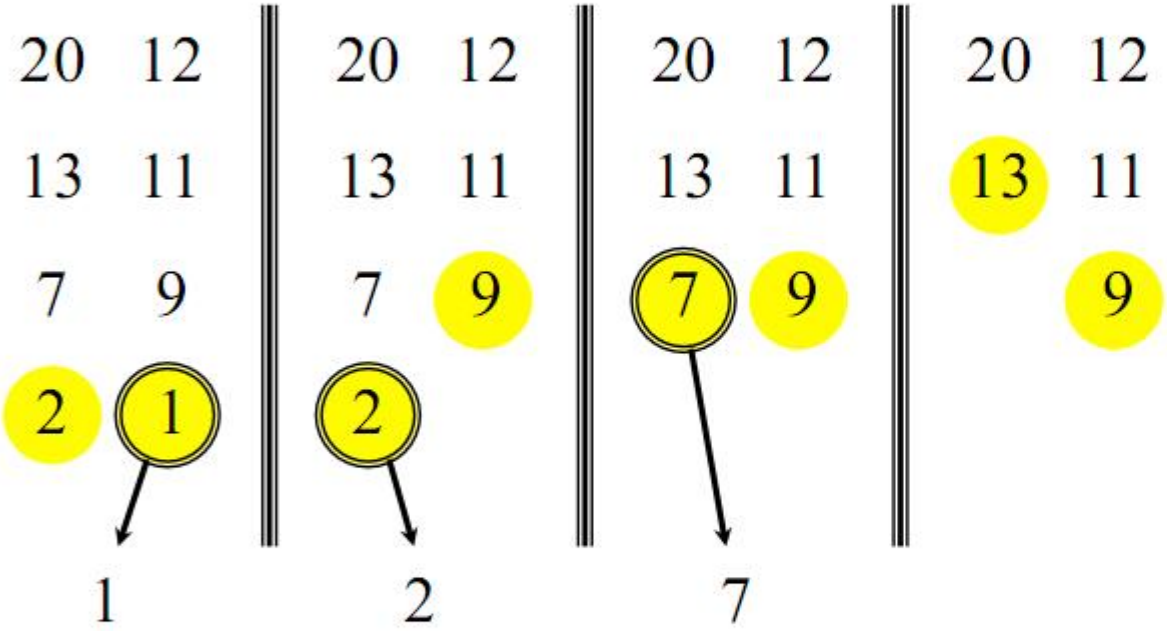
Merging two sorted arrays



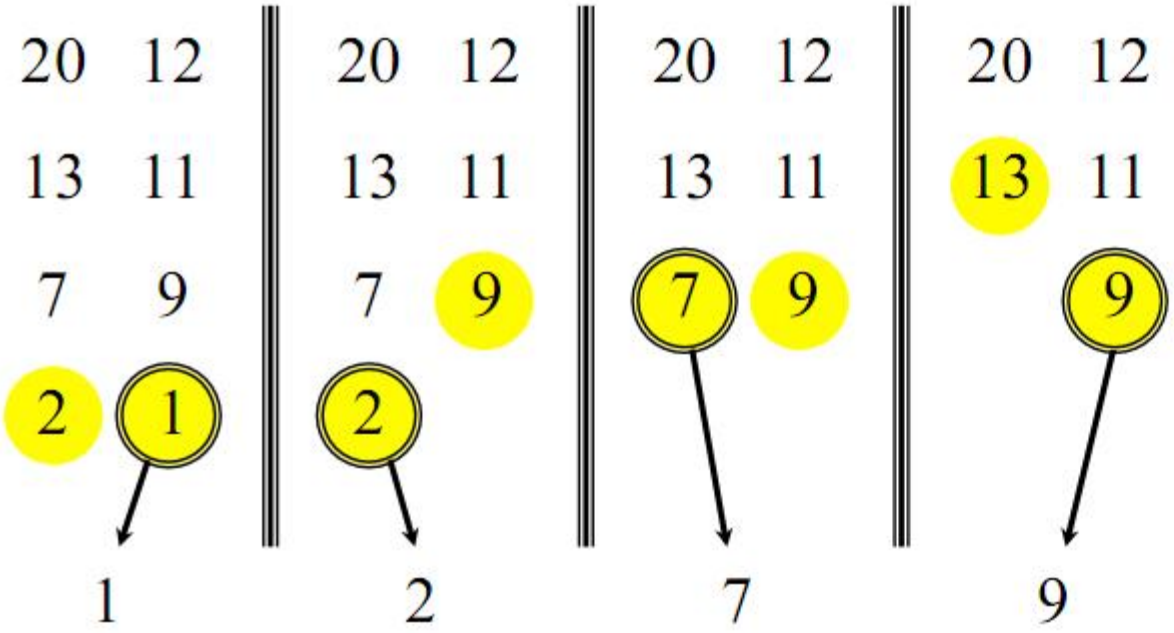
Merging two sorted arrays



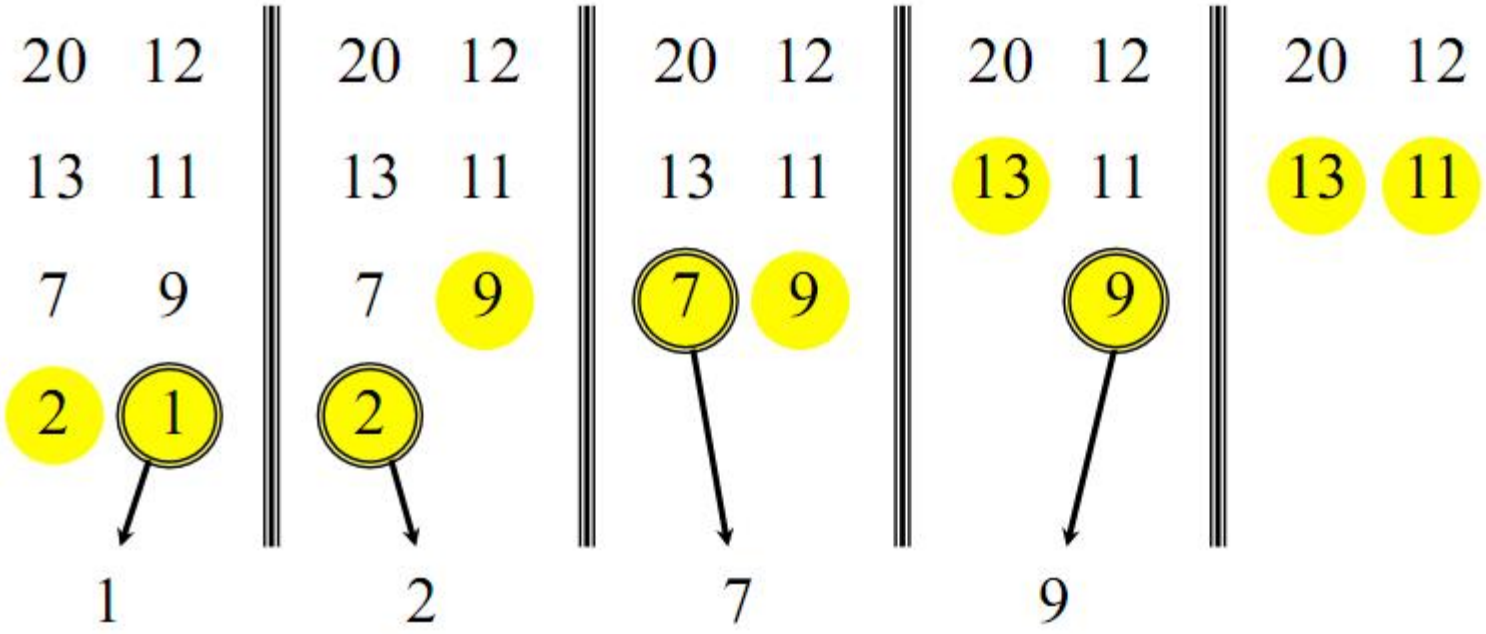
Merging two sorted arrays



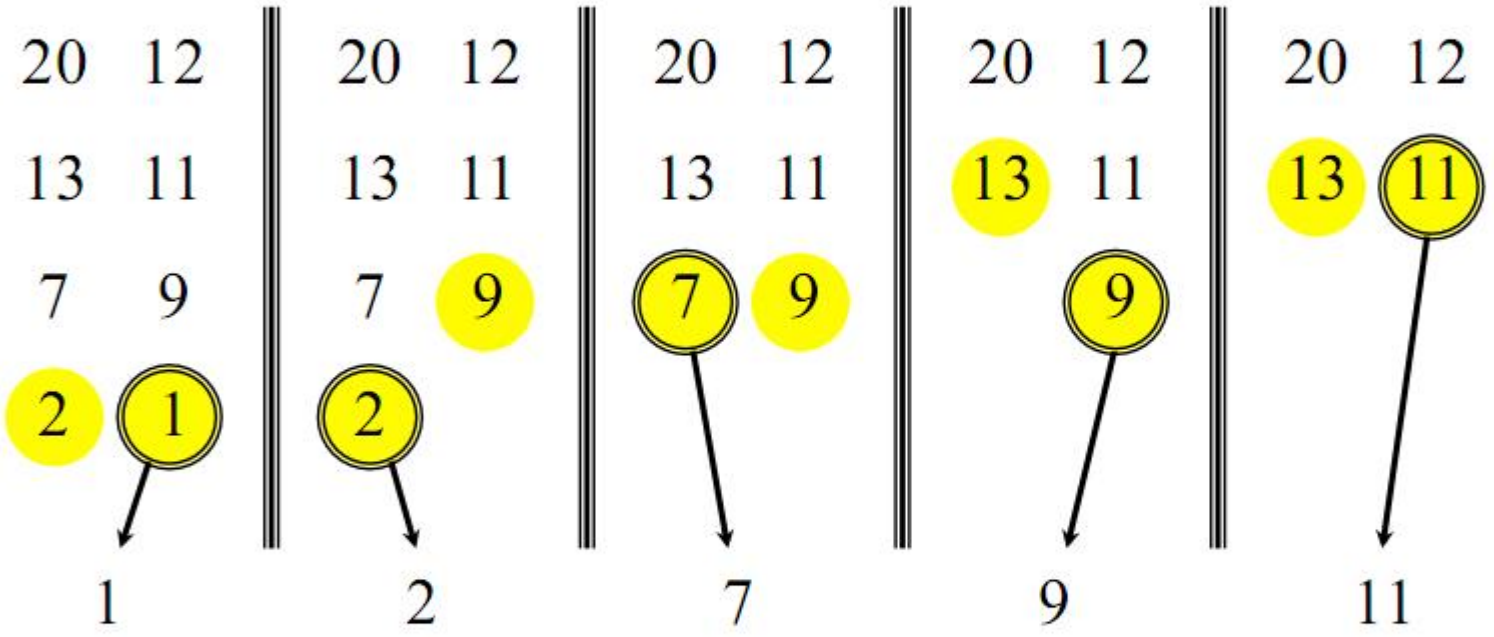
Merging two sorted arrays



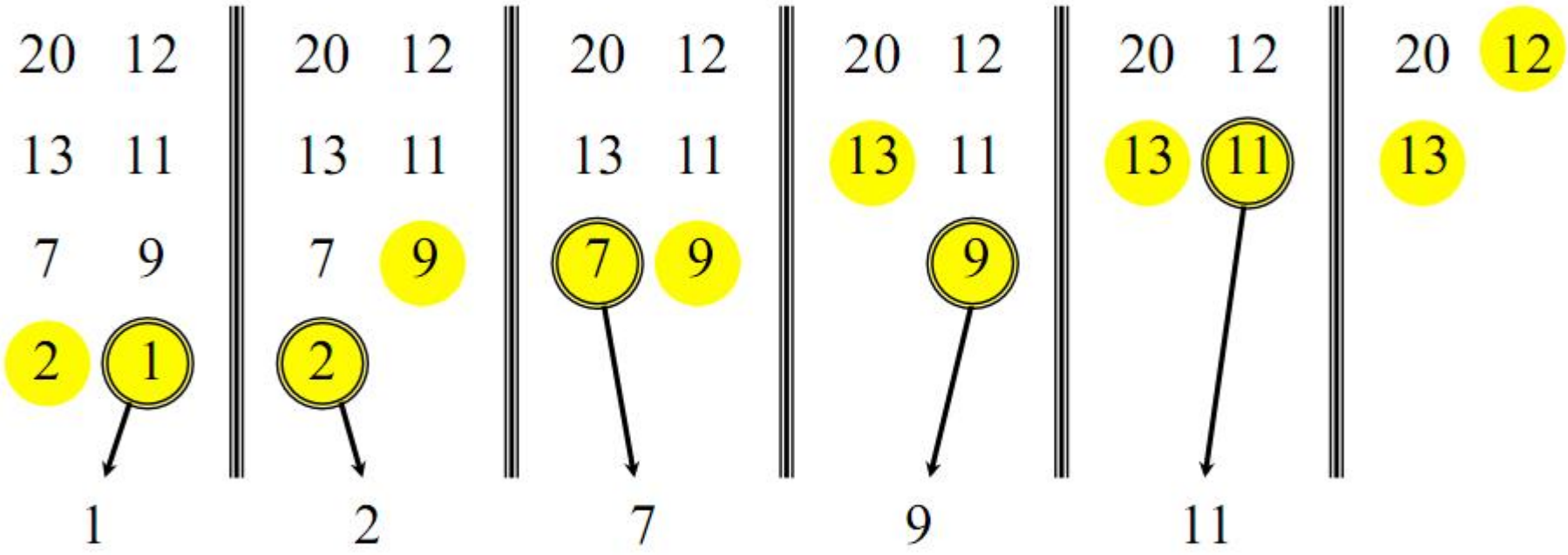
Merging two sorted arrays



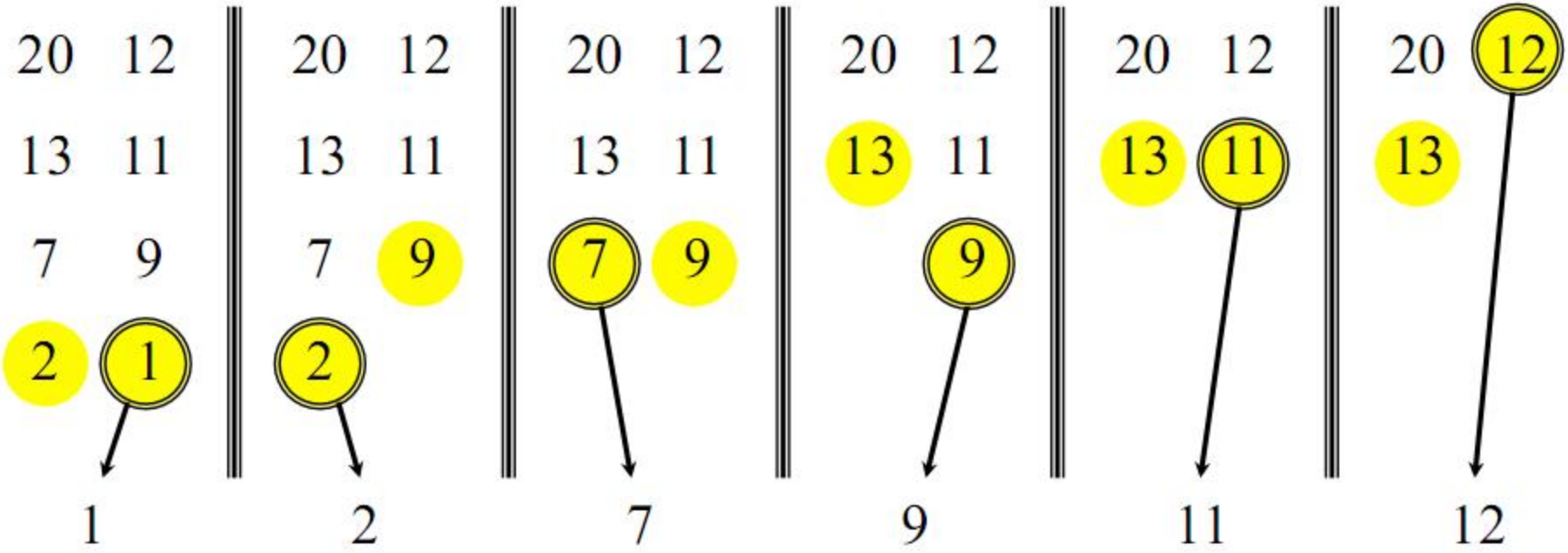
Merging two sorted arrays



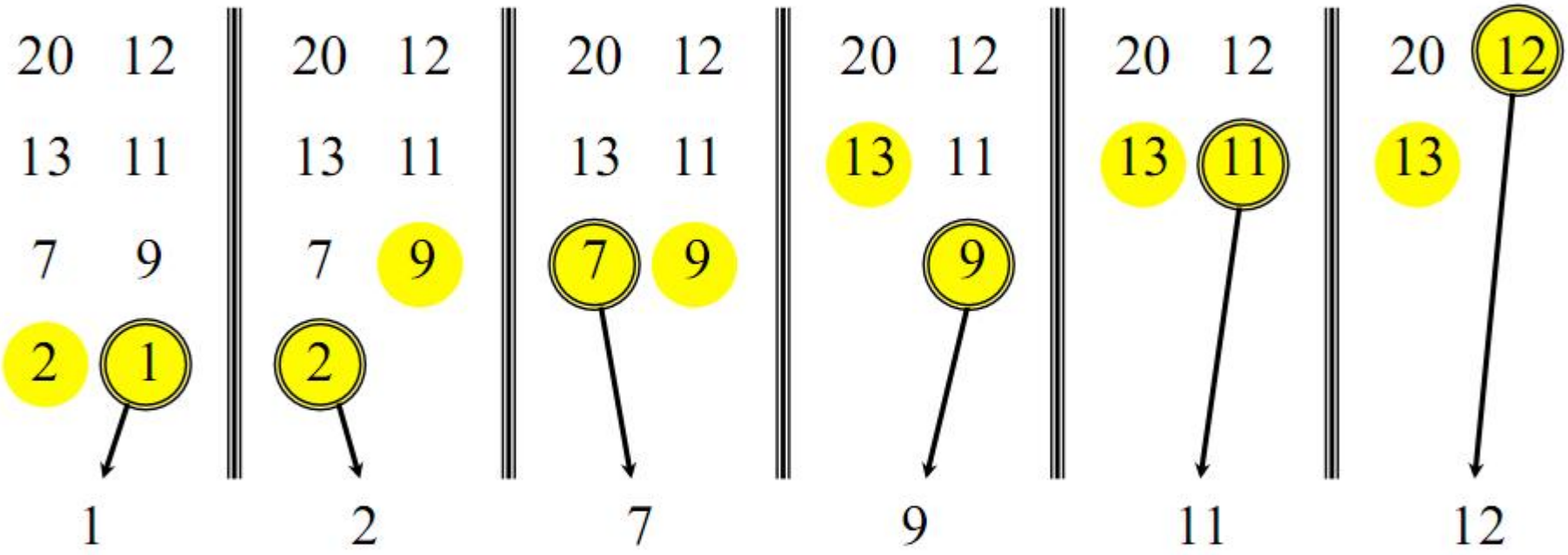
Merging two sorted arrays



Merging two sorted arrays



Merging two sorted arrays



Time = $\Theta(n)$ to merge a total of n elements (linear time).

Analyzing merge sort

MERGE-SORT $A[1 \ldots n]$	$T(n)$
1. If $n = 1$, done.	$\Theta(1)$
2. Recursively sort $A[\,1 \ldots \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \ldots n\,]$.	$2T(n/2)$
3. ***"Merge"*** the two sorted lists	$\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

$T(n) = ?$

Recurrence solving

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

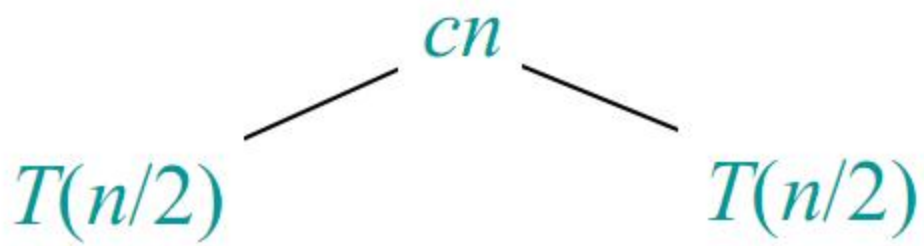
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$T(n)$

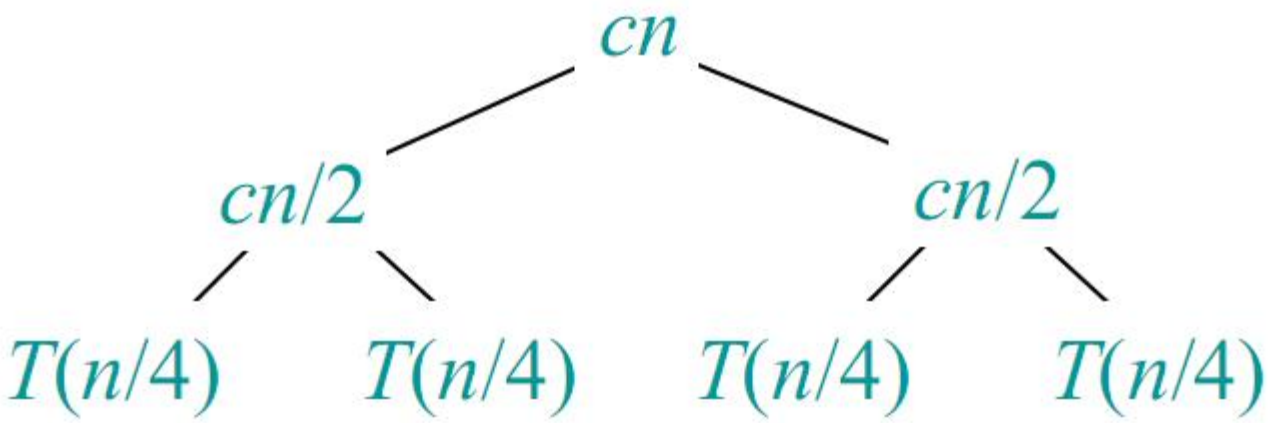
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



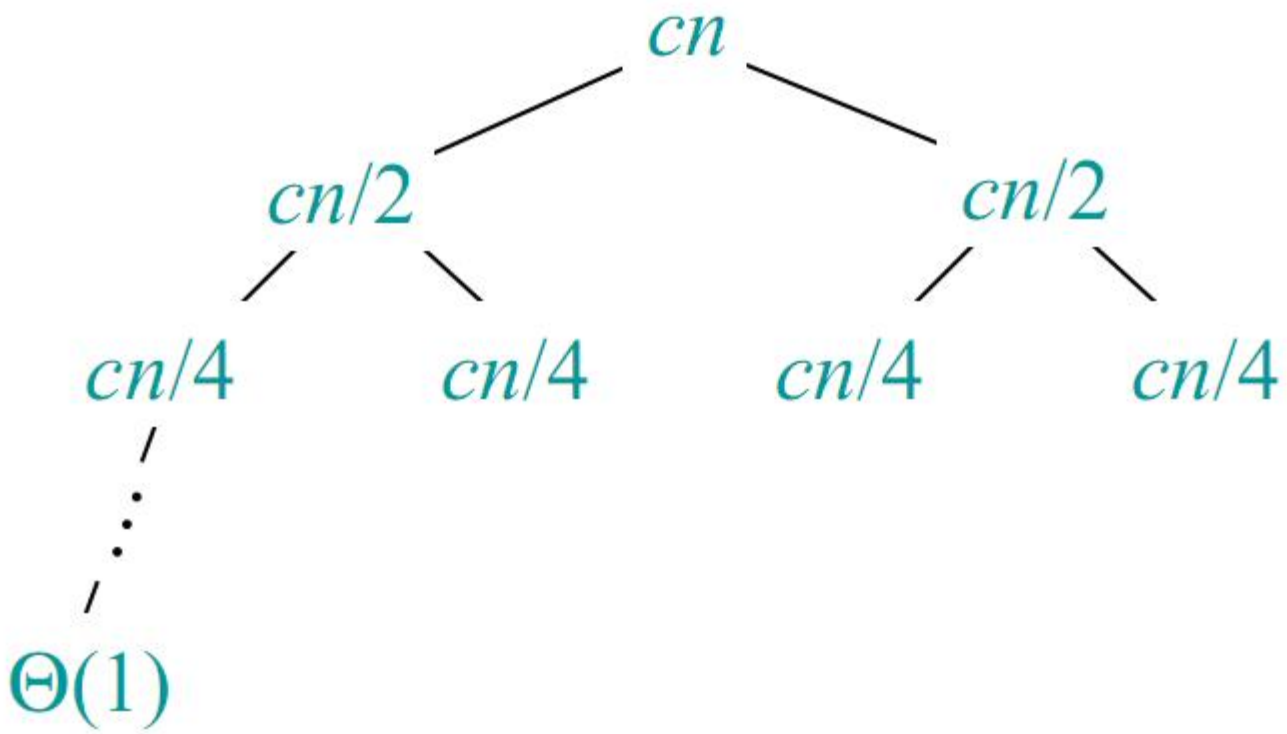
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



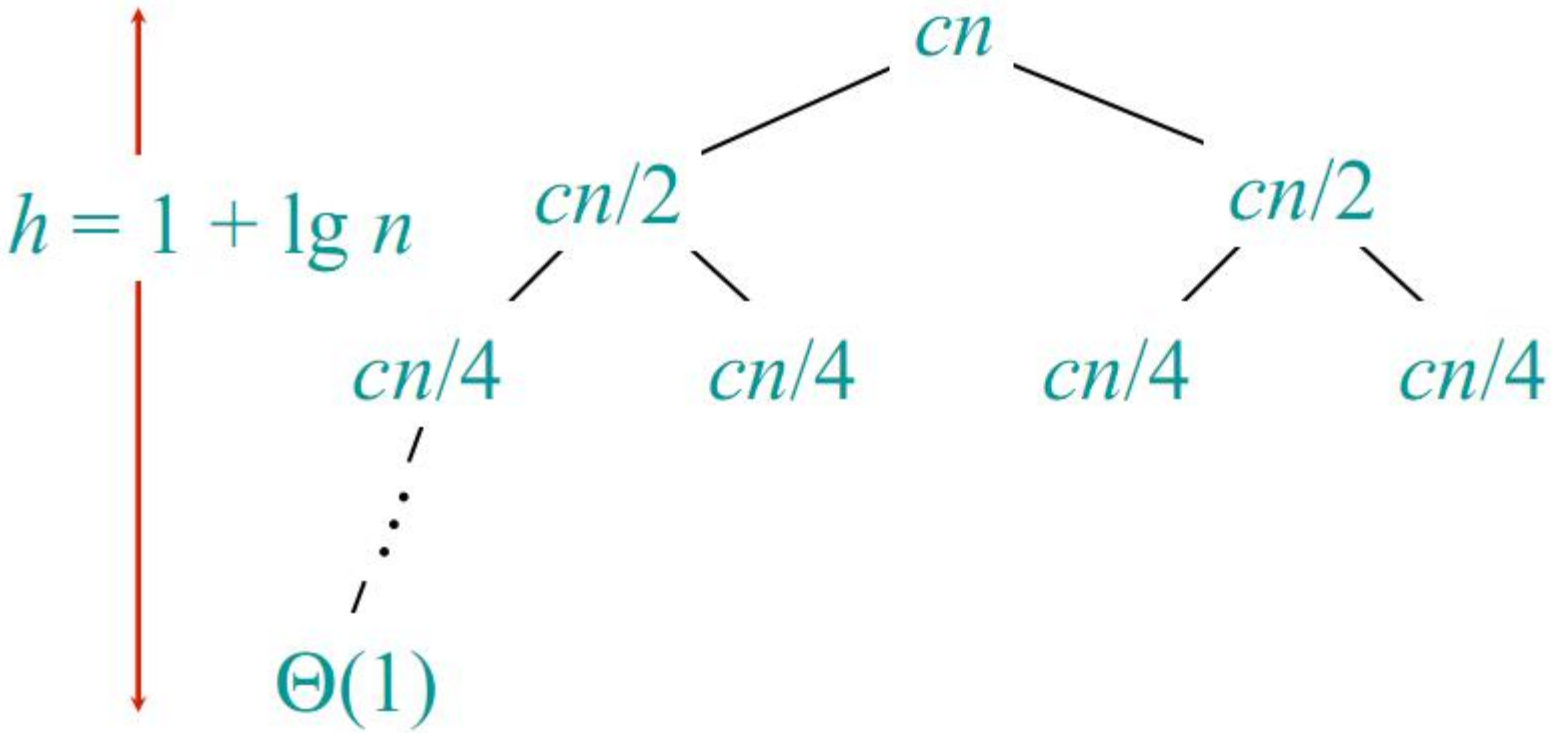
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



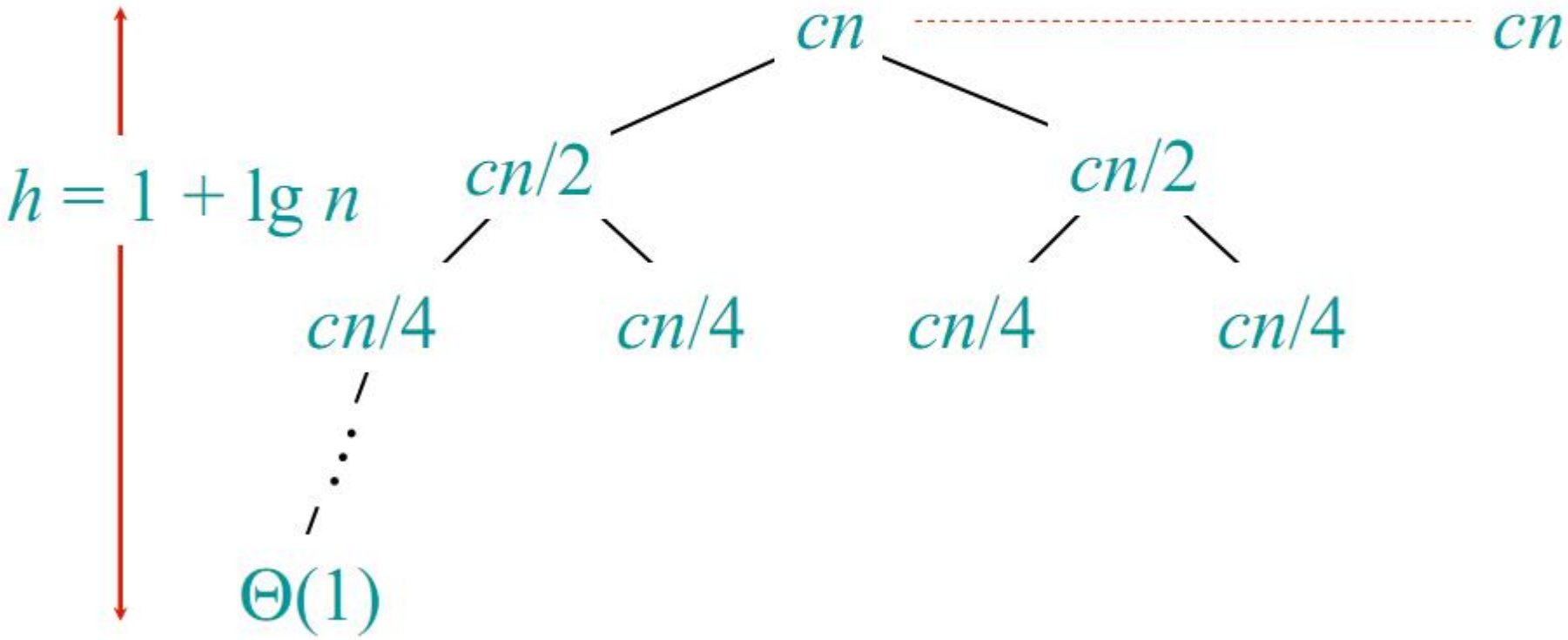
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



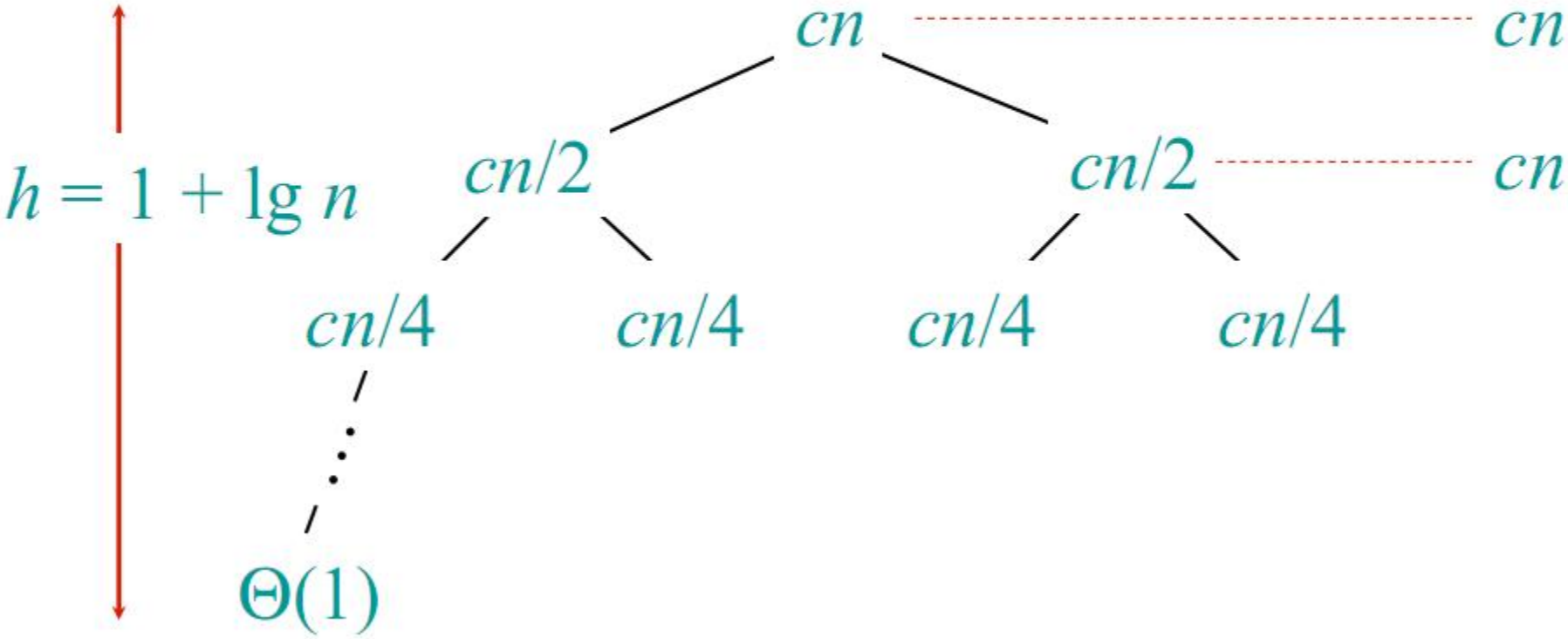
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



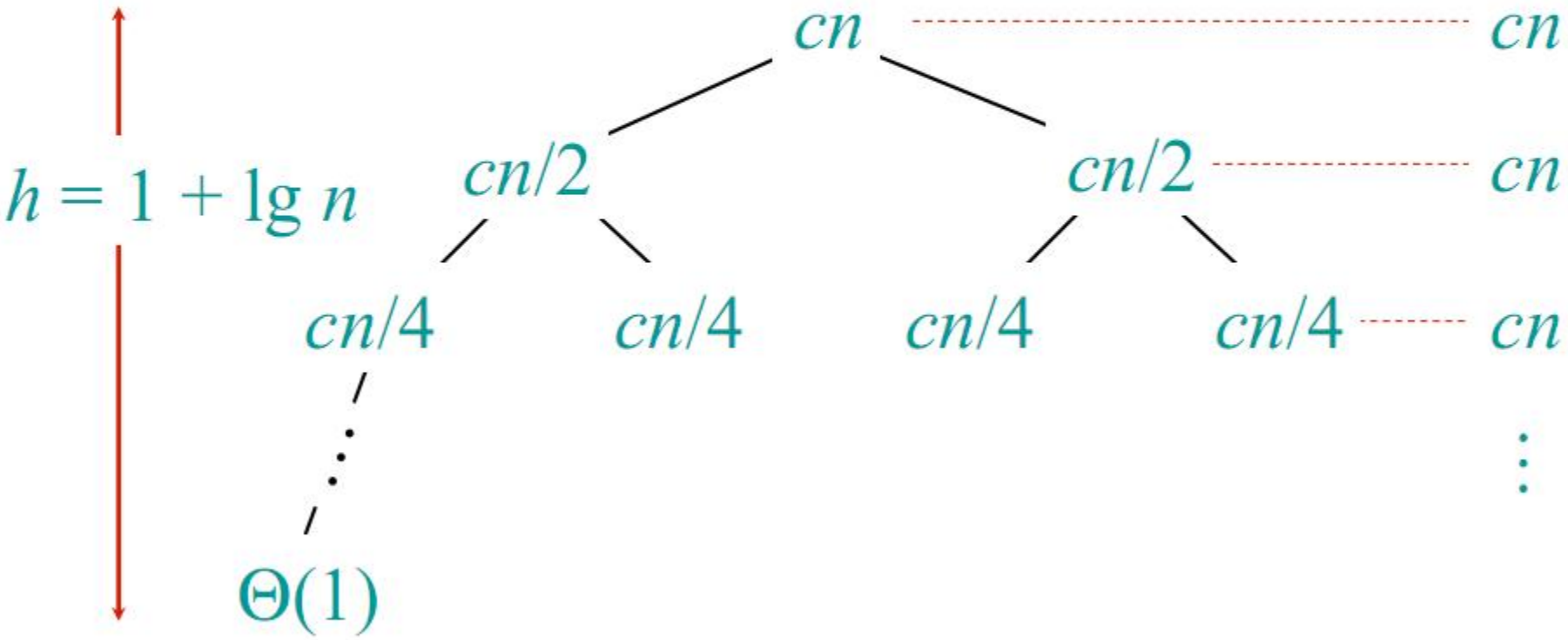
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



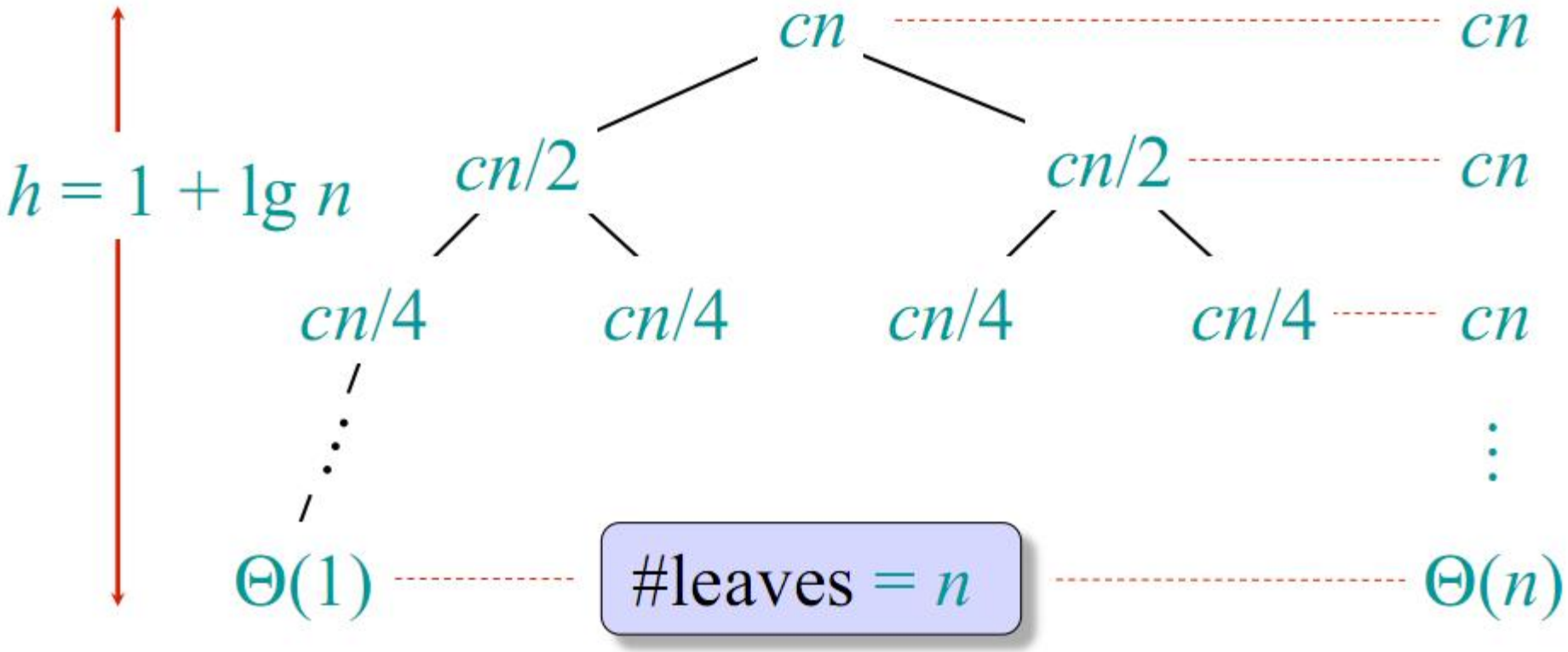
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



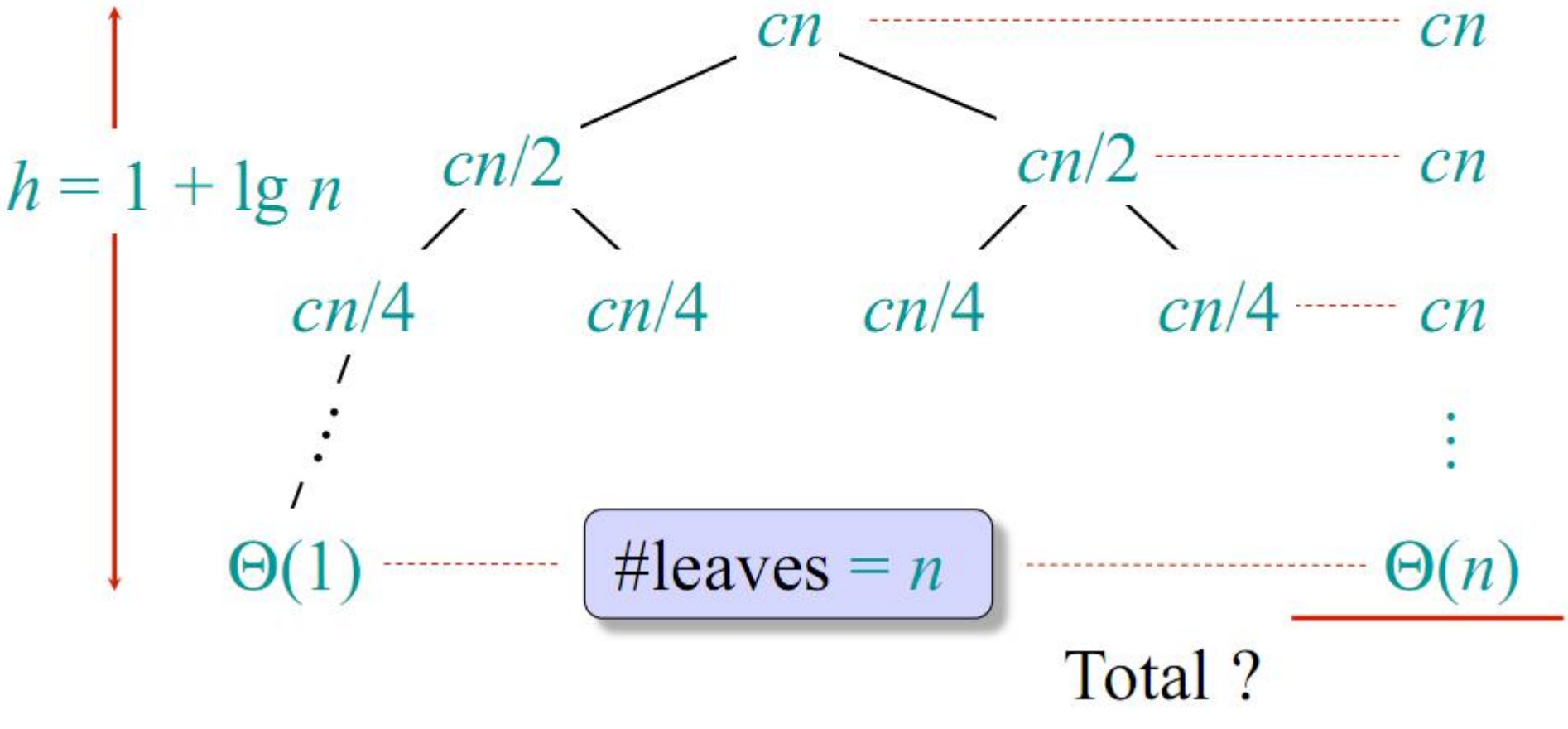
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree

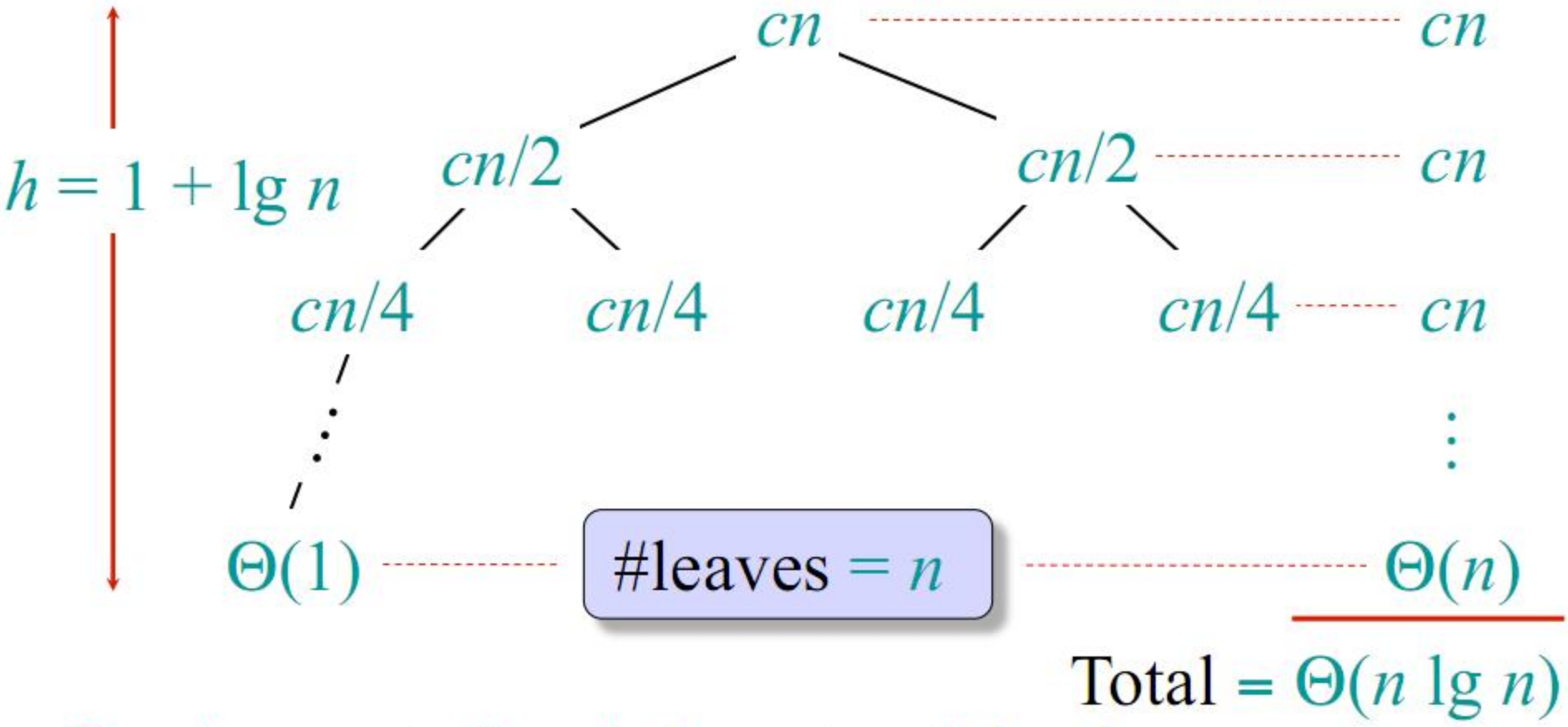
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Total ?

Recursion tree

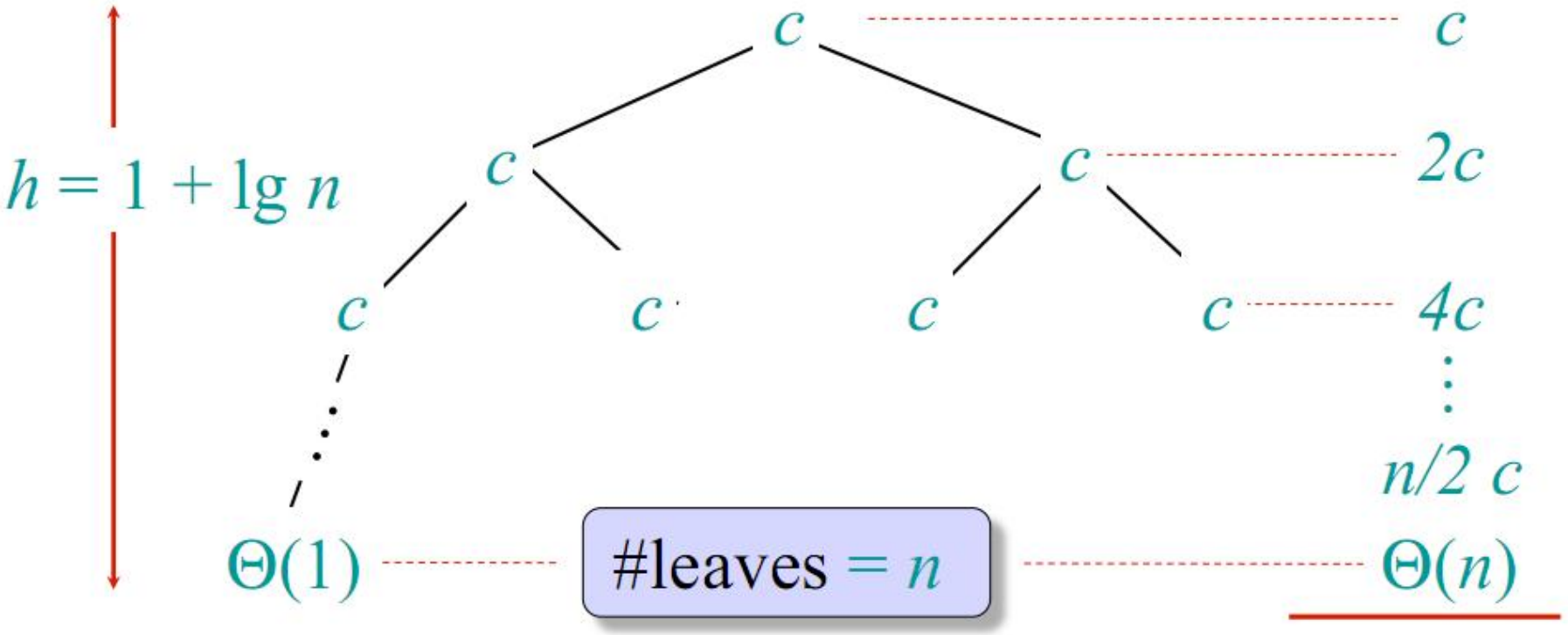
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Equal amount of work done at each level

Tree for different recurrence

Solve $T(n) = 2T(n/2) + c$, where $c > 0$ is constant.

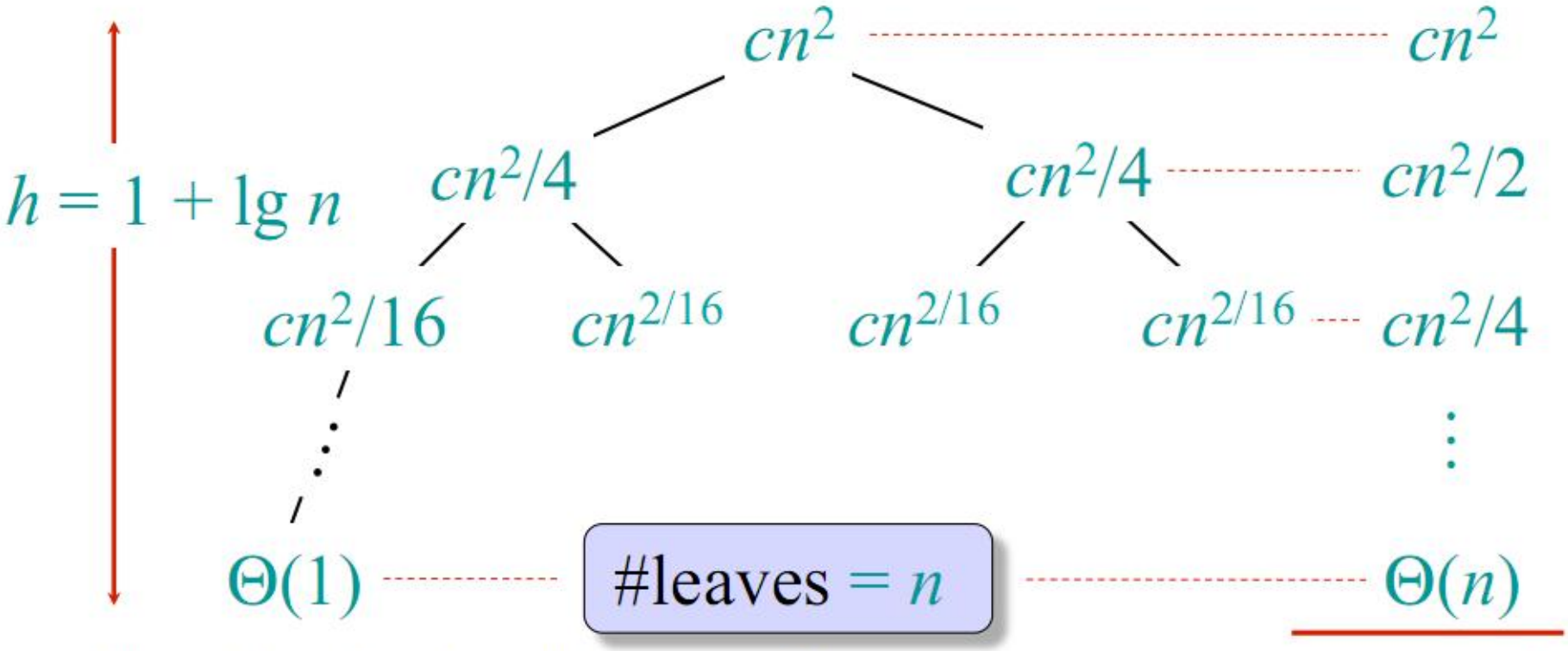


Note that $1 + \frac{1}{2} + \frac{1}{4} + \ldots < 2$

All the work done at the leaves

Tree for yet another recurrence

Solve $T(n) = 2T(n/2) + cn^2$, $c > 0$ is constant.



Note that $1 + \frac{1}{2} + \frac{1}{4} + \ldots < 2$

Total = $\Theta(n^2)$

All the work done at the root

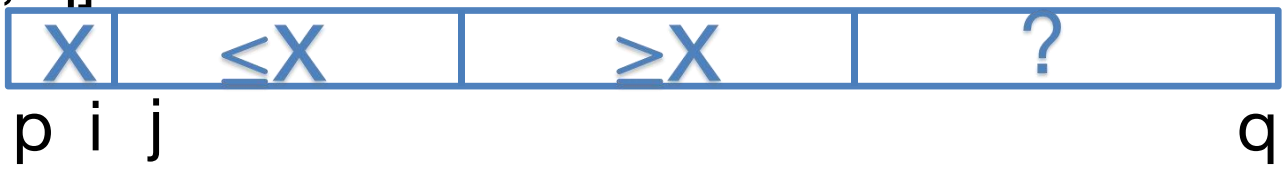
Quick Sort

- Divide : Partition the array into 2 subarrays around pivot x

- Conquer : Recursively sort the subarrays
- Combine : Merge the two subarrays (It is trival) .

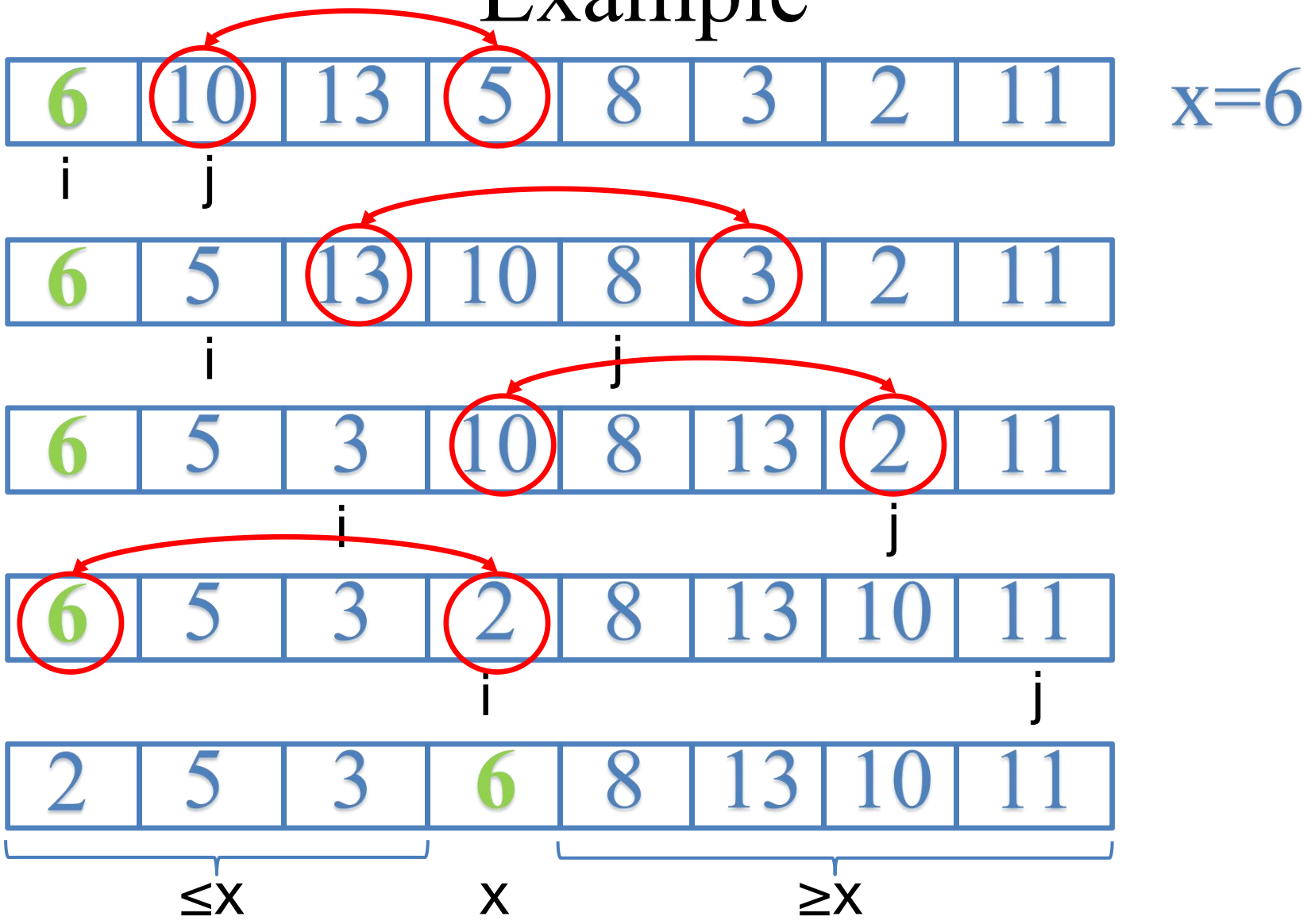
Partitioning

- Select the element p of A[p,…,q] as the pivot
- All elements in A[p+1, i] are less than x, all elements in A[i+1, j] are larger than x, and all elements in A[j+1, q] are unknown

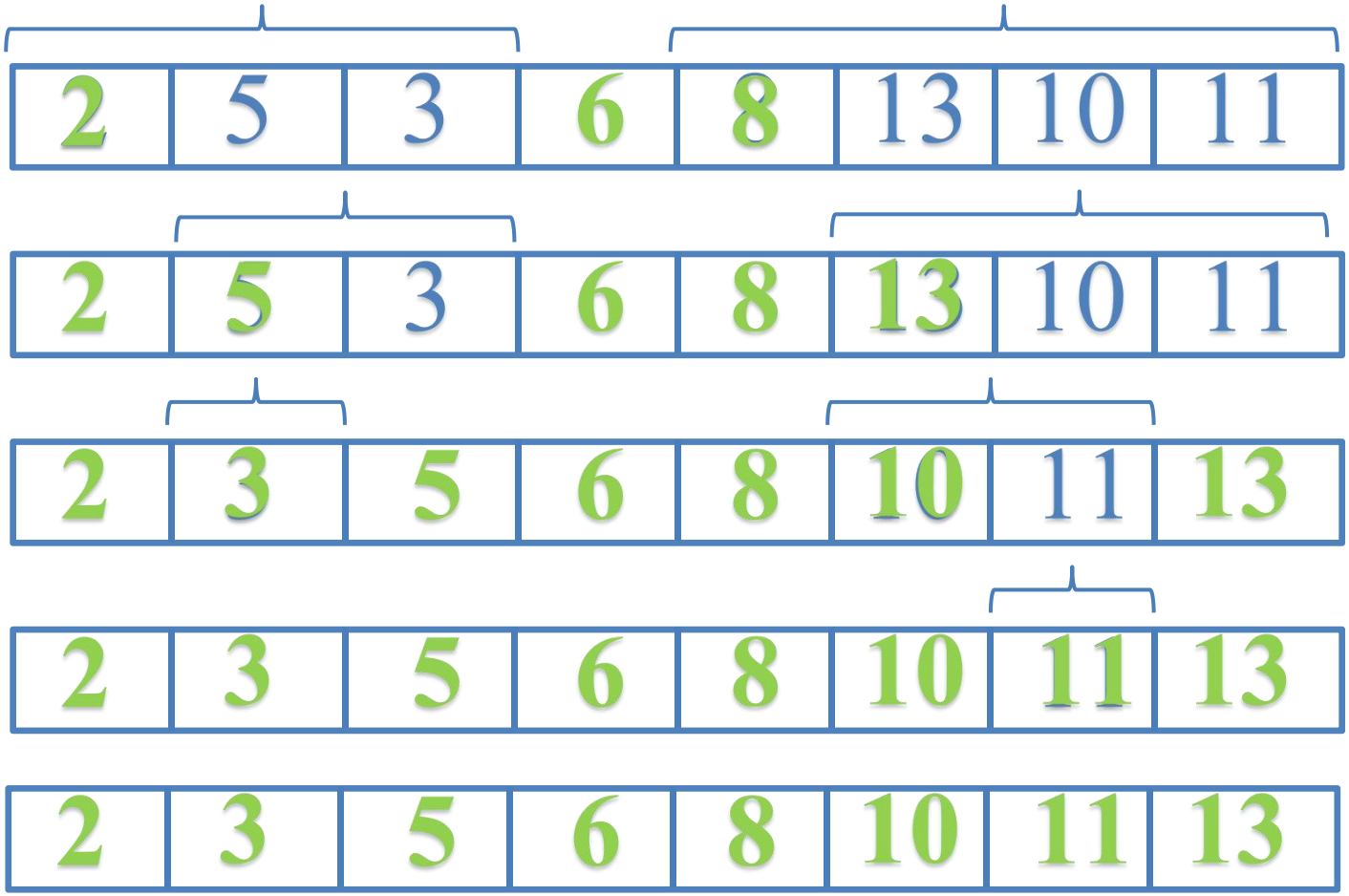


- The complexity is O(n)

Example


6 10 13 5 8 3 2 11
x=6
i
j
6 5 13 10 8 3 2 11
i
j
6 5 3 10 8 13 2 11
i
j
6 5 3 2 8 13 10 11
i
j
2 5 3 6 8 13 10 11
≤x
x
≥x

Sorting Subarrays



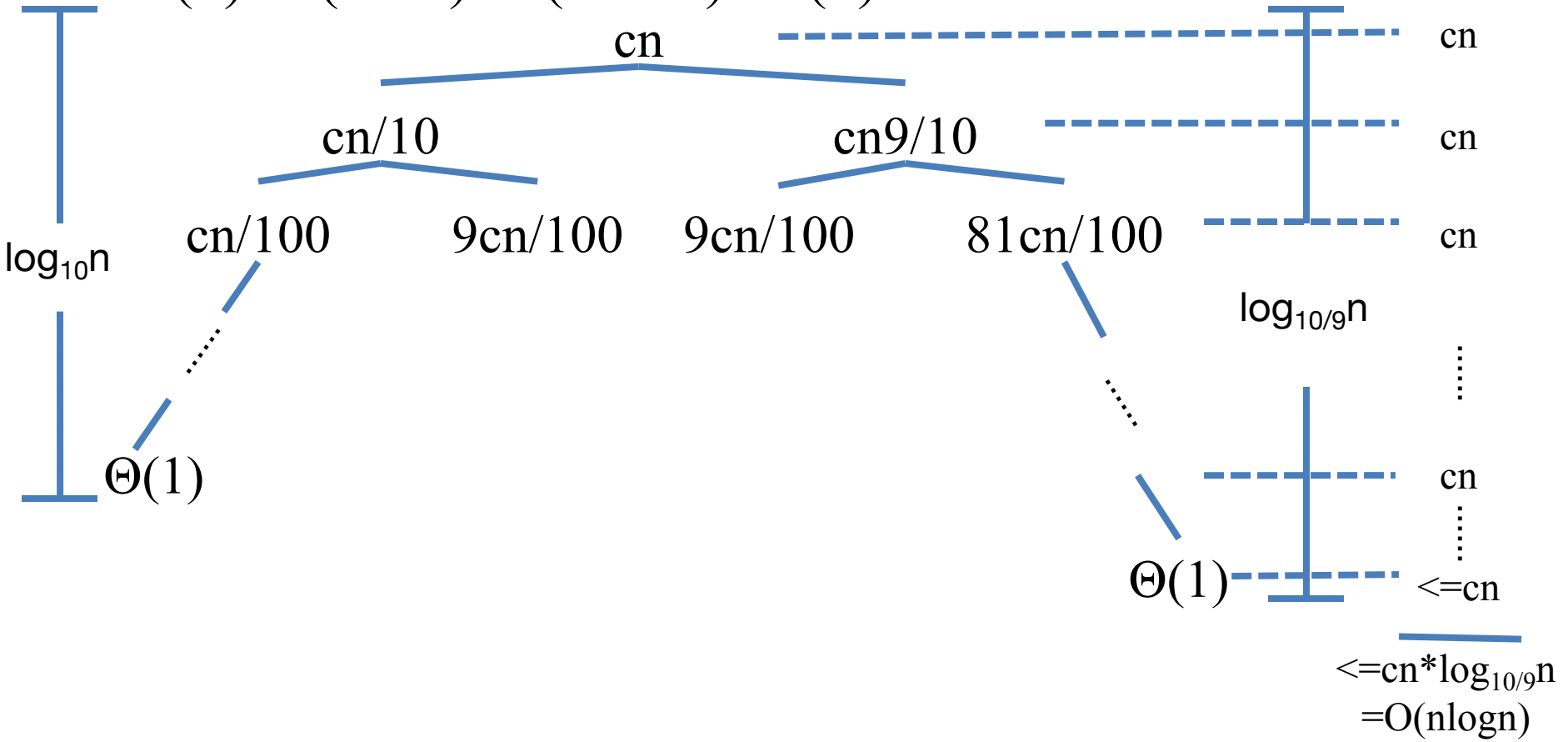
排序完成

Complexity

- Best case : $T(n)=2T(n/2)+ \Theta(n)= \Theta(n\log n)$
- Worst case : $T(n)=T(0)+T(n-1)+\Theta(n) = \Theta(1)+ T(n-1)+\Theta(n) = T(n-1)+\Theta(n) = \Theta(n^2)$
- Other cases……

First Unblanced Case

- Split is always 1:9
- $T(n)=T(n/10)+T(9n/10)+\Theta(n)$



Second Unblanced Case

- Alternate best and worst cases
- Best case: $B(n)=2W(n/2)+\Theta(n)$
- Worst case: $W(n)= B(n-1)+\Theta(n)$
- $B(n)=2(B(\frac{n}{2} - 1) + \Theta(n/2))+\Theta(n)$

  $=2B(\frac{n}{2} - 1)+\Theta(n)$

  $=O(n\log n)$

Randomized Quick Sort

- Selecting a pivot element randomly
- Running time is independent on input running ordering

Analysis

- We first define an indicator random variable x_k
- x_k=1 if generating k:n-k-1 split; otherwise, x_k=0
- Then, the expectation of x_k is

  $E[x_k]=0*pr\{x_k=0\}+1*pr\{x_k=1\}=1/n.$

Analysis

- T(n) = T(0)+T(n-1)+Θ(n) if 0:n-1 split
- T(n) = T(1)+T(n-2)+Θ(n) if 1:n-2 split
- ……
- T(n) = T(n-1)+T(0)+Θ(n) if n-1:0 split
- Then,

$$T(n)=\sum_{k=0}^{n-1} x_k \times (T(k)+T(n-k-1)+\Theta(n))$$

Analysis

- $E(T(n))=E(\sum_{k=0}^{n-1} x_k \times (T(k)+T(n-k-1)+\Theta(n)))$

$$=\sum_{k=0}^{n-1} E(x_k \times (T(k)+T(n-k-1)+\Theta(n)))$$

$$=\sum_{k=0}^{n-1} E(x_k) \times E((T(k)+T(n-k-1)+\Theta(n)))$$

$$=\frac{1}{n}\sum_{k=0}^{n-1} E((T(k)+T(n-k-1)+\Theta(n)))$$

$$=\frac{1}{n}\sum_{k=0}^{n-1} E(T(k))+\frac{1}{n}\sum_{k=0}^{n-1} E(T(n-k-1))+$$

$$\frac{1}{n}\sum_{k=0}^{n-1}\Theta(n)$$

Analysis

- $E(T(n)) = \frac{1}{n}\sum_{k=0}^{n-1} E(T(k)) + \frac{1}{n}\sum_{k=0}^{n-1} E(T(n-k-1)) + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$

$$= \frac{2}{n}\sum_{k=0}^{n-1} E(T(k)) + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

$$= \frac{2}{n}\sum_{k=0}^{n-1} E(T(k)) + \Theta(n)$$

Aborb k=0 and k =1, and we can have a simpler formula:

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

Analysis

- For E(T(n))=$\frac{2}{n}\sum_{k=2}^{n-1}E(T(k))+\Theta(n)$, we can prove E(T(n))=O(nlogn) by using induction and the lemma

$$\sum_{k=2}^{n-1} k\log k <= \frac{1}{2}n^2 \times \log n$$

Comparison sort

All the sorting algorithms we have seen so far are ***comparison sorts***: only use comparisons to determine the relative order of elements.

所有已学排序算法都是基于两个元素比较的排序

The best running time that we've seen for comparison sorting is $O(n \log n)$.

Is $O(n \log n)$ the best we can do?

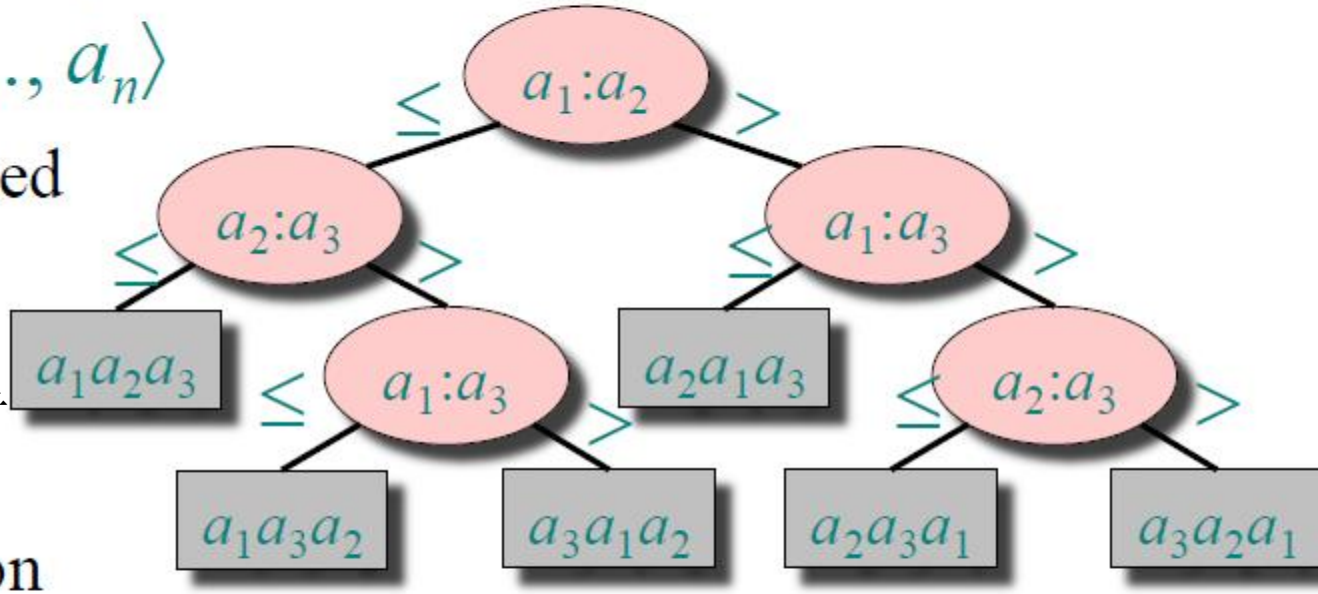
这个运行时间$\Theta(n \lg n)$是我们能做到的最好的吗？

Decision-tree 决策树

A recipe for sorting n things $\langle a_1, a_2, \ldots, a_n \rangle$

- Nodes are suggested comparisons: $a_i:a_j$ means compare a_i to a_j.

节点是两个元素的比较/比对



- Branching direction depends on outcome of comparisons.

分支走向由比对结果决定

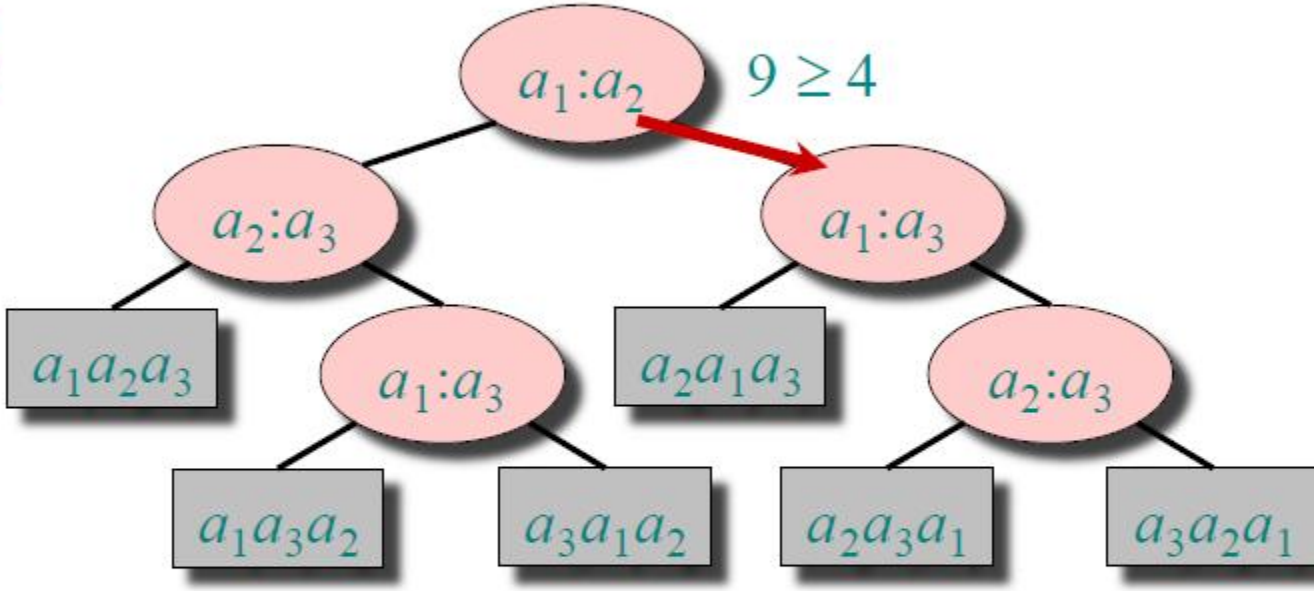
- Leaves are labeled with permutations corresponding to the outcome of the sorting.

叶节点是标记好的排序，对应的是排序的结果

Decision-tree example

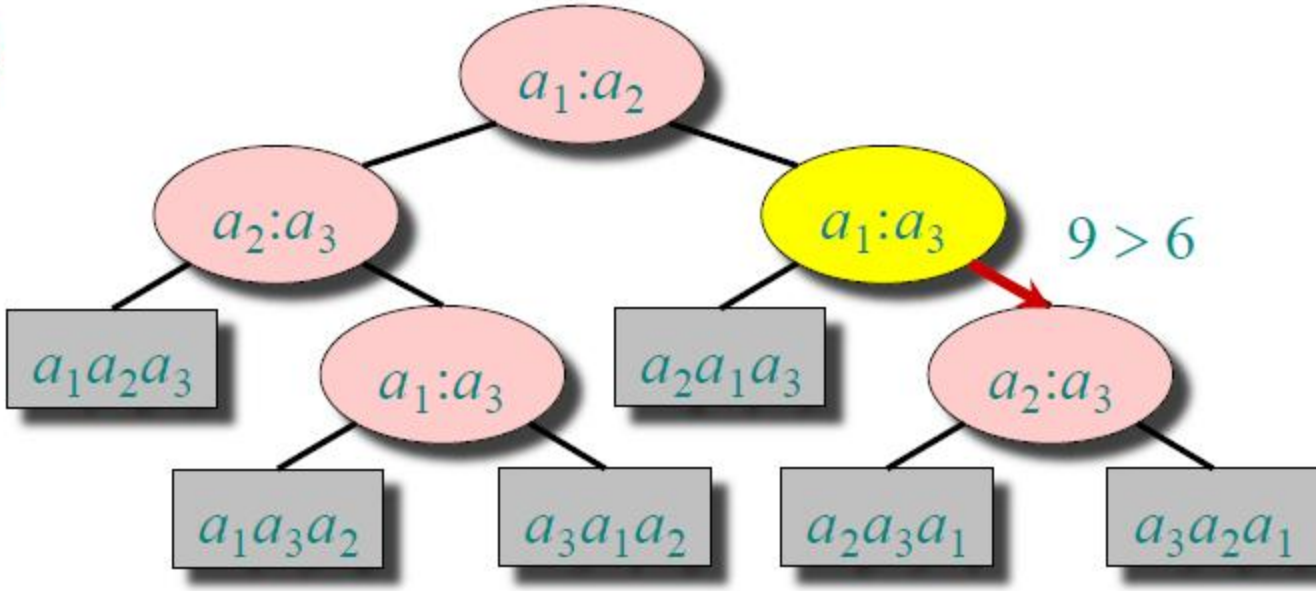
Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle 9, 4, 6 \rangle$:

决策树例子



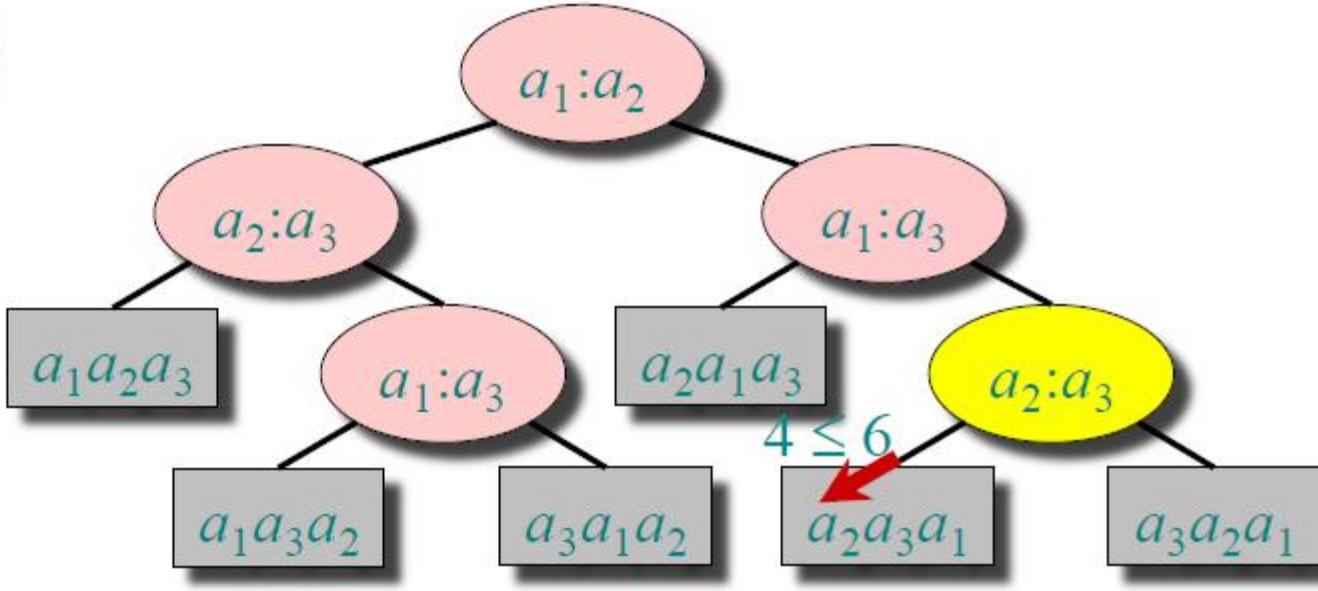
Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:



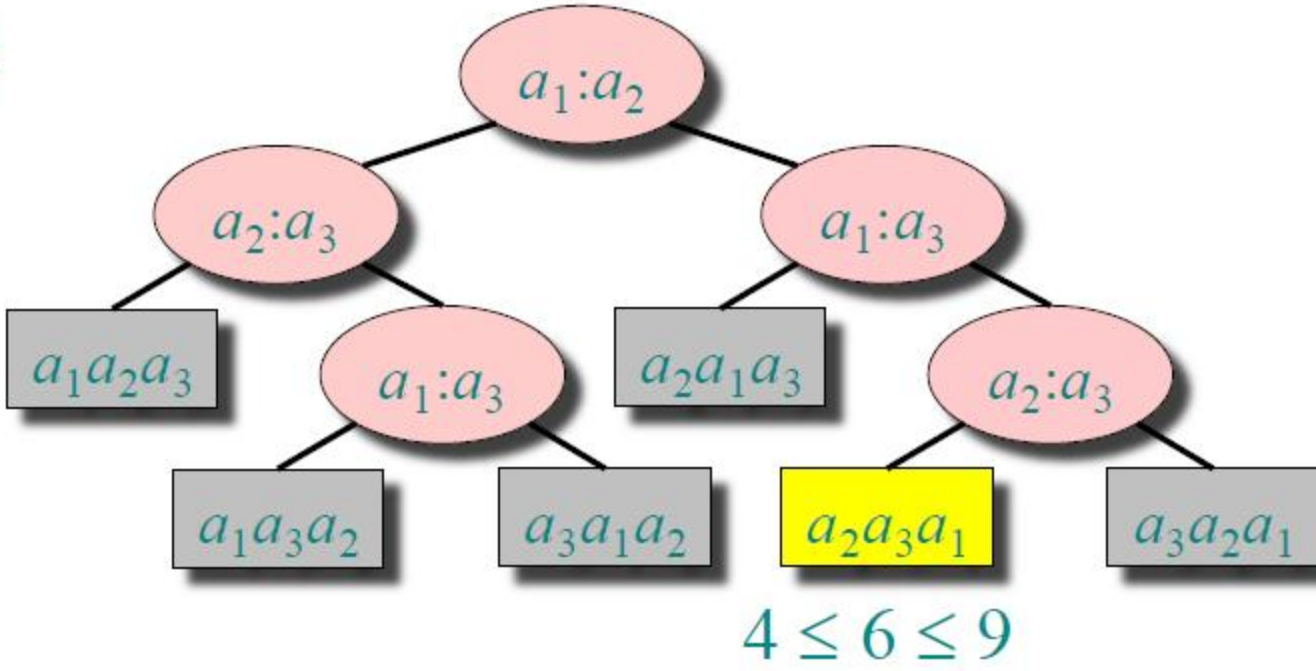
Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:



Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:

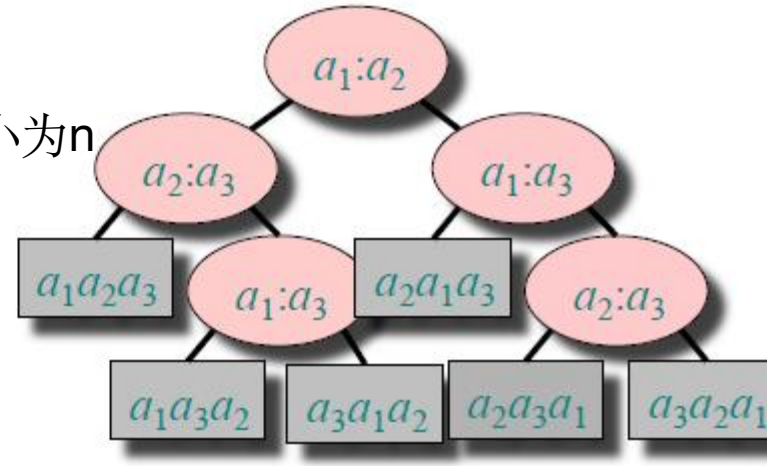


Decision-tree model

一个决策树可以模拟任何比对排序的执行

A decision tree can model the execution of any comparison sort:

- One tree for each input size n. 输入的大小为n
- A path from the root to the leaves of the tree represents a trace of comparisons that the algorithm may perform.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.



一个从跟节点到叶节点的路径表示这个算法可能执行的一系列比对的跟踪。

算法的运行时间=路径的长度

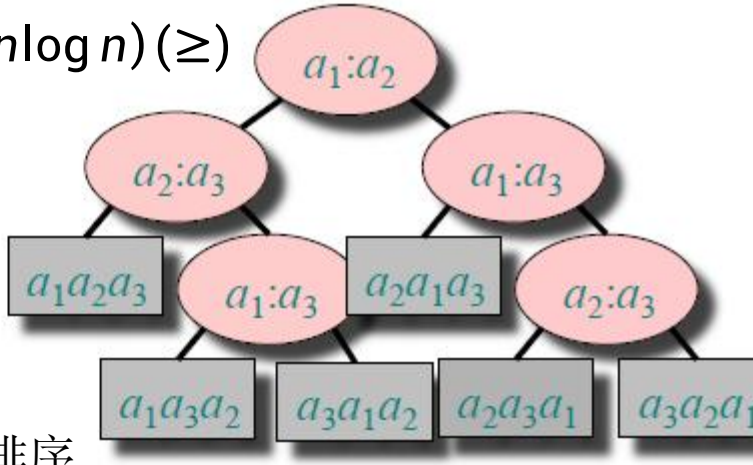
算法的最差情况下的运行时间=树的高度

Lower bound for decision-tree sorting

Theorem. Any decision tree for n elements must have height $\Omega(n \log n)$.

Proof. 一个n个元素的决策树的高度一定是Ω(*n*log *n*)(≥)

The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations.
A height-h binary tree has $\leq 2^h$ leaves.
For it to be able to sort it must be that:



树最多包含（≥）*n*!的叶节点，因为只有n! 可能排序。

$2^h \geq n!$ 一个高度为h的二叉树树最多包含≤ 2^h的叶节点

$h \geq \log(n!)$ 这棵树能够用来排序，那么它必须满足：$2^h \geq n!$

$\geq \log((n/e)^n)$

$= n \log n - n \log e$

$= \Omega(n \log n)$.

Sorting in linear time

计数排序：元素之间没有比对

Counting sort: No comparisons between elements.

输入数组的值在1-k之间

- ***Input***: $A[1\ldots n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- ***Output***: $B[1\ldots n]$, a sorted permutation of A
- ***Auxiliary storage***: $C[1\ldots k]$.

输出数组B是对数组A的排序

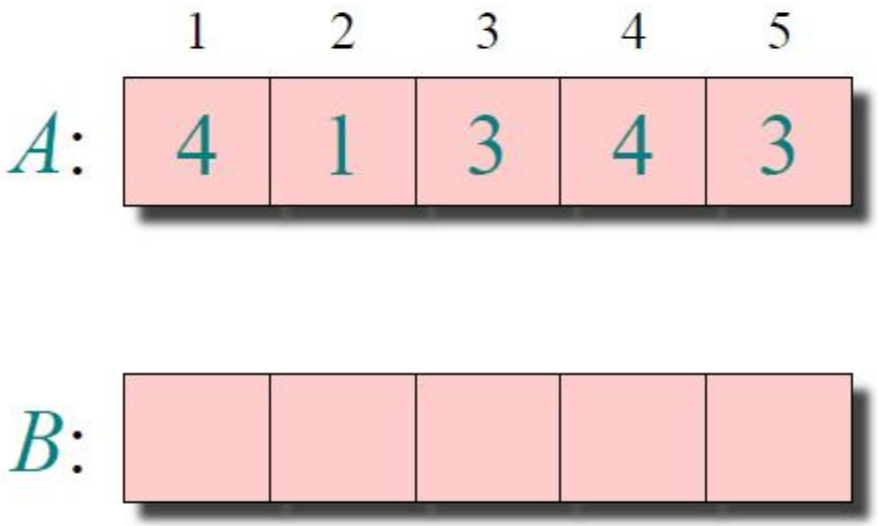
辅助存储空间，长度为k数组C

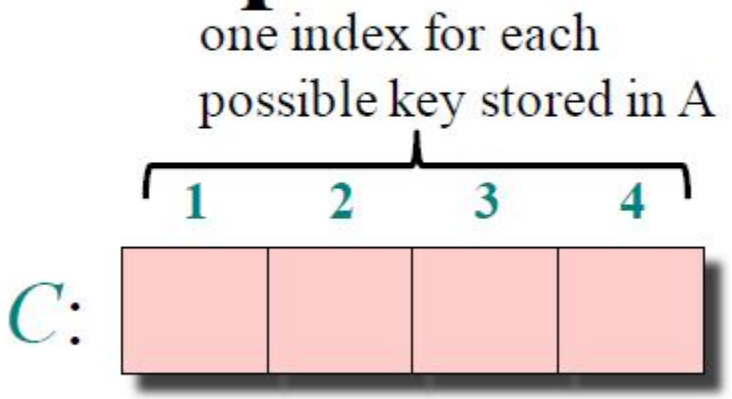
Running time: $O(n+k)$

运行时间为O(n+k)

Counting-sort example

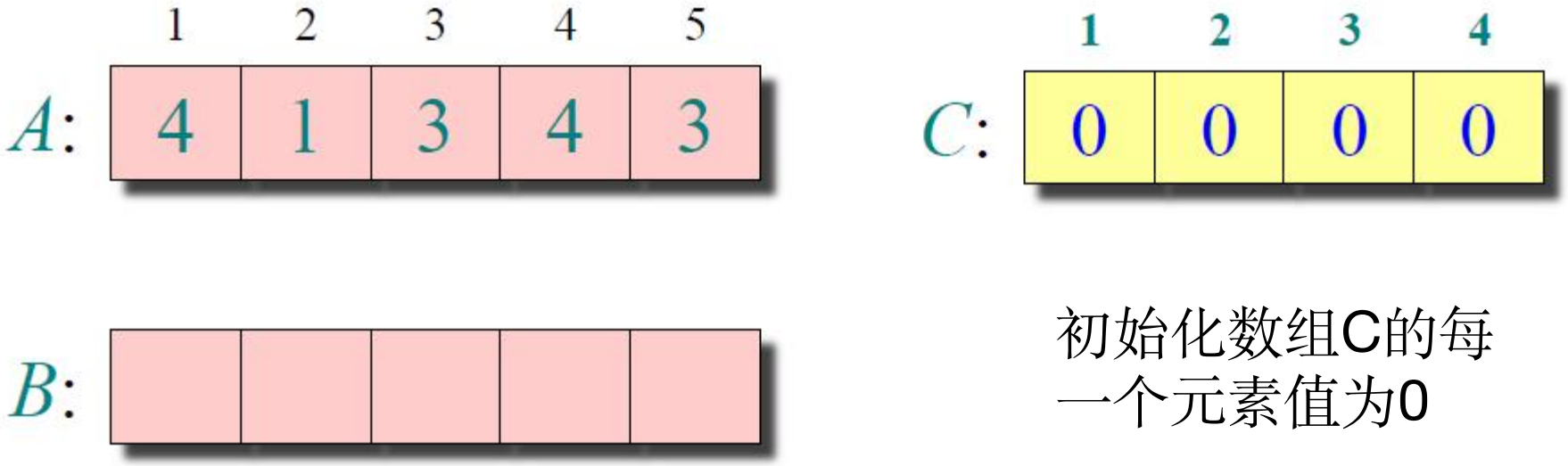
n=5, k=4





数组C的索引对应的数组A中每一个可能的键值

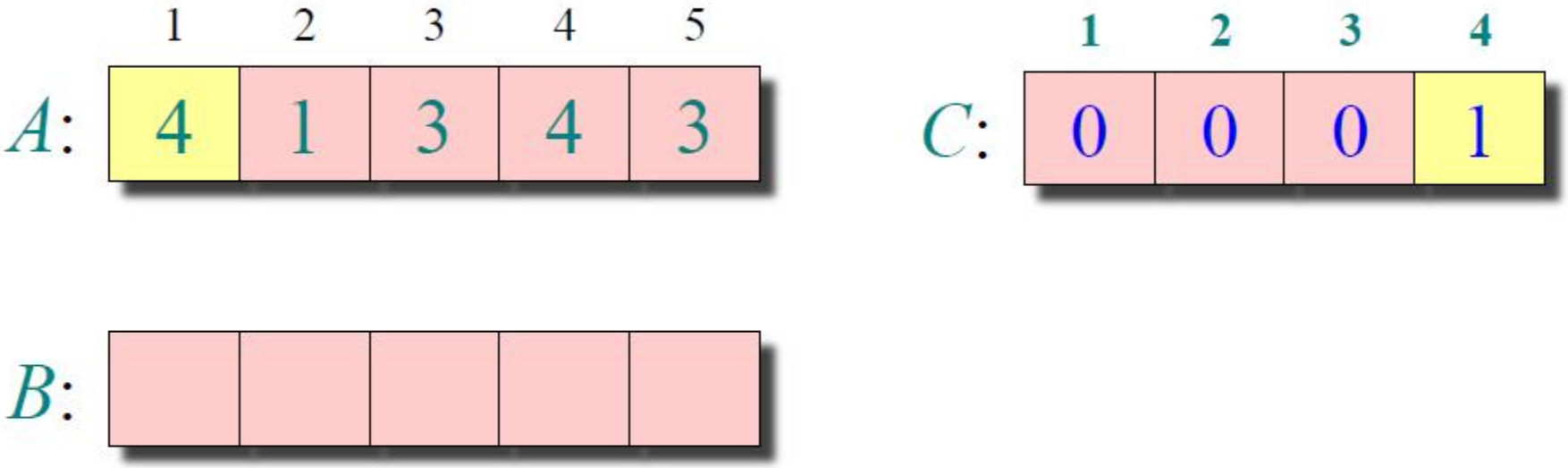
Loop 1: initialization



初始化数组C的每一个元素值为0

for $i \leftarrow 1$ **to** k
do $C[i] \leftarrow 0$

Loop 2: count frequencies

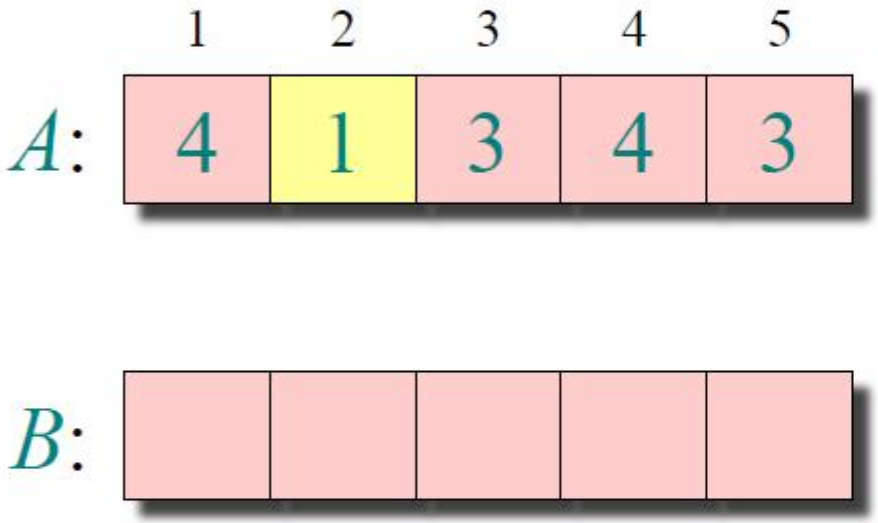


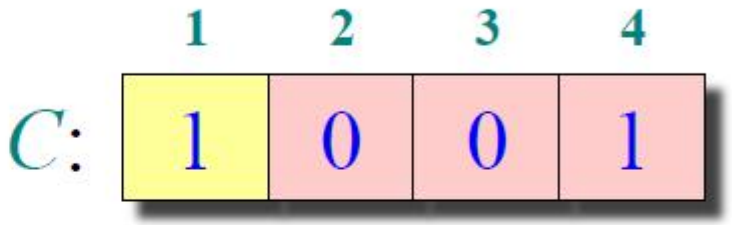
for $j \leftarrow 1$ **to** n

C[i]=数组A中键值为i的元素个数,i=4

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Loop 2: count frequencies



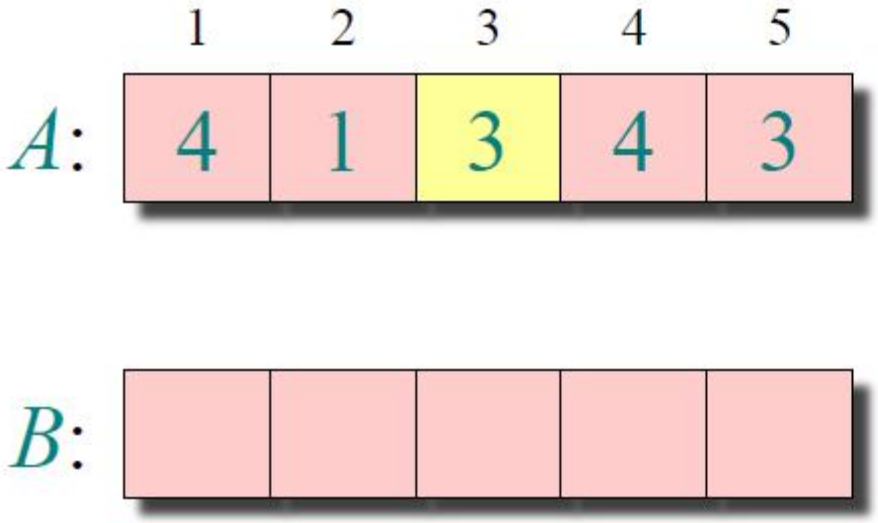


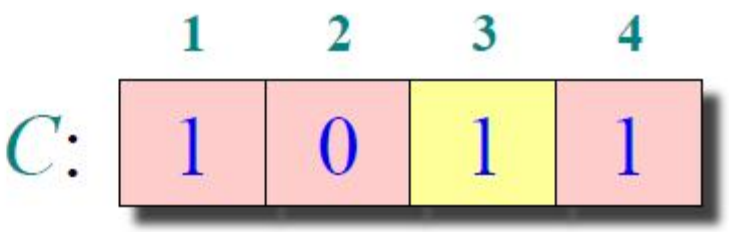
for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

C[i]=数组A中键值为i的元素个数,i=1

Loop 2: count frequencies



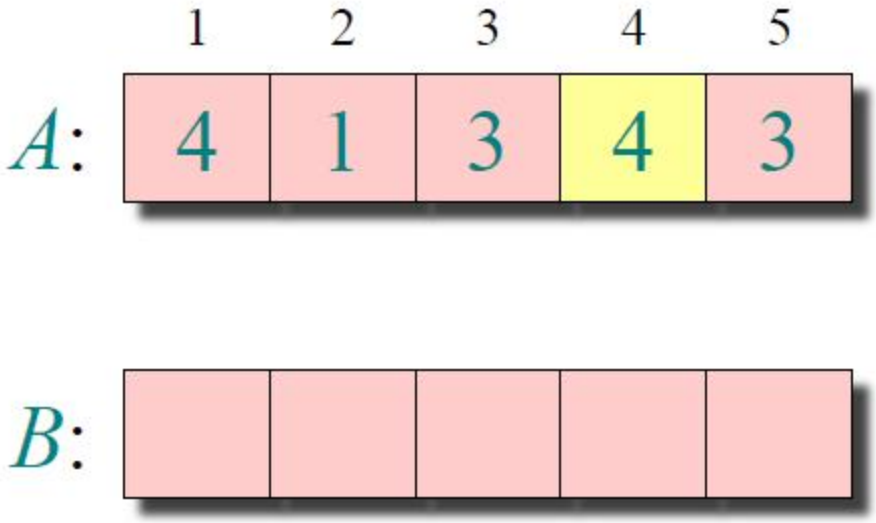


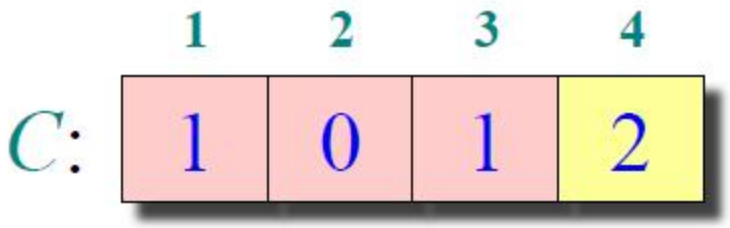
for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

C[i]=数组A中键值为i的元素个数,i=3

Loop 2: count frequencies



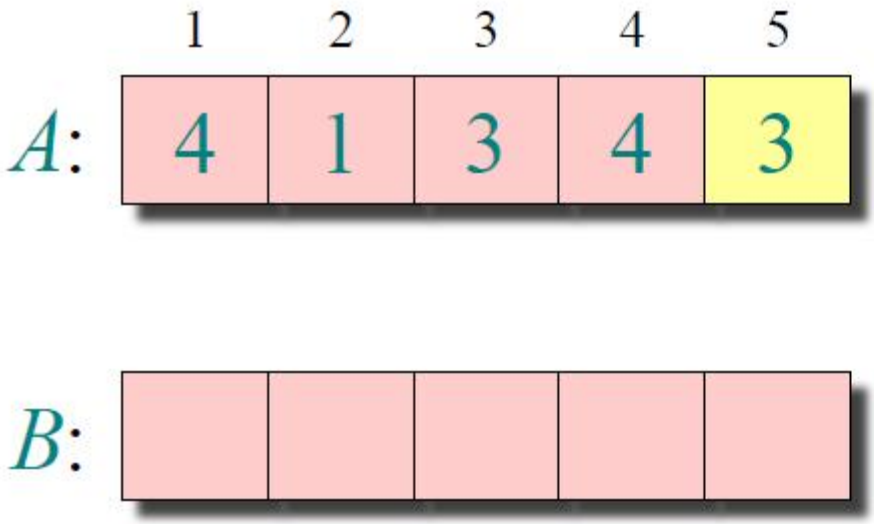


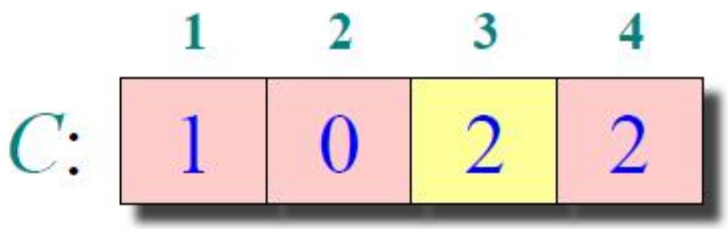
for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

C[i]=数组A中键值为i的元素个数,i=4

Loop 2: count frequencies



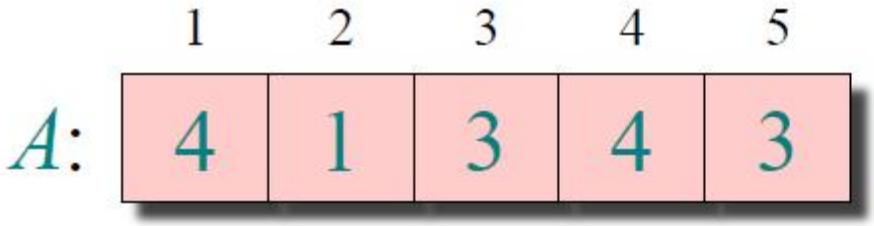


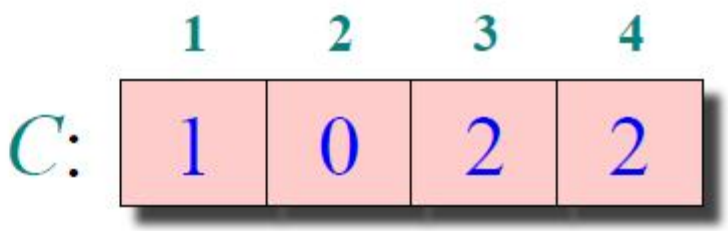
for $j \leftarrow 1$ **to** n

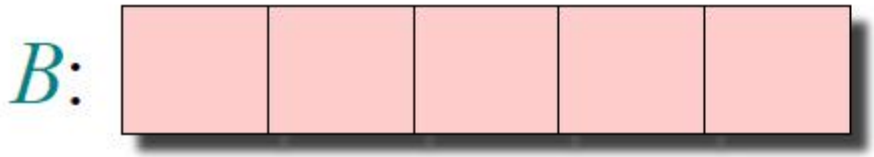
do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

C[i]=数组A中键值为i的元素个数，i=3

Loop 2: count frequencies





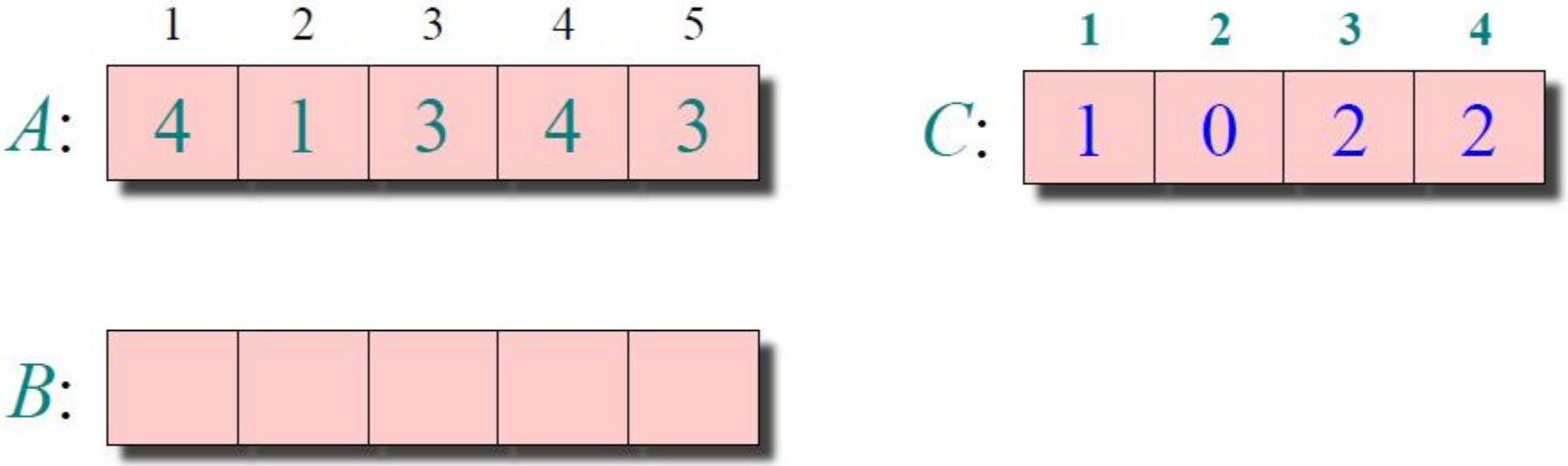


C[i]=数组A中键值为i的元素个数

for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

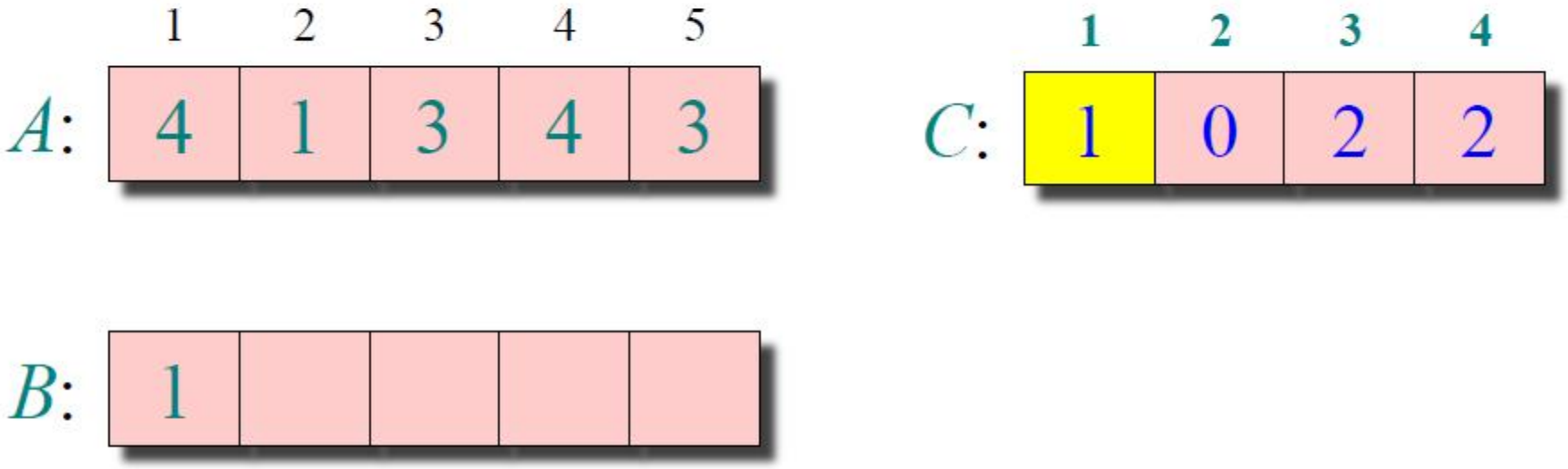
[A parenthesis: a quick finish



对频率数组C从头到尾走一遍，我们可以排序好的键值放到输出数组B中来。

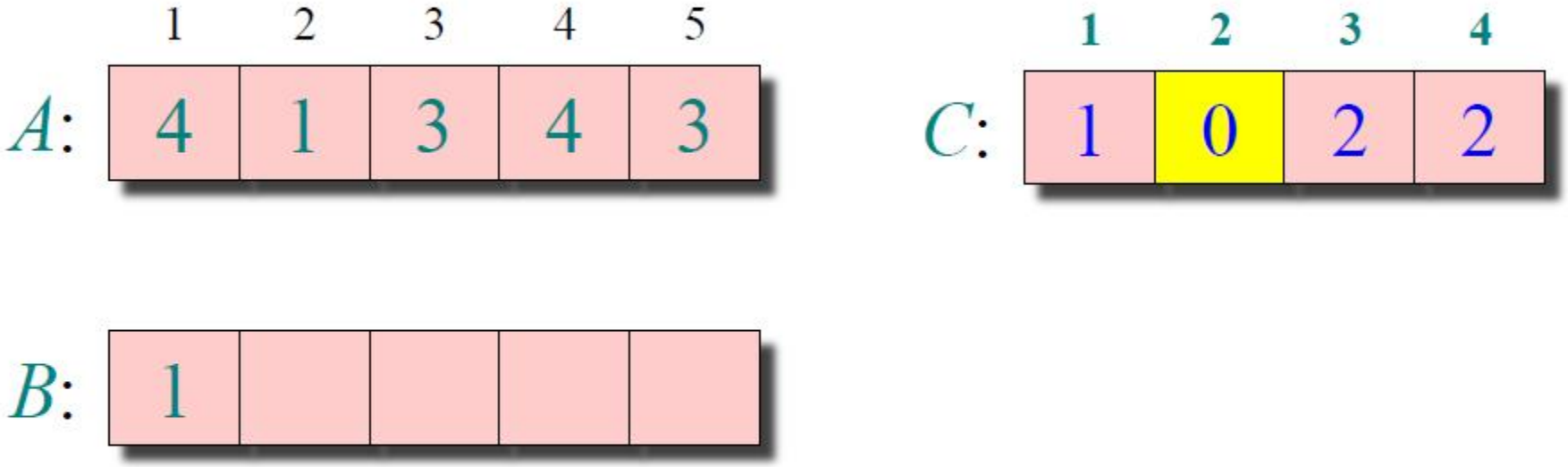
Walk through frequency array and place the appropriate number of each key in output array…

A parenthesis: a quick finish



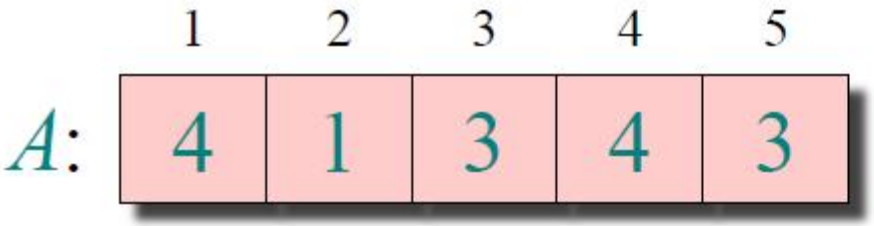
对于C[i]，数组A中键值为i的元素数目为C[i]!
对于C[1]，数组A中键值为1的元素数目为1!

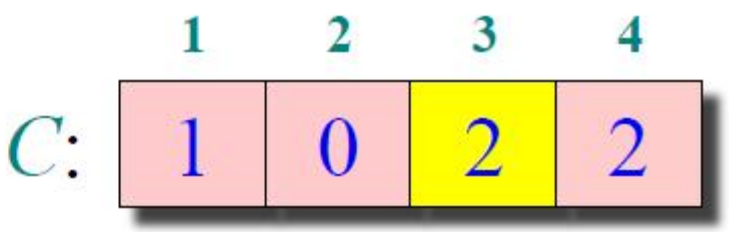
A parenthesis: a quick finish

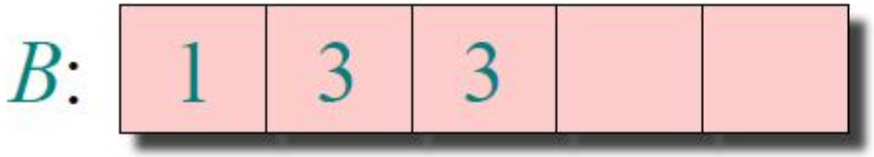


对于C[2]，数组A中键值为2的元素数目为0!

A parenthesis: a quick finish

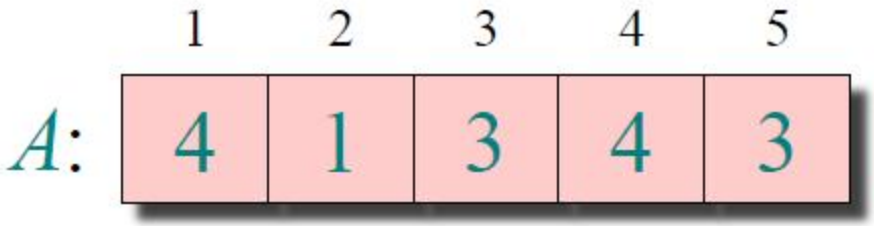


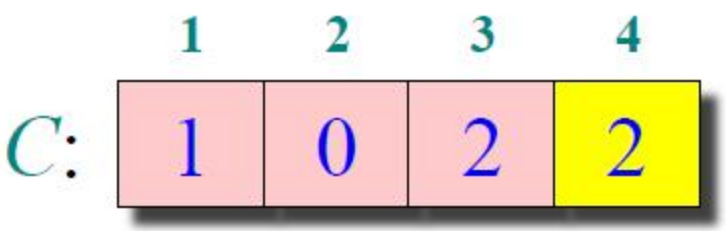


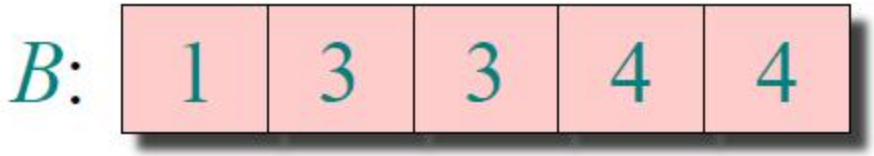


对于C[3]，数组A中键值为3的元素数目为2!

A parenthesis: a quick finish







对于C[4]，数组A中键值为4的元素数目为2!

B是排好了序，但只对键值排了序，没有对A中的元素排序

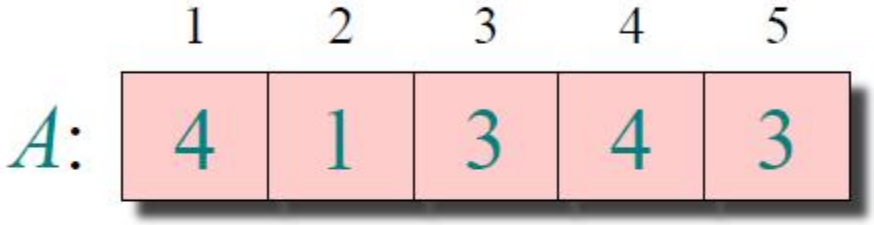
，因为A中元素除了键值还有其他的数据，这里B只有键值的排序而没有对A中元素的交换!

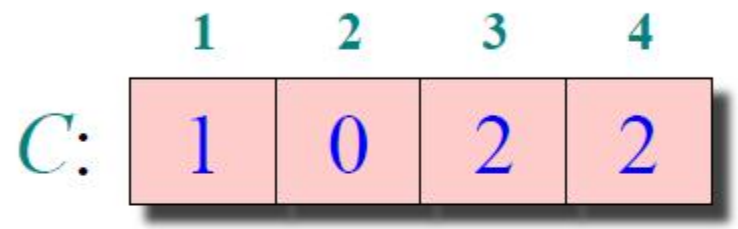
B is sorted!

…but we did not really *permute* the elements of A

(more in a moment)]

Loop 3: from frequencies to cumulative frequencies…





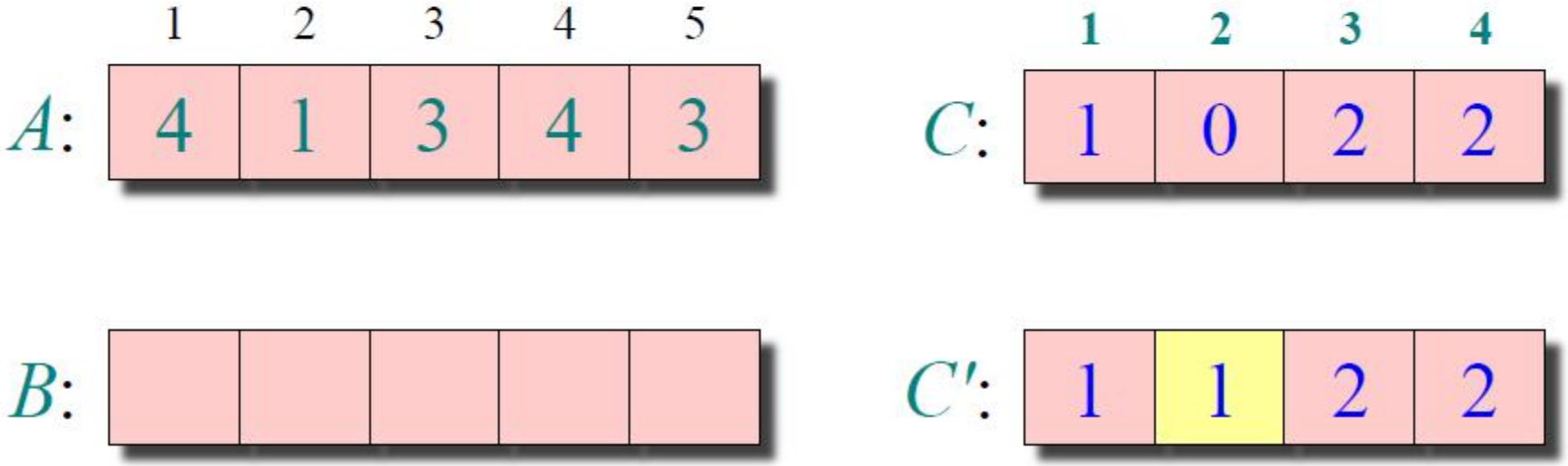
把频率辅助数组C变成累加频率辅助数组C

for $i \leftarrow 2$ **to** k

C[i]=数组A中键值≤i的元素个数

do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

Loop 3: from frequencies to cumulative frequencies…

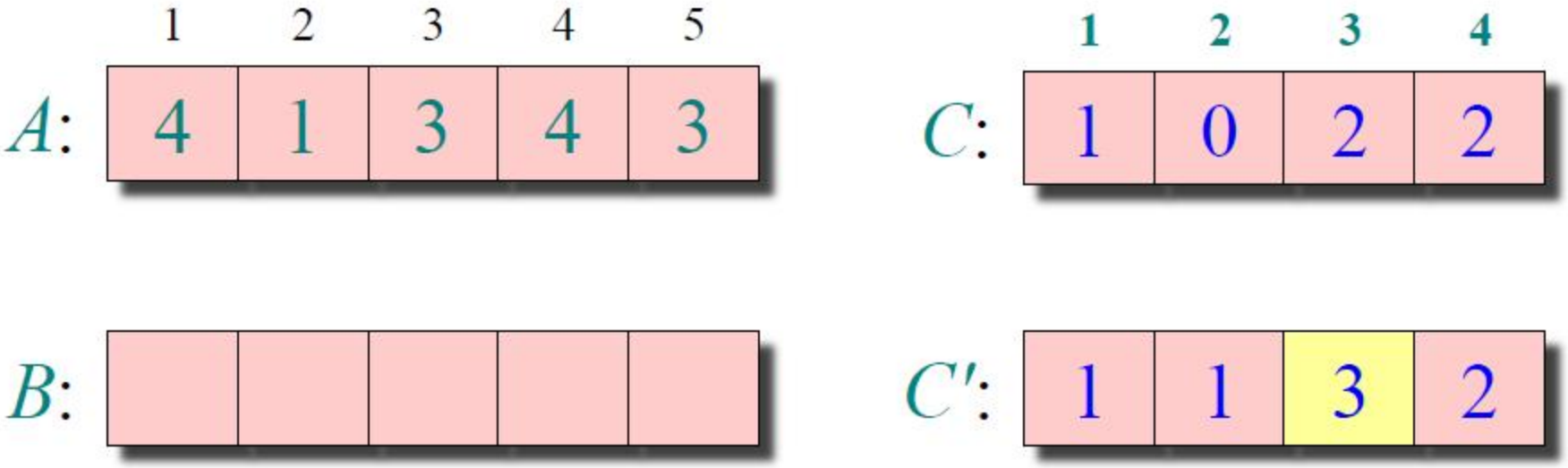


for $i \leftarrow 2$ **to** k

C[i]=数组A中键值≤i的元素个数,i=2

do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

Loop 3: from frequencies to cumulative frequencies…

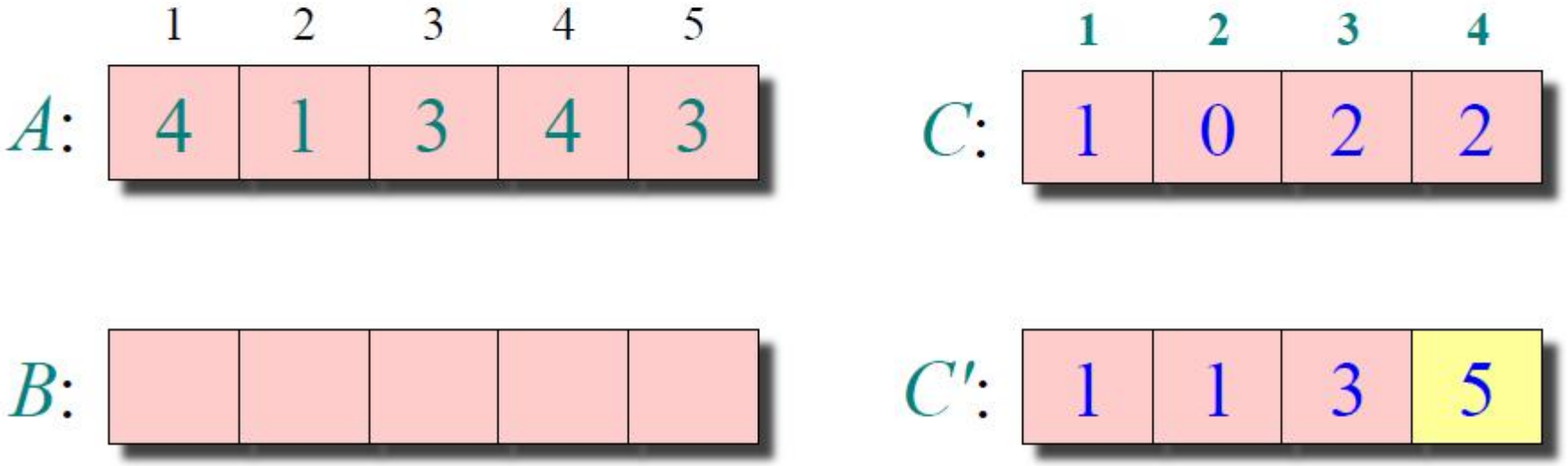


for $i \leftarrow 2$ **to** k

C[i]=数组A中键值≤i的元素个数,i=3

do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

Loop 3: from frequencies to cumulative frequencies…

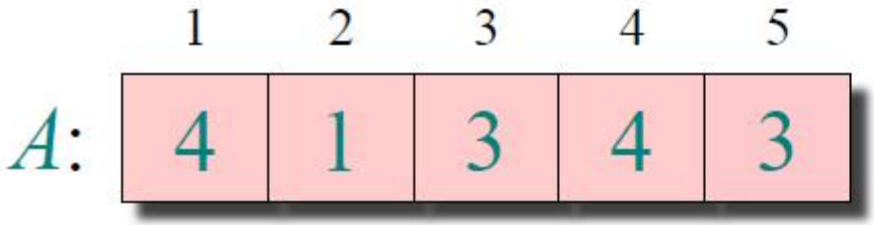


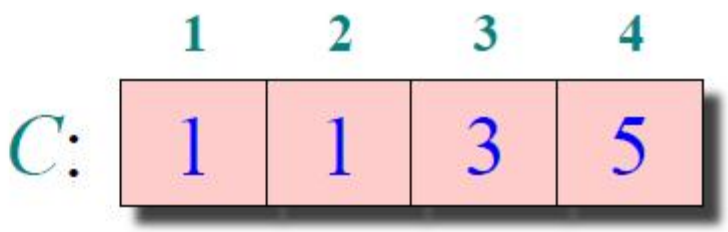
for $i \leftarrow 2$ **to** k

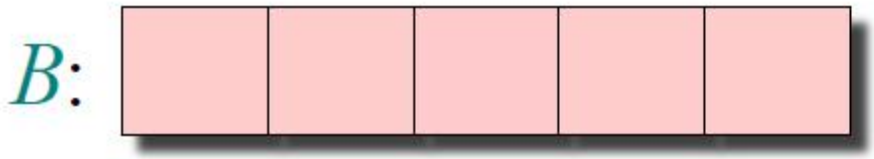
C[i]=数组A中键值≤i的元素个数,i=4

do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

Loop 4: permute elements of A







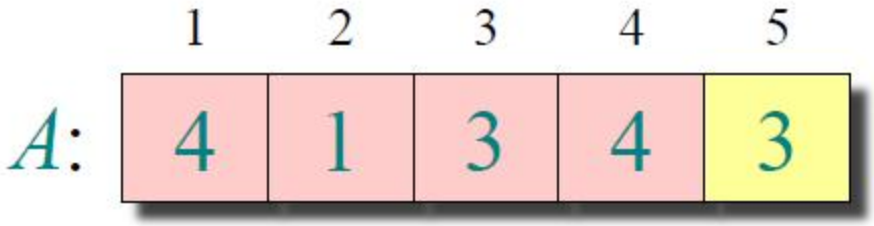
对数组A从尾到头进行元素置换

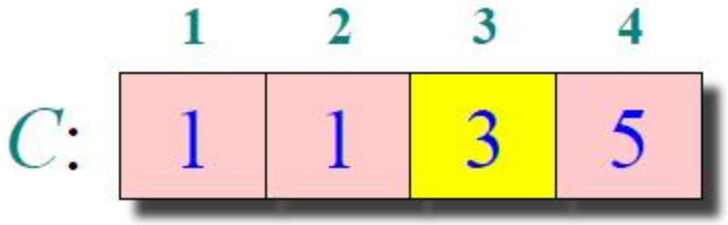
for $j \leftarrow n$ **downto** 1

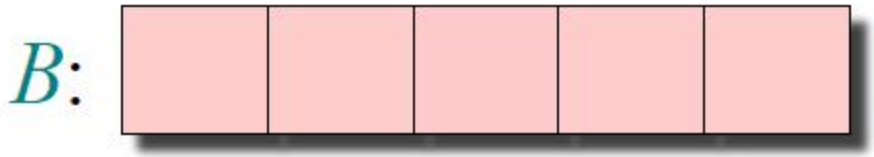
do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A





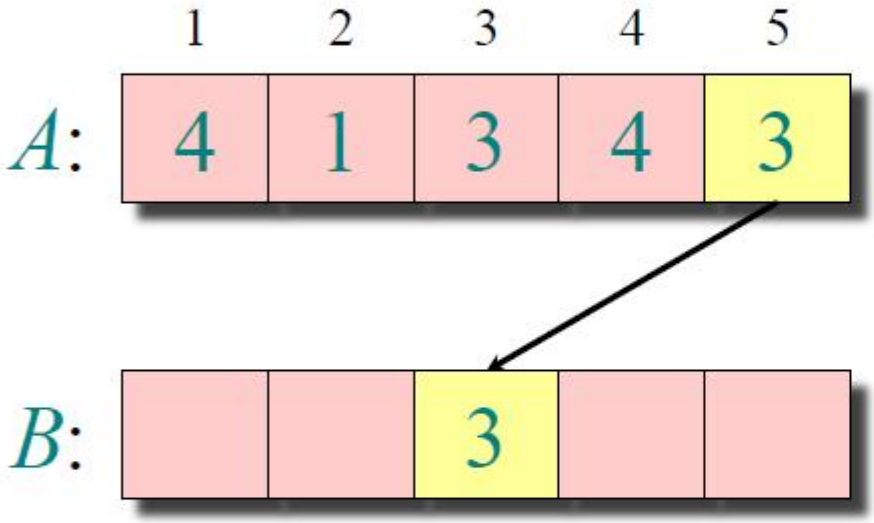


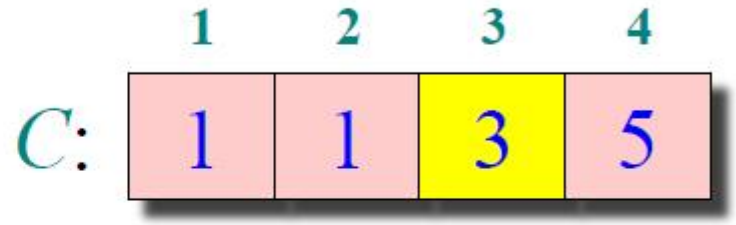
There are exactly 3 elements ≤A[5]. So where should I place A[5]?

正好有3个元素≤ A[5],那么应该把A[5]元素放到哪个位置去?

for $j \leftarrow n$ **downto** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A



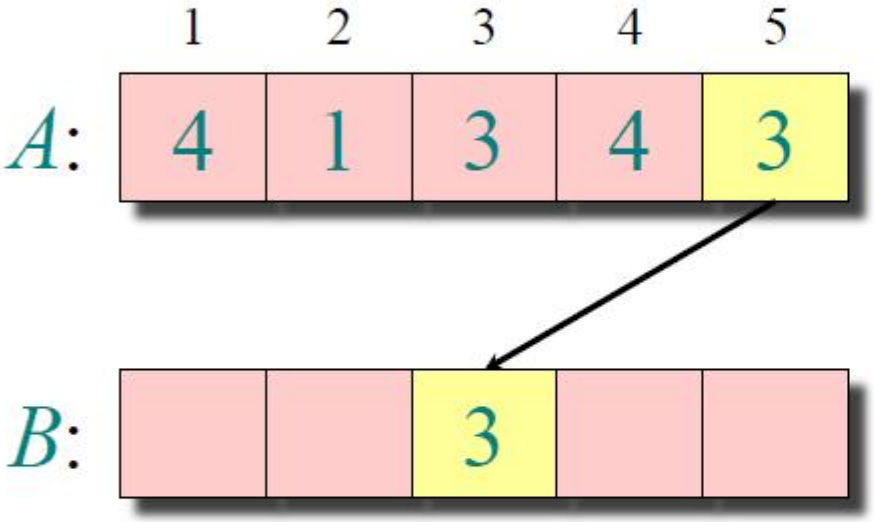


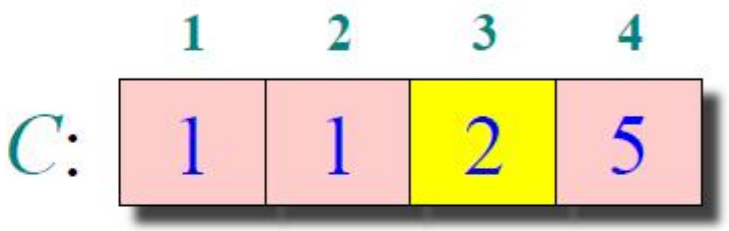
Used-up one 3; update counter in C for the next 3 that shows up...

把A[5]放到B[C[A[5]]]中
用了C[A[5]]的最上面一个元素位置
更新C[A[5]]来给A数组中其他键值为A[5]元素来安排位置!

for $j \leftarrow n$ **downto** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A

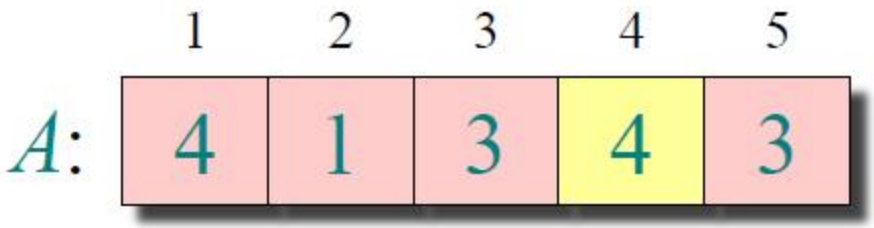


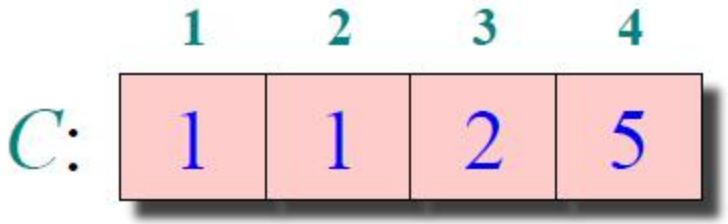


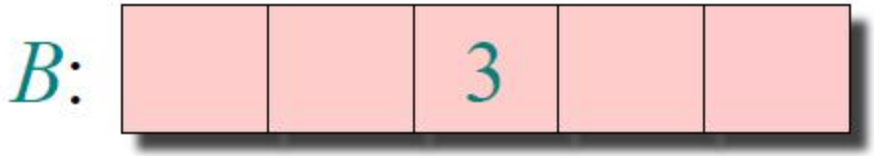
把A[5]放到B[C[A[5]]]中
用了C[A[5]]的最上面一个元素位置
更新C[A[5]]来给A数组中其他键值为A[5]元素来安排位置!

for $j \leftarrow n$ **downto** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A







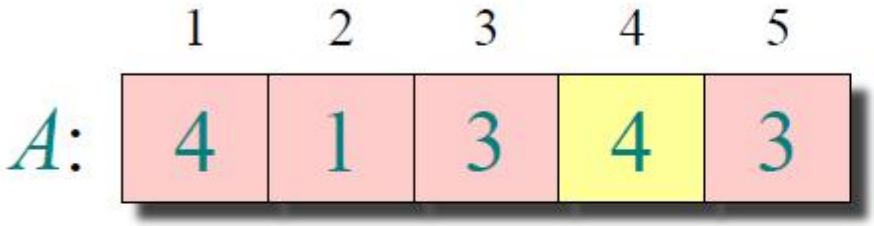
for $j \leftarrow n$ **downto** 1

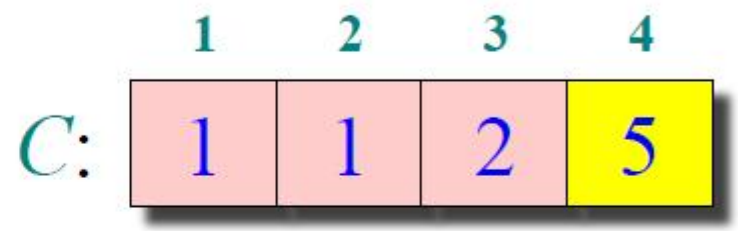
do $B[C[A[j]]] \leftarrow A[j]$

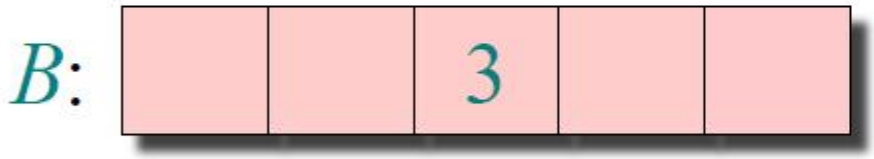
$C[A[j]] \leftarrow C[A[j]] - 1$

把A[4]放到B[C[A[4]]]中
用了C[A[4]]的最上面一个元素位置
更新C[A[4]]来给A数组中其他键值为A[4]元素来安排位置!

Loop 4: permute elements of A





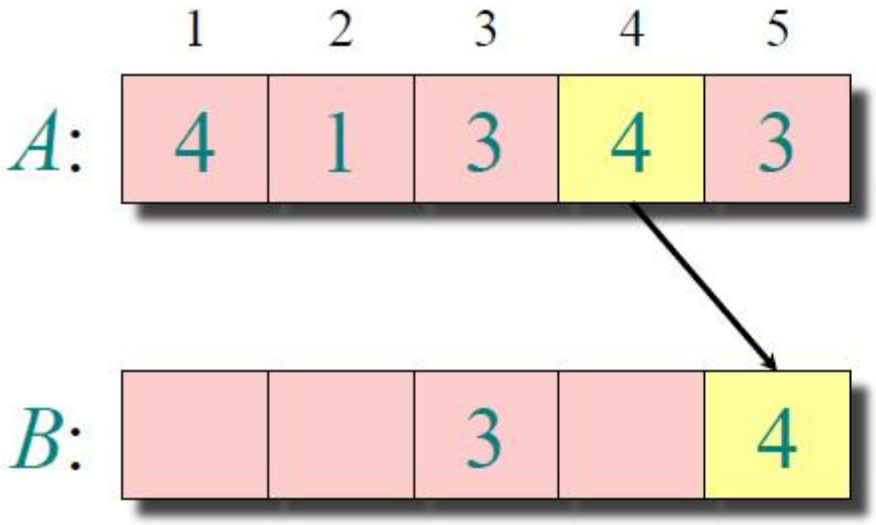


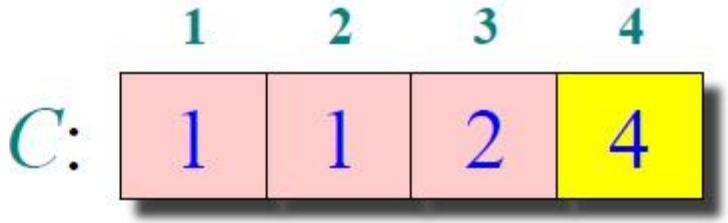
There are exactly 5 elements ≤A[4]. So where should I place A[4]?

把A[4]放到B[C[A[4]]]中
用了C[A[4]]的最上面一个元素位置
更新C[A[4]]来给A数组中其他键值为A[4]元素来安排位置!

for $j \leftarrow n$ **downto** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A





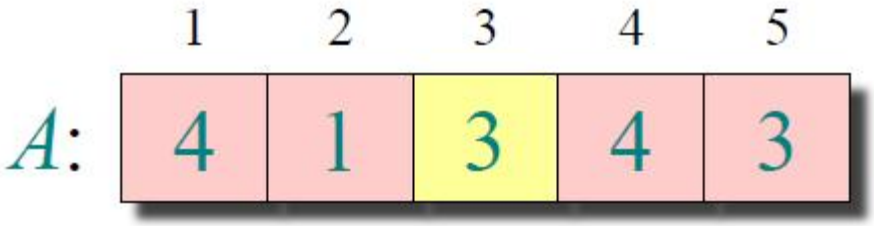
for $j \leftarrow n$ **downto** 1

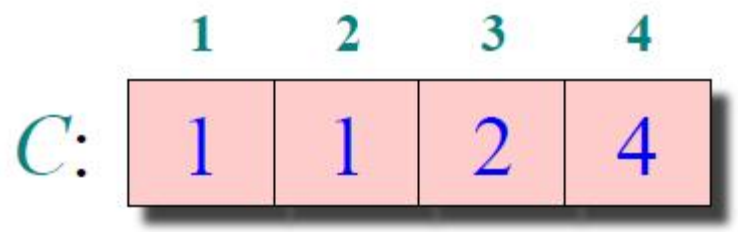
do $B[C[A[j]]] \leftarrow A[j]$

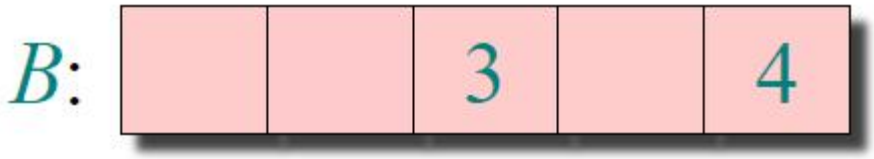
$C[A[j]] \leftarrow C[A[j]] - 1$

把A[4]放到B[C[A[4]]]中
用了C[A[4]]的最上面一个元素位置
更新C[A[4]]来给A数组中其他键值为A[4]元素来安排位置!

Loop 4: permute elements of A







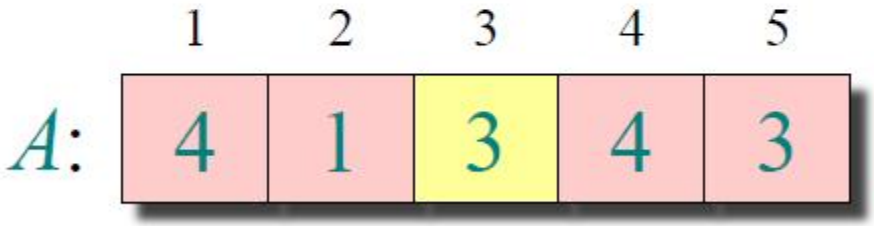
把A[3]放到B[C[A[3]]]中
用了C[A[3]]的最上面一个元素位置
更新C[A[3]]来给A数组中其他键值为A[4]元素来安排位置!

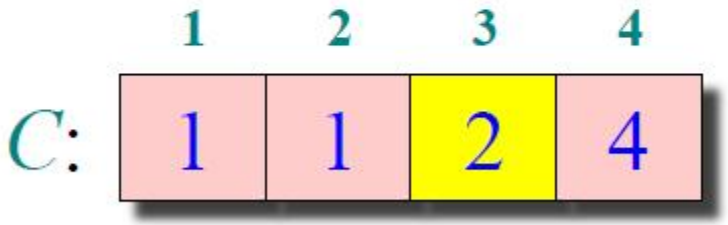
for $j \leftarrow n$ **downto** 1

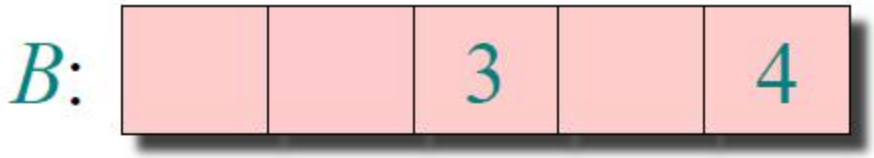
do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A







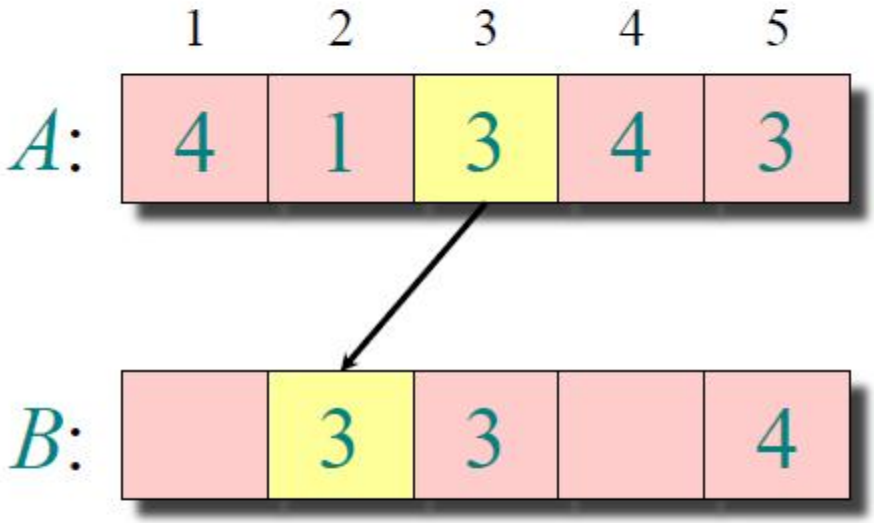
for $j \leftarrow n$ **downto** 1

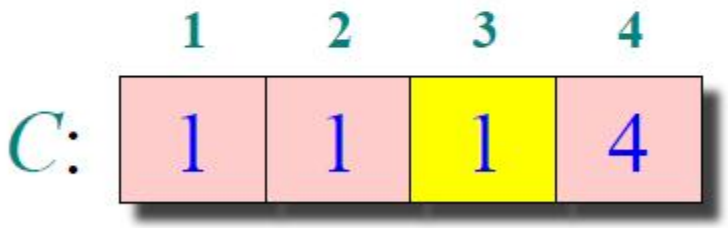
do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

把A[3]放到B[C[A[3]]]中
用了C[A[3]]的最上面一个元素位置
更新C[A[3]]来给A数组中其他键值为A[3]元素来安排位置!

Loop 4: permute elements of A

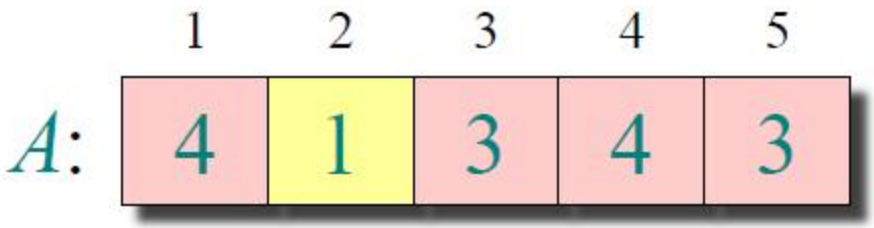


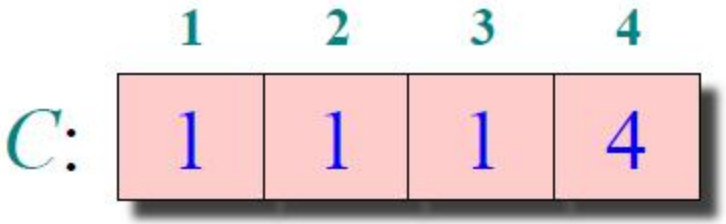


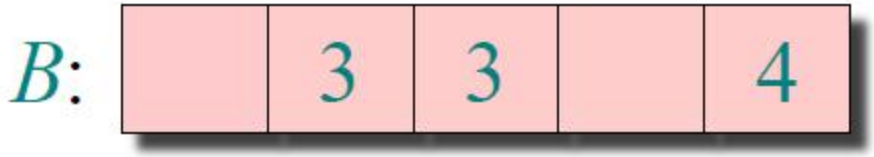
把A[3]放到B[C[A[3]]]中
用了C[A[3]]的最上面一个元素位置
更新C[A[3]]来给A数组中其他键值为A[3]元素来安排位置!

for $j \leftarrow n$ **downto** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A



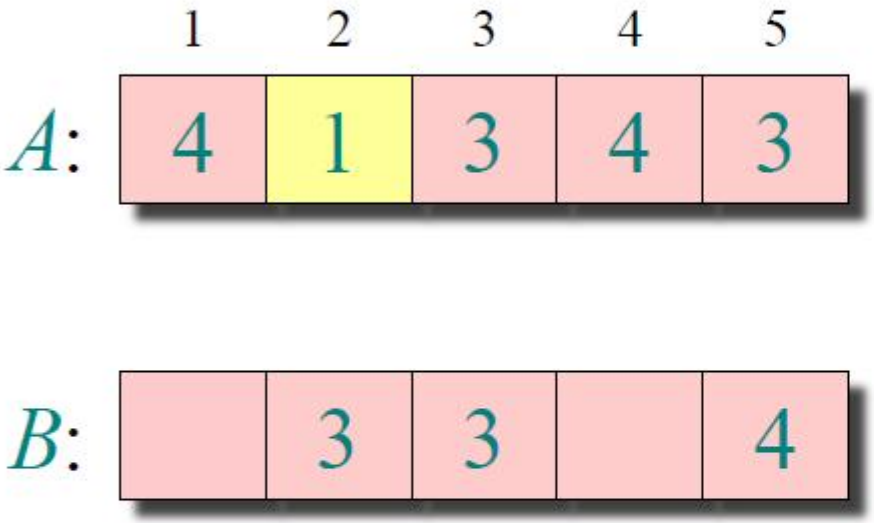


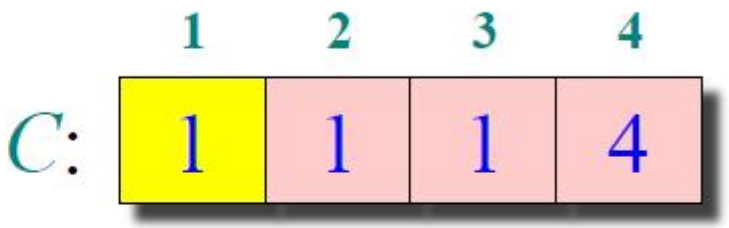


for $j \leftarrow n$ **downto** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

把A[2]放到B[C[A[2]]]中
用了C[A[2]]的最上面一个元素位置
更新C[A[2]]来给A数组中其他键值为A[2]元素来安排位置!

Loop 4: permute elements of A





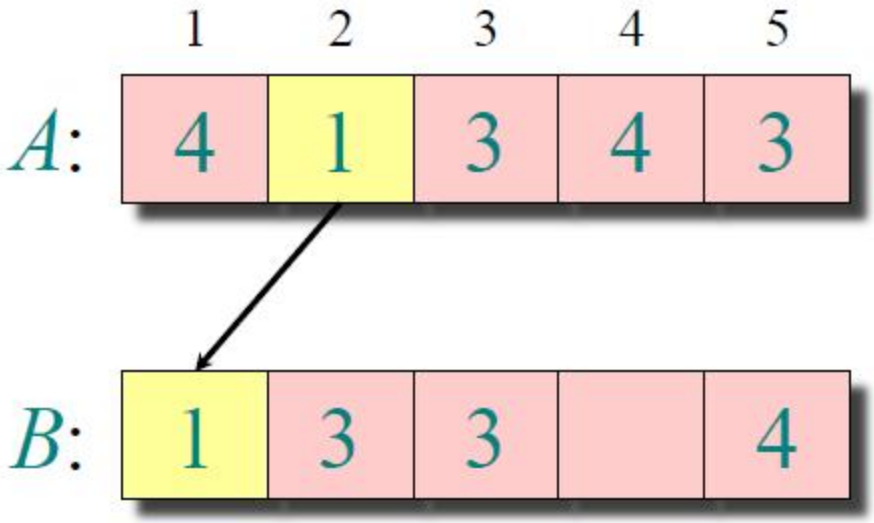
for $j \leftarrow n$ **downto** 1

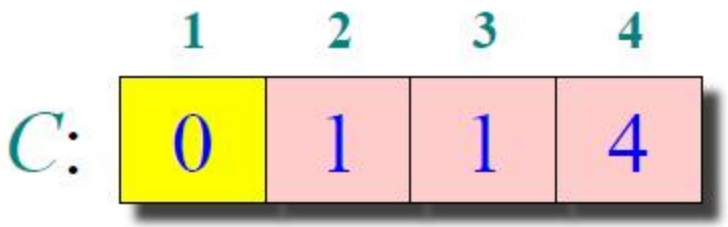
do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

把A[2]放到B[C[A[2]]]中
用了C[A[2]]的最上面一个元素位置
更新C[A[2]]来给A数组中其他键值为A[2]元素来安排位置!

Loop 4: permute elements of A

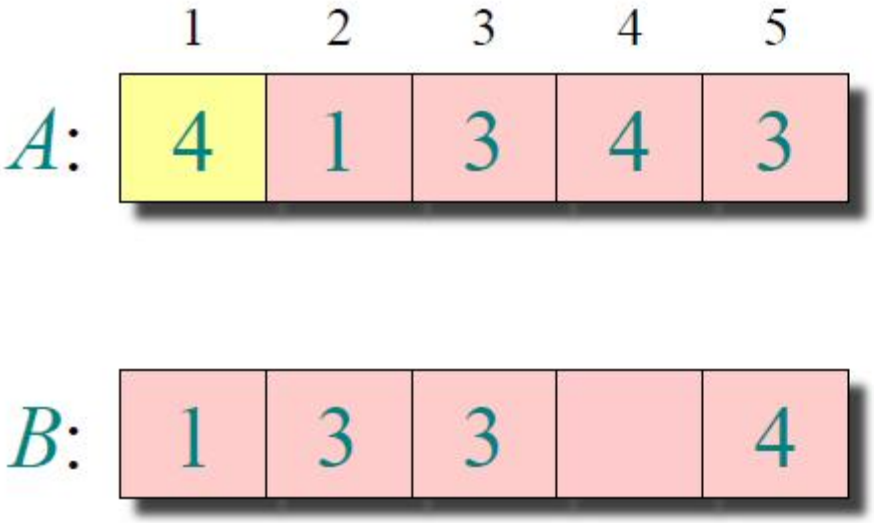




把A[2]放到B[C[A[2]]]中
用了C[A[2]]的最上面一个元素位置
更新C[A[2]]来给A数组中其他键值为A[2]元素来安排位置!

for $j \leftarrow n$ **downto** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

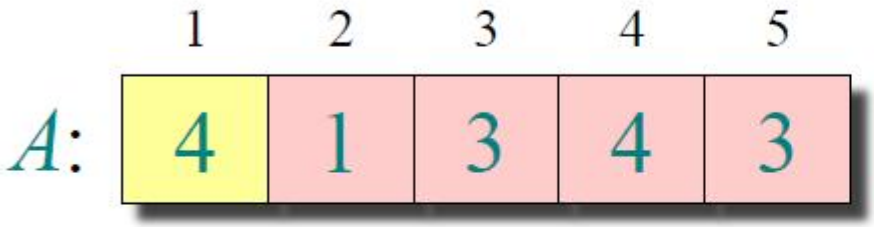
Loop 4: permute elements of A

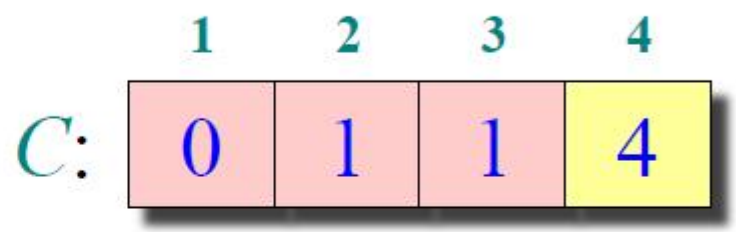


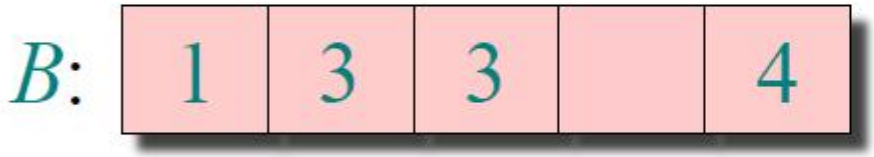
把A[1]放到B[C[A[1]]]中
用了C[A[1]]的最上面一个元素位置
更新C[A[1]]来给A数组中其他键值为A[1]元素来安排位置!

for $j \leftarrow n$ **downto** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4: permute elements of A







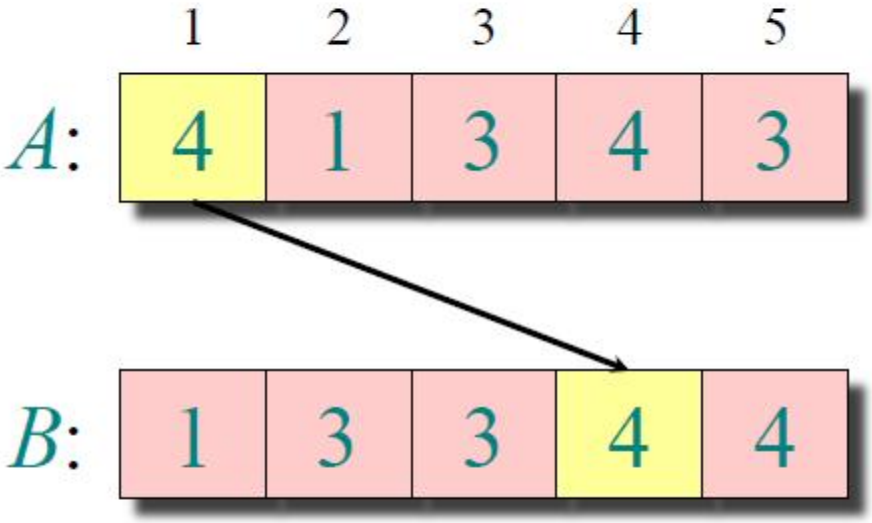
for $j \leftarrow n$ **downto** 1

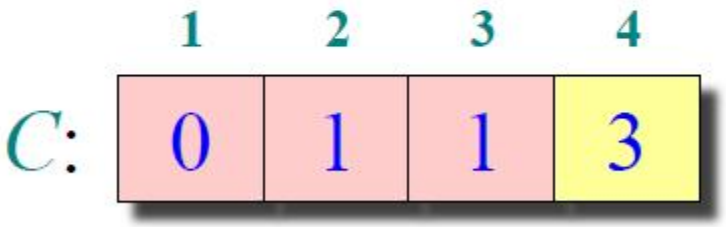
do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

把A[1]放到B[C[A[1]]]中
用了C[A[1]]的最上面一个元素位置
更新C[A[1]]来给A数组中其他键值为A[1]元素来安排位置!

Loop 4: permute elements of A





for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

把A[1]放到B[C[A[1]]]中
用了C[A[1]]的最上面一个元素位置
更新C[A[1]]来给A数组中其他键值为A[1]元素来安排位置!

Counting sort

```
for i ← 1 to k
    do C[i] ← 0
```
$\Theta(k)$

```
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1
```
store in C the frequencies of the different keys in A i.e. $C[i] = |\{\text{key} = i\}|$ — $\Theta(n)$

```
for i ← 2 to k
    do C[i] ← C[i] + C[i–1]
```
store in C the cumulative frequencies of different keys in A, i.e. $C[i] = |\{\text{key} \le i\}|$ — $\Theta(k)$

```
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] – 1
```
using cumulative frequencies build sorted permutation — $\Theta(n)$

$\Theta(n + k)$

Running time

加入k=O(n),那么计数排序需要

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?

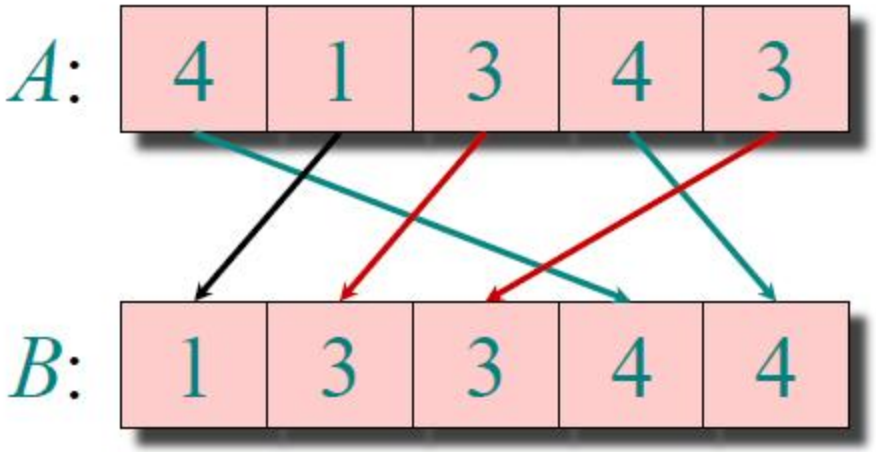
Answer:

- ***Comparison sorting*** takes $\Omega(n \lg n)$ time.
- Counting sort is not a ***comparison sort***.
- In fact, not a single comparison between elements occurs!

Stable sorting

计数排序是一种稳定排序：它给同样键值的元素们保留了原先排序!

Counting sort is a ***stable*** sort: it preserves the input order among equal elements.

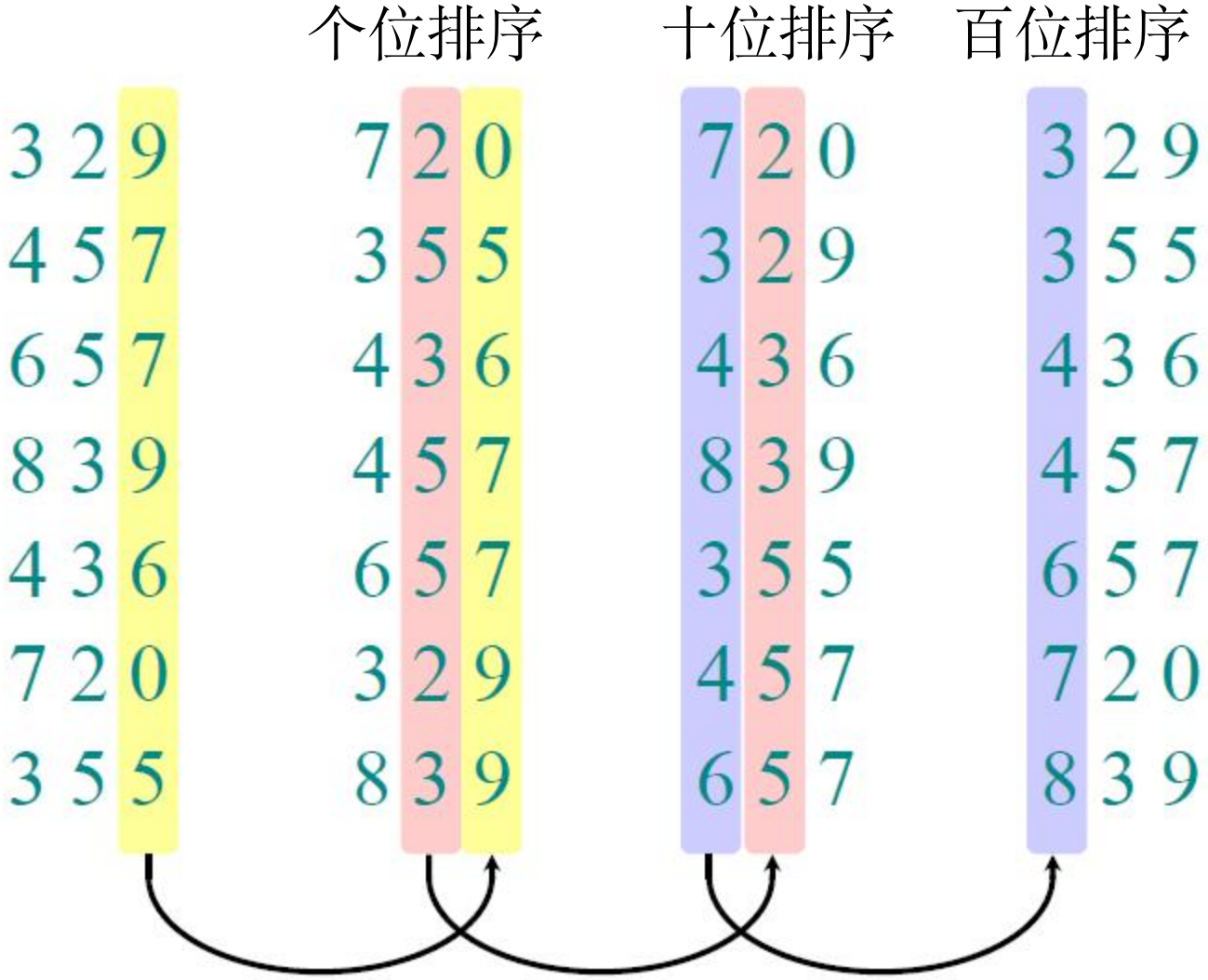


Radix sort 基数排序

- ***Origin***: Herman Hollerith's card-sorting machine for the 1890 U.S. Census. (See Appendix  .) 1890年用来解决美国的人口普查!
- Digit-by-digit sort. 一个位数接一个位数的排序!
- Hollerith's original (bad) idea: sort on most-significant digit first. 原来的（差的）想法：高位优先
- Good idea: Sort on ***least-significant digit first*** with auxiliary ***stable*** sort.

好想法：低位优先

Operation of radix sort



Correctness of radix sort

对位数上的数学归纳

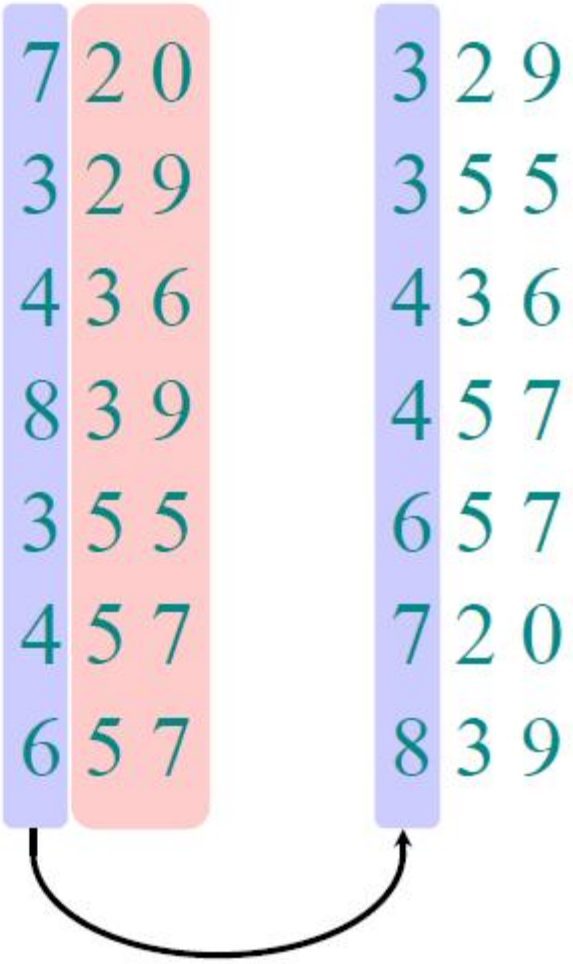
Induction on digit position

- Assume that the numbers are sorted by their low-order $t-1$ digits.

假设t-1位数们，都是排好序的

- Sort on digit t

对t位数的排序操作能保证键值只有t位的数组的排序是正确的。



Correctness of radix sort

对位数上的数学归纳

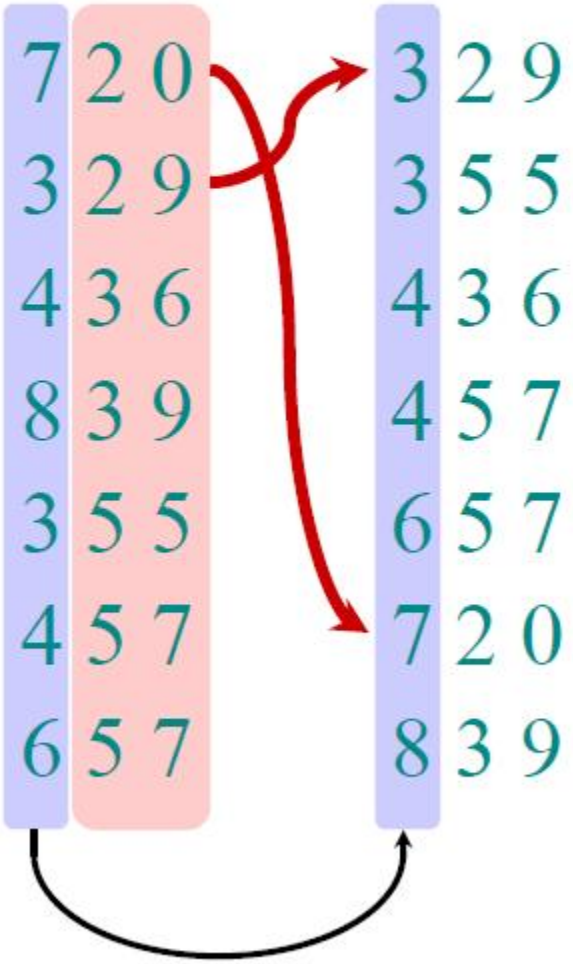
Induction on digit position

- Assume that the numbers are sorted by their low-order $t-1$ digits.

  假设t-1位数们，都是排好序的

- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.

    任意两个在t位数上键值不一样的元素，在t位上比对交换大小排序后，与这两个元素在数位1-t位上的大小顺序是一致的。



Correctness of radix sort

对位数上的数学归纳

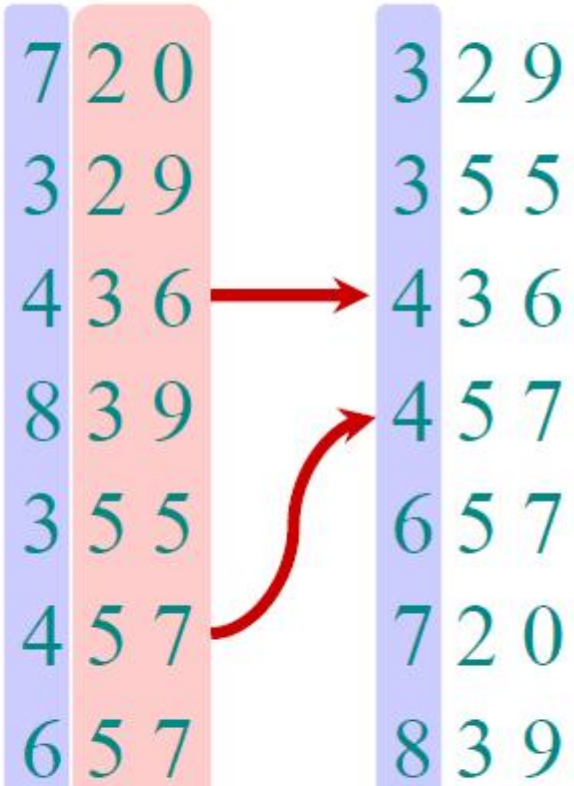
Induction on digit position

- Assume that the numbers are sorted by their low-order $t-1$ digits.

  假设t-1位数们，都是排好序的

- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input ⇒ correct order.

(just used stability property!)



任意两个在t位数上键值一样的元素，与这两个元素在数位1-t位上的大小顺序依然是一致的。（得证！）

Runtime Analysis of radix sort

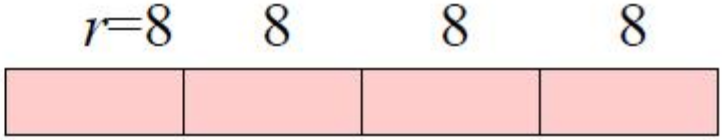
- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.

每一个键值由b个二进制组成

- Each word can be viewed as having b/r base-2^r digits.

基数为2^r个比特，一个键值可以化成b/r个位数

Example: b=32-bit word



- If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time.
- So overall $\Theta(b/r\,(n + 2^r))$ time.

每一个位数的排序时间为$\Theta(n + 2^r)$

所有位数的排序时间为$\Theta(b/r\,(n + 2^r))$

- Setting $r = \log n$ gives $\Theta(n)$ time per pass, or $\Theta(n\,b/\log n)$ total

设定r为log n，则每一个位数的排序时间为$\Theta(n)$,那么所有位数的排序时间为$\Theta(nb/\log n)$