


《数据结构与算法》课程组
重庆大学计算机学院



Data Structures & Algorithms

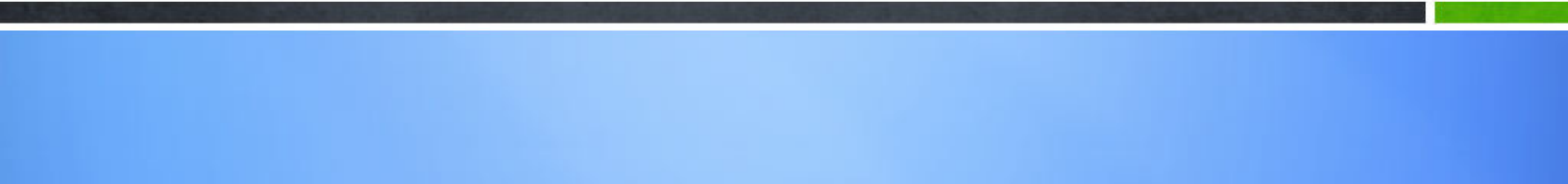




ADVANCED SORTING ALGORITHMS



11.1 Shell Sort



Shellsort (diminishing increment sort)

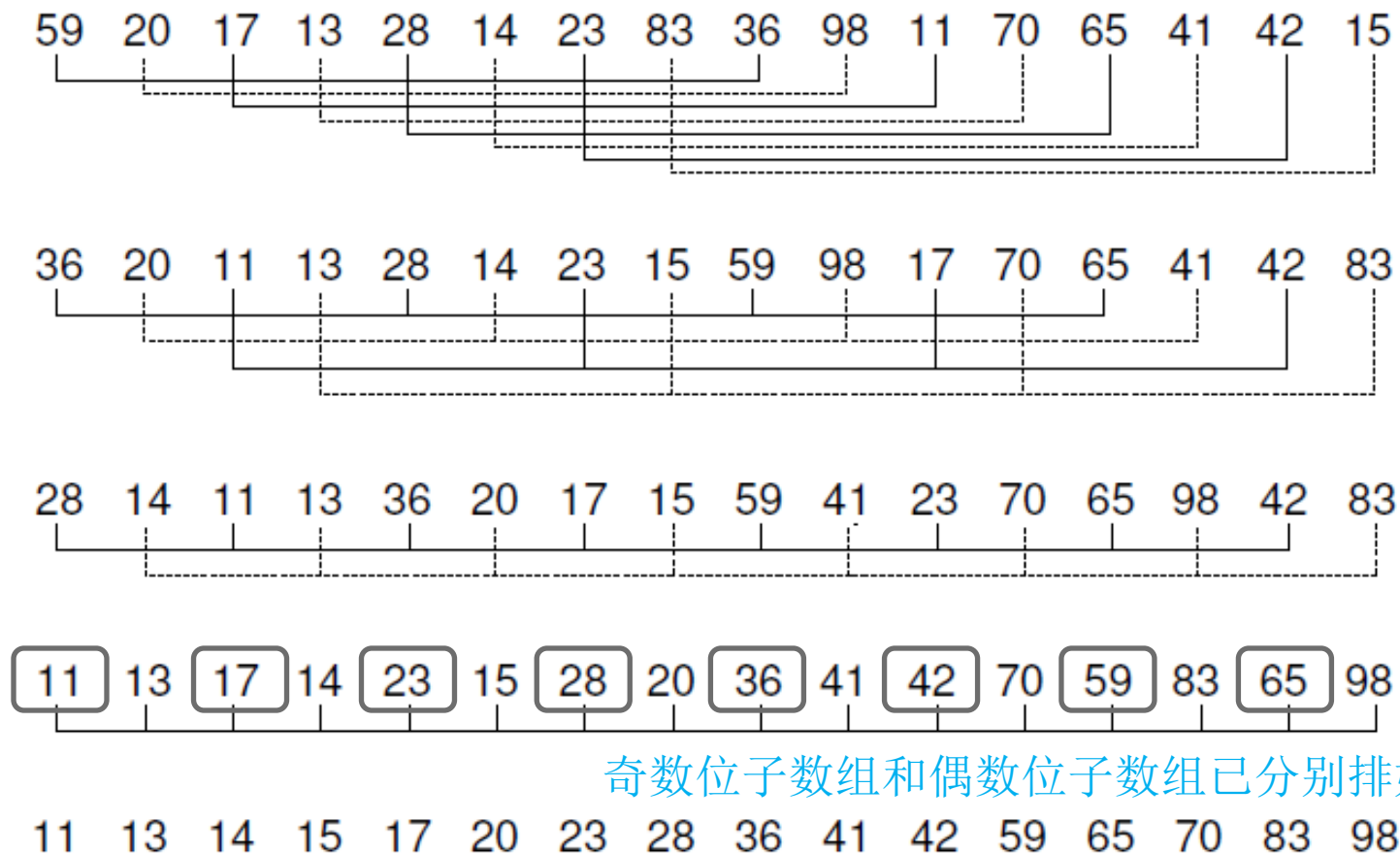
- 对几乎已经排好序的序列，插入排序效率高！接近线性排序的效率
- 一般情况下，插入排序效率低，因为每次只能交换相邻元素的位置

- Shellsort makes comparisons and swaps between **non-adjacent** elements
- Shellsort's strategy is to make the list “**mostly sorted**” so that a final Insertion Sort can finish the job.
- Shellsort is substantially better than Insertion Sort, or any of exchange sorts (?).

Shellsort: Idea

- Shellsort breaks the array of elements into “**virtual**” (logic) sublists.
- Each sublist is sorted using an **Insertion Sort**.
- Another group of sublists is then chosen and sorted,
- and so on.

Shellsort: Example



What is the worst case in time complexity?

Shellsort: Implementation

```
// Modified version of Insertion Sort for varying increments
template <typename E, typename Comp>
void inssort2(E A[], int n, int incr) {
    for (int i=incr; i<n; i+=incr) virtual list
        for (int j=i; (j>=incr) &&
                (Comp::prior(A[j], A[j-incr])); j-=incr)
            swap(A, j, j-incr);
}

template <typename E, typename Comp>
void shellsort(E A[], int n) { // Shellsort
    for (int i=n/2; i>2; i/=2) reducing increment by half
        for (int j=0; j<i; j++) // Sort each sublist
            inssort2<E, Comp>(&A[j], n-j, i);
    inssort2<E, Comp>(A, n, 1);
}
```

Shellsort: history

- 希尔排序最早出现在设计者希尔（Donald Shell）在 1959 年所发表的论文“A high-speed sorting procedure”中。
- 1961年，IBM 公司的女程序员 Marlene Metzner Norton（玛琳·梅茨纳·诺顿）首次使用 FORTRAN 语言编程实现了希尔排序算法。
- 在其程序中使用了一种简易有效的方法设置希尔排序所需的增量序列：**第一个增量取待排序记录个数的一半，然后逐次减半，最后一个增量为 1**。该算法后来被称为 Shell-Metzner 算法。
- 但Metzner 本人在2003年的一封电子邮件中说道：“我没有为这种算法做任何事，我的名字不应该出现在算法的名字中”。

Shellsort: Complexity

- The average-case performance of Shellsort (for “divisions by three” increments) is $O(n^{1.5})$.
- 间隔(increment): $2, 4, 8, \dots, 2^k$ 效率低 $O(n^2)$
- Hibbard间隔: $1, 3, 7, \dots, 2^k - 1$ 效率高 $O(n^{1.5})$

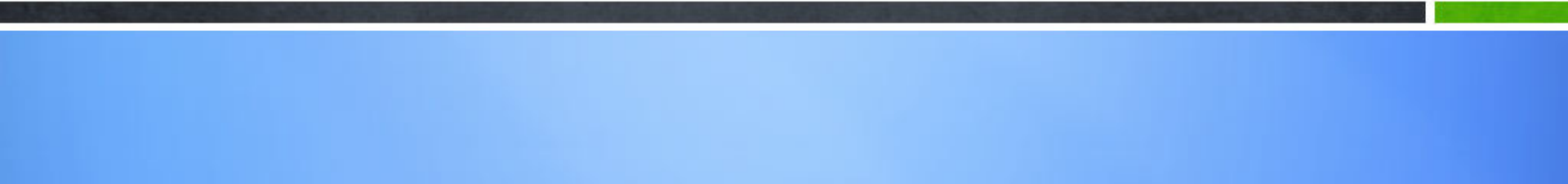
证明很复杂！！

- 作为最早突破 $O(n^2)$ 复杂度的排序算法之一，希尔排序(Shell Sort)一直是闪耀着编程艺术之美的存在。

--- 《计算机程序设计的艺术》The Art of Computer Programming, Donald E. Knuth 著



11.2 Linear-time Sort



Outline

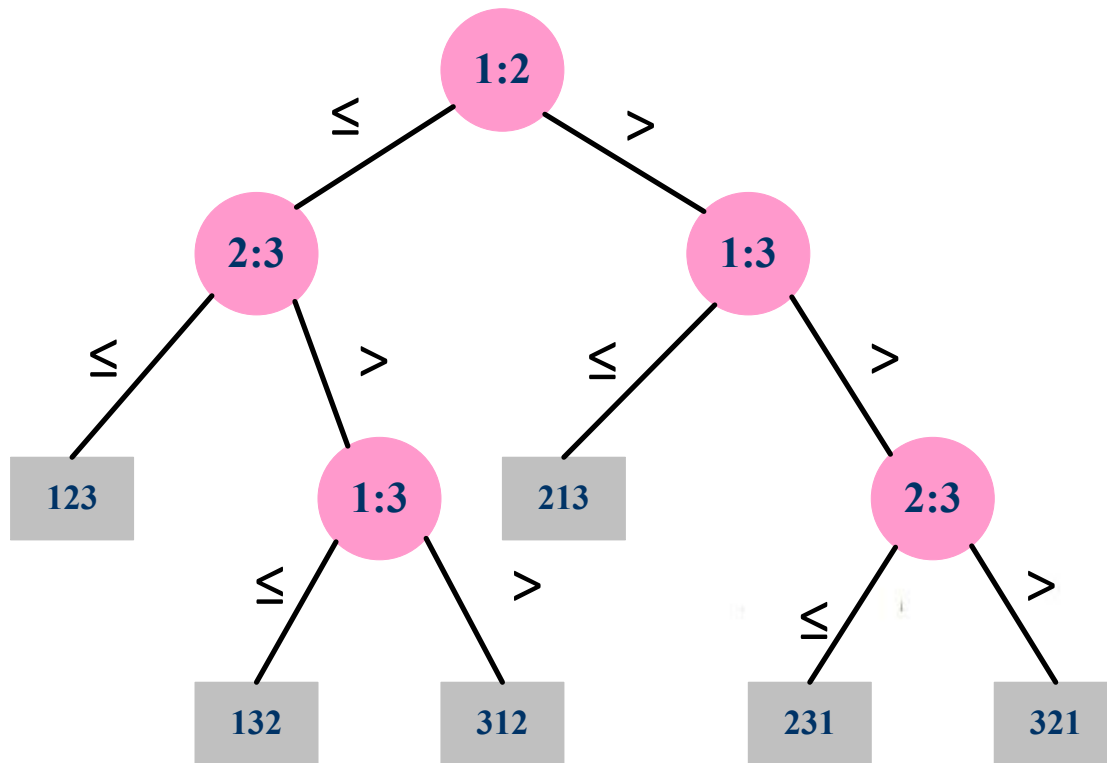
- **1 How fast can we sort?**
- **2 Counting Sort**
- **3 Radix Sort**
- **4 Bucket Sort**

How fast can we sort?

- All the sorting algorithms we have seen so far are comparison sorts: use comparisons to determine the relative order of elements.
 - E.g., **merge sort, quick sort, heap sort.**
 - The best worst-case running time that we've seen for comparison sort is $O(n \log n)$.
 - Actually, this is indeed the **best worst-case running time** of **comparison sort.**

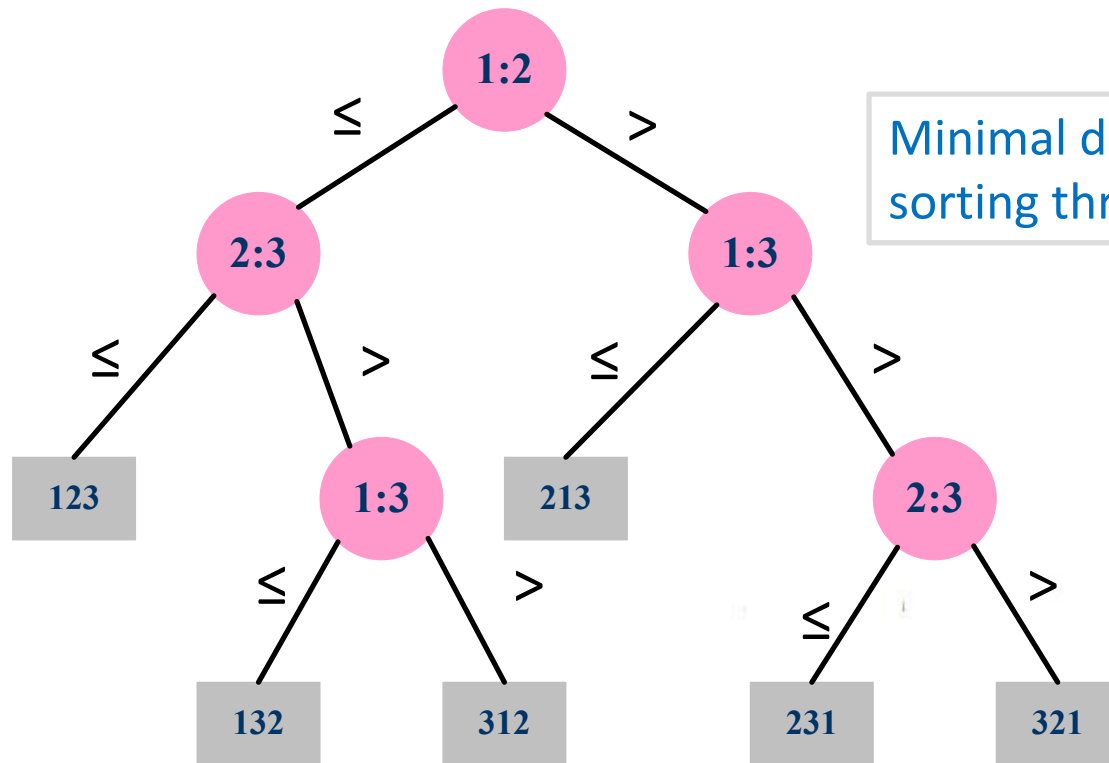
Decision-tree example

- Sort $\langle a_1, a_2, a_3 \rangle$
 - Each **internal node** is labeled $i:j$ for $i, j \in \{1, 2, 3\}$
 - The **left sub-tree** shows subsequent comparisons with $a_i \leq a_j$
 - The **right sub-tree** show subsequent comparisons with $a_i > a_j$



Decision-tree example

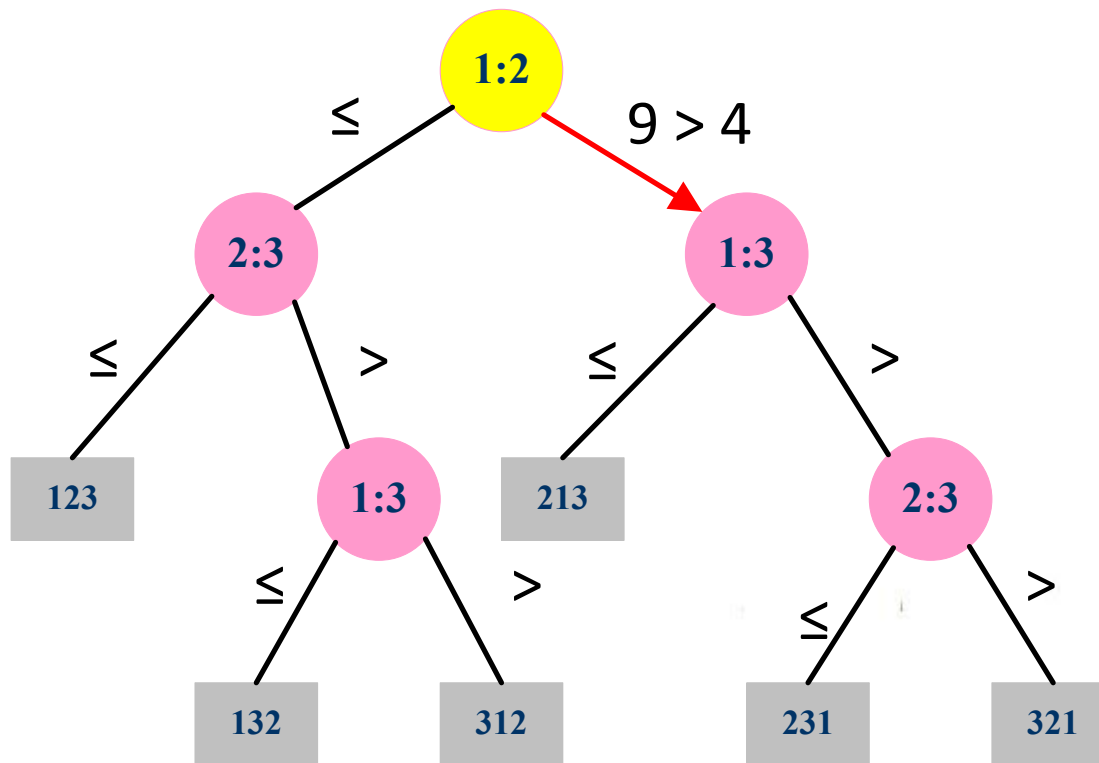
- Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$
 - Each internal node is labeled $i:j$ for $i, j \in \{1, 2, 3\}$
 - The left sub-tree shows subsequent comparisons with $a_i \leq a_j$
 - The right sub-tree show subsequent comparisons with $a_i > a_j$
 - **Each Leaf denotes a permutation of the to-be-sorted array**



Minimal decision-tree for
sorting three elements!

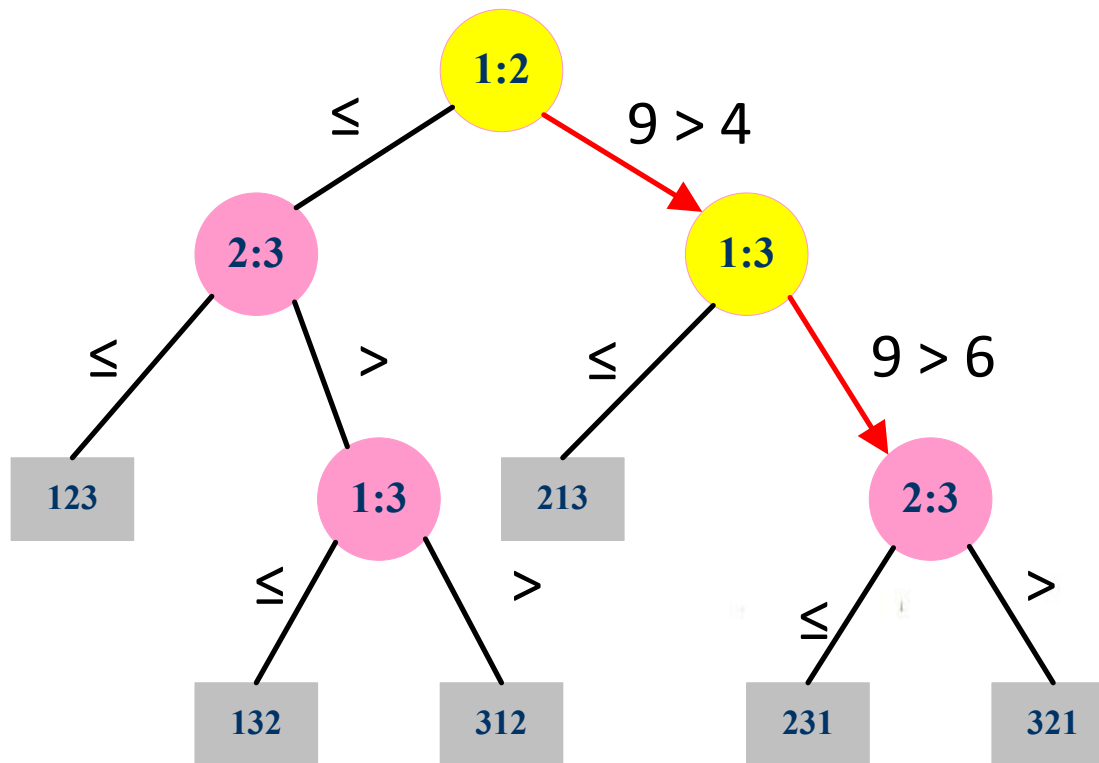
Decision-tree example

- Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$
 - Each internal node is labeled $i:j$ for $i, j \in \{1, 2, 3\}$
 - The left sub-tree shows subsequent comparisons with $a_i \leq a_j$
 - The right sub-tree show subsequent comparisons with $a_i > a_j$
 - Each Leaf denotes a permutation of the to-be-sorted array



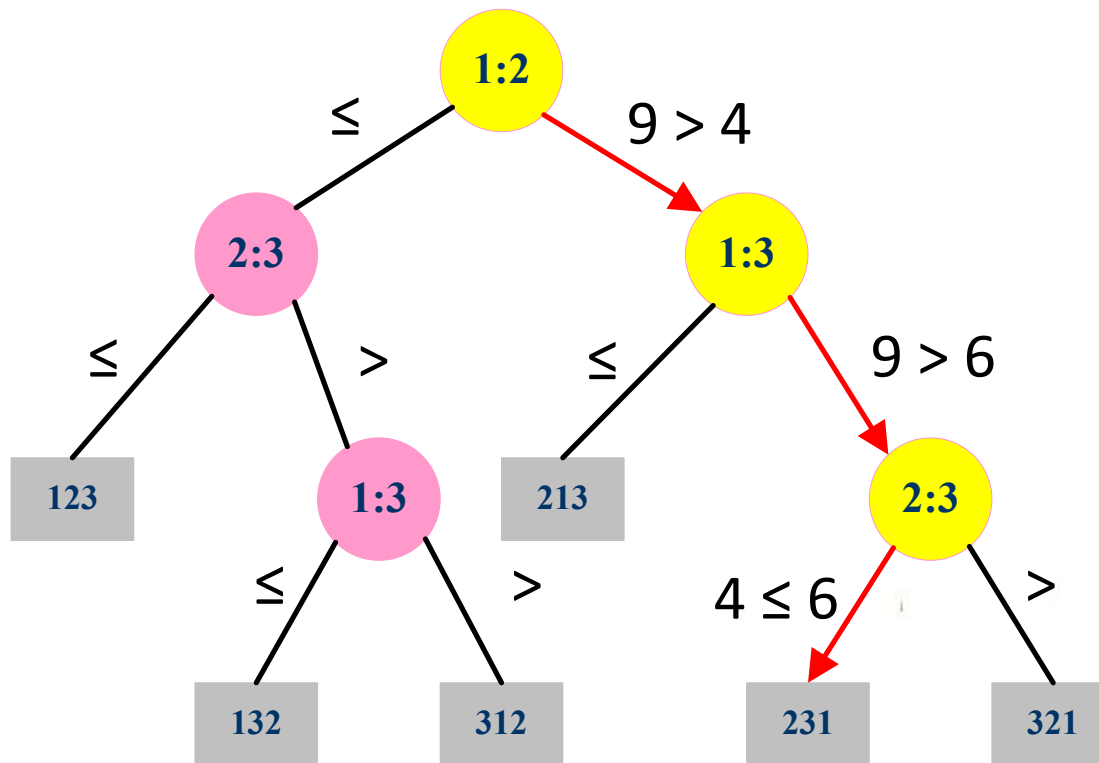
Decision-tree example

- Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$
 - Each internal node is labeled $i:j$ for $i, j \in \{1, 2, 3\}$
 - The left sub-tree shows subsequent comparisons with $a_i \leq a_j$
 - The right sub-tree show subsequent comparisons with $a_i > a_j$
 - Each Leaf denotes a permutation of the to-be-sorted array



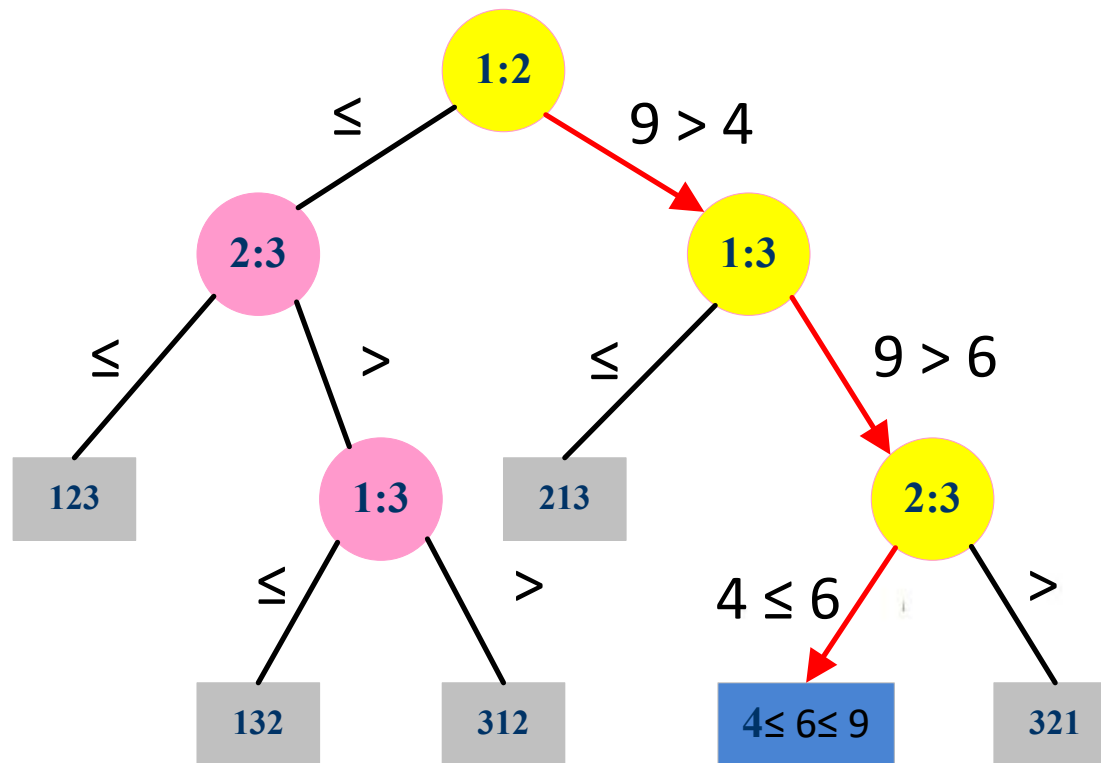
Decision-tree example

- Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$
 - Each internal node is labeled $i:j$ for $i, j \in \{1, 2, 3\}$
 - The left sub-tree shows subsequent comparisons with $a_i \leq a_j$
 - The right sub-tree show subsequent comparisons with $a_i > a_j$
 - Each Leaf denotes a permutation of the to-be-sorted array



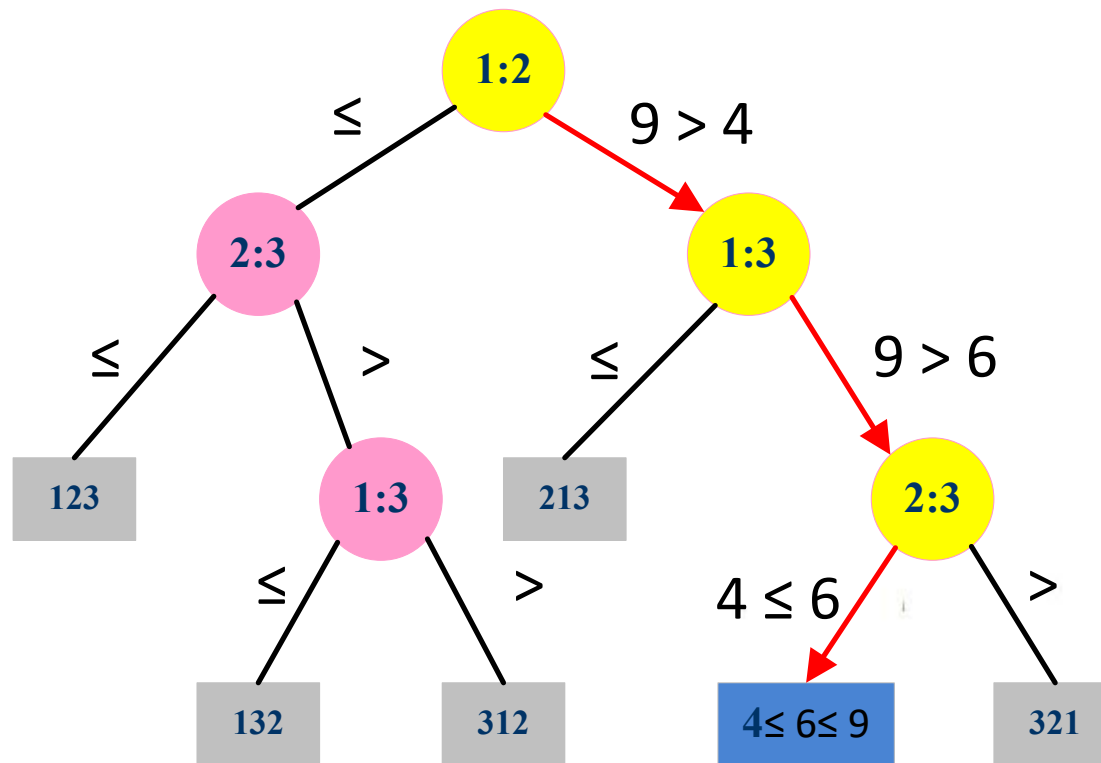
Decision-tree example

- Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$
 - Each internal node is labeled $i:j$ for $i, j \in \{1, 2, 3\}$
 - The left sub-tree shows subsequent comparisons with $a_i \leq a_j$
 - The right sub-tree show subsequent comparisons with $a_i > a_j$
 - Each Leaf denotes a permutation of the to-be-sorted array



Decision-tree example

- Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$
 - The tree contains the comparisons along all possible paths.
 - The running time of the algorithm = the length of the path taken.
 - The Worst-case running time = the height of the tree.



Decision Tree

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

- One tree for each input size n .
- Each leaf indicates a specific total ordering of all elements.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

Lower Bound for Comparison Sorting

Theorem. Any comparison sorting algorithm requires $\Omega(n \log n)$ **comparisons** in the worst case

- *Proof:*

- An n -element-array have $n!$ different permutations.
- We need at least **one leaf** to denote **one permutation**.
- The tree must contain at least $n!$ **leaves**.
- A **height- h** binary tree has at most 2^h leaves
- Thus, **the lower bound of h** can be denoted by:

$$h \geq \log(n!)$$

$$\geq \log(n/e)^n \quad \text{-----} \rightarrow \quad \text{Stirling's Approximation}$$

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

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

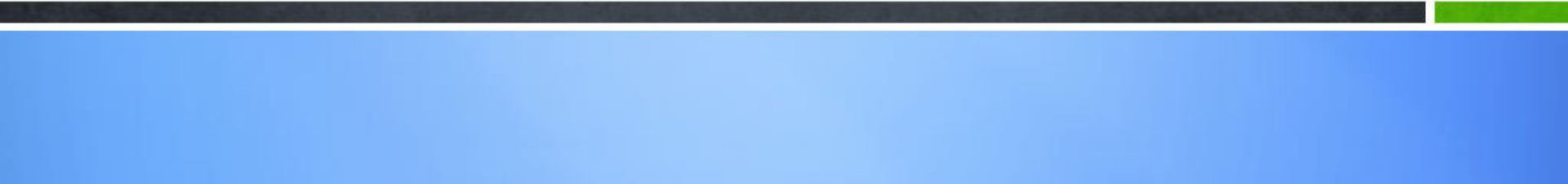
Heapsort and merge sort are asymptotically optimal comparison sorting algorithms.

Can We Sort Faster?

- **Is there a faster algorithm?**
- **Can we sort in linear time?**
- **HOW?**



11.2.1 Counting Sort



Counting Sort

- Counting sort: sort without comparison
 - Input: $A[1..n]$, where $A[j] \in \{1, 2, \dots, k\}$
 - Output: sorted array $B[1..n]$
 - Auxiliary storage: $C[1..k]$

P.S.: Note that the size of the auxiliary array is decided by the **largest** element in A .

Counting principles

- For each element x in A , if there are 17 elements less than x in A , then x deserve the output position 18.
- What if multiple elements equal to x in A ?
 - Put the 1st in position 18, 2nd in position 19, 3rd in position 20, etc. (successive!)
- What if there are 17 elements not greater than x in A ?
 - Put the last in position 17, the second-last in position 16, etc.

Counting Sort Pseudo Code

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

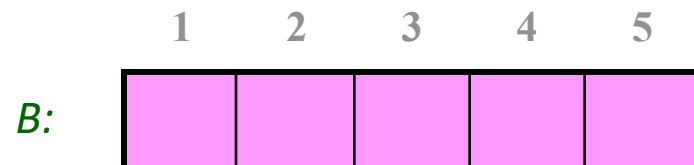
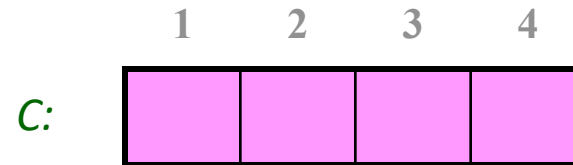
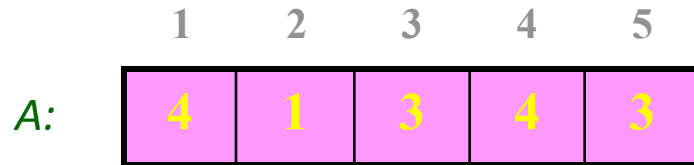
11 C[A[j]] ← C[A[j]]-1

Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

```
1  for i ← 1 to k
2      do C[i] ← 0
3  for j ← 1 to length[A]
4      do C[A[j]] ← C[A[j]]+1
5  //C[i] now contains the number of elements equal to i.
6  for i ← 2 to k
7      do C[i] ← C[i]+ C[i-1]
8  //C[i] now contains the number of elements less than or equal to i.
9  for j ← length[A] downto 1
10     do B[C[A[j]]] ← A[j]
11     C[A[j]] ← C[A[j]]-1
```



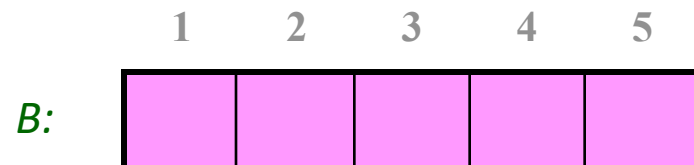
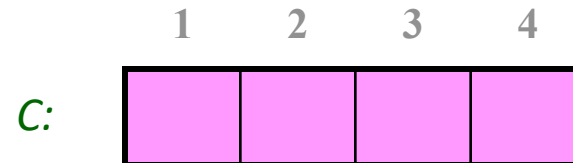
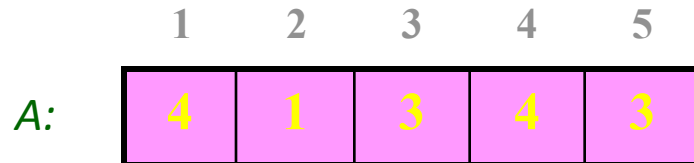
n=5, k=4

Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

```
1  for i ← 1 to k
2    do C[i] ← 0
3  for j ← 1 to length[A]
4    do C[A[j]] ← C[A[j]]+1
5  //C[i] now contains the number of elements equal to i.
6  for i ← 2 to k
7    do C[i] ← C[i]+ C[i-1]
8  //C[i] now contains the number of elements less than or equal to i.
9  for j ← length[A] downto 1
10   do B[C[A[j]]] ← A[j]
11   C[A[j]] ← C[A[j]]-1
```

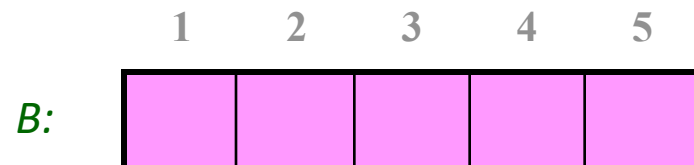
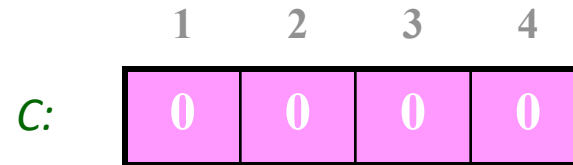
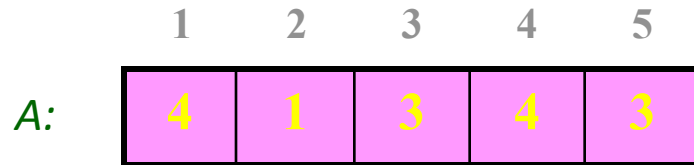


Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

```
1  for i ← 1 to k
2    do C[i] ← 0
3  for j ← 1 to length[A]
4    do C[A[j]] ← C[A[j]]+1
5  //C[i] now contains the number of elements equal to i.
6  for i ← 2 to k
7    do C[i] ← C[i]+ C[i-1]
8  //C[i] now contains the number of elements less than or equal to i.
9  for j ← length[A] downto 1
10   do B[C[A[j]]] ← A[j]
11   C[A[j]] ← C[A[j]]-1
```



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

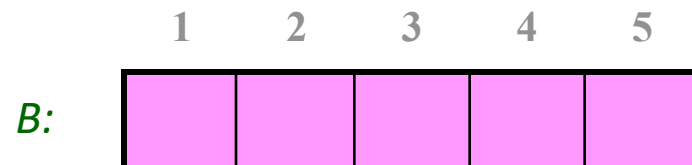
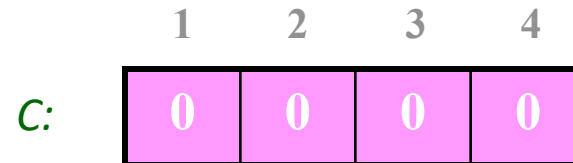
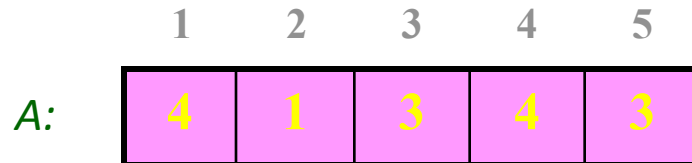
7 do C[i] ← C[i]+ C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

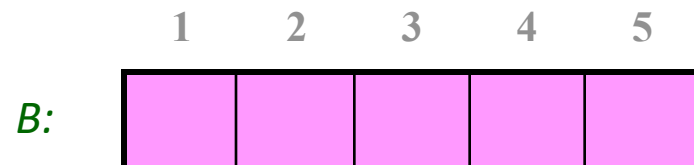
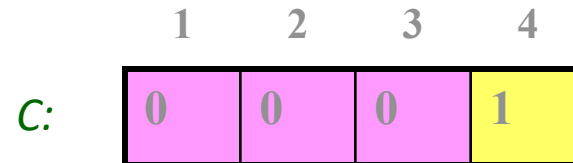
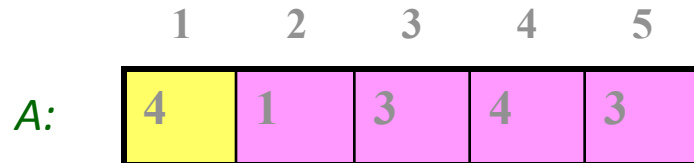
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

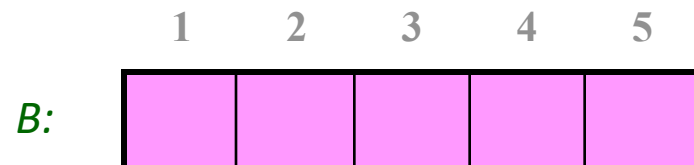
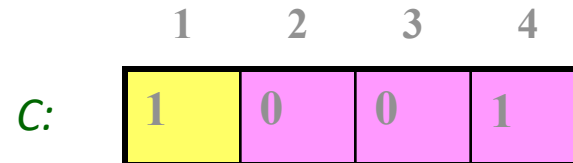
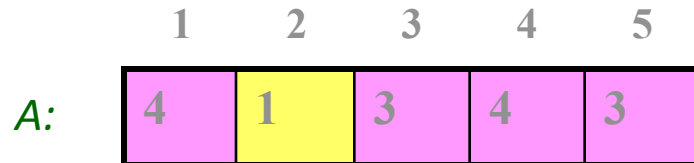
7 do C[i] ← C[i]+ C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

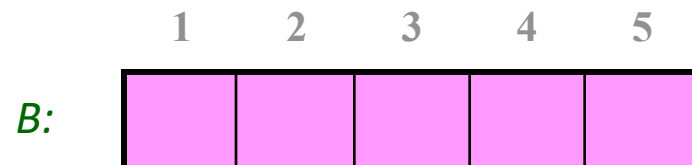
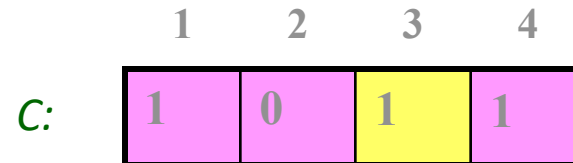
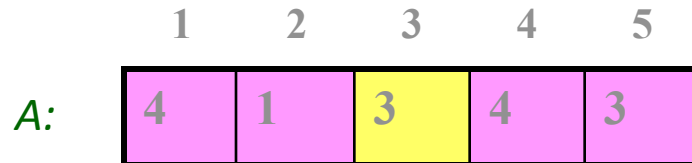
7 do C[i] ← C[i]+ C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

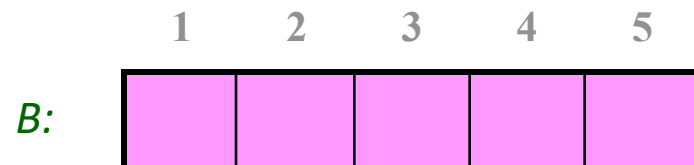
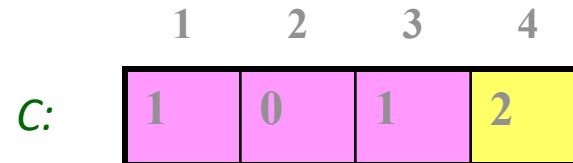
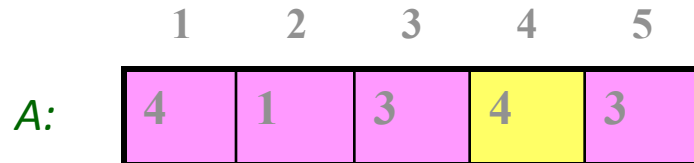
7 do C[i] ← C[i]+ C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

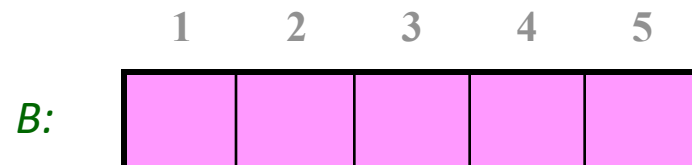
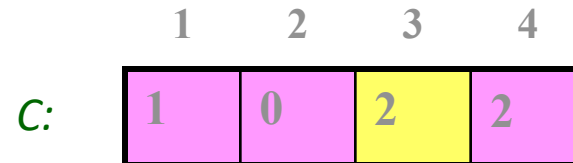
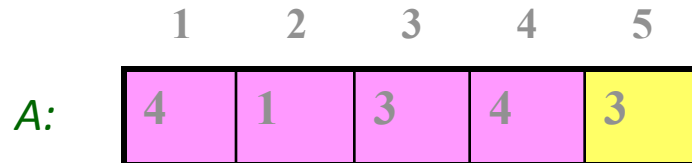
7 do C[i] ← C[i]+ C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

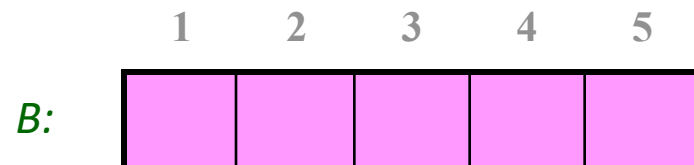
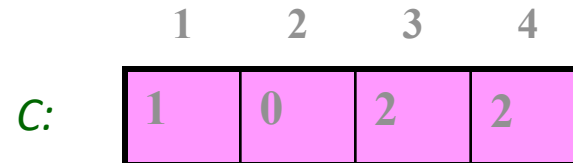
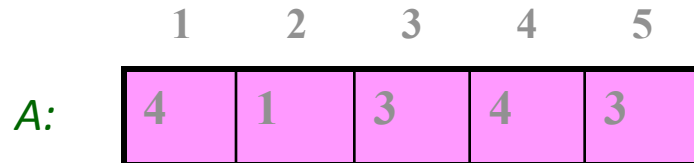
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

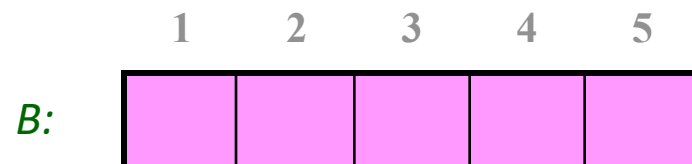
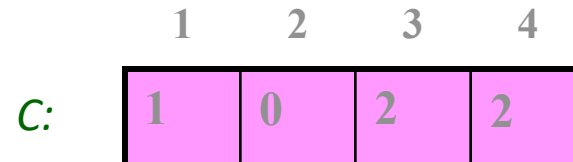
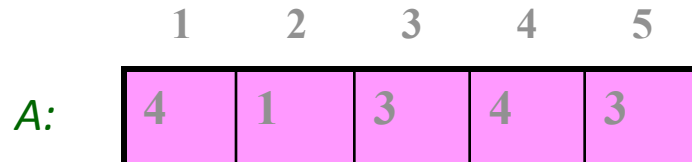
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

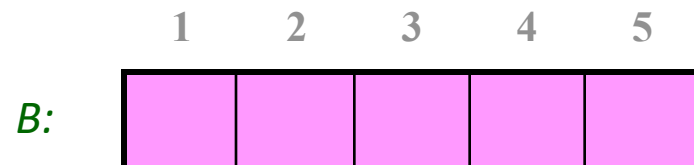
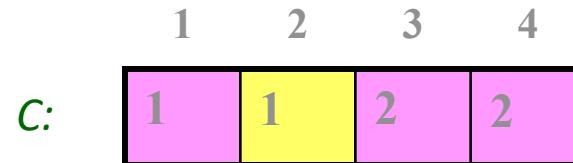
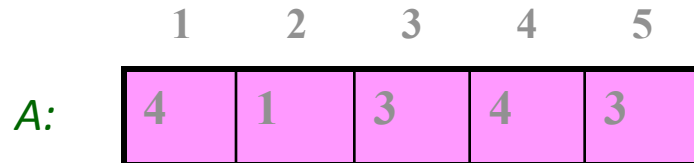
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

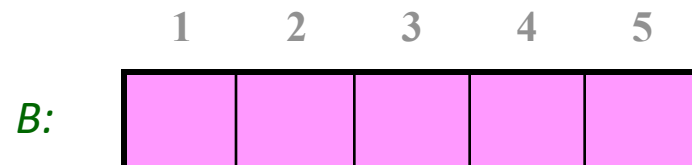
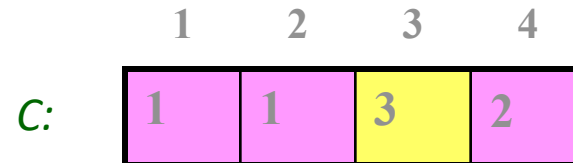
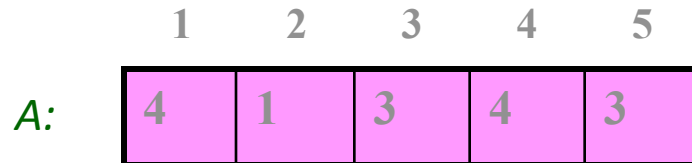
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

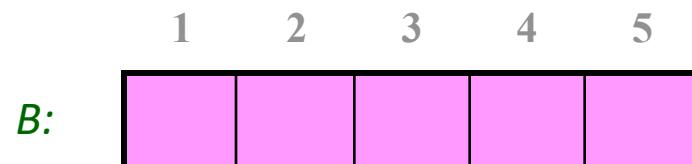
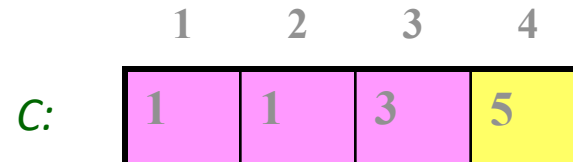
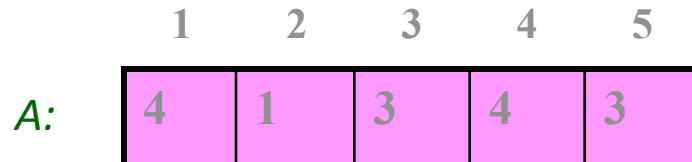
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

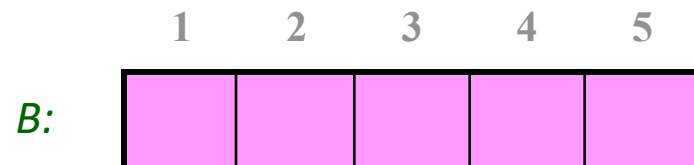
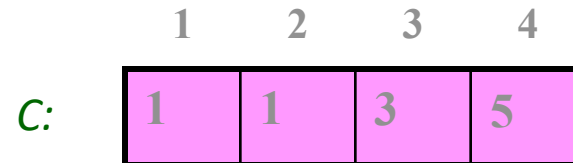
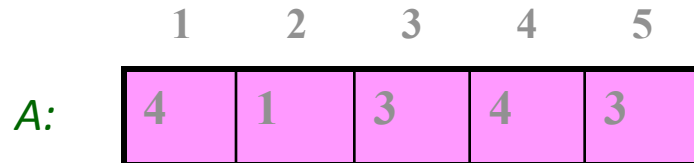
7 do C[i] ← C[i]+ C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

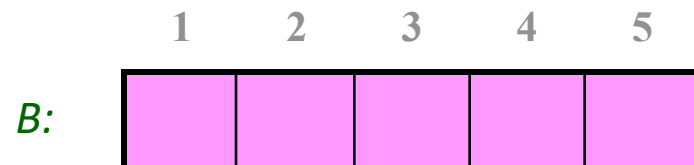
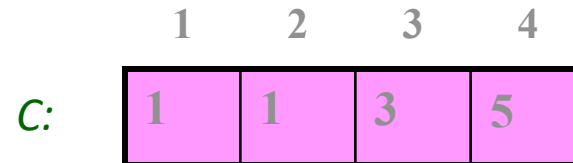
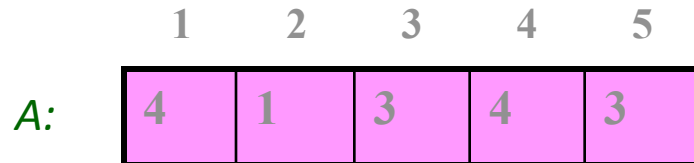
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

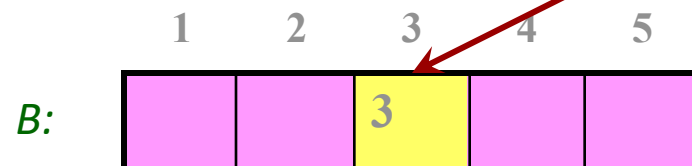
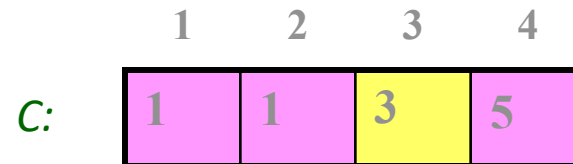
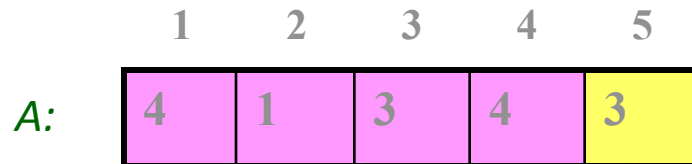
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

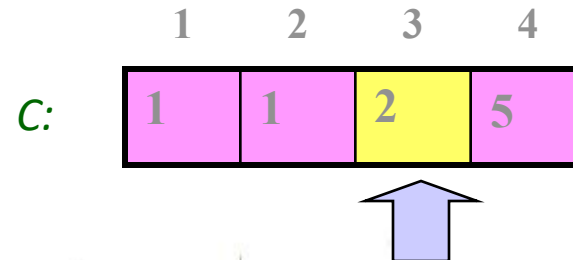
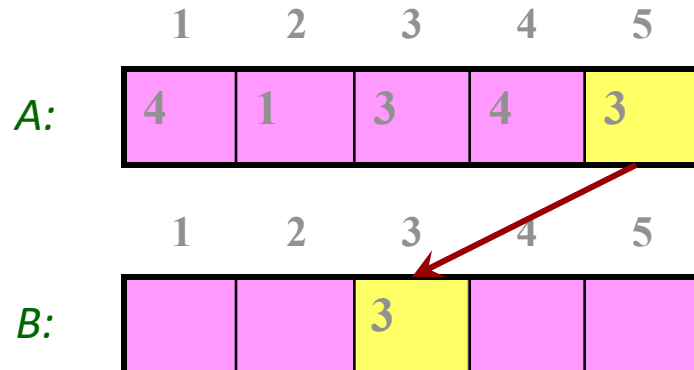
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

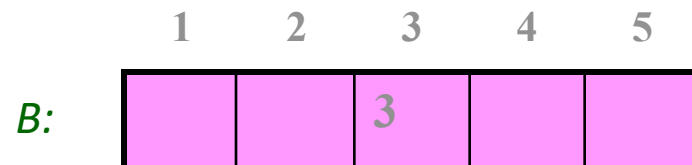
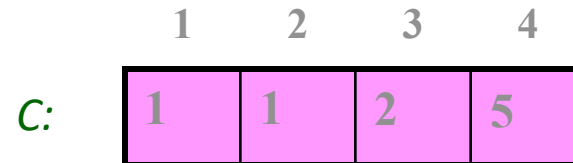
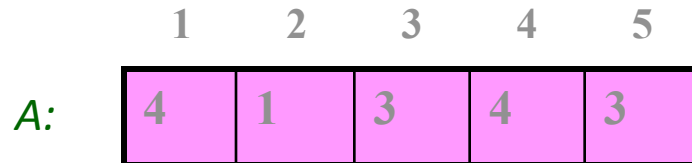
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

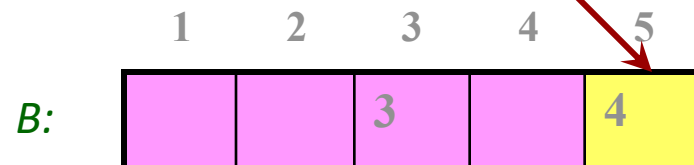
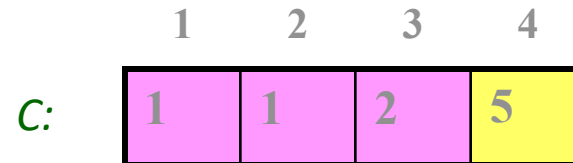
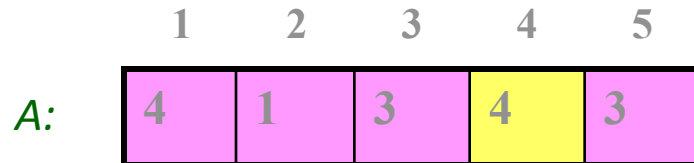
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

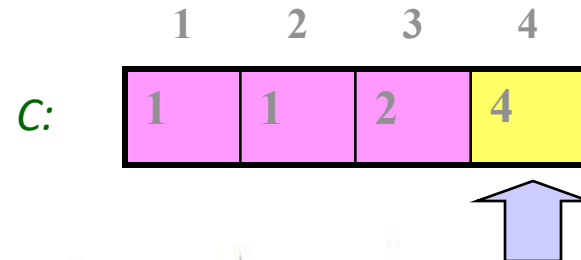
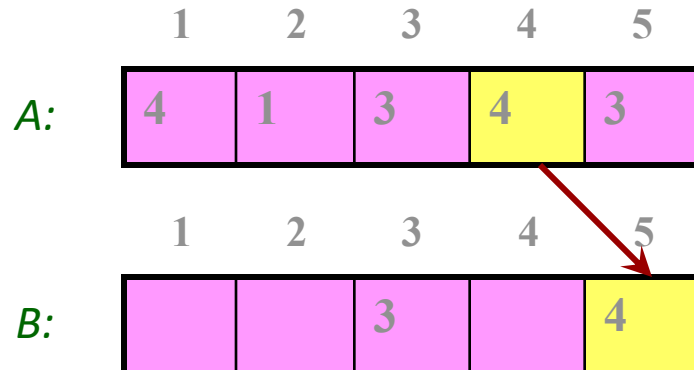
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

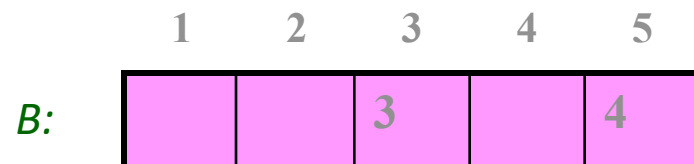
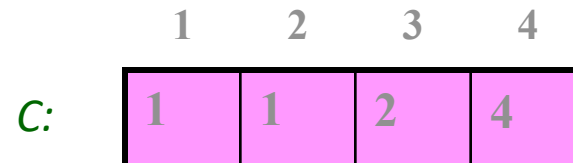
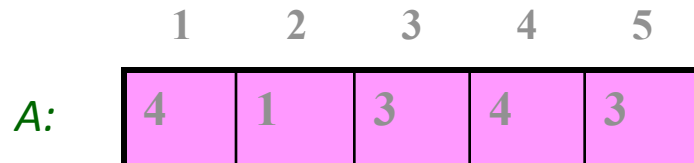
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

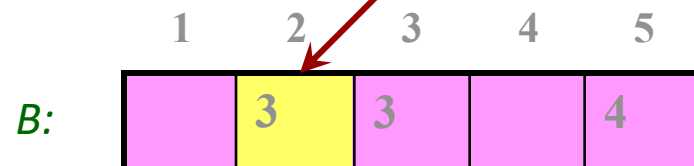
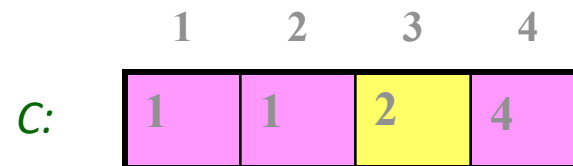
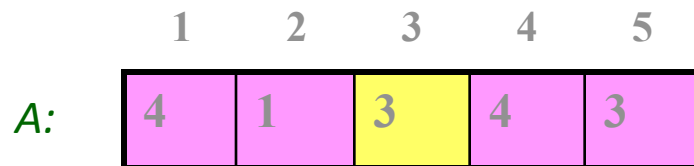
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

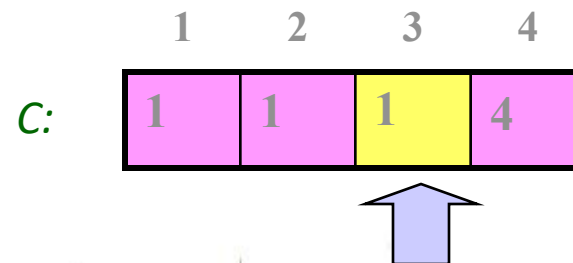
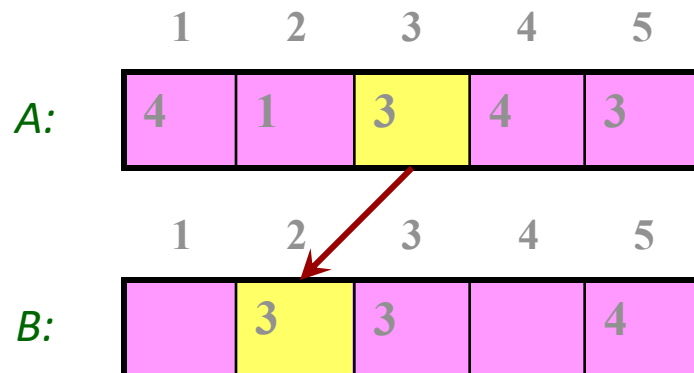
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

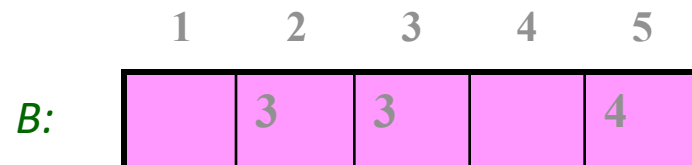
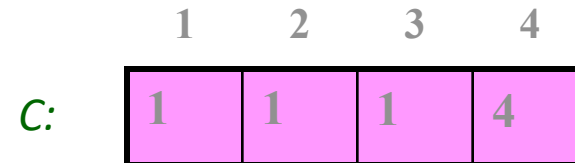
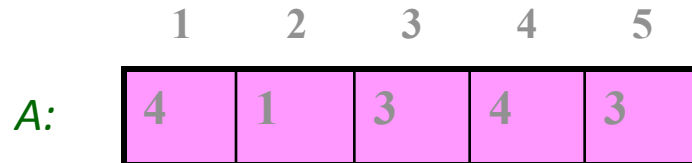
7 do C[i] ← C[i]+ C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

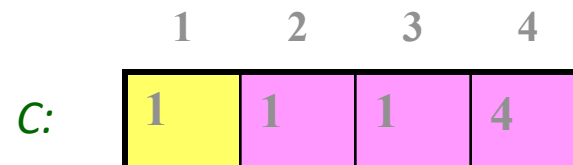
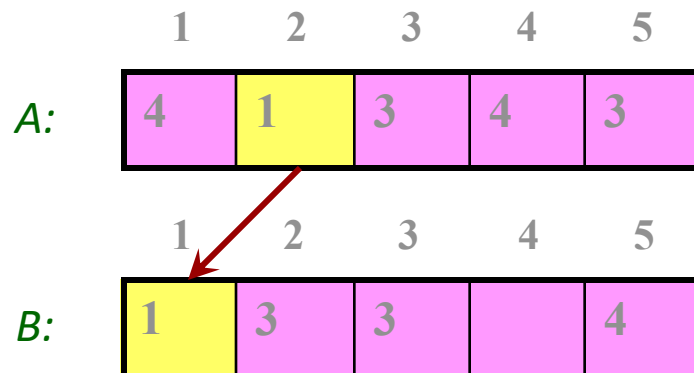
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

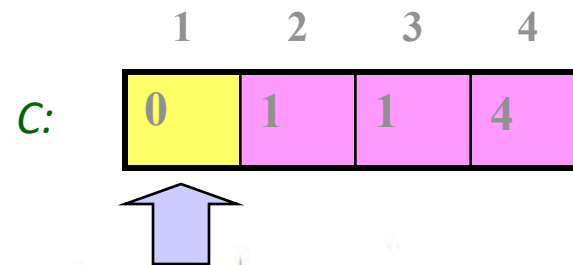
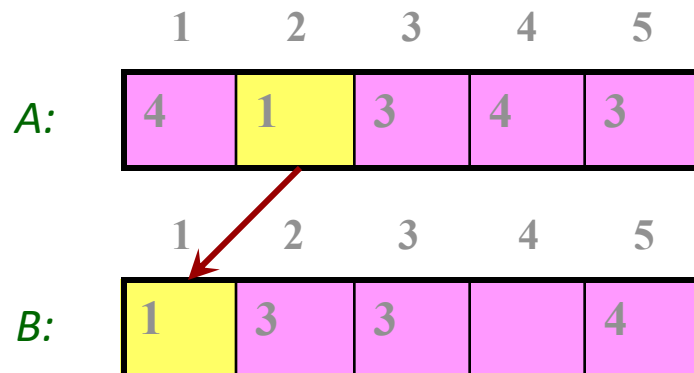
7 do C[i] ← C[i]+ C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

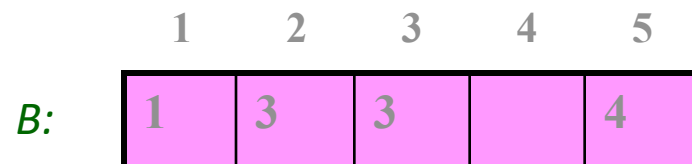
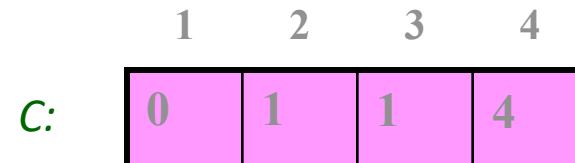
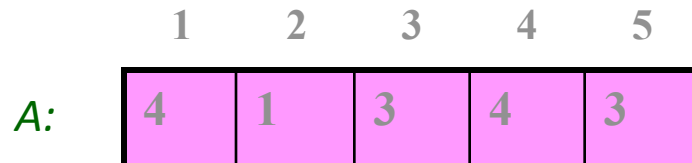
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

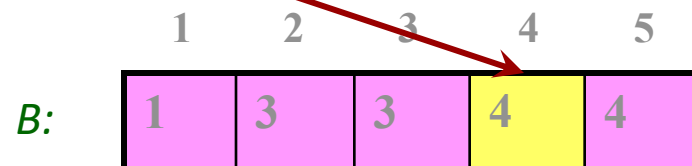
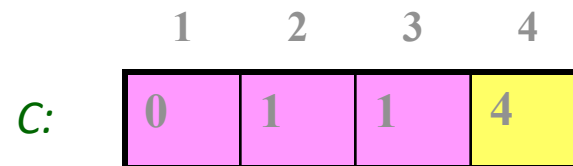
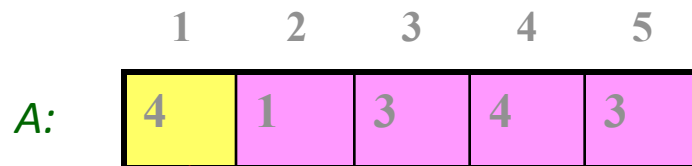
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

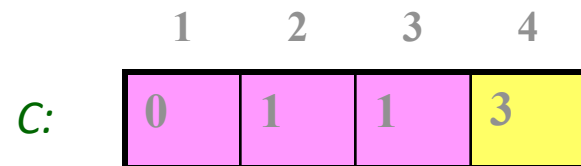
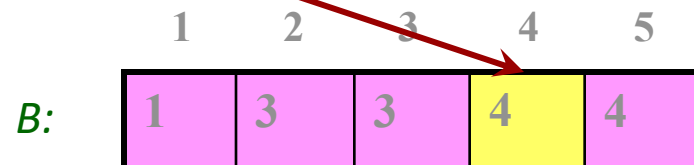
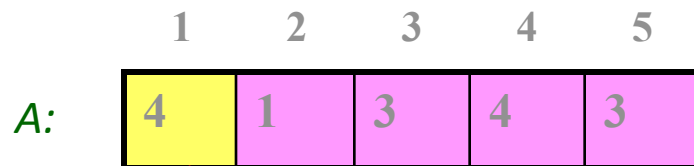
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Counting-sort example

COUNTING SORT

COUNTING-SORT (A, B, k)

1 for i ← 1 to k

2 do C[i] ← 0

3 for j ← 1 to length[A]

4 do C[A[j]] ← C[A[j]]+1

5 //C[i] now contains the number of elements equal to i.

6 for i ← 2 to k

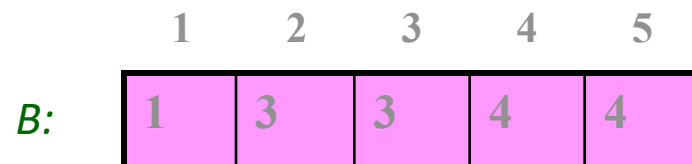
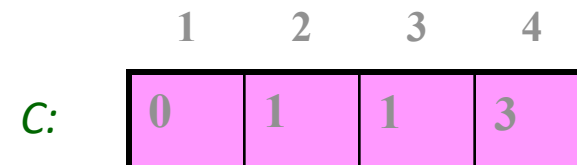
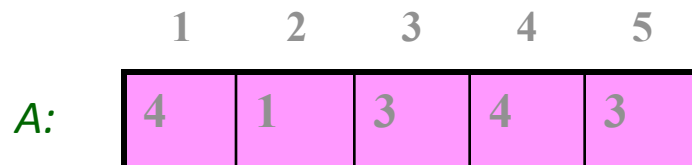
7 do C[i] ← C[i] + C[i-1]

8 //C[i] now contains the number of elements less than or equal to i.

9 for j ← length[A] downto 1

10 do B[C[A[j]]] ← A[j]

11 C[A[j]] ← C[A[j]]-1



Analyzing Counting Sort

COUNTING SORT

COUNTING-SORT (A, B, k)

```
1  for i ← 0 to k          }  $O(k)$ 
2      do C[i] ← 0
3  for j ← 1 to length[A]   }  $O(n)$ 
4      do C[A[j]] ← C[A[j]]+1
5  //C[i] now contains the number of elements equal to i.
6  for i ← 1 to k          }  $O(k)$ 
7      do C[i] ← C[i] + C[i-1]
8  //C[i] now contains the number of elements less than or equal to i.
9  for j ← length[A] downto 1 }  $O(n)$ 
10     do B[C[A[j]]] ← A[j]
11     C[A[j]] ← C[A[j]]-1
```

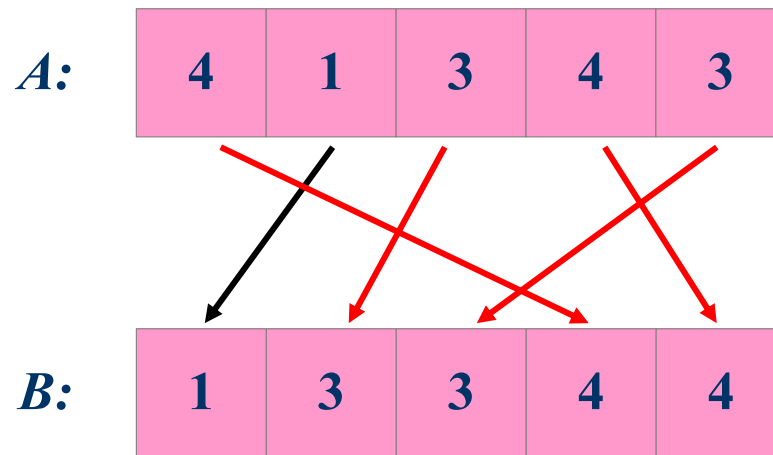
- The overall time is $O(k + n)$

Running time of Counting Sort

- If $k = O(n)$, then counting sort takes $O(n)$ time.
 - However, comparison sort take $\Omega(n \log n)$ time!
- What enables this linear implementation?
 - Counting sort is **not a comparison sort**
 - Or, single comparisons between elements never occurs

Stability of Counting Sort

- **Counting sort is stable:** it reserves the input order among equal elements.



- **Exercise:** what is the limitation of counting sort?

Applications

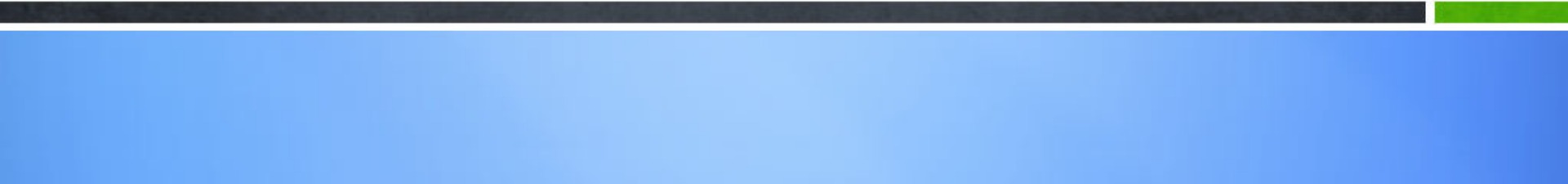
题目：2021年全国有1071万学生参加高考，根据考试成绩对学生进行排序。时间复杂度为 $O(n)$ ，可使用复杂度 $O(1)$ 的辅助空间。

分析：

- 如果使用插入、归并、冒泡、快速等比较排序法，时间复杂度为 $O(n^2)$ 或 $O(n \log(n))$ 。
- 题目要求按考试成绩排序，而考试成绩的值在 $[0, 750]$ 区间内。因此可以先统计每个分数有多少学生，有哪些学生，再按照分数从高到低依次排列全体学生（相同分数的按考号排列）。



11.2.2 Radix Sort



Radix Sort

- It was used by Herman Hollerith's card-sorting machine for the 1890 U.S. Census.
- Card sorters worked on one column at a time.
- It is the algorithm for using the machine that extends the technique to **multi-column sorting**.
- The human operator was part of the algorithm!
- **Key idea:** sort on the “**least significant digit**” first and on the remaining digits in sequential order. **The sorting method used to sort each digit must be “stable”.**
 - If we start with the “most significant digit”, we'll need extra storage.

An Example

Input		After sorting on LSD		After sorting on middle digit		After sorting on MSD
392		631		928		356
356		392		631		392
446		532		532		446
928	→	495	→	446	→	495
631		356		356		532
532		446		392		631
495		928		495		928

Digit-by-digit sort!

Radix-Sort(A, d)

RadixSort(A, d)

1. for $i \leftarrow 1$ to d
2. do *use a stable sort to sort array A on digit i*

Correctness of Radix Sort

By induction on the number of digits sorted.

- Assume that radix sort works for $d - 1$ digits.
- Show that it works for d digits.

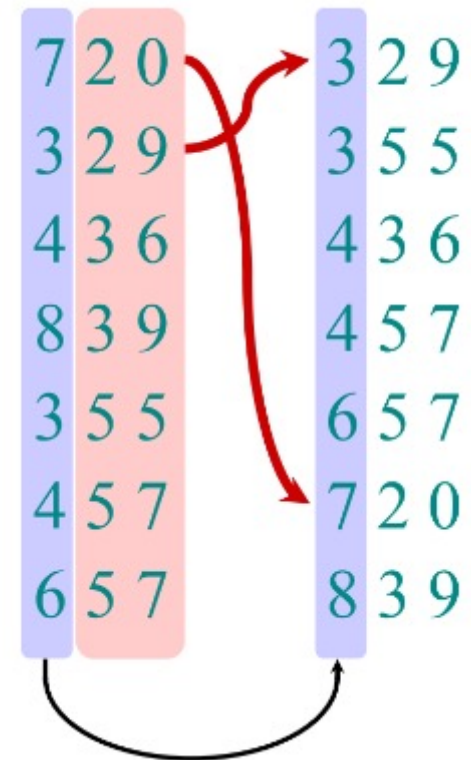
Radix sort of d digits \equiv radix sort of the low-order $d - 1$ digits followed by a sort on digit d .

Correctness of Radix Sort

- By induction hypothesis, the sort of the low-order $d - 1$ digits works, so the elements are **in order according to their low-order $d - 1$ digits**.
- **The sort on digit d will order the elements by their d^{th} digit.**

Consider two elements, a and b , with d^{th} digits a_d and b_d :

- If $a_d < b_d$, the sort will place a before b , since $a < b$ regardless of the low-order digits.
- If $a_d > b_d$, the sort will place a after b , since $a > b$ regardless of the low-order digits.

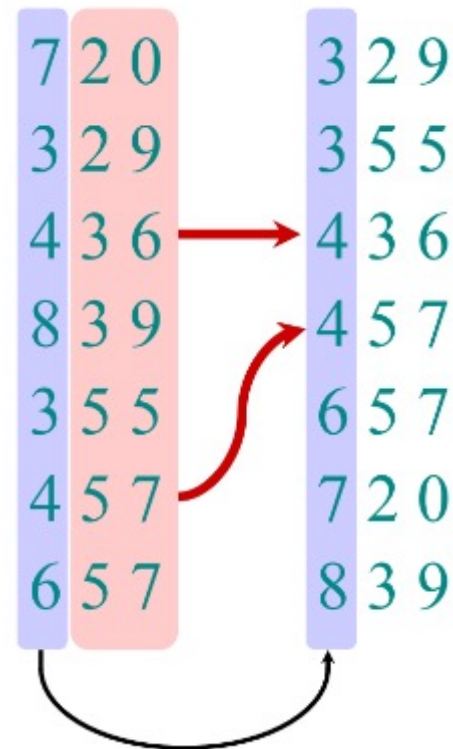


Correctness of Radix Sort

- By induction hypothesis, the sort of the low-order $d - 1$ digits works, so the elements are **in order according to their low-order $d - 1$ digits**.
- **The sort on digit d will order the elements by their d^{th} digit.**

Consider two elements, a and b , with d^{th} digits a_d and b_d :

- If $a_d = b_d$, the sort will **leave a and b in the same order**, since the sort is stable. But that order is already correct, since the correct order of is determined by the low-order digits when their d^{th} digits are equal.

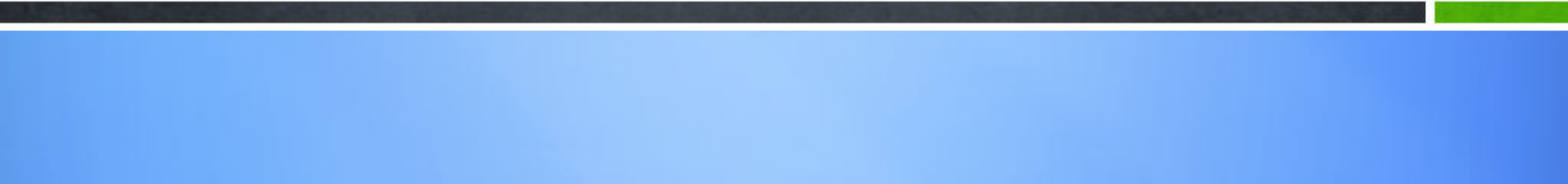


Algorithm Analysis

- Assume **counting sort** is used for each pass. Each pass over n d -digit numbers then takes time $\Theta(n+k)$.
- There are d passes, so the **total time for radix sort is $\Theta(d(n+k))$** .
- **When d is a constant and $k = O(n)$, radix sort runs in linear time.**
- Radix sort, if uses counting sort as the intermediate stable sort, **does not sort in place**.
 - If primary memory storage is an issue, quicksort or other sorting methods may be preferable.



11.2.3 Bucket Sort



Bucket Sort

- Assumes input is generated by a random process that distributes the elements uniformly over $[0, 1)$.
- Idea:
 - Divide $[0, 1)$ into n equal-sized buckets.
 - Distribute the n input values into the buckets.
 - Sort each bucket.
 - Then go through the buckets in order, listing elements in each one.

An Example

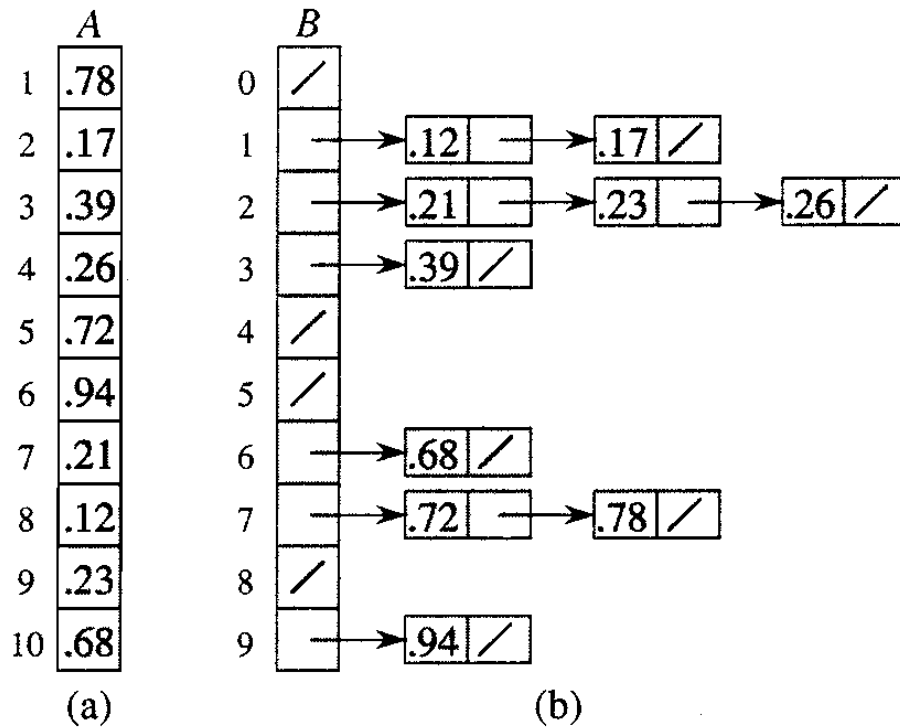


Figure 9.4 The operation of BUCKET-SORT. (a) The input array $A[1..10]$. (b) The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \dots, B[9]$.

Bucket-Sort (A)

Input: $A[1..n]$, where $0 \leq A[i] < 1$ for all i .

Auxiliary array: $B[0..n-1]$ of linked lists, each list initially empty.

BucketSort(A)

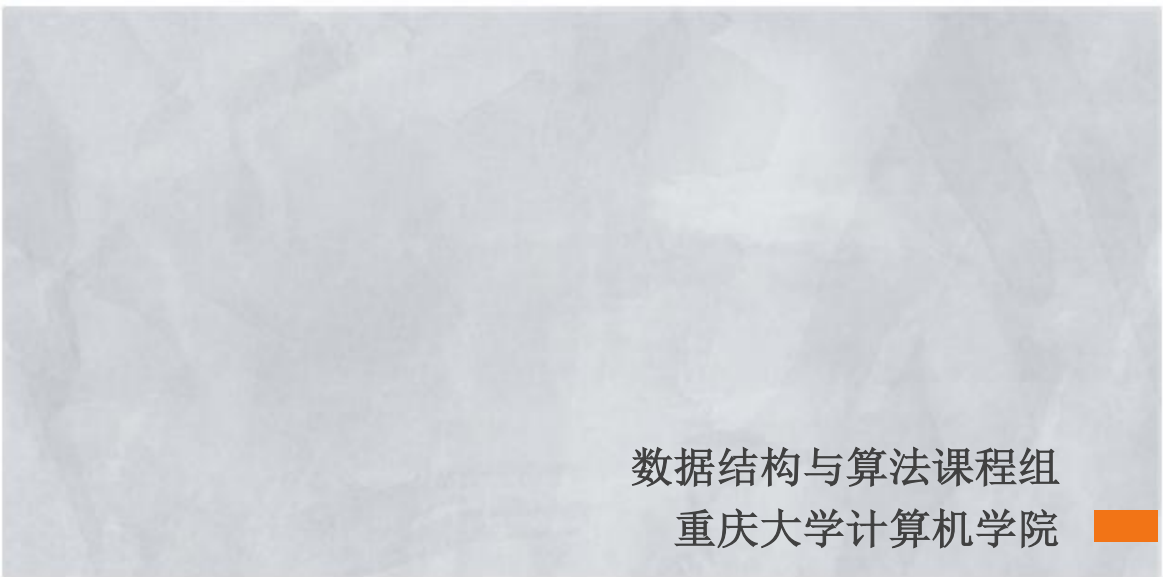

1. $n \leftarrow \text{length}[A]$
2. for $i \leftarrow 1$ to n
3. do insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. for $i \leftarrow 0$ to $n-1$
5. do sort list $B[i]$ with insertion sort
6. concatenate the lists $B[i]$ s together in order
7. return the concatenated lists

Correctness of BucketSort


- Consider $A[i], A[j]$. Assume w.o.l.o.g, $A[i] \leq A[j]$.
- Then, $\lfloor n \times A[i] \rfloor \leq \lfloor n \times A[j] \rfloor$.
- So, $A[i]$ is placed into the same bucket as $A[j]$ or into a bucket with a lower index.
 - If same bucket, insertion sort fixes up.
 - If earlier bucket, concatenation of lists fixes up.

Analysis

- Relies on no bucket getting too many values.
- All lines except **insertion sorting** in line 5 take $O(n)$ altogether.
- We “expect” each bucket to have few elements, since the average is 1 element per bucket.
- Intuitively, if each bucket gets **a constant number of elements**, it takes $O(1)$ time to sort each bucket \Rightarrow **$O(n)$ sort time for all buckets.**



数据结构与算法课程组
重庆大学计算机学院



End of Section.

