



T&R Team of Algorithm Design
College of Computer Science and Engineering, CQU



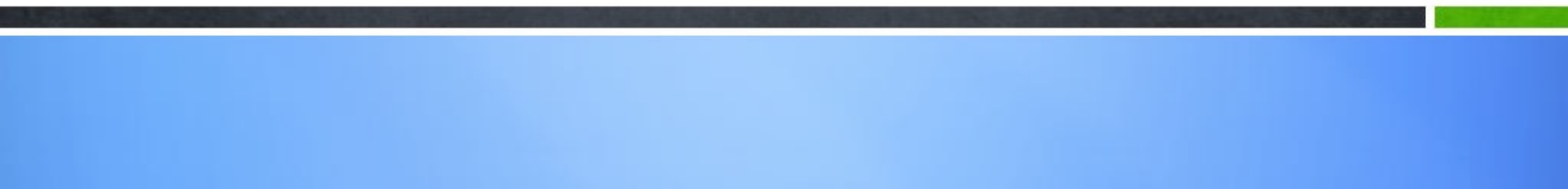
Algorithm Analysis & Design

Introduction to Algorithm





11.1 Shell Sort



Shellsort (diminishing increment sort)

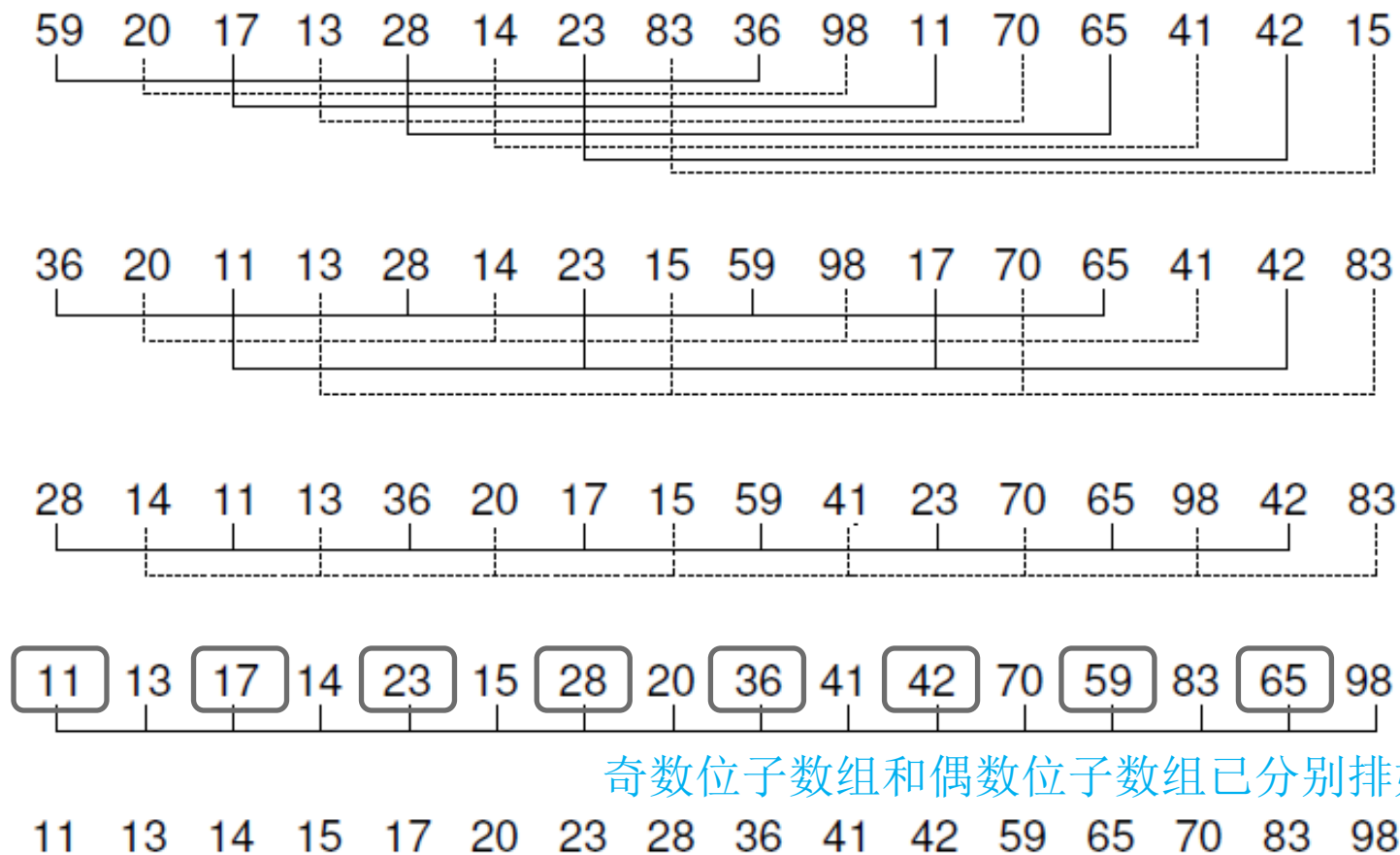
- 对几乎已经排好序的序列，插入排序效率高！接近线性排序的效率
- 一般情况下，插入排序效率低，因为每次只能交换相邻元素的位置

- 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

Shellsort: Complexity

- The average-case performance of Shellsort (for “divisions by three” increments) is $O(n^{1.5})$.

KEY THEOREM:

对数组 $a[1, \dots, n]$ 希尔排序:

(1) 先用间距(increment) h 插入排序, 使

$$\forall i \in [1, n-h]: a[i] < a[i+h];$$

(2) 再用间距 l 插入排序 (通常 $h > l$), 同样

$$\forall j \in [1, n-l]: a[j] < a[j+l].$$

可以证明第 (2) 次排序后, $\forall i \in [1, n-h]: a[i] < a[i+h]$ 成立!

设序列 $X = \langle x_1, x_2, \dots, x_i, \dots, x_n \rangle$ 和 $Y = \langle y_1, y_2, \dots, y_i, \dots, y_n \rangle$ 。

$$\forall i \in [1, n]: x_i \leq y_i$$

对 X 和 Y 分别升序排序可得 X' 和 Y' , 证明 $\forall i \in [1, n]: x'_i \leq y'_i$

Shellsort: Complexity

- The average-case performance of Shellsort (**for “divisions by three” increments**) is $O(n^{1.5})$.

KEY THEOREM:

对数组 $a[1, \dots, n]$ 希尔排序:

(1) 先用间距(increment) h 插入排序, 使

$$\forall i \in [1, n - h]: a[i] < a[i + h];$$

(2) 再用间距 l 插入排序 (通常 $h > l$), 同样

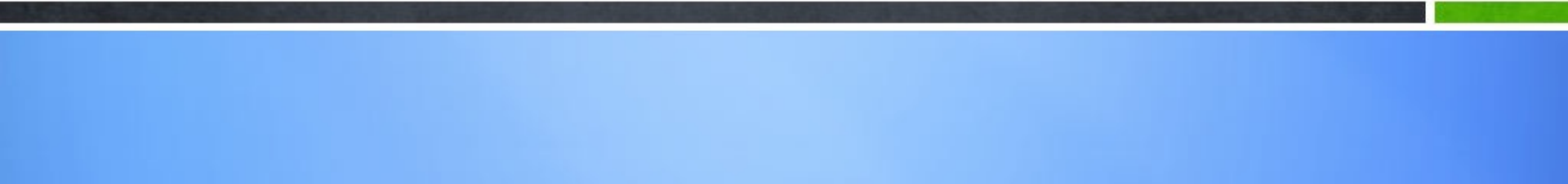
$$\forall j \in [1, n - l]: a[j] < a[j + l].$$

可以证明第 (2) 次排序后, $\forall i \in [1, n - h]: a[i] < a[i + h]$ 成立!

- 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 $O(n^{1.5})$



Linear-time Sort

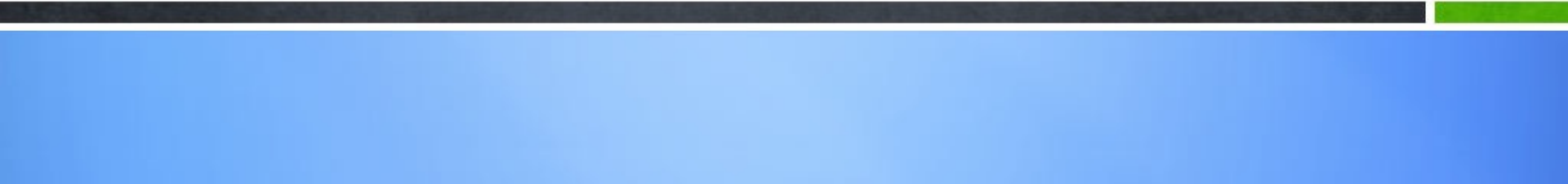


Outline

- **1 How fast can we sort?**
- **2 Counting Sort**
 - > **Application: Longest zero-sum subarray**
- **3 Radix Sort**
 - > **Application: Suffix array**
- **4 Bucket Sort**



1. How fast can we sort?

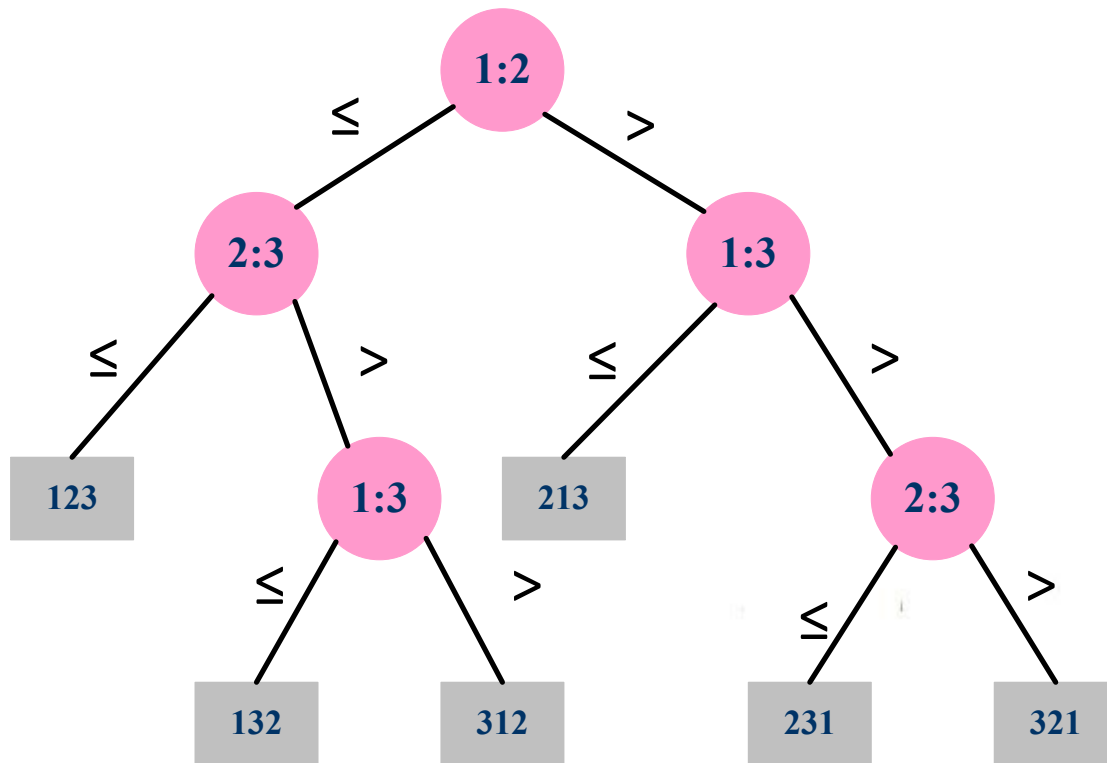


How fast can we sort?

- 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.

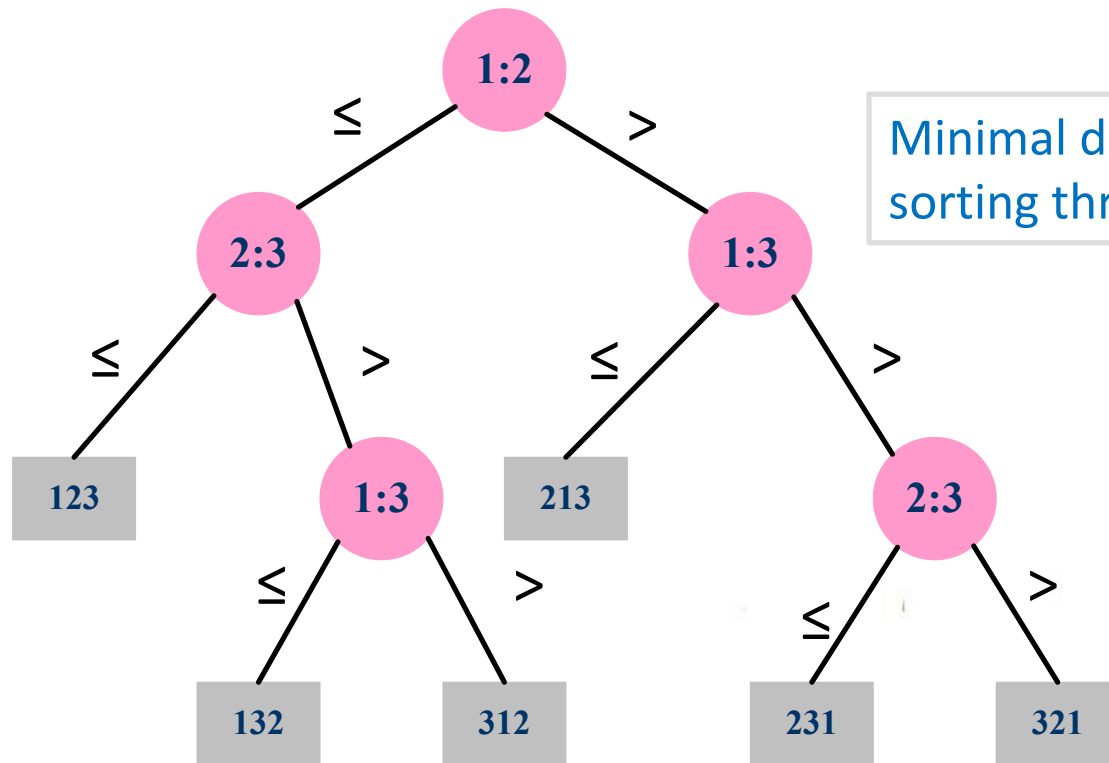
Decision-tree example

- 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

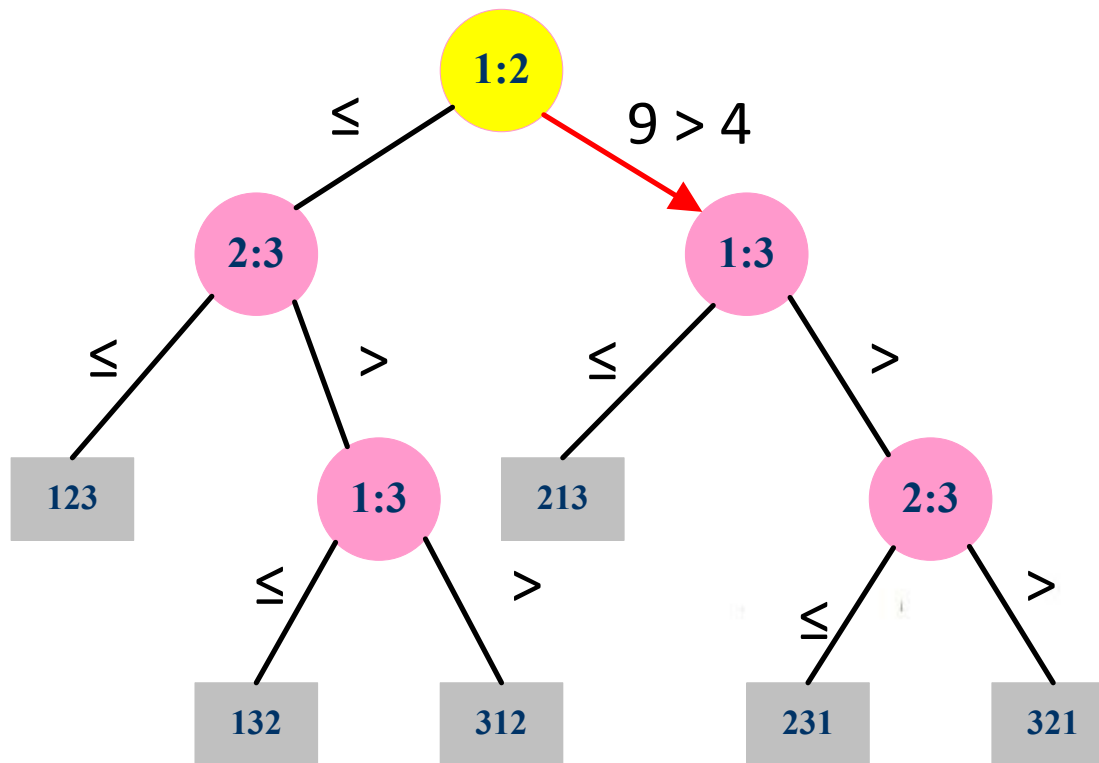
- Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$
 - Each internal node is labeled $i:j$ for $i, j \in \{1, 2, 3\}$
 - The left sub-tree shows subsequent comparisons with $a_i \leq a_j$
 - The right sub-tree show subsequent comparisons with $a_i > a_j$
 - **Each Leaf denotes a permutation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



Minimal decision-tree for
sorting three elements!

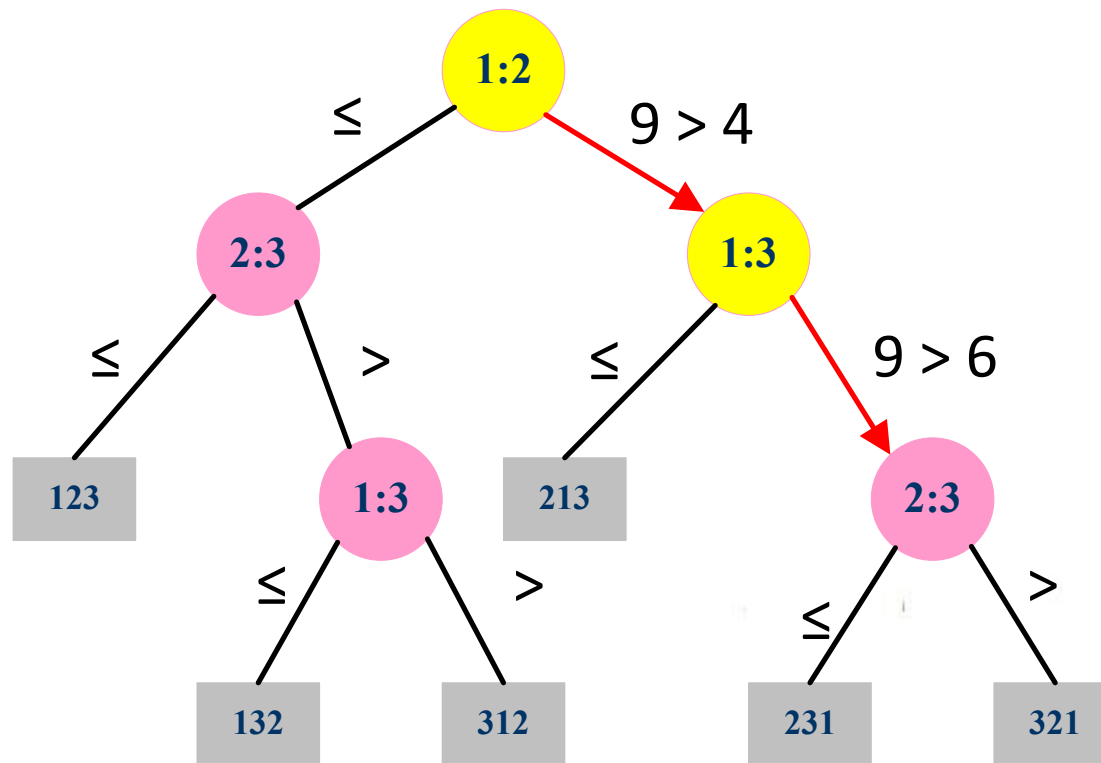
Decision-tree example

- 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



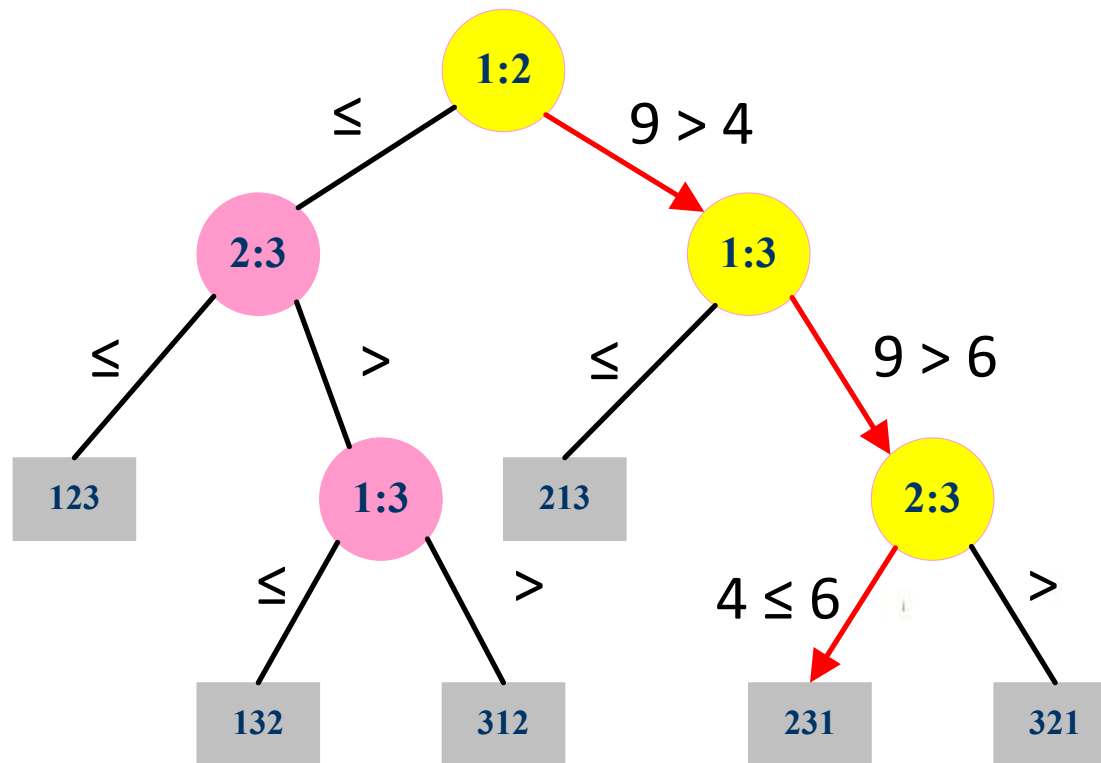
Decision-tree example

- 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



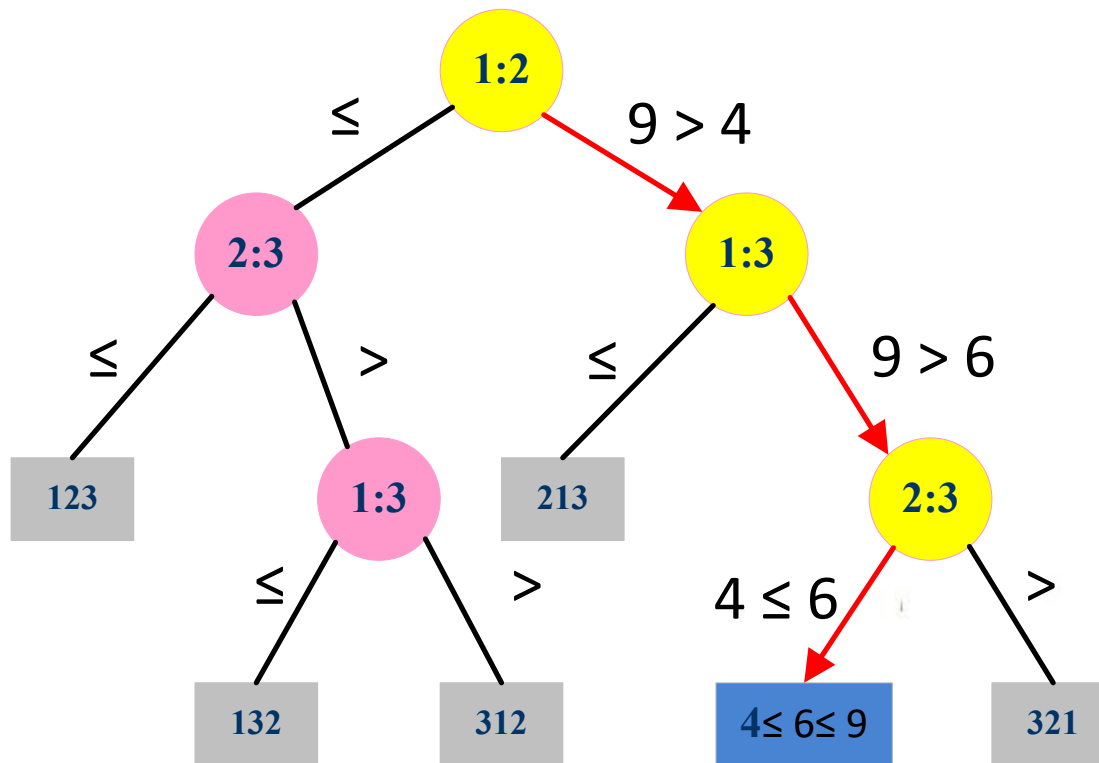
Decision-tree example

- 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



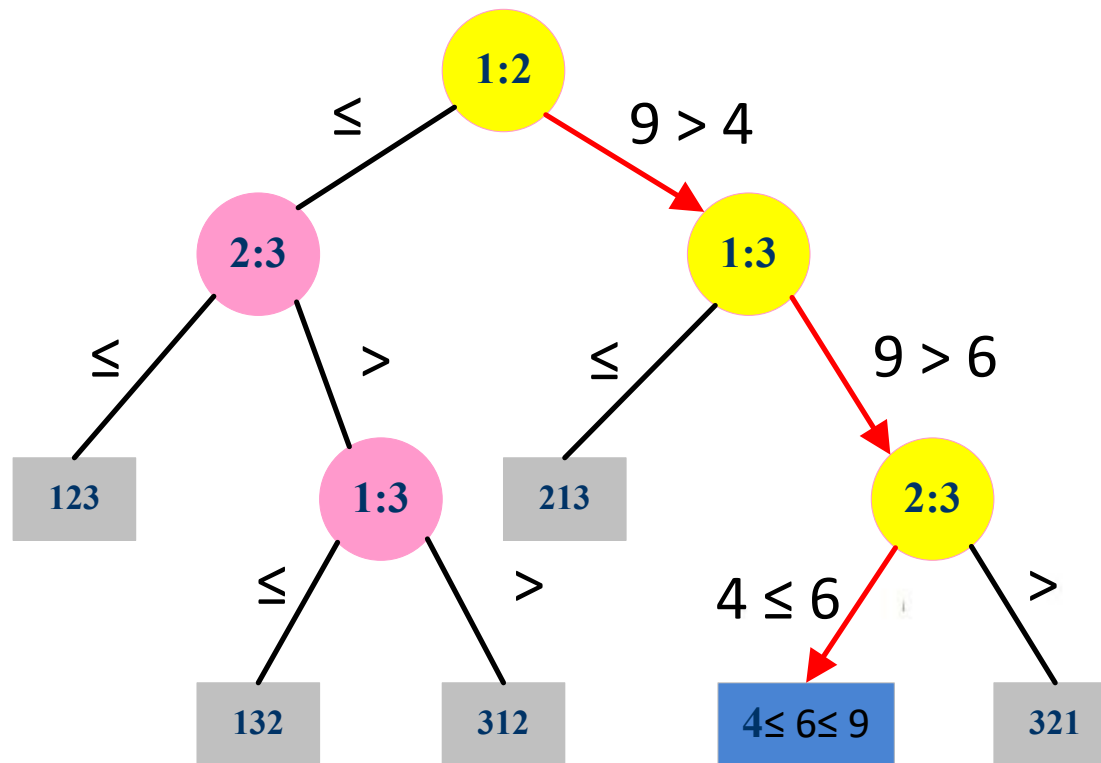
Decision-tree example

- 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:

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

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

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

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

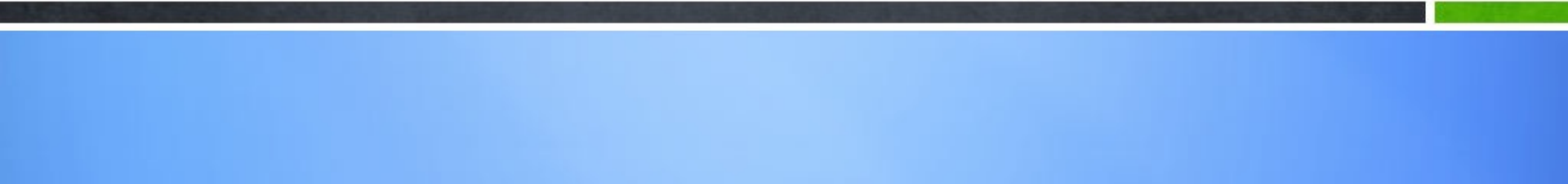
Heapsort and merge sort are asymptotically optimal comparison sorting algorithms.

Can We Sort Faster?

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



2 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]

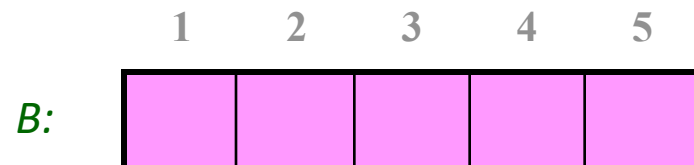
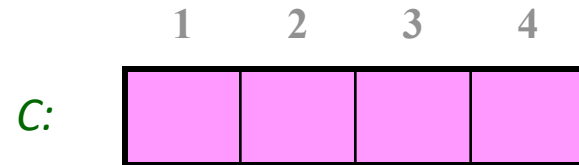
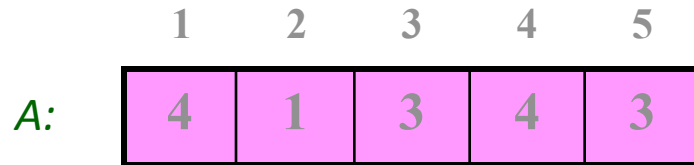
11 C[A[j]] ← C[A[j]]-1

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
```



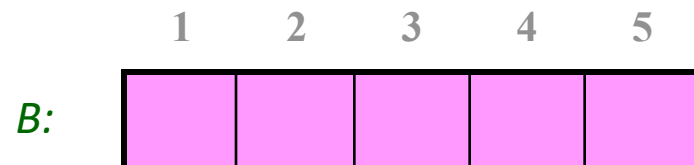
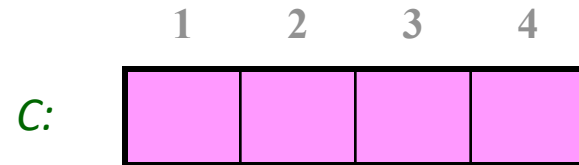
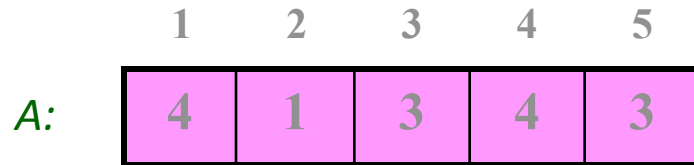
n=5, k=4

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
```

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
0	0	0	0

B:

1	2	3	4	5

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

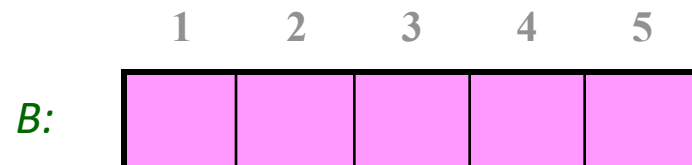
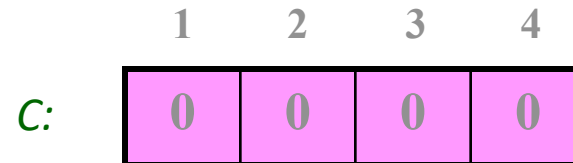
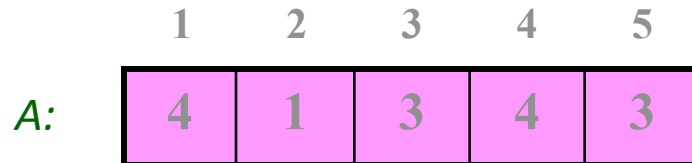
7 do C[i] ← C[i] + C[i-1]

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

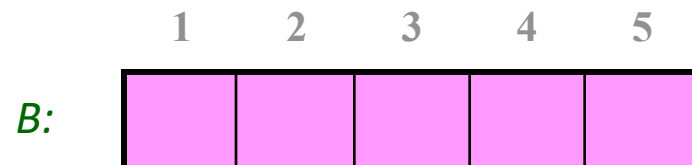
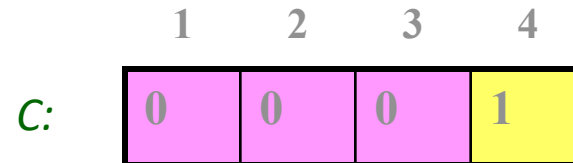
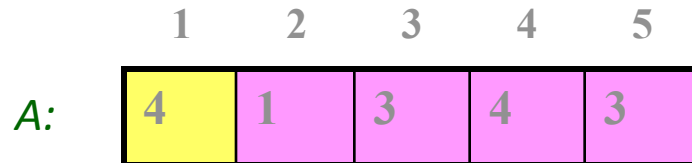
7 do C[i] ← C[i]+ C[i-1]

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

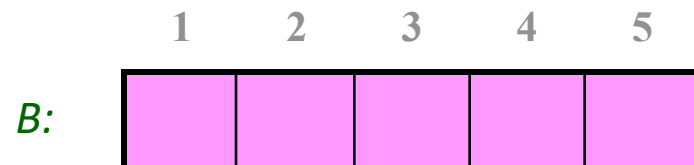
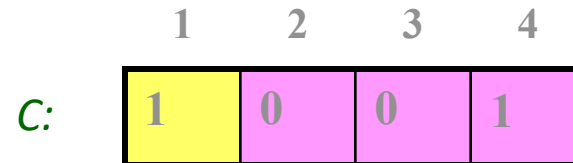
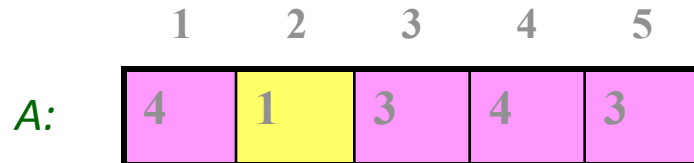
7 do C[i] ← C[i]+ C[i-1]

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

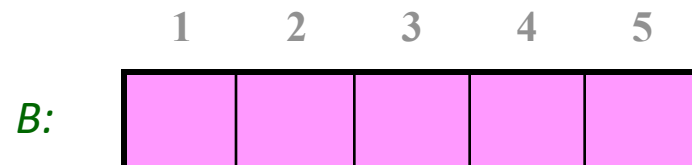
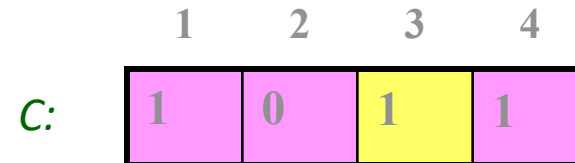
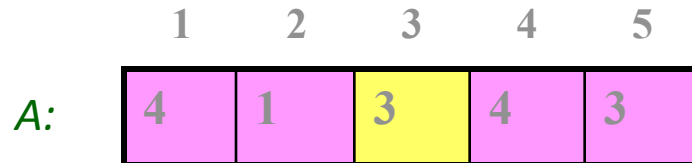
7 do C[i] ← C[i] + C[i-1]

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

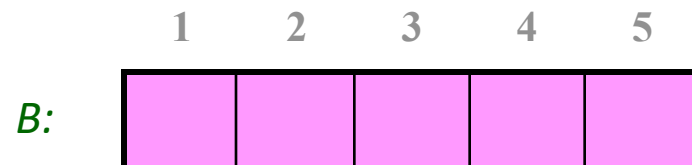
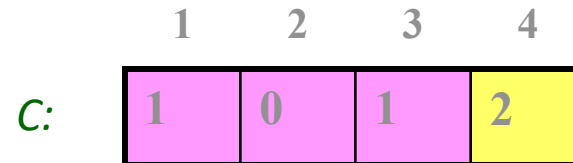
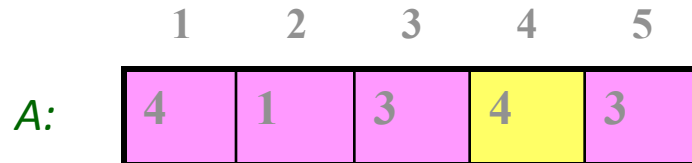
7 do C[i] ← C[i]+ C[i-1]

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

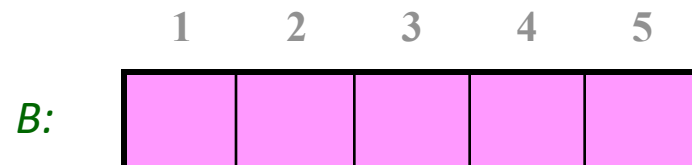
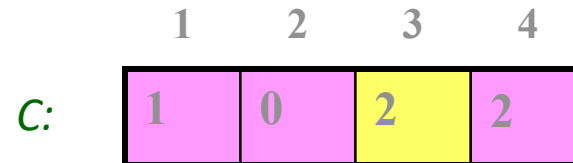
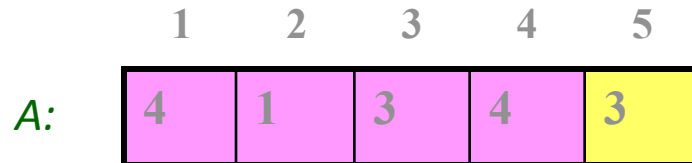
7 do C[i] ← C[i]+ C[i-1]

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

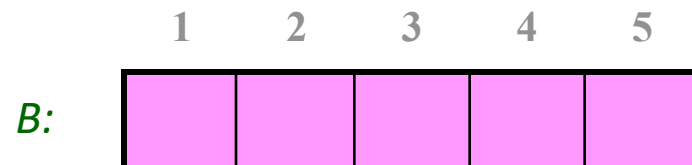
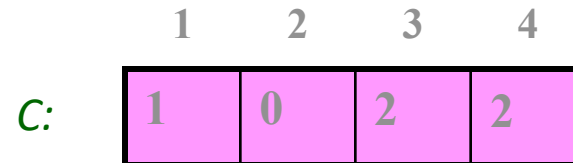
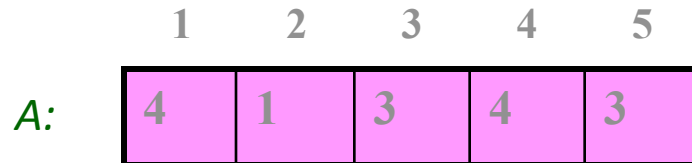
7 do C[i] ← C[i]+ C[i-1]

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

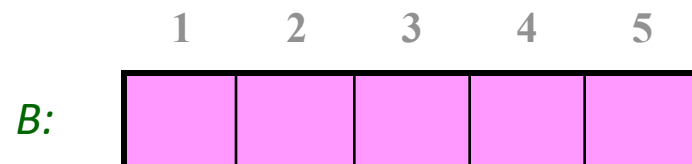
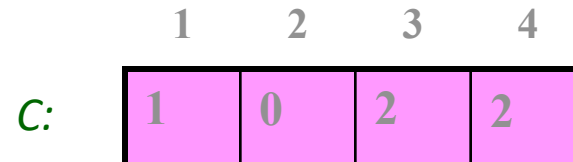
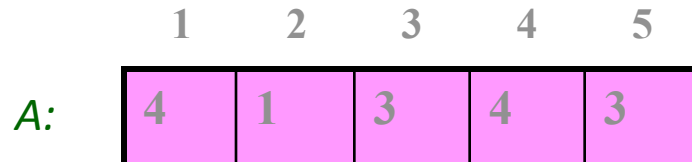
7 do C[i] ← C[i] + C[i-1]

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

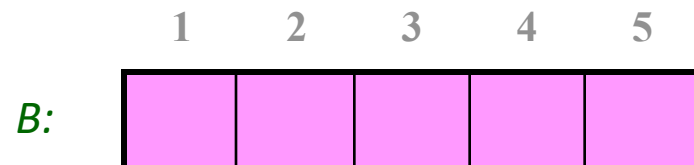
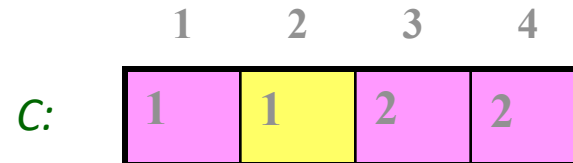
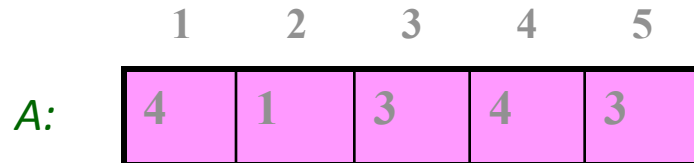
7 do C[i] ← C[i] + C[i-1]

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

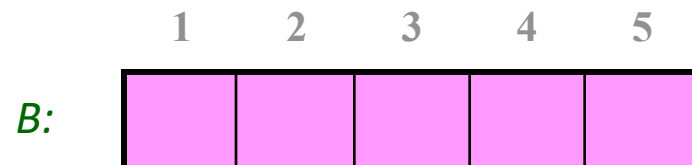
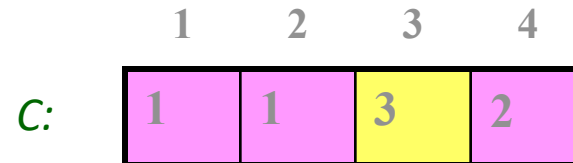
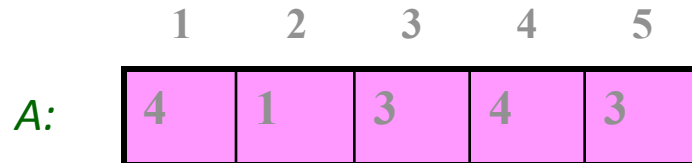
7 do C[i] ← C[i] + C[i-1]

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

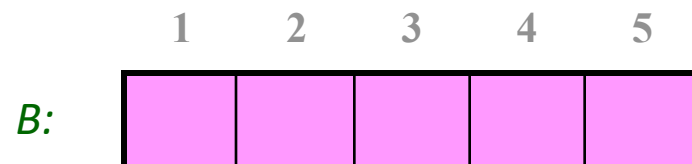
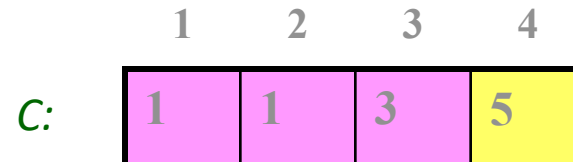
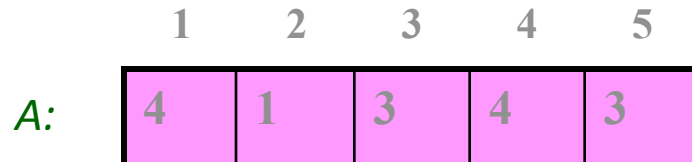
7 do C[i] ← C[i] + C[i-1]

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

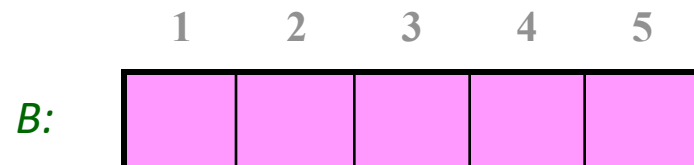
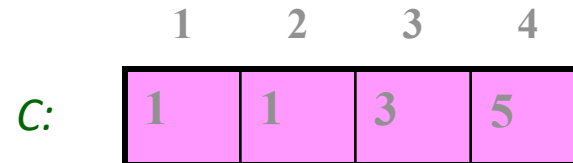
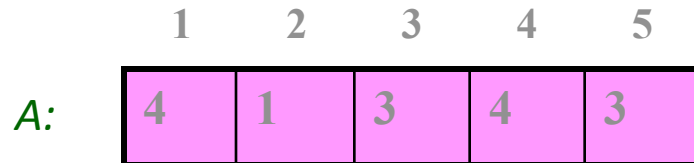
7 do C[i] ← C[i]+ C[i-1]

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

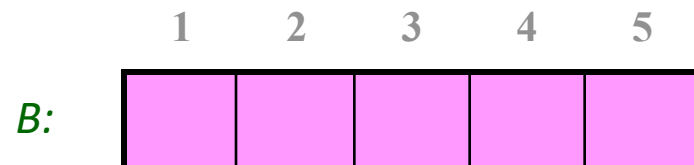
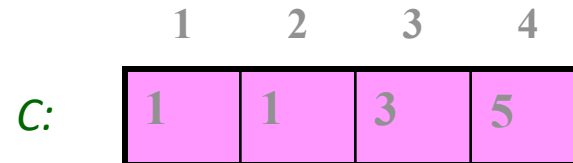
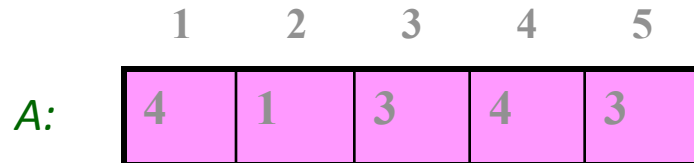
7 do C[i] ← C[i] + C[i-1]

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

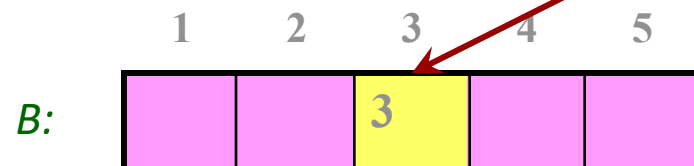
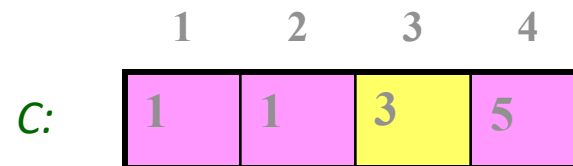
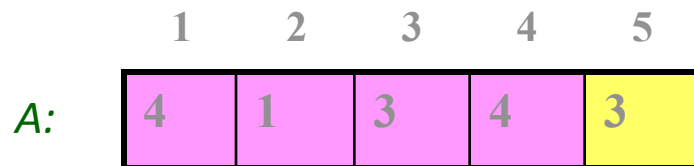
7 do C[i] ← C[i] + C[i-1]

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

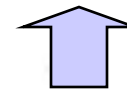
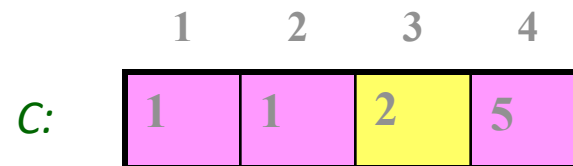
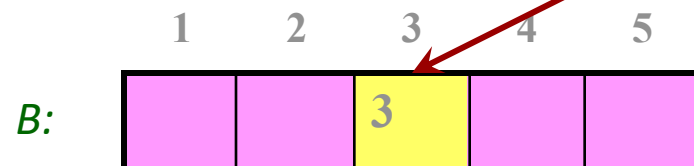
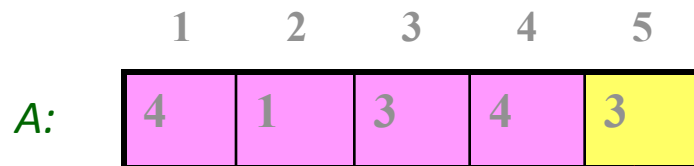
7 do C[i] ← C[i] + C[i-1]

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

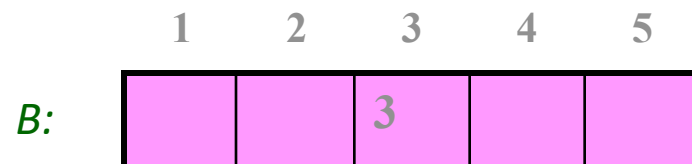
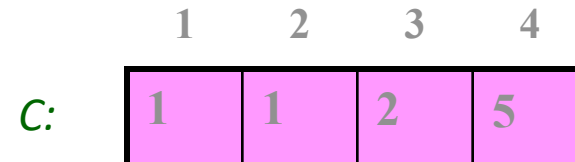
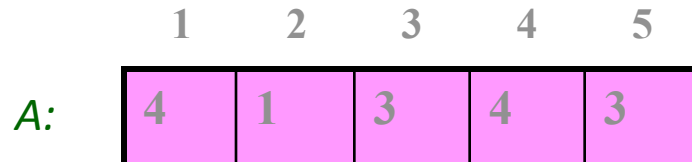
7 do C[i] ← C[i] + C[i-1]

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

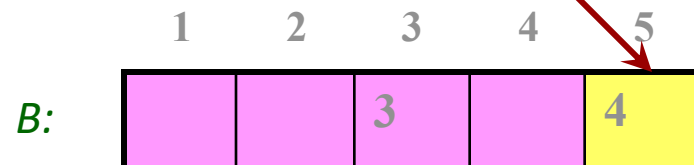
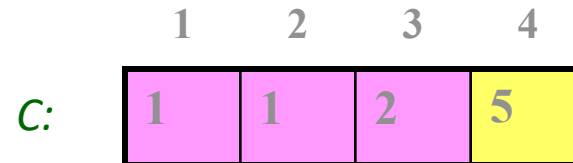
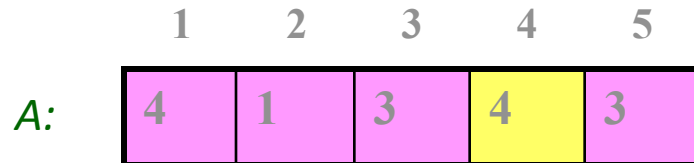
7 do C[i] ← C[i]+ C[i-1]

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

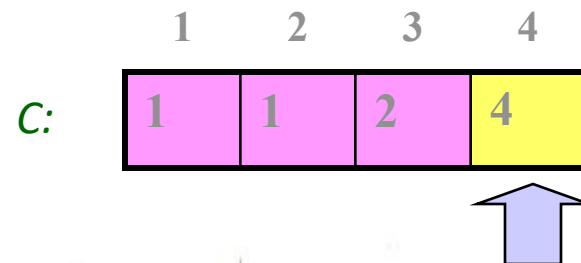
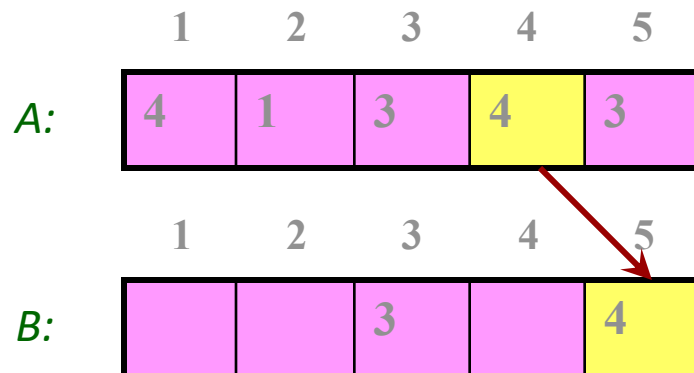
7 do C[i] ← C[i] + C[i-1]

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

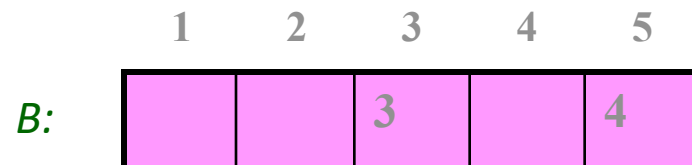
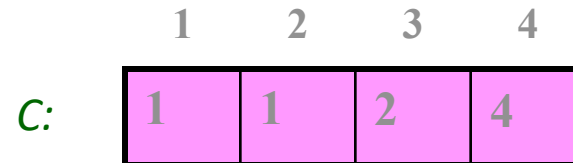
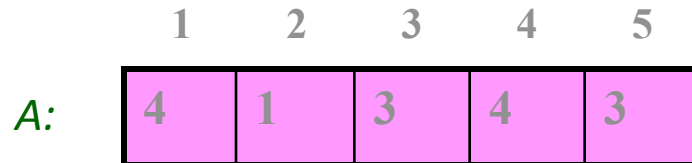
7 do C[i] ← C[i] + C[i-1]

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

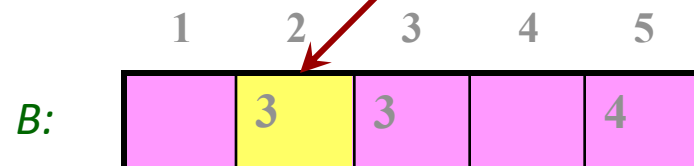
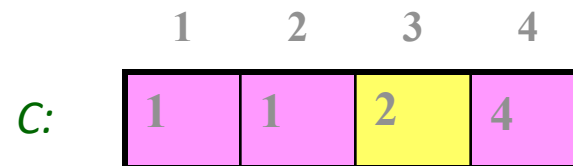
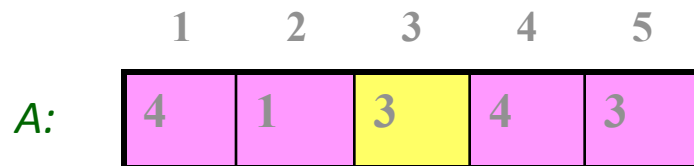
7 do C[i] ← C[i] + C[i-1]

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

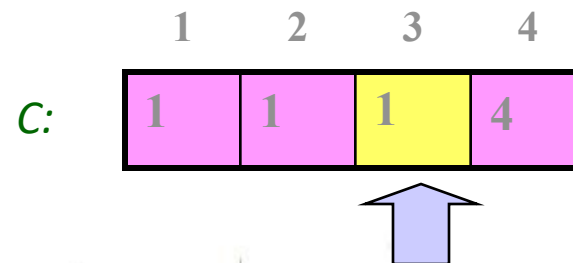
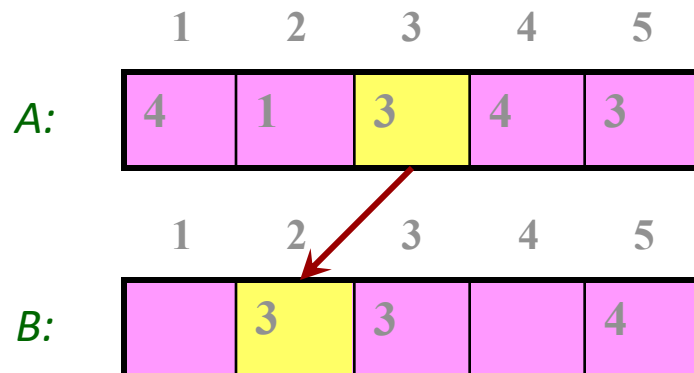
7 do C[i] ← C[i] + C[i-1]

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

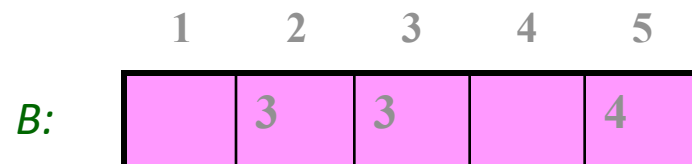
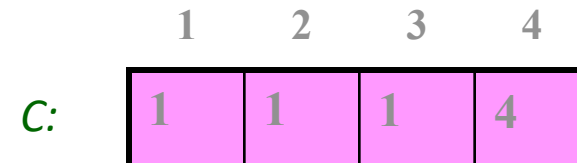
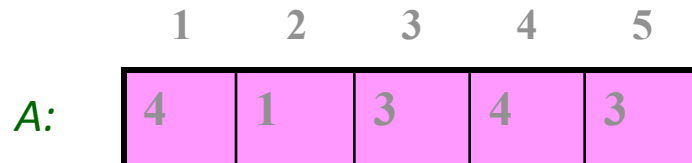
7 do C[i] ← C[i]+ C[i-1]

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

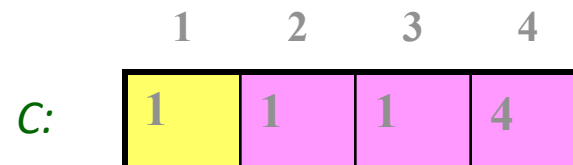
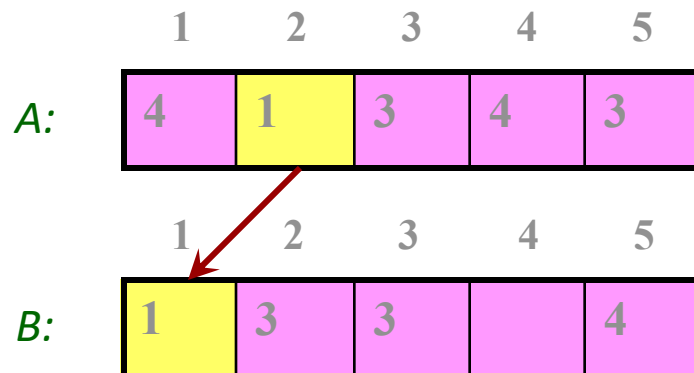
7 do C[i] ← C[i]+ C[i-1]

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

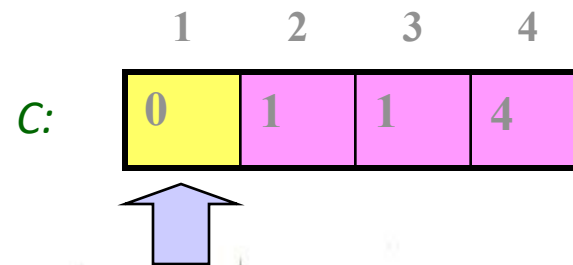
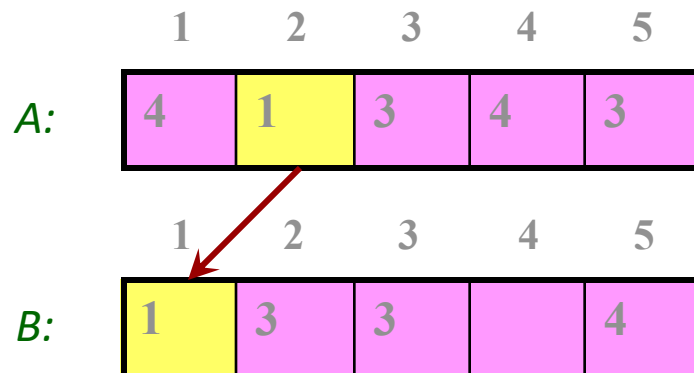
7 do C[i] ← C[i] + C[i-1]

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

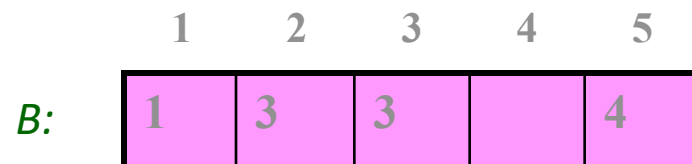
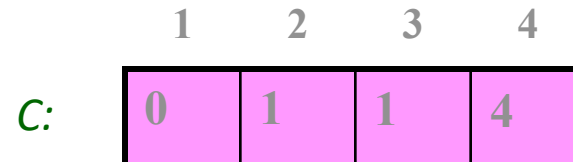
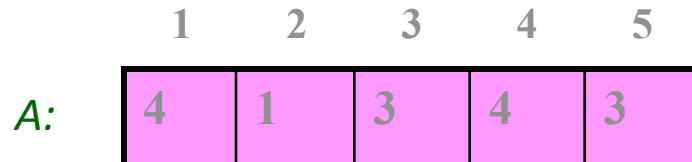
7 do C[i] ← C[i] + C[i-1]

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

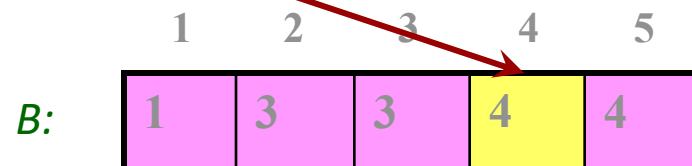
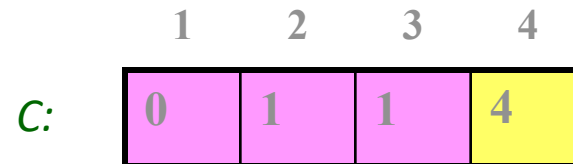
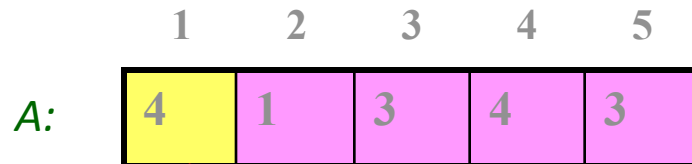
7 do C[i] ← C[i] + C[i-1]

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

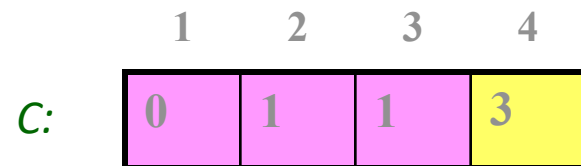
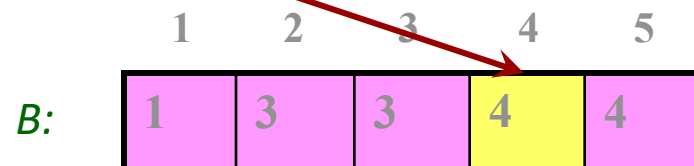
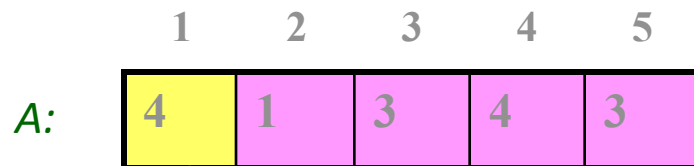
7 do C[i] ← C[i]+ C[i-1]

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

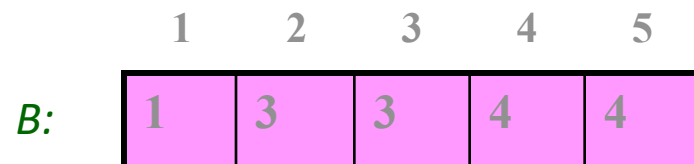
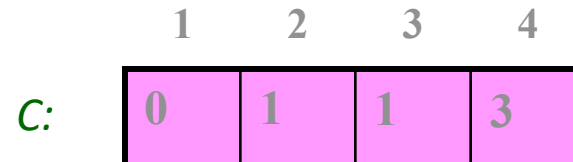
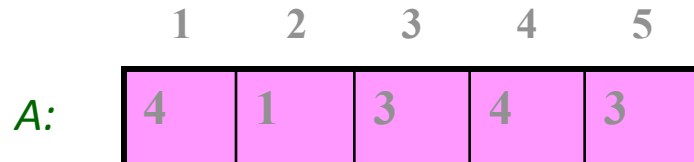
7 do C[i] ← C[i]+ C[i-1]

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
```

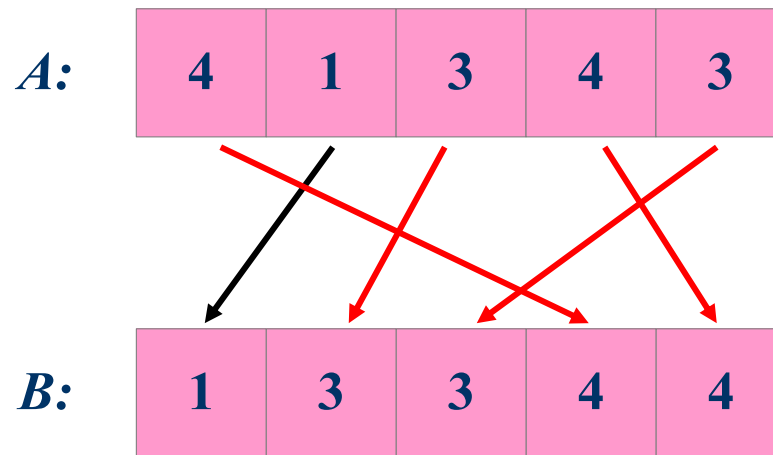
- The overall time is $O(k + n)$

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）。



Longest Zero-Sum Sub Array

Application of counting

Longest Zero-Sum Sub Array (和为零的最长子数组)

- 数组: 5, -1, -2, 3, 4, 0, -6, 1, 5, 1, -3, 1, 2, 1

和为零的子数组:

0

-1, -2, 3

-3, 1, 2

-6, 1, 5

0, -6, 1, 5

....

和为零的最长子数组:

-2, 3, 4, 0, -6, 1

Longest Zero-Sum Sub Array （和为零的最长子数组）

- Algorithm 1: Full search

LongestZeroSum(A[1..n])

```
L ← 0
for i ← 1 to n do      //子数组开始位置
    sum ← 0;
    for j ← i to n do  //结束位置
        sum ← sum + A[j] //递进求和
        if sum = 0 and L < j - i + 1, //比较最大值
            L = j - i + 1
return L
```

Time complexity: $\theta(n^2)$

Longest Zero-Sum Sub Array （和为零的最长子数组）

- Algorithm 2: Divide and Conquer

LongestZeroSum(A, l, r)

```
if l = r           //长度1
    if A[l] = 0,
        return 1
    else
        return 0
```

```
p ← (l+r)/2      //在中间位置分开
```

```
L ← LongestZeroSum(A, l, p)
R ← LongestZeroSum(A, p+1, r)
```

```
M ← Middle(A, l, p, r)
    // longest zero-sum sub-array containing A[p] and A[p+1].
```

```
return max(L, R, M)
```

Longest Zero-Sum Sub Array (和为零的最长子数组)

Algorithm 2: Divide and Conquer

$$\begin{array}{ccc} & \text{L} = 3 & \\ & \hline 5, -1, -2, 3, 4, 0, -6 & || & 1, 5, 1, -3, 1, 2, 1 \\ & & \hline \text{LongestZeroSum}(A, 1, 7) = 3 & & \text{LongestZeroSum}(A, 8, 14) = 3 \end{array}$$

Longest Zero-Sum Sub Array (和为零的最长子数组)

Algorithm 2: Divide and Conquer

$L = 3$ $R = 3$
5, -1, -2, 3, 4, 0, -6 || 1, 5, 1, -3, 1, 2, 1

$\text{Middle}(A, 1, 7, 14) = 6$

How to compute **Middle**?

Longest Zero-Sum Sub Array (和为零的最长子数组)

Algorithm 2: Divide and Conquer

$$\begin{array}{c} L = 3 \qquad \qquad \qquad R = 3 \\ \hline 5, -1, -2, 3, 4, 0, -6 \quad || \quad 1, 5, 1, -3, 1, 2, 1 \\ \hline \hline \hline \text{Middle}(A, 1, 7, 14) = 6 \end{array}$$

$$LS[1] = -6 \quad (-6)$$

$$LS[2] = -6 \quad (0 + -6)$$

$$LS[3] = -2 \quad (4 + 0 + -6)$$

$$LS[4] = 1 \quad (3 + 4 + 0 + -6)$$

$$LS[5] = -1 \quad (-2 + 3 + 4 + 0 + -6)$$

$$LS[6] = -2 \quad (-1 + -2 + 3 + 4 + 0 + -6)$$

$$LS[7] = 3 \quad (5 + -1 + -2 + 3 + 4 + 0 + -6)$$

$$RS[1] = 1 \quad (1)$$

$$RS[2] = 6 \quad (1 + 5)$$

$$RS[3] = 7 \quad (1 + 5 + 1)$$

$$RS[4] = 4 \quad (1 + 5 + 1 + -3)$$

$$RS[5] = 5 \quad (1 + 5 + 1 + -3 + 1)$$

$$RS[6] = 7 \quad (1 + 5 + 1 + -3 + 1 + 2)$$

$$RS[7] = 8 \quad (1 + 5 + 1 + -3 + 1 + 2 + 1)$$

$$LS[1] + RS[2] = 0; \quad LS[2] + RS[2] = 0; \quad LS[5] + RS[1] = 0$$

$$\text{Time complexity: } T(n) = 2T\left(\frac{n}{2}\right) + \theta(n^2)$$

Longest Zero-Sum Sub Array (和为零的最长子数组)

Algorithm 2: Divide and Conquer (revised)

$L = 3$

$R = 3$

$5, -1, -2, 3, 4, \underline{0}, \underline{-6} \parallel 1, 5, 1, -3, 1, 2, 1$

Middle(A, 1, 7, 14) = 6

```
LS[1] = -6
LS[2] = -6
LS[3] = -2
LS[4] = 1
LS[5] = -1
LS[6] = -2
LS[7] = 3
```

```
-RS[1] = -1
-RS[2] = -6
-RS[3] = -7
-RS[4] = -4
-RS[5] = -5
-RS[6] = -7
-RS[7] = -8
```

Middle(A, l, p, r)

1. Compute $LS[1..p]$, $RS[1..(r-p)]$
2. $B[P..Q] \leftarrow \{0, \dots, 0\}$ // $P = \min\{LS[1..p], RS[1..(r-p)]\}$
// $Q = \max\{LS[1..p], RS[1..(r-p)]\}$
3. for $i \leftarrow 1$ to p do $B[LS[i]] = i$

```
// if LS[i]=LS[j]&& i<j, then B[LS[j]]=j
```

- ```

4. $M \leftarrow 0$
5. for $j \leftarrow 1$ to $r-p$ do
 if $B[-RS[j]] > 0$ and $M < B[-RS[j]] + j$
 $M \leftarrow B[-RS[j]] + j$
6. return M

```

## Longest Zero-Sum Sub Array (和为零的最长子数组)

## Algorithm 2: Divide and Conquer (revised)

$$\begin{array}{cccccccc}
 & & \text{L} = 3 & & & & & \text{R} = 3 \\
 & \text{---} & & & & & & \text{---} \\
 5, & -1, & -2, & 3, & 4, & 0, & -6 & || & 1, & 5, & 1, & -3, & 1, & 2, & 1 \\
 & & & & & \text{---} & & & & & & & & & \\
 & & & & & \text{---} & & & & & & & & & \\
 & \text{---} & & & & & & & \text{---} & & & & & & \\
 & & & & & & \text{Middle(A, 1, 7, 14) = 6} & & & & & & & & 
 \end{array}$$

```
LS[1] = -6
LS[2] = -6
LS[3] = -2
LS[4] = 1
LS[5] = -1
LS[6] = -2
LS[7] = 3
```

```
-RS[1] = -1
-RS[2] = -6
-RS[3] = -7
-RS[4] = -4
-RS[5] = -5
-RS[6] = -7
-RS[7] = -8
```

Middle(A, l, p, r)

1. Compute  $LS[1..p]$ ,  $RS[1..(r-p)]$
2.  $B[P..Q] \leftarrow \{0, \dots, 0\}$
3. for  $i \leftarrow 1$  to  $p$  do  $B[LS[i]] = i$

4.  $M \leftarrow 0$

- ```

5. for j ← 1 to r-p do
    if  $B[-RS[j]] > 0$  and  $M < B[-RS[j]] + j$ 
         $M \leftarrow B[-RS[j]] + j$ 

```

6. return M

```
i = 1, B[-6] = 1
i = 2, B[-6] = 2
i = 3, B[-2] = 3
i = 4, B[1] = 4
i = 5, B[-1] = 5
i = 6, B[-2] = 6
i = 7, B[3] = 7
```


Longest Zero-Sum Sub Array (和为零的最长子数组)

Algorithm 2: Divide and Conquer (revised)

$$\begin{array}{cccccccc}
 & & \text{L} = 3 & & & & & \text{R} = 3 \\
 & \text{---} & & & & & & \text{---} \\
 5, & -1, & -2, & 3, & 4, & 0, & -6 & || & 1, & 5, & 1, & -3, & 1, & 2, & 1 \\
 & & & & & \text{---} & & & & & & & & & \\
 & & & & & \text{---} & & & & & & & & & \\
 & \text{---} & & & & & & & & & & & & & \text{Middle(A, 1, 7, 14) = 6}
 \end{array}$$

```
LS[1] = -6
LS[2] = -6
LS[3] = -2
LS[4] = 1
LS[5] = -1
LS[6] = -2
LS[7] = 3
```

```
-RS[1] = -1
-RS[2] = -6
-RS[3] = -7
-RS[4] = -4
-RS[5] = -5
-RS[6] = -7
-RS[7] = -8
```

Middle(A, l, p, r)

1. Compute $LS[1..p]$, $RS[1..(r-p)]$
2. $B[P..Q] \leftarrow \{0, \dots, 0\}$
3. for $i \leftarrow 1$ to p do $B[LS[i]] = i$

4. $M \leftarrow 0$

- ```

5. for j ← 1 to r-p do
 if B[-RS[j]] > 0 and M < B[-RS[j]] + j
 M ← B[-RS[j]] + j

```

6. return M

```
i = 1, B[-6] = 1
i = 2, B[-6] = 2
i = 3, B[-2] = 3
i = 4, B[1] = 4
i = 5, B[-1] = 5
i = 6, B[-2] = 6
i = 7, B[3] = 7
```

j = 1, B[-1] = 5, M = 5+1=6  
j = 2, B[-6] = 2, M = 6  
j = 3, B[-7] = 0, M = 6  
....

# Longest Zero-Sum Sub Array （和为零的最长子数组）

Algorithm 2: Divide and Conquer (revised)

LongestZeroSum(A, l, r)

```
if l = r,
 if A[l] = 0,
 return 1
 else
 return 0
```

$p \leftarrow (l+r)/2$

$L \leftarrow \text{LongestZeroSum}(A, l, p)$

$R \leftarrow \text{LongestZeroSum}(A, p+1, r)$

$M \leftarrow \text{Middle}(A, l, p, r)$

Return max(L, R, M)

Middle(A, l, p, r)

1. Compute LS[1..p], RS[1..(r-p)]  $\theta(n)$
2.  $B[P..Q] \leftarrow \{0, \dots, 0\}$   $\theta(Q - P)$
3. for  $i \leftarrow 1$  to  $p$  do  $B[LS[i]] = i$   $\theta(n)$
4.  $M \leftarrow 0$
5. for  $j \leftarrow 1$  to  $r-p$  do  $\theta(n)$   
    if  $B[-RS[j]] > 0$  and  $M < B[-RS[j]] + j$   
         $M \leftarrow B[-RS[j]] + j$
6. return M

Time complexity:  $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$

# Longest Zero-Sum Sub Array (和为零的最长子数组)

Algorithm 3: Successive Counting

$$5, \overbrace{-1, -2, 3}^{L=3}, 4, 0, -6 \quad || \quad 1, 5, 1, \overbrace{-3, 1, 2}^{R=3}, 1$$

$$LS[1] = -6 \quad (-6)$$

$$LS[2] = -6 \quad (0 + -6)$$

$$LS[3] = -2 \quad (4 + 0 + -6)$$

$$LS[4] = 1 \quad (3 + 4 + 0 + -6)$$

$$LS[5] = -1 \quad (-2 + 3 + 4 + 0 + -6)$$

$$LS[6] = -2 \quad (-1 + -2 + 3 + 4 + 0 + -6)$$

$$LS[7] = 3 \quad (5 + -1 + -2 + 3 + 4 + 0 + -6)$$

$$\left. \begin{array}{l} LS[3] = -2 \\ LS[4] = 1 \\ LS[5] = -1 \end{array} \right\} A[2]+A[3]+A[4]=0$$

$$RS[1] = 1$$

$$RS[2] = 6$$

$$RS[3] = 7$$

$$RS[4] = 4$$

$$RS[5] = 5$$

$$RS[6] = 7$$

$$RS[7] = 8$$

# Longest Zero-Sum Sub Array (和为零的最长子数组)

Algorithm 3: Successive Counting

**5, -1, -2, 3, 4, 0, -6, 1, 5, 1, -3, 1, 2, 1**

$$S[1] = 5 \quad (5)$$

$$S[2] = 4 \quad (5 + -1)$$

$$S[3] = 2 \quad (5 + -1 + -2)$$

$$S[4] = 5 \quad (5 + -1 + -2 + 3)$$

$$S[5] = 9 \quad (5 + -1 + -2 + 3 + 4)$$

$$S[6] = 9 \quad (5 + -1 + -2 + 3 + 4 + 0)$$

$$S[7] = 3 \quad (5 + -1 + -2 + 3 + 4 + 0 + -6)$$

$$S[8] = 4 \quad (5 + -1 + -2 + 3 + 4 + 0 + -6 + 1)$$

$$S[9] = 9 \quad (5 + -1 + -2 + 3 + 4 + 0 + -6 + 1 + 5)$$

$$S[10] = 10 \quad (5 + -1 + -2 + 3 + 4 + 0 + -6 + 1 + 5 + 1)$$

$$S[11] = 7 \quad (5 + -1 + -2 + 3 + 4 + 0 + -6 + 1 + 5 + 1 + -3)$$

$$S[12] = 8 \quad (5 + -1 + -2 + 3 + 4 + 0 + -6 + 1 + 5 + 1 + -3 + 1)$$

$$S[13] = 10 \quad (5 + -1 + -2 + 3 + 4 + 0 + -6 + 1 + 5 + 1 + -3 + 1 + 2)$$

$$S[14] = 11 \quad (5 + -1 + -2 + 3 + 4 + 0 + -6 + 1 + 5 + 1 + -3 + 1 + 2 + 1)$$

# Longest Zero-Sum Sub Array （和为零的最长子数组）

- Algorithm 3: Successive Counting

LongestZeroSum(A[1...n])

$S[1] = A[1]$

for  $i \leftarrow 2$  to  $n$  do  $S[i] = S[i-1] + A[i]$

$B[P..Q] = \{0, \dots, 0\}$  //  $P = \min\{S[1], \dots, S[n]\}$ ,  $Q = \max\{S[1], \dots, S[n]\}$

$L = 0$

for  $i \leftarrow 1$  to  $n$  do

if  $B[S[i]] = 0$

$B[S[i]] = i$

// record the initial position of  $S[i]$

else

$L = \max\{i - B[S[i]], L\}$

return  $L$

# Longest Zero-Sum Sub Array (和为零的最长子数组)

- Algorithm 3: Successive Counting

LongestZeroSum(A[1..n])

$S[1] = A[1]$

for  $i \leftarrow 2$  to  $n$  do  $S[i] = S[i-1] + A[i]$

$B[P..Q] = \{0, \dots, 0\}$

$L = 0$

for  $i \leftarrow 1$  to  $n$  do

    if  $B[S[i]] = 0$ ,

$B[S[i]] = i$

    else

$L = \max\{i - B[S[i]], L\}$

return  $L$

5, -1, -2, 3, 4, 0, -6, 1, 5, 1, -3, 1, 2, 1

$S[1] = 5$        $i=1$ :  $B[5]=0, L=0 \rightarrow B[5]=1, L=0$

$S[2] = 4$        $i=2$ :  $B[4]=0, L=0 \rightarrow B[4]=2, L=0$

$S[3] = 2$        $i=3$ :  $B[2]=0, L=0 \rightarrow B[2]=3, L=0$

$S[4] = 5$        $i=4$ :  $B[5]=1, L=0 \rightarrow B[5]=1, L=4-1=3$

$S[5] = 9$        $i=5$ :  $B[9]=0, L=3 \rightarrow B[9]=5, L=3$

$S[6] = 9$        $i=6$ :  $B[9]=5, L=3 \rightarrow B[9]=5, L=3$

$S[7] = 3$        $i=7$ :  $B[3]=0, L=3 \rightarrow B[3]=7, L=3$

$S[8] = 4$        $i=8$ :  $B[4]=2, L=3 \rightarrow B[4]=2, L=8-2=6$

$S[9] = 9$        $i=9$ :  $B[9]=5, L=6 \rightarrow B[9]=5, L=6$

$S[10] = 10$

$S[11] = 7$

$S[12] = 8$

$S[13] = 10$

$S[14] = 11$

# Longest Zero-Sum Sub Array （和为零的最长子数组）

- Algorithm 3: Successive Counting

LongestZeroSum(A[1..n])

$S[1] = A[1]$

for  $i \leftarrow 2$  to  $n$  do  $S[i] = S[i-1] + A[i]$   $\theta(n)$

$B[P..Q] = \{0, \dots, 0\}$   $\theta(Q - P)$

$L = 0$

for  $i \leftarrow 1$  to  $n$  do  $\theta(n)$

    if  $B[S[i]] = 0$ ,

$B[S[i]] = i$

    else  $L = \max\{i - B[S[i]], L\}$

return  $L$

Time complexity:  $T(n) = \theta(n)$

# Longest Zero-Sum Sub Array (和为零的最长子数组)

- **Algorithm 4: Comparison sorting**
  - The counting works well when  $(\max\{S[1], \dots, S[n]\} - \min\{S[1], \dots, S[n]\})$  is kept small enough.
  - But the efficiency in both time and space will drop down as the **difference gets large**. (e.g.  $> 10^8$ )
  - In this case, **comparison sorting** can be used to find a pair of identical prefix sums that are the furthest apart in the array.
  - For this purpose, we attach a second key (i.e., the index) to each prefix sum.

$$\forall i \in [1, n]: S[i] \rightarrow (S[i], i)$$

- For any  $1 \leq i, j \leq n$ ,  $(S[i], i) < (S[j], j)$  holds if  $S[i] < S[j]$  or  $S[i] = S[j] \wedge i < j$ .



# Longest Zero-Sum Sub Array (和为零的最长子数组)

- Algorithm 4: Comparison sorting

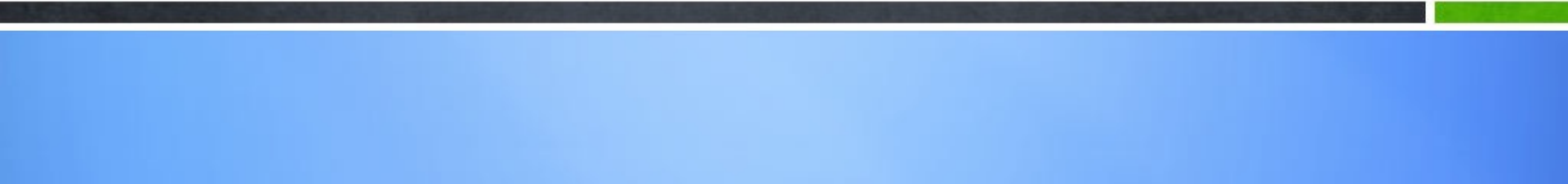
5, -1, -2, 3, 4, 0, -6, 1, 5, 1, -3, 1, 2, 1

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

|            |                        |                          |
|------------|------------------------|--------------------------|
| S[1] = 5   | (S[1], 1) = (5, 1)     | (2, 3)                   |
| S[2] = 4   | (S[2], 2) = (4, 2)     | (3, 7)                   |
| S[3] = 2   | (S[3], 3) = (2, 3)     | (4, 2)                   |
| S[4] = 5   | (S[4], 4) = (5, 4)     | (4, 8) } $8 - 2 = 6$     |
| S[5] = 9   | (S[5], 5) = (9, 5)     | (5, 1)                   |
| S[6] = 9   | (S[6], 6) = (9, 6)     | (5, 4) } $4 - 1 = 3$     |
| S[7] = 3   | (S[7], 7) = (3, 7)     | (7, 11)                  |
| S[8] = 4   | (S[8], 8) = (4, 8)     | (8, 12)                  |
| S[9] = 9   | (S[9], 9) = (9, 9)     | (9, 5)                   |
| S[10] = 10 | (S[10], 10) = (10, 10) | (9, 6)                   |
| S[11] = 7  | (S[11], 11) = (7, 11)  | (9, 9) } $9 - 5 = 4$     |
| S[12] = 8  | (S[12], 12) = (8, 12)  | (10, 10)                 |
| S[13] = 10 | (S[13], 13) = (10, 13) | (10, 13) } $13 - 10 = 3$ |
| S[14] = 11 | (S[14], 14) = (11, 14) | (11, 14)                 |



## **3. 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<br>on LSD | After sorting<br>on middle<br>digit | After sorting<br>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.

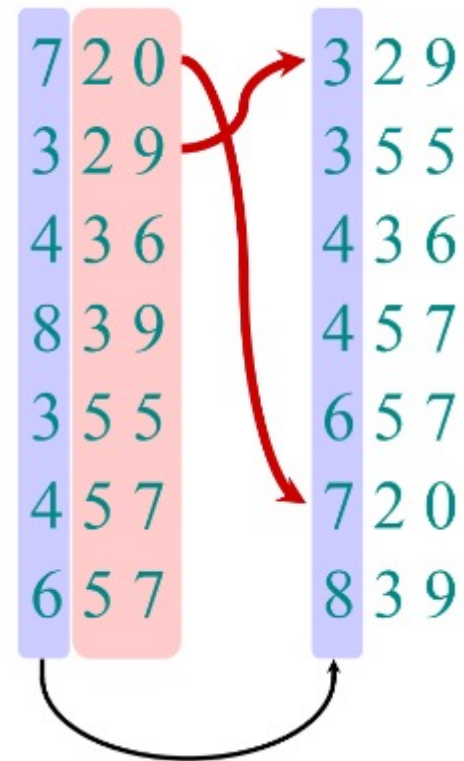
Radix sort of  $d$  digits  $\equiv$  radix sort of the low-order  $d - 1$  digits followed by a sort on digit  $d$ .

# Correctness of Radix Sort

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

Consider two elements,  $a$  and  $b$ , with  $d^{\text{th}}$  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.

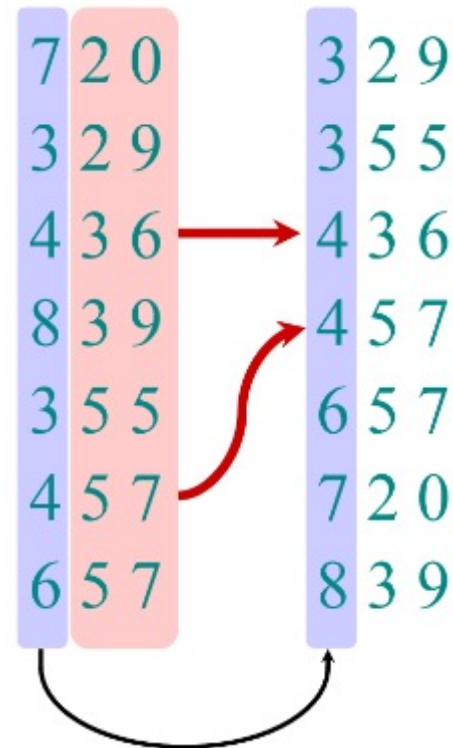


# Correctness of Radix Sort

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

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

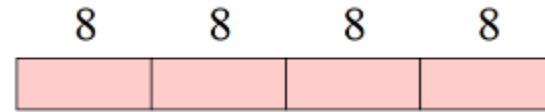
- If  $a_d = b_d$ , the sort will **leave  $a$  and  $b$  in the same order**, since **the sort is stable**. But that order is already correct, since the correct order of is determined by the low-order digits when their  $d^{\text{th}}$  digits are equal.



# Algorithm Analysis

- Assume counting sort is the auxiliary stable sort.
- Sort  $n$  words of  $b$  bits each.
- Each word can be viewed as having  $b/r$  base- $2^r$  digits.

- **Example:** 32-bit word



- $r = 8 \Rightarrow b/r = 4$  passes of counting sort on base- $2^8$  digits;
- $r = 16 \Rightarrow b/r = 2$  passes of counting sort on base- $2^{16}$  digits.



# Algorithm Analysis

- If each  $b$ -bit word is broken into  $r$ -bit pieces, each pass of **counting sort** takes  $\Theta(n + 2^r)$  time.
- Since there are  $b/r$  passes, we have

$$T(n, b) = \Theta\left(\frac{b}{r} (n + 2^r)\right)$$

**Goal** : Choose  $r$  to minimize  $T(n, b)$ :

**Intuition**: Increasing  $r$  means fewer passes, but as  $r > \log(n)$ , the time grows exponentially.

# Algorithm Analysis

- If each  $b$ -bit word is broken into  $r$ -bit pieces, each pass of **counting sort** takes  $\Theta(n + 2^r)$  time.
- Since there are  $b/r$  passes, we have

$$T(n, b) = \Theta\left(\frac{b}{r} (n + 2^r)\right)$$

**Goal** : Choose  $r$  to minimize  $T(n, b)$ :

- Assume  $r = \log(n)$ , Then  $T(n, b) = \Theta\left(\frac{bn}{\log(n)}\right)$
- If all sorting data are in range  $[0, n^d)$ , i.e.,  $b = \log(n)$ , we have  $T(n, b) = \Theta(dn)$

# Algorithm Analysis

---

- In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

**Example** (32-bit numbers):

- At most 3 passes when sorting more than 2000 numbers.
- Merge sort and quicksort do at least  $\log(2000) = 11$  passes.

**Downside:** Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

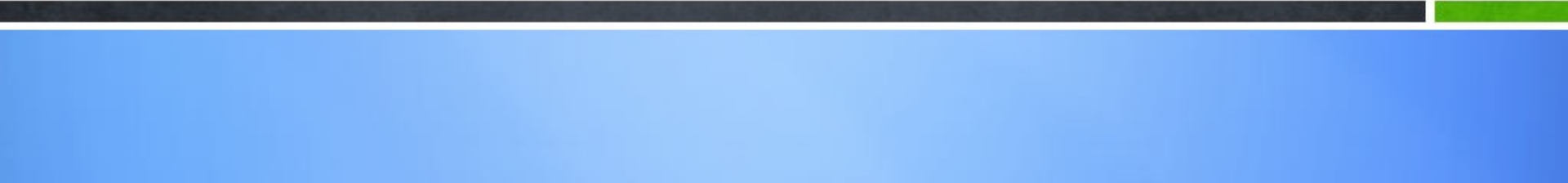
# Algorithm Analysis

- Each pass over  $n$   $d$ -digit numbers then takes time  $\Theta(n+k)$ . (Assuming counting sort is used for each pass.)
- 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.



# Suffix Array

Application of radix sort



# Suffix Array (后缀数组)

- 后缀数组 (suffix array) 是一个通过对字符串的所有**后缀**经过**排序**后得到的数组
- 后缀数组被乌迪·曼伯尔与尤金·迈尔斯于1990年提出，比后缀树简单并节省空间。它们也被Gaston Gonnet 于1987年独立发现，并命名为“PAT数组”
- 被广泛运用于全文索引、数据压缩算法、以及生物信息学。**例：统计子串频次、子串相似度、最长公共子串等**

# Suffix Array (后缀数组)

- (Leetcode 1044) 最长重复子串  
给出一个字符串 S，考虑其所有重复子串（连续子串，在S中出现两次以上，可重叠）。返回具有最长长度的重复子串。

例：输入 “banana”  
输出 “ana”

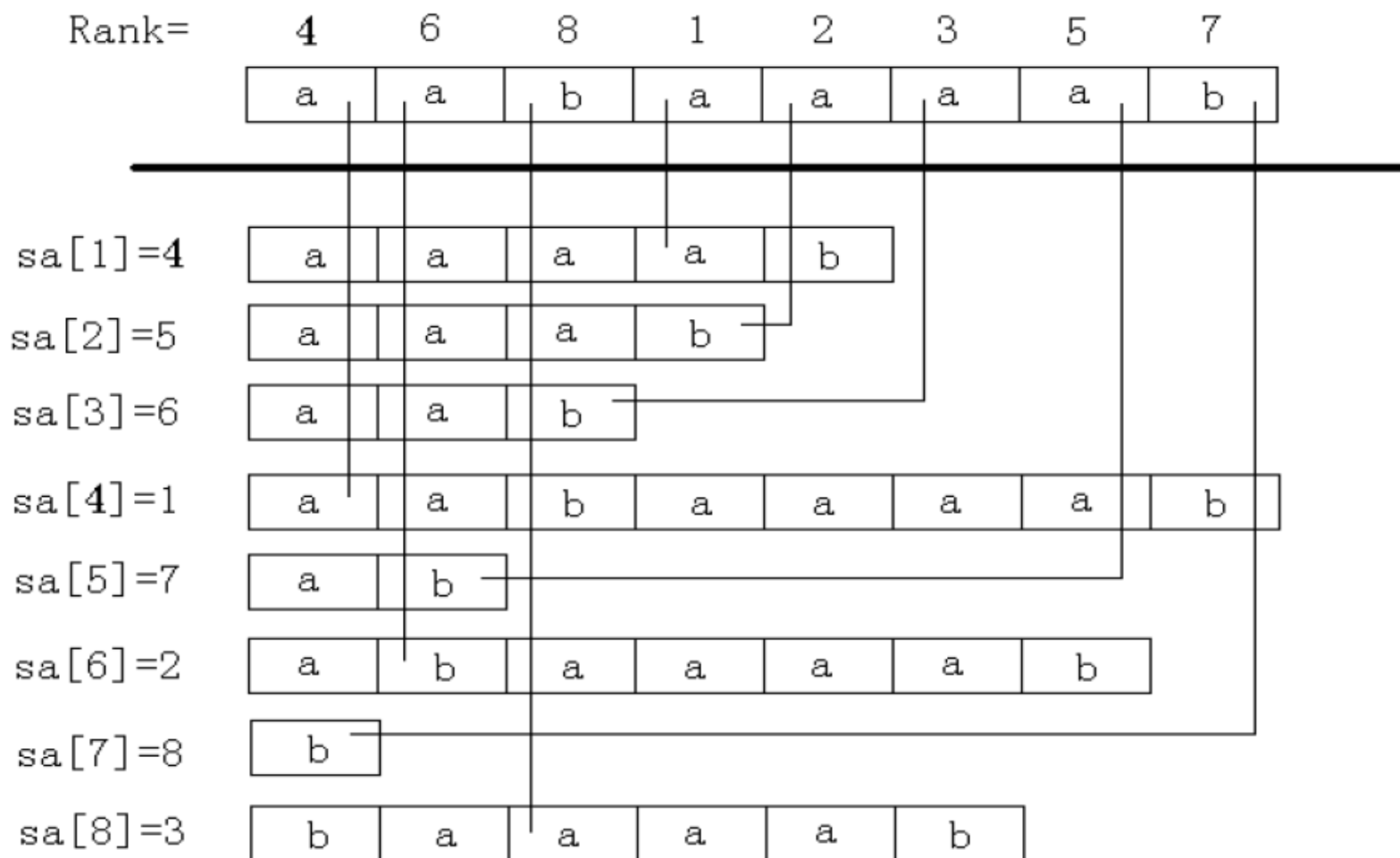
解题思路：将重复子串问题转化为后缀串间的最长公共前缀串问题。借助后缀数组，快速计算最长公共前缀。

# Suffix Array (后缀数组)

- **子串**: 字符串 $S[1]S[2]\dots S[n]$ , 子串 $S[i, j]=S[i]S[i+1]\dots S[j]$
- **后缀**: 后缀是指从某个位置 $i$ 开始到整个串末尾结束的一个特殊子串。  $\text{suffix}(i)=S[i, n]$
- **字典序**: 字符串 $X[1..n]$ ,  $Y[1..m]$ 且 $X<Y$ 成立的充分必要条件
  - (1)  $n < m$  且  $\forall i \in \{1, \dots, n\}: X[i] = Y[i]$  或者
  - (2)  $\exists 1 \leq j \leq \min(m, n) \forall 1 \leq i < j: X[j] < Y[j] \wedge X[i] = Y[i]$
- **后缀数组SA**: 把 $n$ 个后缀从小到大排序,  $SA[i]$ 记录排在第 $i$ 位的后缀的开始位置, 即
$$\forall 1 \leq i < n: \text{suffix}(SA[i]) < \text{suffix}(SA[i + 1])$$
- **名次数组Rank**: 名次数组 $Rank[i]$ 保存的是 $\text{suffix}(i)$ 在所有后缀中从小到大排列的“名次”
$$\forall 1 \leq i \leq n: Rank[SA[i]] = SA[Rank[i]] = i$$



# Suffix Array (后缀数组)



# Suffix Array (后缀数组)

长度（不超过） $2^k$ 的子串序列：

$S[1, 2^k], S[2, 2^k + 1], \dots, S[i, i + 2^k] (i + 2^k \leq n), \dots, S[n - 2^k + 1, n], \dots, S[n, n]$

- $SA[i] (1 \leq i \leq n)$  表示从小到大排在第  $i$  位的子串的开始位置  
(绝对顺序，每个  $i$  对应唯一的子串)



$$\forall 1 < i \leq n: S[SA[i - 1], SA[i - 1] + 2^k - 1] \leq S[SA[i], SA[i] + 2^k - 1]$$

- $Rank[i] (1 \leq i \leq n)$  表示  $S[i, i + k]$  在所有子串中从小到大排第几位  
(相对顺序，相同子串其 **Rank** 值相同)



$$S[i, i + 2^k - 1] < S[j, j + 2^k - 1] \Leftrightarrow Rank[i] < Rank[j]$$

$$S[i, i + 2^k - 1] = S[j, j + 2^k - 1] \Leftrightarrow Rank[i] = Rank[j]$$

# Suffix Array (后缀数组)

长度（不超过） $2^k$ 的子串序列：

$S[1,2^k], S[2,2^k + 1], \dots, S[i, i + 2^k] (i + 2^k \leq n), \dots, S[n - 2^k + 1, n], \dots, S[n, n]$

- $SA^k[i] (1 \leq i \leq n)$  表示从小到大排在第*i*位的子串的开始位置  
(绝对顺序，每个*i*对应唯一的子串)
- $Rank^k[i] (1 \leq i \leq n)$  表示 $S[i, i+k]$ 在所有子串中从小到大排第几位  
(相对顺序，相同子串其Rank值相同)

$k = 0$

| $i$         | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-------------|---|---|---|---|---|---|---|---|
| $SA^0[i]$   | 1 | 2 | 4 | 5 | 6 | 7 | 3 | 8 |
| $S[i]$      | a | a | b | a | a | a | a | b |
| $Rank^0[i]$ | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 2 |

# Suffix Array (后缀数组)

长度（不超过） $2^k$ 的子串序列：

$S[1, 2^k], S[2, 2^k + 1], \dots, S[i, i + 2^k] (i + 2^k \leq n), \dots, S[n - 2^k + 1, n], \dots, S[n, n]$

•  $SA^k[1, \dots, n] \Rightarrow Rank^k[1, \dots, n]$

```
1. $Rank^k[SA^k[1]] \leftarrow 1$
2. for ($i \leftarrow 2, rk \leftarrow 1; i \leq n; i \leftarrow i + 1$)
 do if $S[SA^k[i], SA^k[i] + 2^k - 1] = S[SA^k[i - 1], SA^k[i - 1] + 2^k - 1]$
 then $Rank^k[SA^k[i]] \leftarrow Rank^k[SA^k[i - 1]]$
 else
 $rk \leftarrow rk + 1$
 $Rank^k[SA^k[i]] \leftarrow rk$
```

时间复杂度  $\Theta(n)$

# Suffix Array (后缀数组)

长度（不超过） $2^k$ 的子串序列：

$S[1, 2^k], S[2, 2^k + 1], \dots, S[i, i + 2^k] (i + 2^k \leq n), \dots, S[n - 2^k + 1, n], \dots, S[n, n]$

- $Rank^k[1, \dots, n] \Rightarrow SA^k[1, \dots, n]$

```
1. $C[1, \dots, n] \leftarrow 0$ Counting Sort
2. for($i \leftarrow 1; i \leq n; i \leftarrow i + 1$)
 do $C[Rank^k[i]] \leftarrow C[Rank^k[i]] + 1$
3. for($i \leftarrow 2; i \leq n; i \leftarrow i + 1$)
 do $C[i] \leftarrow C[i] + C[i - 1]$
4. for($i \leftarrow n; i > 0; i \leftarrow i - 1$)
 do $SA^k[C[Rank^k[i]]] \leftarrow i$
 $C[Rank^k[k]] \leftarrow C[Rank^k[k]] - 1$
```

时间复杂度  $\Theta(n)$

# Suffix Array (后缀数组)

- 快速排序  
时间复杂度  $O(n^2 \log(n))$
- DC3  
时间复杂度  $cn$ , 但常数 $c$ 通常很大
- 倍增法  
时间复杂度  $O(n \log(n))$

思路： 对每个字符开始的长度为 $2^k$ 的子字符串进行排序， $k=0$ 开始，每次循环增加1，直到 $2^k$ 大于 $n$ ，每个字符开始的长度为 $2^k$ 的子字符串便相当于所有的后缀。

# Suffix Array (后缀数组)

$Rank^k[i]$

$Rank^k[i + 2^k + 1]$

|        |          |      |                  |              |      |                  |
|--------|----------|------|------------------|--------------|------|------------------|
| $S[i]$ | $S[i+1]$ | .... | $S[i + 2^k - 1]$ | $S[i + 2^k]$ | .... | $S[i + 2^{k+1}]$ |
|--------|----------|------|------------------|--------------|------|------------------|

$Rank^{k+1}[i] = ?$

$Rank^k[j]$

$Rank^k[j + 2^k + 1]$

|        |          |      |              |            |      |                |
|--------|----------|------|--------------|------------|------|----------------|
| $S[j]$ | $S[j+1]$ | .... | $S[j+2^k-1]$ | $S[j+2^k]$ | .... | $S[j+2^{k+1}]$ |
|--------|----------|------|--------------|------------|------|----------------|

$Rank^{k+1}[j] = ?$

已知 $Rank^k[1, \dots, n]$ , 如何求解 $Rank^{k+1}[1, \dots, n]$

# Suffix Array (后缀数组)

| $Rank^k[i]$ |          |      |                  | $Rank^k[i + 2^k + 1]$ |      |                  |
|-------------|----------|------|------------------|-----------------------|------|------------------|
| $S[i]$      | $S[i+1]$ | .... | $S[i + 2^k - 1]$ | $S[i + 2^k]$          | .... | $S[i + 2^{k+1}]$ |

$$Rank^{k+1}[i] = ?$$

| $Rank^k[j]$ |          |      |                | $Rank^k[j + 2^k + 1]$ |      |                |
|-------------|----------|------|----------------|-----------------------|------|----------------|
| $S[j]$      | $S[j+1]$ | .... | $S[j+2^k - 1]$ | $S[j+2^k]$            | .... | $S[j+2^{k+1}]$ |

$$Rank^{k+1}[j] = ?$$

$Rank^{k+1}[i] \leq Rank^{k+1}[j]$  成立的条件:

$$(1) Rank^k[i] < Rank^k[j] \Rightarrow Rank^{k+1}[i] < Rank^{k+1}[j]$$

$$(2) Rank^k[i] = Rank^k[j] \wedge Rank^k[i + 2^k] \leq Rank^k[j + 2^k] \\ \Rightarrow Rank^{k+1}[i] \leq Rank^{k+1}[j]$$



# Suffix Array (后缀数组)

| $Rank^k[i]$ |          |      |                  | $Rank^k[i + 2^k + 1]$ |      |                  |
|-------------|----------|------|------------------|-----------------------|------|------------------|
| $S[i]$      | $S[i+1]$ | .... | $S[i + 2^k - 1]$ | $S[i + 2^k]$          | .... | $S[i + 2^{k+1}]$ |

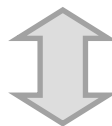
$$Rank^{k+1}[i] = ?$$

| $Rank^k[j]$ |          |      |                | $Rank^k[j + 2^k + 1]$ |      |                |
|-------------|----------|------|----------------|-----------------------|------|----------------|
| $S[j]$      | $S[j+1]$ | .... | $S[j+2^k - 1]$ | $S[j+2^k]$            | .... | $S[j+2^{k+1}]$ |

$$Rank^{k+1}[j] = ?$$

$Rank^{k+1}[i] \leq Rank^{k+1}[j]$  成立的条件:

$$(Rank^k[i] \mid Rank^k[i + 2^k])_{n+1} \leq (Rank^k[j] \mid Rank^k[j + 2^k])_{n+1}$$



$$Rank^{k+1}[i] \leq Rank^{k+1}[j]$$

# Suffix Array (后缀数组)

k=0 开始

$$(Rank^k[1] \mid Rank^k[2^k + 1])_{n+1}$$

$$(Rank^k[i] \mid Rank^k[2^k + i])_{n+1}$$

$$(Rank^k[n - 2^k] \mid Rank^k[n])_{n+1}$$

$$(Rank^k[n - 2^k + 1] \mid 0)_{n+1}$$

$$(Rank^k[n] \mid 0)_{n+1}$$

$2^k > n$  结束

基位+计数排序

$$SA^{k+1}[1, \dots, n]$$

判重

$$Rank^{k+1}[SA[i-1]] = Rank^{k+1}[SA[i]]$$



$$Rank^k[SA[i-1]] = Rank^k[SA[i]]$$

$\wedge$

$$Rank^k[SA[i-1] + 2^k] = Rank^k[SA[i] + 2^k]$$

转换

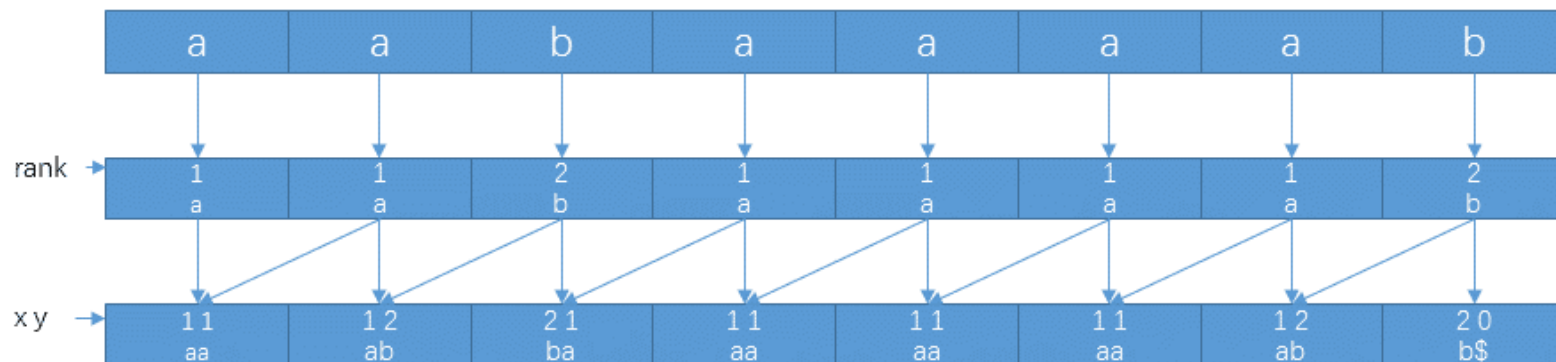
$k \leftarrow k + 1$

$$Rank^{k+1}[1, \dots, n]$$

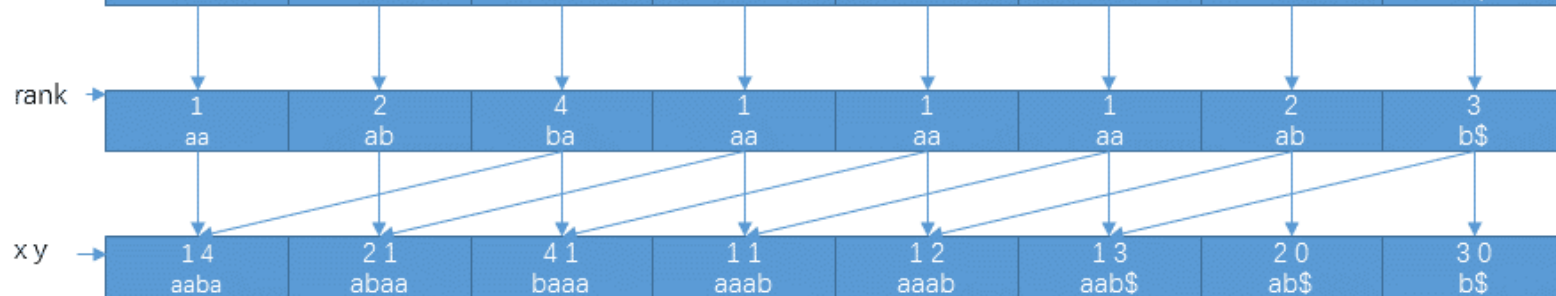
时间复杂度  $O(n \log(n))$

# Suffix Array (后缀数组)

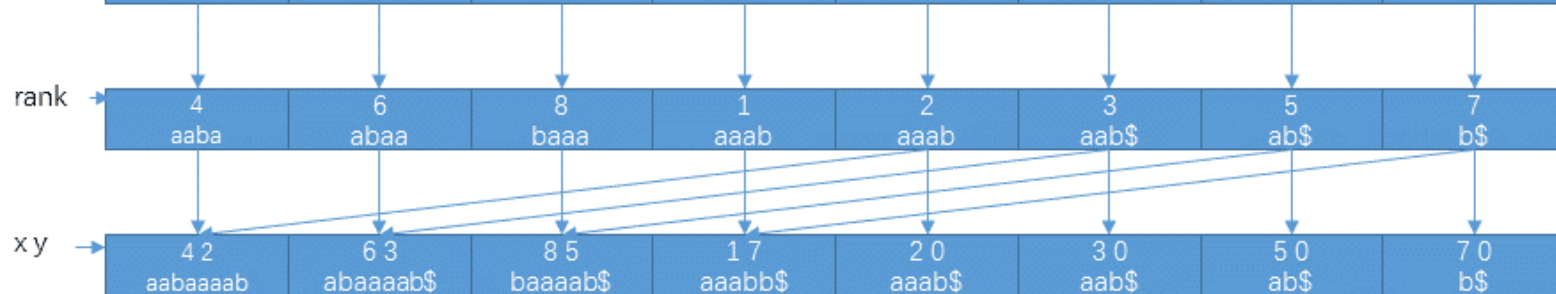
第一次排序



第二次排序



第三次排序



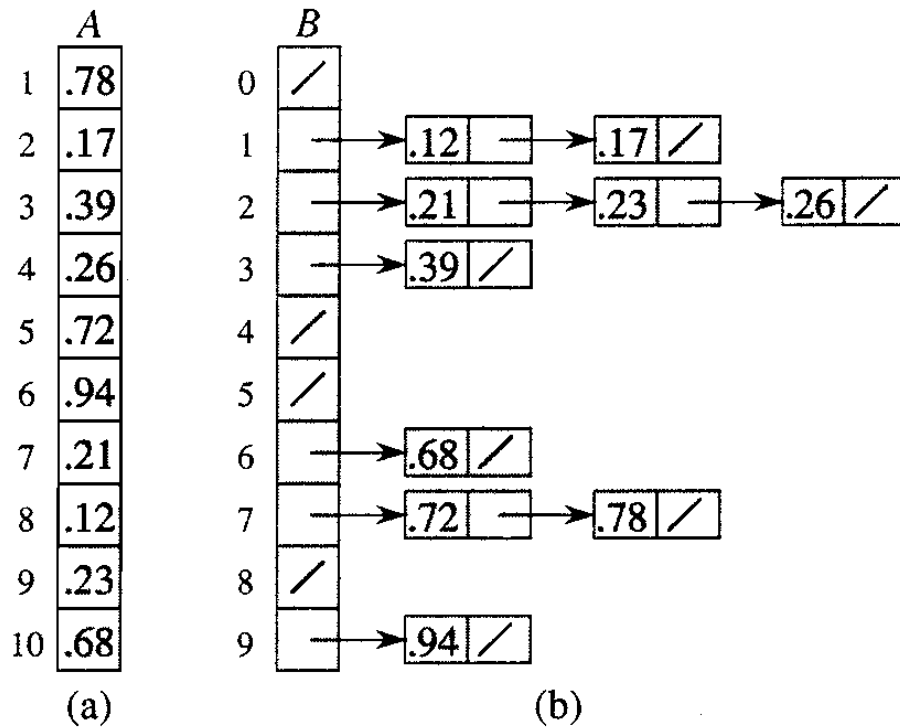
第四次排序



# 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.
- 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.
- We “expect” each bucket to have few elements, since the average is 1 element per bucket.
- But we need to do a careful analysis.



# Analysis – Contd.

- Suppose  $n_i$  = no. of elements are placed in bucket  $B[i]$ .
- Insertion sort runs in quadratic time. Hence, time for bucket sort is:

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

Taking expectations of both sides and using linearity of expectation, we have

$$\begin{aligned} E[T(n)] &= E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \quad (\text{by linearity of expectation}) \quad (8.1) \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \quad (E[aX] = aE[X]) \end{aligned}$$

# Analysis – Contd.

- **Claim:**  $E[n_i^2] = 2 - 1/n.$  (8.2)
- **Proof:**
- **Define indicator random variables.**
  - $X_{ij} = I\{A[j] \text{ falls in bucket } i\}$
  - **Probability** $\{A[j] \text{ falls in bucket } i\} = 1/n.$
  - $n_i = \sum_{j=1}^n X_{ij}$

指示随机变量(indicator random variable)是概率分析中非常重要的一种离散随机变量，其用来表征某事件是否发生。更加具体的，假设事件A发生，变量取值为1，否则取值为0. 数学上表述为：

$$I\{A\} = \begin{cases} 1, & \text{if } A \text{ happens} \\ 0, & \text{otherwise} \end{cases}$$

# Analysis – Contd.

$$\begin{aligned} E[n_i^2] &= E\left[\left(\sum_{j=1}^n X_{ij}\right)^2\right] \\ &= E\left[\sum_{j=1}^n \sum_{k=1}^n X_{ij} X_{ik}\right] \\ &= E\left[\sum_{j=1}^n X_{ij}^2 + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ j \neq k}} X_{ij} X_{ik}\right] \\ &= \sum_{j=1}^n E[X_{ij}^2] + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ j \neq k}} E[X_{ij} X_{ik}] \text{ , by linearity of expectation.} \end{aligned}$$

(8.3)

# Analysis – Contd.

$$\begin{aligned} E[X_{ij}^2] &= 0^2 \cdot \Pr\{A[j] \text{ doesn't fall in bucket } i\} + \\ &\quad 1^2 \cdot \Pr\{A[j] \text{ falls in bucket } i\} \\ &= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n} \\ &= \frac{1}{n} \end{aligned}$$

$E[X_{ij}X_{ik}]$  for  $j \neq k$ :

Since  $j \neq k$ ,  $X_{ij}$  and  $X_{ik}$  are independent random variables.

$$\begin{aligned} \Rightarrow E[X_{ij}X_{ik}] &= E[X_{ij}]E[X_{ik}] \\ &= \frac{1}{n} \cdot \frac{1}{n} \\ &= \frac{1}{n^2} \end{aligned}$$


# Analysis – Contd.

(8.3) is hence,


$$\begin{aligned} E[n_i^2] &= \sum_{j=1}^n \frac{1}{n} + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} \frac{1}{n^2} \\ &= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} \\ &= 1 + \frac{n-1}{n} \\ &= 2 - \frac{1}{n}. \end{aligned}$$

Substituting (8.2) in (8.1), we have,

$$\begin{aligned} E[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n) \\ &= \Theta(n) + O(n) \\ &= \Theta(n) \end{aligned}$$



算法分析课程组  
重庆大学计算机学院



End of Section.

