Sorting in Linear Time

ITX2010, CSX3003, IT2230

Data Structures and Algorithms, Information Structures

Learning Objectives

Students will be able to:

- Prove that any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case:
  - Merge sort and Heap sort are asymptotically optimal.

- Examine three algorithms (Use operations rather than comparison to determine the sorted order and run in linear time) – including
  - Counting sort
  - Radix sort
  - Bucket sort

Chapter Outline

1. Lower bounds for sorting
   1) Comparison sort
   2) The decision tree model
   3) A lower bound for the worst case
2. Counting sort
   1) Concept
   2) Algorithm
   3) Running time
   4) Stability
   5) Strength and weakness
3. Radix sort
   1) Concept
   2) Algorithm
   3) Running time
   4) Radix sort and Comparison sort
4. Bucket sort
   1) Concept
   2) Algorithm
   3) Running time

8.1

Lower bounds for sorting

1) Comparison sort
2) The decision tree model
3) A lower bound for the worst case

Comparison Sort

Lower bounds for sorting

- Comparing elements to obtain its order of a sequence list (a_1, a_2,…,a_n). **[1]**

$<a_1 , a_2 , a_3>$

- To determine a relative order, when considering comparisons between two numbers (a_i and a_j),

How many possible comparisons will be initiated for a three-key set ?

Comparison Sort

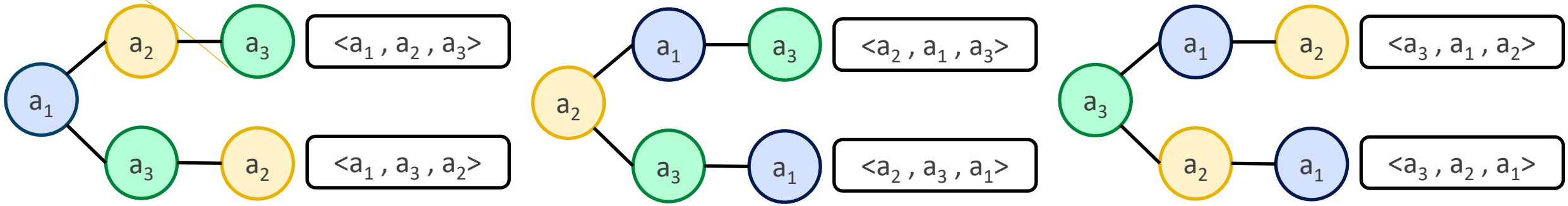
Lower bounds for sorting

- **Permutation rule:** the number of r-permutations for an n-key set:

$$ {}^{n}_{r}P = P(n,r) = \frac{n!}{(n-r)!} $$

When comparing 2 numbers at a time in a set of 3 keys,

$$ {}^{3}_{2}P = P(3,2) = \frac{3!}{(3-2)!} = 6\ \# $$



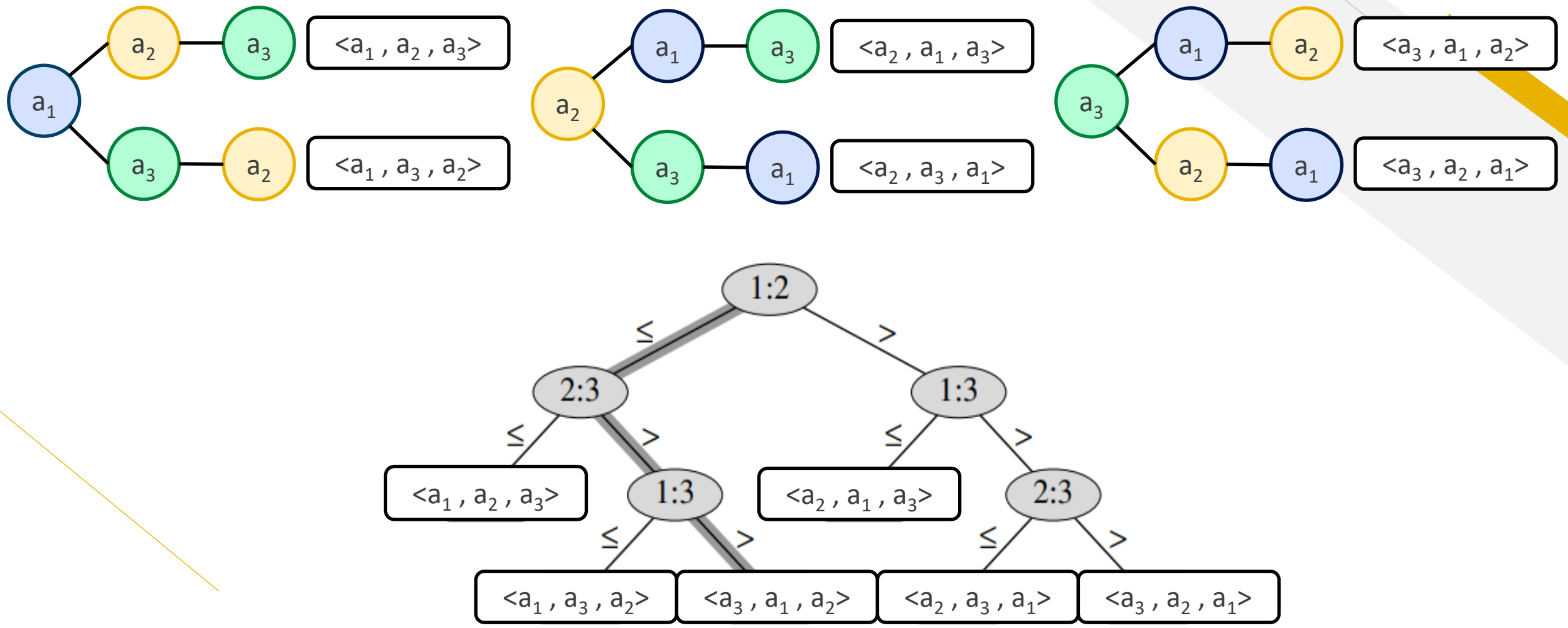


Fig 8-1 Decision tree for insertion sort operating on three elements **[1]**

The decision tree model

Lower bounds for sorting

- Decision tree is a full binary tree that can represent entire comparisons { $a_i < a_j$, $a_i \leq a_j$, $a_i > a_j$, $a_i \geq a_j$, $a_i = a_j$ } between elements in sorting.
- Execution of sorting corresponds to tracing a path front the root to a leaf.

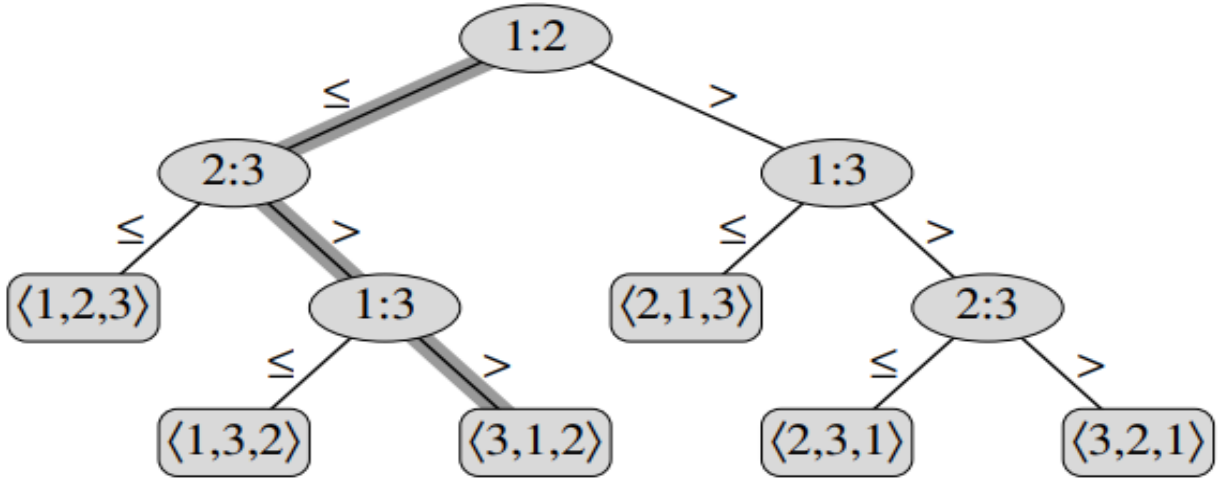


Fig 8-1 Decision tree for insertion sort operating on three elements [1]

A lower bound for the worst case

Lower bounds for sorting

- The length of longest path from root to its reachable leaves -> represents the worst-case number of comparisons.
- The worst-case number of comparisons equals the height of decision tree !

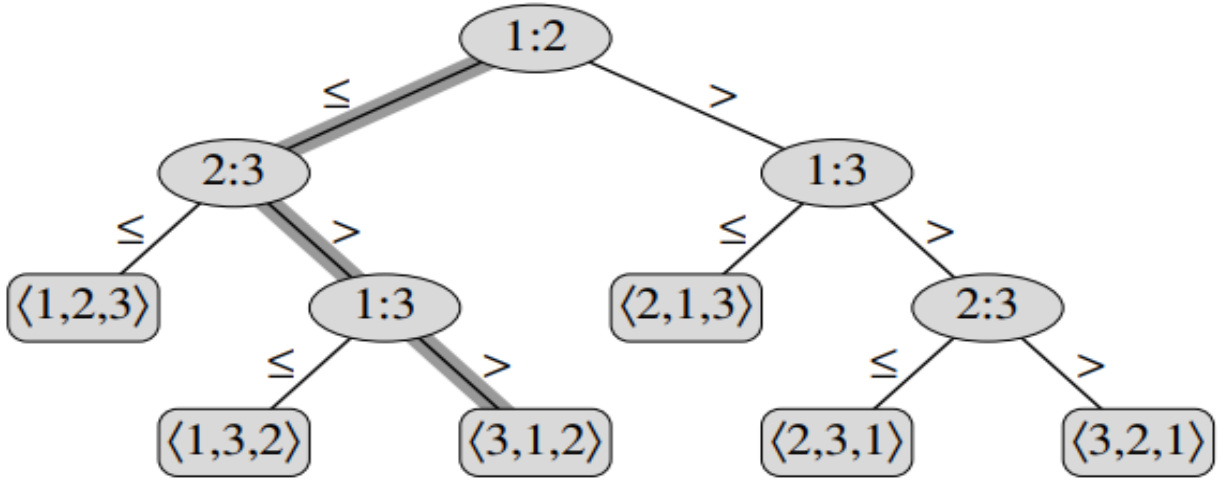


Fig 8-1 Decision tree for insertion sort operating on three elements [1]

A lower bound for the worst case

Lower bounds for sorting

- **Theorem 8.1 :** Any comparison sort algorithm requires Ω(n lg n) comparisons in the worst case.
  - Given a decision tree whose height is h with l reachable leaves of sorting n elements.
  - Since a binary tree of height h has no more than 2^h leaves:
$$n! \leq l \leq 2^h$$
  - Take logarithm both side to find the height:
$$h \geq \lg(n!) = \Omega(n \lg n)$$
- **Corollary 8.2:** Heap sort and Merge sort are asymptotically optimal comparison sorts.
  - The O(n lg n) upper bounds on the runtimes for heapsort and merge sort match the Ω(n lg n) worst-case lower bound from Theorem 8.1 !

8.2

Counting sort

1) Counting sort concept
2) Counting sort algorithm
3) Counting sort running time
4) Counting sort stability
5) Counting sort strength and weakness

Counting sort concept

Counting sort

- A sorted list is sorted by counting each element's occurrence then doing arithmetic to find its sequence in the list.

- Counting sort assumes that each of the n input is an integer in the range 0 to k. [1]
  - When k = O(n), the sort runs in $\Theta(n)$ time.

- Counting sort requires the following parameters:
  1. Input array: A[1..n] whose length is length (A) = n
  2. Output array: B[1..n] hold sorted list
  3. A maximum element in array A: k

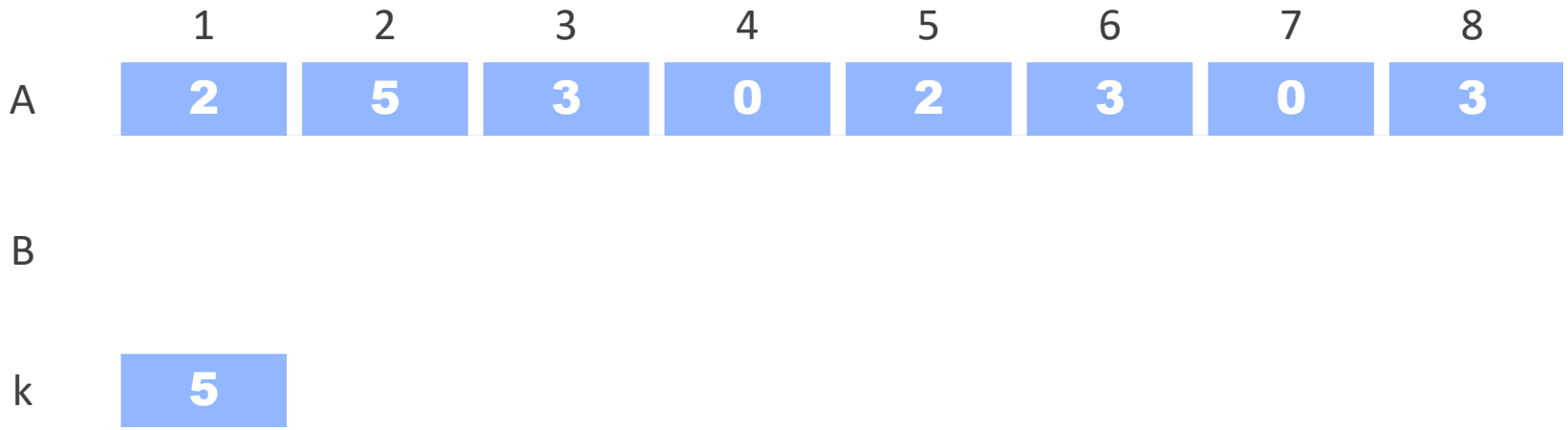
COUNTING-SORT(A, B, k)

Counting sort concept

Counting sort

1. Assign an unsorted array A and initial an empty sorted list B,
2. Find out the maximum element in A store in k
3. Execute the counting sort

```
17  if __name__ == "__main__":
18      #Assign array A and B
19      A = [2, 5, 3, 0, 2, 3, 0, 3];
20      B=[];
21      #Finds max val in the list
22      k=max(A);
23      print("Array A : %s" % str(A));
24      print("Sorted array B : %s" % str(Counting_Sort(A,B,k)));
```



Counting sort concept

Counting sort algorithm [1]

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 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 ← 1 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
```

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

,B ,k= 5

```
COUNTING-SORT(A, B, k)
1  for i ← 0 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
7
8
9
10
11
```

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

	0	1	2	3	4	5
C	2	0	2	3	0	1

(a)

data	0	1	2	3	4	5
C						
2	0	1	2	3	4	5
C			[1]			
5	0	1	2	3	4	5
C			1			[1]
3	0	1	2	3	4	5
C			1	[1]		1
0	0	1	2	3	4	5
C	[1]		1	1		1
2	0	1	2	3	4	5
C	1		[2]	1		1
3	0	1	2	3	4	5
C	1		2	[2]		1
0	0	1	2	3	4	5
C	[2]		2	2		1
3	0	1	2	3	4	5
C	2		2	[3]		1

(a)

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 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 ← 1 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
10
11
```

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

Index	0	1	2	3	4	5
C	2		2	3		1
0	0	1	2	3	4	5
C	2					
1	0	1	2	3	4	5
C	2	2				
2	0	1	2	3	4	5
C	2	2	4			
3	0	1	2	3	4	5
C	2	2	4	7		
4	0	1	2	3	4	5
C	2	2	4	7	7	
5	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

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

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 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 ← 1 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
```

	1	2	3	4	5	6	7	8
B							3	

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

(c)

Index	Index	A[8] = 3
	C	7
B	3	

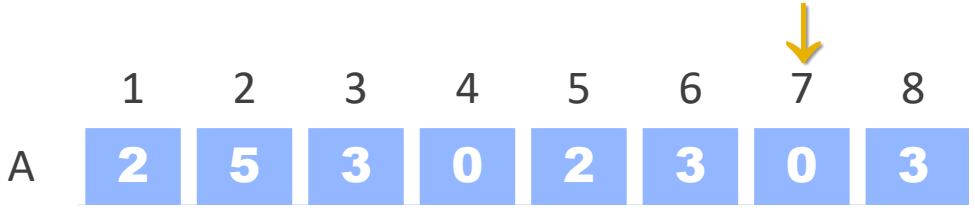
Index	A[8] = 3
C	6

Index	0	1	2	3	4	5
C	2	2	4	7	7	8

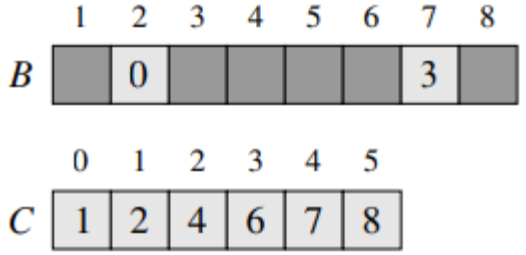
Index	1	2	3	4	5	6	7	8
B							[3]	

Index	0	1	2	3	4	5
C	2	2	4	[6]	7	8

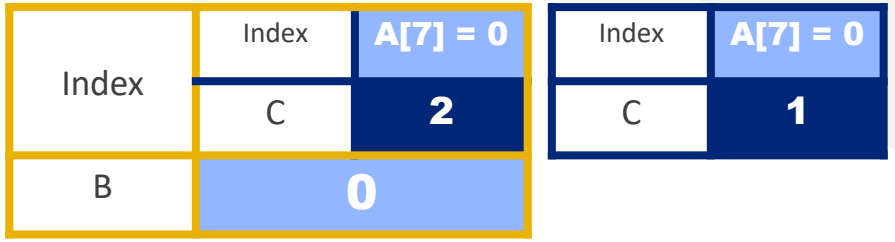
(c)

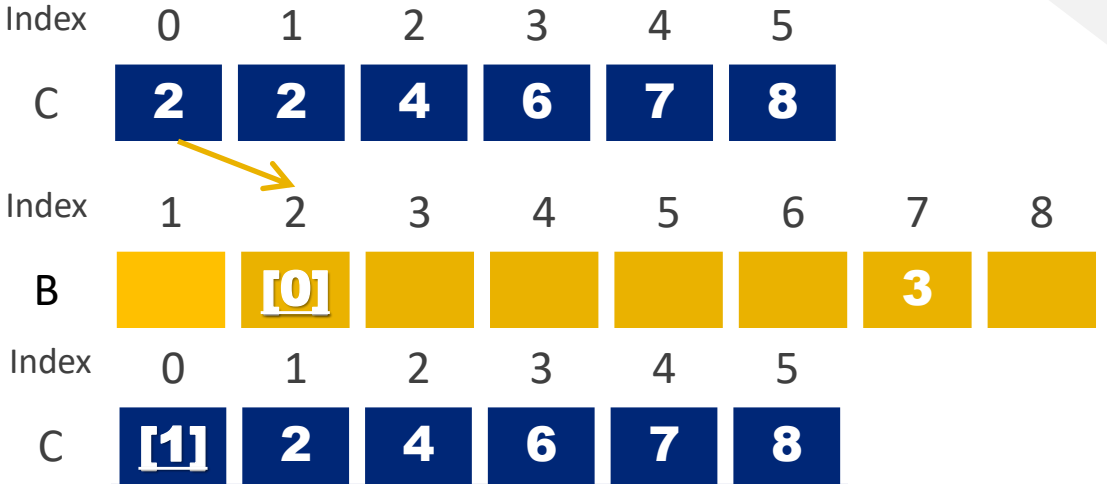



```
COUNTING-SORT(A, B, k)
 1  for i ← 0 to k
 2       do C[i] ← 0
 3  for j ← 1 to length[A]
 4       do C[A[j]] ← C[A[j]] + 1
 5  ▷ C[i] now contains the number of elements equal to i.
 6  for i ← 1 to k
 7       do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] now contains the number of elements less than or 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
```



(d)





(d)

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

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 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 ← 1 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
```

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

	0	1	2	3	4	5
C	1	2	4	5	7	8

(e)

Index	Index	A[6] = 3
	C	6
B	3	

Index	A[6] = 3
C	5

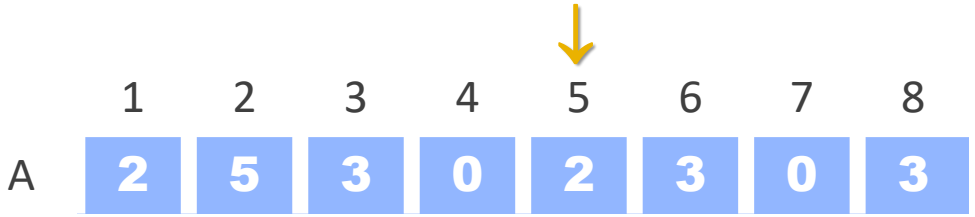
Index	0	1	2	3	4	5
C	1	2	4	6	7	8

Stability is maintained !

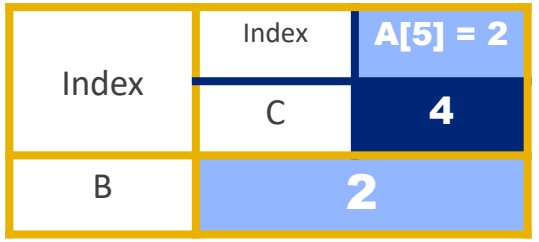
Index	1	2	3	4	5	6	7	8
B		0				[3]	3	

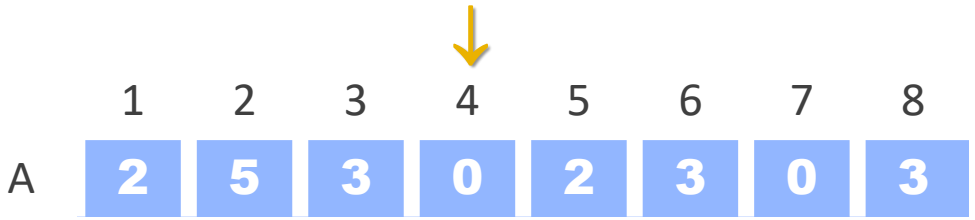
Index	0	1	2	3	4	5
C	1	2	4	5	7	8

(e)

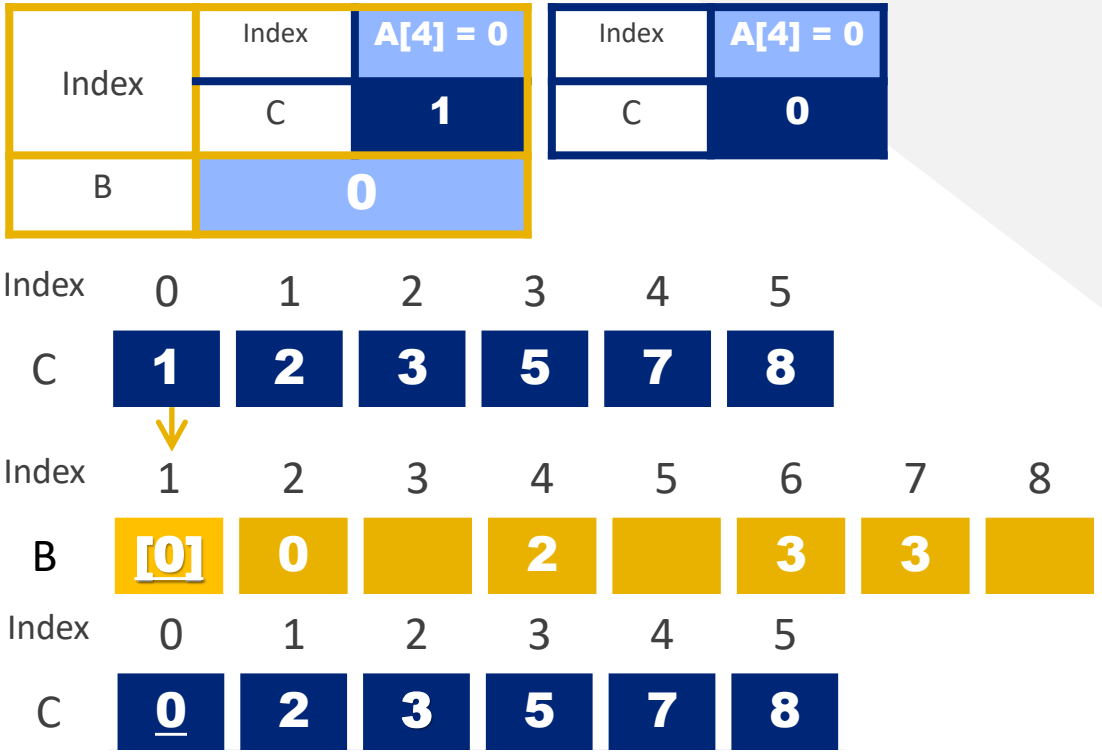


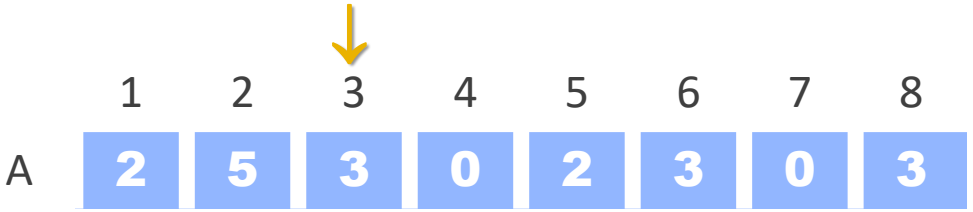

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 to k
 2       do C[i] ← 0
 3  for j ← 1 to length[A]
 4       do C[A[j]] ← C[A[j]] + 1
 5  ▷ C[i] now contains the number of elements equal to i.
 6  for i ← 1 to k
 7       do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] now contains the number of elements less than or 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
```



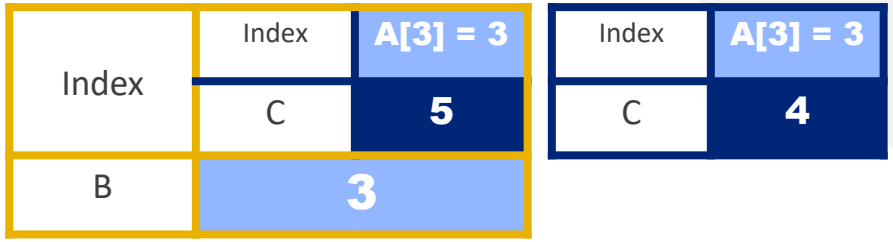


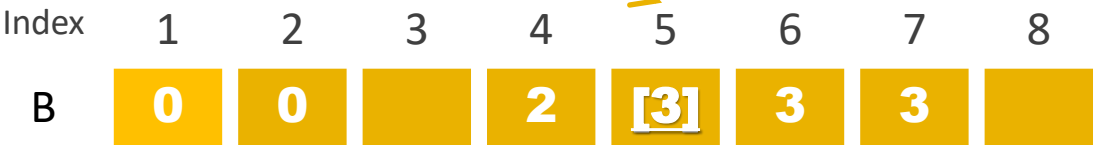

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 to k
 2       do C[i] ← 0
 3  for j ← 1 to length[A]
 4       do C[A[j]] ← C[A[j]] + 1
 5  ▷ C[i] now contains the number of elements equal to i.
 6  for i ← 1 to k
 7       do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] now contains the number of elements less than or 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
```

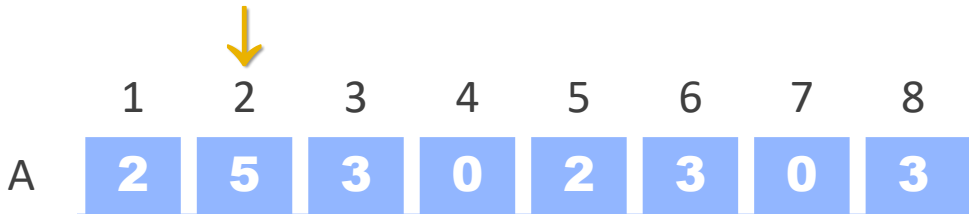




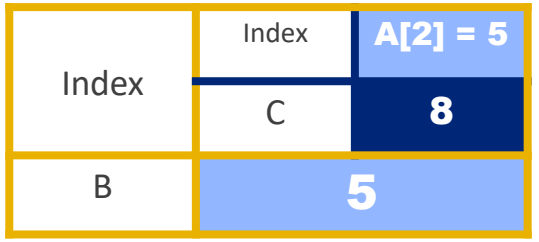

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 to k
 2       do C[i] ← 0
 3  for j ← 1 to length[A]
 4       do C[A[j]] ← C[A[j]] + 1
 5  ▷ C[i] now contains the number of elements equal to i.
 6  for i ← 1 to k
 7       do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] now contains the number of elements less than or 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
```

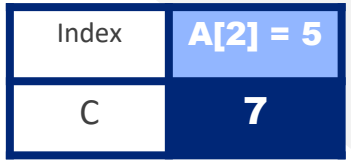


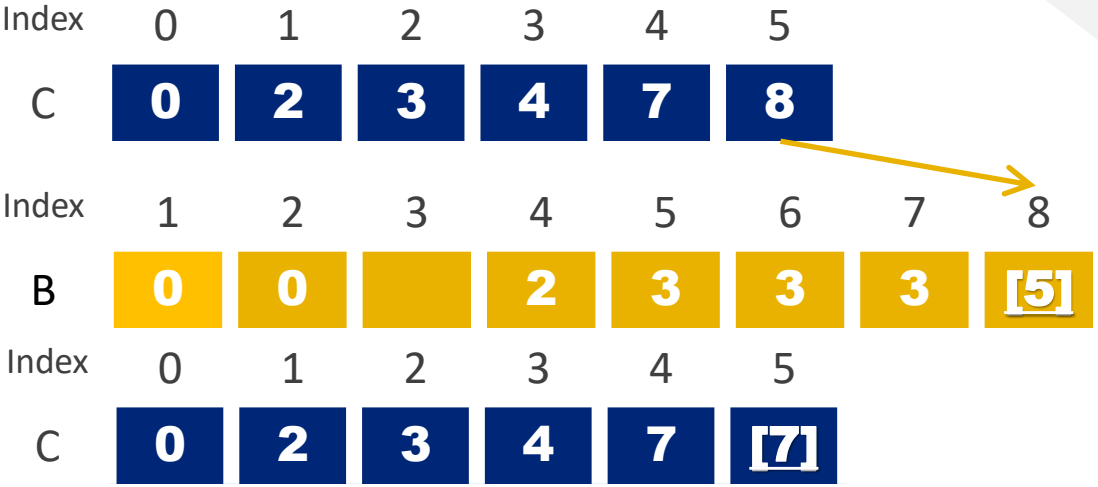





```
COUNTING-SORT(A, B, k)
 1  for i ← 0 to k
 2       do C[i] ← 0
 3  for j ← 1 to length[A]
 4       do C[A[j]] ← C[A[j]] + 1
 5  ▷ C[i] now contains the number of elements equal to i.
 6  for i ← 1 to k
 7       do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] now contains the number of elements less than or 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
```







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

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 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 ← 1 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
```

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

(f)

Index	Index	A[1] = 2
	C	3
B	2	

Index	A[1] = 2
C	2

Index	0	1	2	3	4	5
C	0	2	3	4	7	7

Index	1	2	3	4	5	6	7	8
B	0	0	[2]	2	3	3	3	5

Index	0	1	2	3	4	5
C	0	2	[2]	4	7	7

(f)

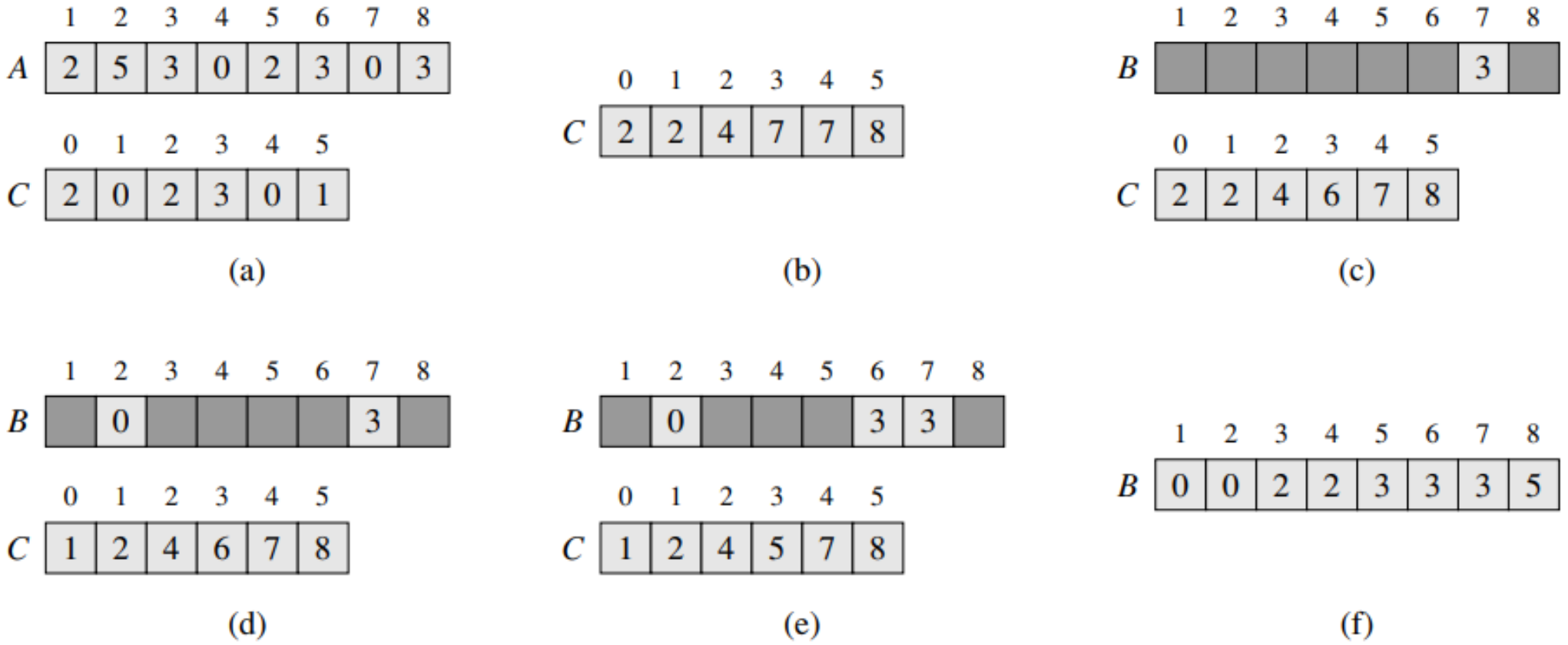


Fig 8-2 The operation of Counting-Sort on an input array A[1..8] [1]

```
1
2   def Counting_Sort(A,B,k):
3       # Initialize each elements count as 0
4       C = [0 for i in range(0, k +1)];
5       # C[i] now contins the number of elements equal to i.
6       # Finds frequency of each element in array A
7       for i in range(0, len(A)):
8           C[A[i]] += 1;
9       # C[i] now contains the number of elements less than or
          equal to i
10      # Adds elements in sorted order based on frequency.
11      for i in range(0, k+1):
12          while (C[i]>0):
13              B.append(i);
14              C[i] -= 1;
15      return B;
16
17  if __name__ == "__main__":
18      #Assign array A and B
19      A = [2, 5, 3, 0, 2, 3, 0, 3];
20      B=[];
21      #Finds max val in the list
22      k=max(A);
23      print("Array A : %s" % str(A));
24      print("Sorted array B : %s" % str(Counting_Sort(A,B,k)));
```

```
Array A : [2, 5, 3, 0, 2, 3, 0, 3]
Sorted array B : [0, 0, 2, 2, 3, 3, 3, 5]
```

COUNTING-SORT(A, B, k)

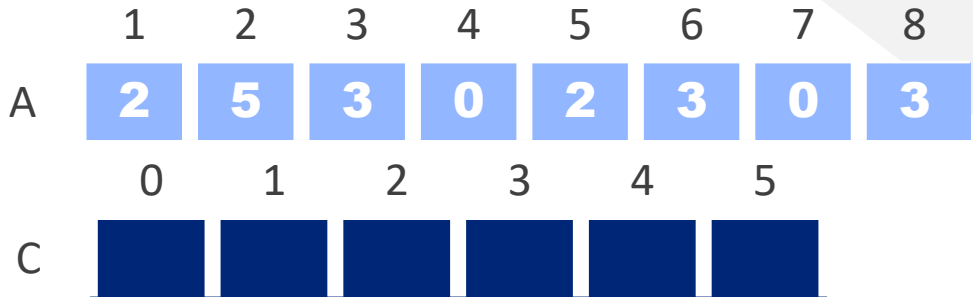
```
1  for i ← 0 to k
2      do C[i] ← 0
3  for j ← 1 to length[A]
4      do C[A[j]] ← C[A[j]] + 1
5  ▷ C[i] now contains the number of elements equal to i.
6  for i ← 1 to k
7      do C[i] ← C[i] + C[i − 1]
8  ▷ C[i] now contains the number of elements less than or equal to i.
9  for j ← length[A] downto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

Counting sort concept

Counting sort running time

```
COUNTING-SORT(A, B, k)
1  for i ← 0 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 ← 1 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
```

- Line 1-2, running time is $\Theta(k)$.
  - If k=5 of an eight-key array



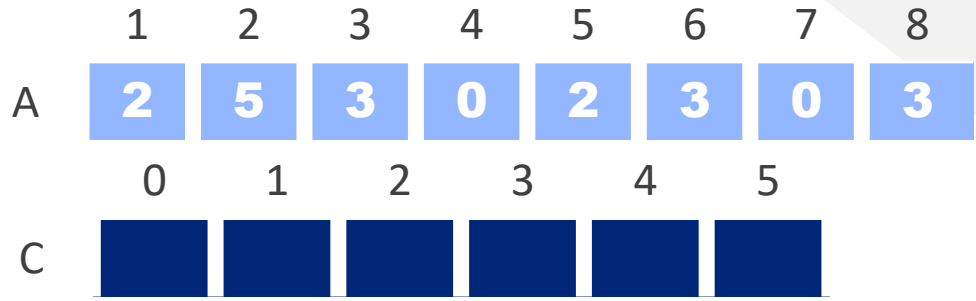
  - If k=1,000,000 of a three-data array

Counting sort concept

Counting sort running time

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 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 ← 1 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
```

- Line 3-4, running time is $\Theta(n)$.
  - If n=8,



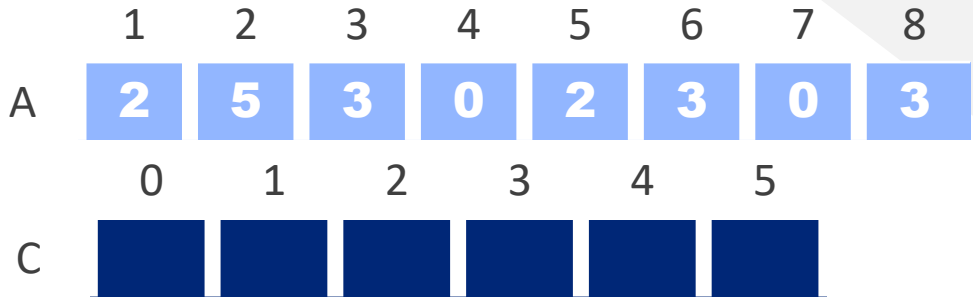
  - If n=3,

Counting sort concept

Counting sort running time

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 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 ← 1 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
```

- Line 6-7, running time is $\Theta(k)$.
  - If k=5,



  - If k=1,000,000

Counting sort concept

Counting sort running time

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 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 ← 1 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
```

- Line 9-11, running time is $\Theta(n)$.
  - If n=8,

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

  - If n=3

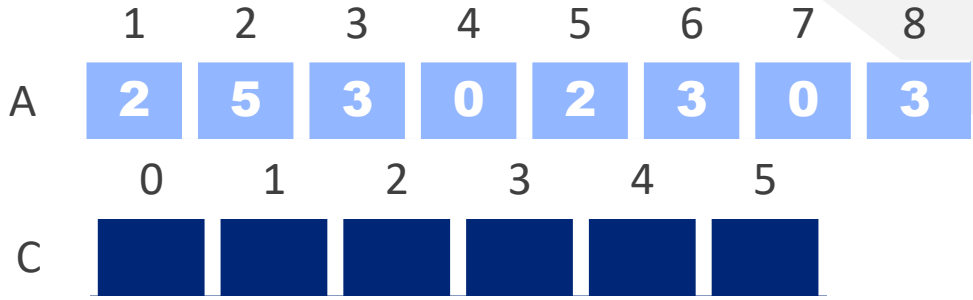
	1	2	3
A	5	1,000,000	300

Counting sort concept

Counting sort running time

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 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 ← 1 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
```

- Overall running time is $\Theta(n+k)$.
  - If k=5,



  - If k=1,000,000

Counting sort concept

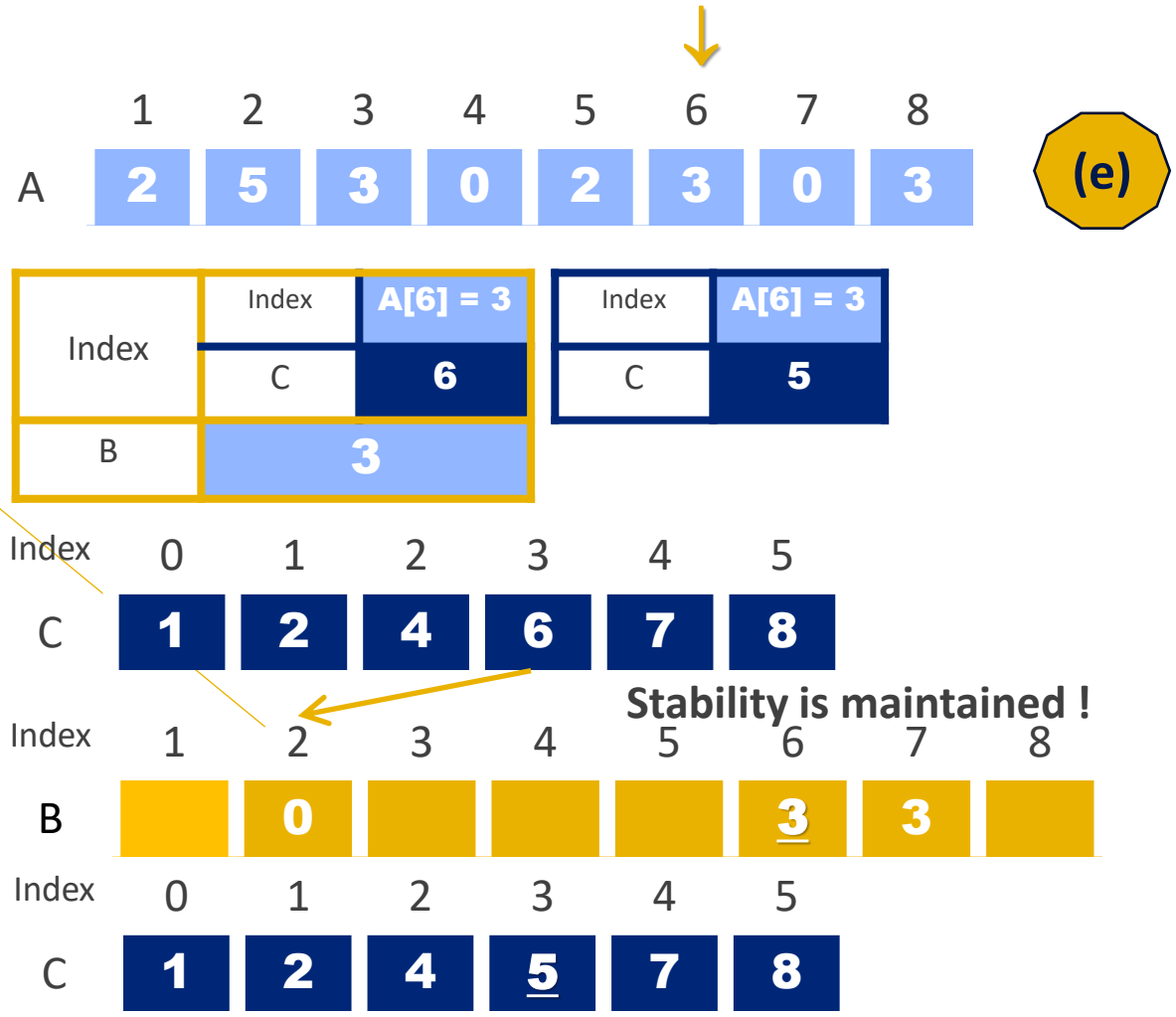
Counting sort running time

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 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 ← 1 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 beats the lower bound of $\Omega(n \lg n)$! **[1]**
  - Because it is not a comparison sort.
  - No comparisons between inputs elements occur anywhere in the code.
  - The $\Omega(n \lg n)$ lower bound for sorting does not apply when we depart from the comparison-sort model.

Counting sort concept

Counting sort stability



- Counting sort is stable ! [1]
  - Number with the same value appear in the output array in the same order as they do in the input array.
- Stability is important only when satellite data are carried around with the element being sort.
- Thus, it is often used as a subroutine in the radix sort

Counting sort strength and weakness

Counting sort

- Strength:
  - It is asymptotically faster than comparison-based sorting algorithms like quicksort or merge sort.
- Weakness:
  - It works when data ranges are known and not large.
  - When the data range is large, space cost and spent time of counting array (array C) is high, respectively.

8.3

Radix sort

1) Radix sort concept
2) Radix sort algorithm
3) Radix sort running time
4) Radix sort and Comparison sort

Radix sort concept

Radix sort

- It sorts each place by starting from the least significant place first – such place may be:
  - Digit of d-digit number (unit, ten, hundred, …),
  - Field in a work sheet (day, month, year, …)

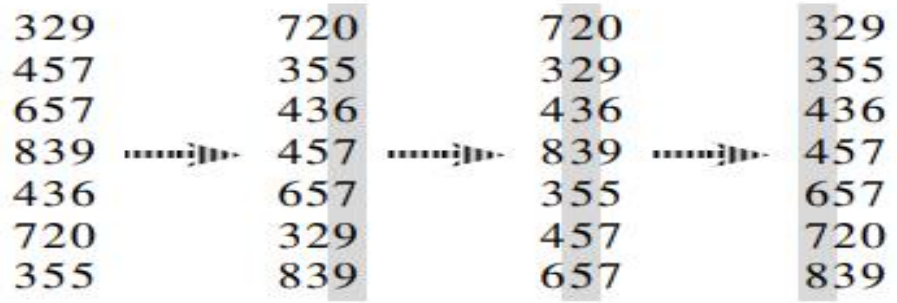


Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

Radix sort algorithm

Radix sort

```
RADIX-SORT(A, d)
1  for i ← 1 to d
2      do use a stable sort to sort array A on digit i
```

[1]

	1	2	3	4	5	6	7		
A	329	457	657	839	436	720	355	,d=	100 10 1

```
1  def Radix_Sort(A):
2      # Get maximum element
3      max_element = max(A)
4      # Apply counting sort to sort elements based on
         place value.
5      d = 1
6      while max_element // d > 0:
7          Counting_Sort(A, d)
8          d *= 10
```

1. Find the maximum element in the input array in order to go through all significant places. (max_element = 839)
2. Go through each digit. (d = 1)

```python
# Using counting sort to sort the elements in the basis of significant places
def Counting_Sort(A, d):
    size = len(A)
    B = [0] * size
    C = [0] * 10
    # Calculate count of elements (C[])
    for i in range(0, size):
        index = A[i] // d
        C[index % 10] += 1
    print("C[] =",C)
    # Calculate cummulative count (C'[])
    for i in range(1, 10):
        C[i] += C[i - 1]
    print("C'[] =",C)
    # Place the elements in sorted order
    i = size - 1
    print("A[] =",A)
    while i >= 0:
        index = A[i] // d
        B[C[index % 10] - 1] = A[i]
        print("B[] =",B)
        C[index % 10] -= 1
        print("C[] =",C)
        i -= 1
    for i in range(0, size):
        A[i] = B[i]
#Main
A = [329, 457, 657, 839, 436, 720, 355]
Radix Sort(A)
```

```
d = 1
C[] = [1, 0, 0, 0, 0, 1, 1, 2, 0, 2]
C'[] = [1, 1, 1, 1, 1, 2, 3, 5, 5, 7]
A[] = [329, 457, 657, 839, 436, 720, 355]
B[] = [0, 355, 0, 0, 0, 0, 0]
C[] = [1, 1, 1, 1, 1, 1, 3, 5, 5, 7]
B[] = [720, 355, 0, 0, 0, 0, 0]
C[] = [0, 1, 1, 1, 1, 1, 3, 5, 5, 7]
B[] = [720, 355, 436, 0, 0, 0, 0]
C[] = [0, 1, 1, 1, 1, 1, 2, 5, 5, 7]
B[] = [720, 355, 436, 0, 0, 0, 839]
C[] = [0, 1, 1, 1, 1, 1, 2, 5, 5, 6]
B[] = [720, 355, 436, 0, 657, 0, 839]
C[] = [0, 1, 1, 1, 1, 1, 2, 4, 5, 6]
B[] = [720, 355, 436, 457, 657, 0, 839]
C[] = [0, 1, 1, 1, 1, 1, 2, 3, 5, 6]
B[] = [720, 355, 436, 457, 657, 329, 839]
C[] = [0, 1, 1, 1, 1, 1, 2, 3, 5, 5]
d = 10
C[] = [0, 0, 2, 2, 0, 3, 0, 0, 0, 0]
C'[] = [0, 0, 2, 4, 4, 7, 7, 7, 7, 7]
A[] = [720, 355, 436, 457, 657, 329, 839]
B[] = [0, 0, 0, 839, 0, 0, 0]
C[] = [0, 0, 2, 3, 4, 7, 7, 7, 7, 7]
B[] = [0, 329, 0, 839, 0, 0, 0]
C[] = [0, 0, 1, 3, 4, 7, 7, 7, 7, 7]
B[] = [0, 329, 0, 839, 0, 0, 657]
C[] = [0, 0, 1, 3, 4, 6, 7, 7, 7, 7]
B[] = [0, 329, 0, 839, 0, 457, 657]
C[] = [0, 0, 1, 3, 4, 5, 7, 7, 7, 7]
B[] = [0, 329, 436, 839, 0, 457, 657]
C[] = [0, 0, 1, 2, 4, 5, 7, 7, 7, 7]
B[] = [0, 329, 436, 839, 355, 457, 657]
C[] = [0, 0, 1, 2, 4, 4, 7, 7, 7, 7]
B[] = [720, 329, 436, 839, 355, 457, 657]
C[] = [0, 0, 0, 2, 4, 4, 7, 7, 7, 7]
```

```python
# Using counting sort to sort the elements in the basis of significant places
def Counting_Sort(A, d):
    size = len(A)
    B = [0] * size
    C = [0] * 10
    # Calculate count of elements (C[])
    for i in range(0, size):
        index = A[i] // d
        C[index % 10] += 1
    print("C[] =",C)
    # Calculate cummulative count (C'[])
    for i in range(1, 10):
        C[i] += C[i - 1]
    print("C'[] =",C)
    # Place the elements in sorted order
    i = size - 1
    print("A[] =",A)
    while i >= 0:
        index = A[i] // d
        B[C[index % 10] - 1] = A[i]
        print("B[] =",B)
        C[index % 10] -= 1
        print("C[] =",C)
        i -= 1
    for i in range(0, size):
        A[i] = B[i]
A = [329, 457, 657, 839, 436, 720, 355]
Radix_Sort(A)
print("Sorted array:", A)
```

```
d = 10
C[] = [0, 0, 2, 2, 0, 3, 0, 0, 0, 0]
C'[] = [0, 0, 2, 4, 4, 7, 7, 7, 7, 7]
A[] = [720, 355, 436, 457, 657, 329, 839]
B[] = [0, 0, 0, 839, 0, 0, 0]
C[] = [0, 0, 2, 3, 4, 7, 7, 7, 7, 7]
B[] = [0, 329, 0, 839, 0, 0, 0]
C[] = [0, 0, 1, 3, 4, 7, 7, 7, 7, 7]
B[] = [0, 329, 0, 839, 0, 0, 657]
C[] = [0, 0, 1, 3, 4, 6, 7, 7, 7, 7]
B[] = [0, 329, 0, 839, 0, 457, 657]
C[] = [0, 0, 1, 3, 4, 5, 7, 7, 7, 7]
B[] = [0, 329, 436, 839, 0, 457, 657]
C[] = [0, 0, 1, 2, 4, 5, 7, 7, 7, 7]
B[] = [0, 329, 436, 839, 355, 457, 657]
C[] = [0, 0, 1, 2, 4, 4, 7, 7, 7, 7]
B[] = [720, 329, 436, 839, 355, 457, 657]
C[] = [0, 0, 0, 2, 4, 4, 7, 7, 7, 7]
d = 100
C[] = [0, 0, 0, 2, 2, 0, 1, 1, 1, 0]
C'[] = [0, 0, 0, 2, 4, 4, 5, 6, 7, 7]
A[] = [720, 329, 436, 839, 355, 457, 657]
B[] = [0, 0, 0, 0, 657, 0, 0]
C[] = [0, 0, 0, 2, 4, 4, 4, 6, 7, 7]
B[] = [0, 0, 0, 457, 657, 0, 0]
C[] = [0, 0, 0, 2, 3, 4, 4, 6, 7, 7]
B[] = [0, 355, 0, 457, 657, 0, 0]
C[] = [0, 0, 0, 1, 3, 4, 4, 6, 7, 7]
B[] = [0, 355, 0, 457, 657, 0, 839]
C[] = [0, 0, 0, 1, 3, 4, 4, 6, 6, 7]
B[] = [0, 355, 436, 457, 657, 0, 839]
C[] = [0, 0, 0, 1, 2, 4, 4, 6, 6, 7]
B[] = [329, 355, 436, 457, 657, 0, 839]
C[] = [0, 0, 0, 0, 2, 4, 4, 6, 6, 7]
B[] = [329, 355, 436, 457, 657, 720, 839]
C[] = [0, 0, 0, 0, 2, 4, 4, 5, 6, 7]
Sorted array: [329, 355, 436, 457, 657, 720, 839]
```

Radix sort running time

Radix sort

```
RADIX-SORT(A, d)
1  for i ← 1 to d
2      do use a stable sort to sort array A on digit i
```

[1]

	1	2	3	4	5	6	7	, d=
A	329	457	657	839	436	720	355	100
								10
								1

```python
def Radix_Sort(A):
    # Get maximum element
    max_element = max(A)
    # Apply counting sort to sort elements based on
      place value.
    d = 1
    while max_element // d > 0:
        Counting_Sort(A, d)
        d *= 10
```

Lemma 8.3 [1]

Given n d-digit numbers,

each digit takes k possible values,

It sorts n numbers in: $\Theta(d(n+k))$

Lemma 8.4 [1]

Given n b-bit numbers,

any positive integer $r \leq b \rightarrow d = \lceil b/r \rceil$,

It sorts n numbers in: $\Theta((b/r)(n+2^r))$

Radix sort and Comparison sort

Radix sort

- If b often takes O(lg n) so:
  - $r \leq b \approx \lg n$, then radix sort's running time is $\Theta(n)$.
  - It is better than quicksort's average case time of $\Theta(n \lg n)$.

- Although radix sort may make fewer passes than quicksort over n-input number, each pass may require significantly longer

Lemma 8.4 [1]

Given n b-bit numbers,

any positive integer $r \leq b$ -> $d = \lceil b/r \rceil$,

It sorts n numbers in: $\Theta((b/r)(n+2^r))$

Radix sort and Comparison sort

Radix sort

- Is radix sort preferable to a comparison sort – such as Quicksort ? [1]

1. It depends on the input data and the characteristic of implementations, of the underlying machine.
   - E.g., quicksort often uses hardware caches more effectively than radix sort!

2. The version of radix sort that uses counting sort as the intermediate stable sort -> does not sort in place, which many of the $\Theta(n \lg n)$-time comparison sort do.
   - When primary memory storage is a premium, an in-place algorithm such as quick sort may be preferable!

8.4

Bucket sort

1) Concept
2) Algorithm
3) Running time

Bucket sort concept

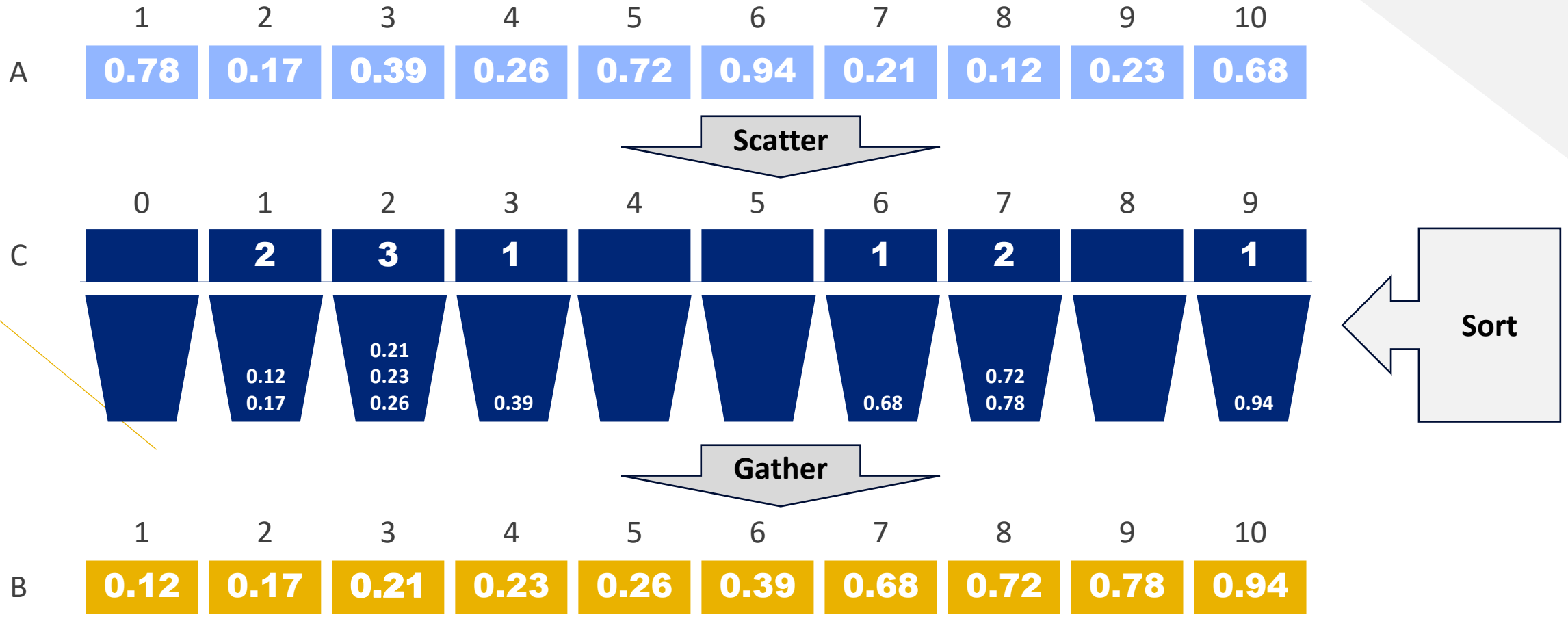
Bucket sort

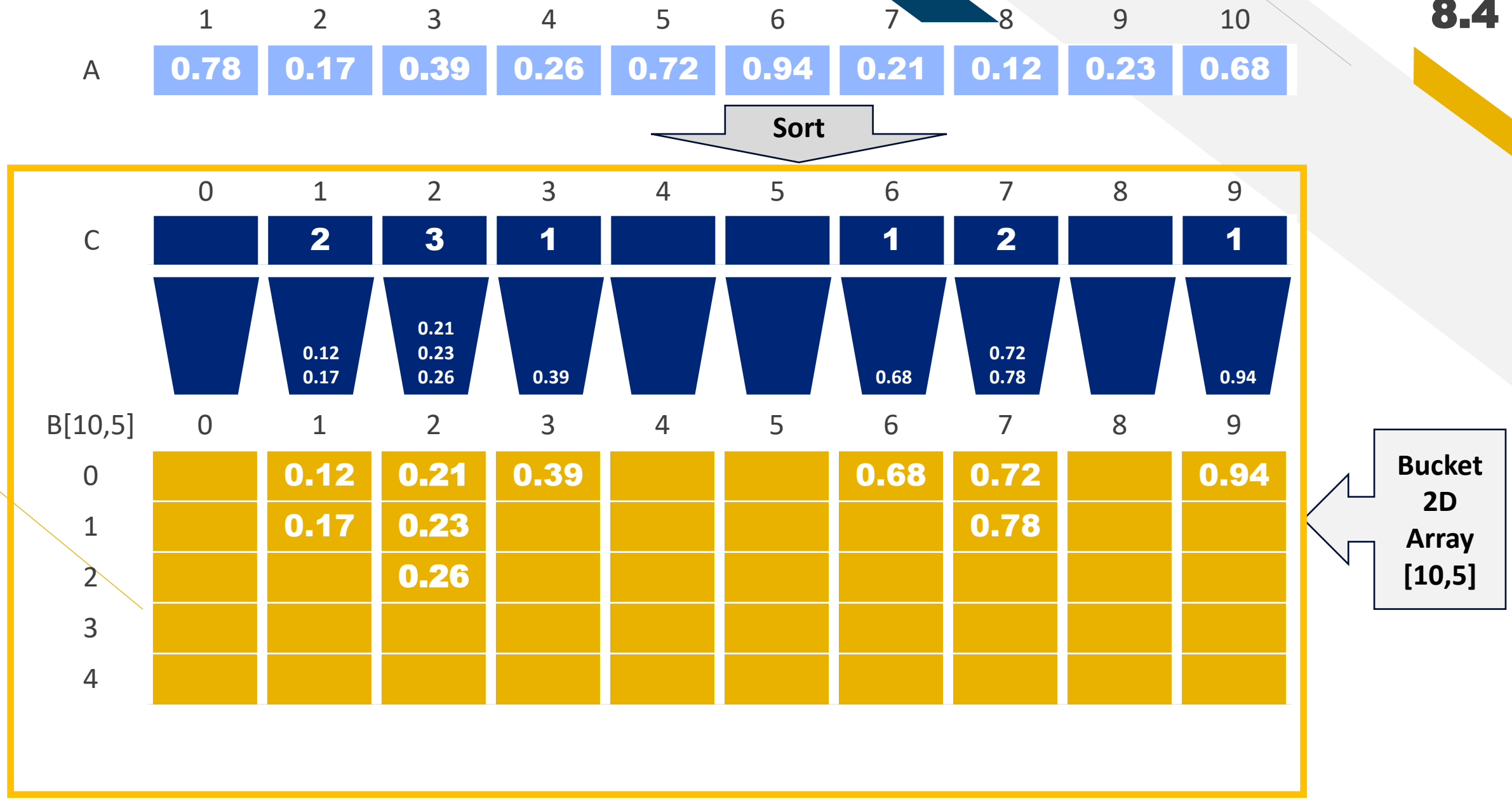
- It runs in linear time when the input is drawn from a uniform distribution. [1]
- Like counting sort, it is fast because it assumes something about the input (None of comparison occurs).

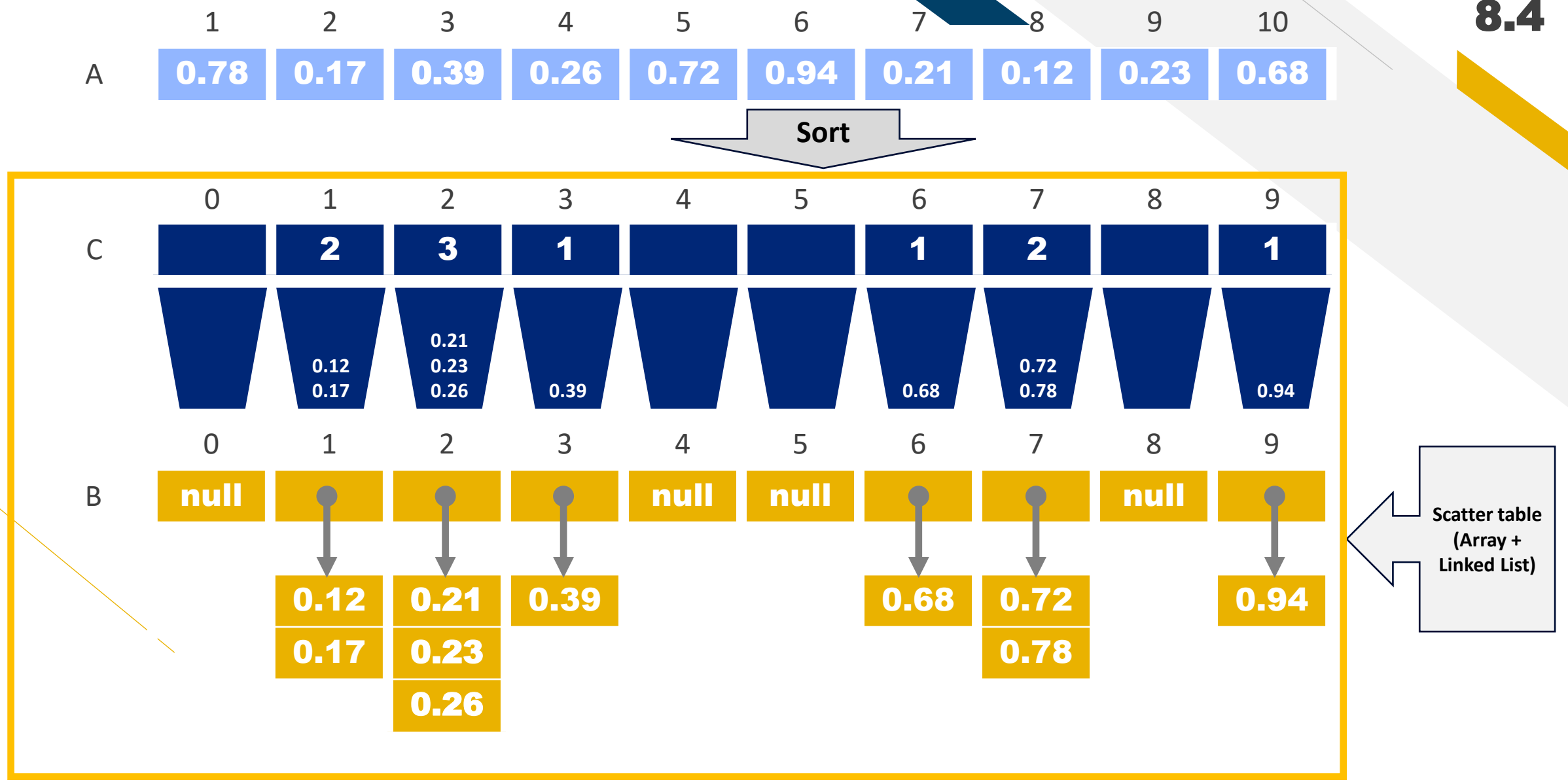
Bucket sort	Counting sort
It assumes that the input is generated by a random process that distribute elements uniformly over the interval [0,1).	It assumes that the input consists of integers in a small range.

Bucket sort algorithm

Bucket sort




1 2 3 4 5 6 7 8 9 10
A
0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68
Sort
0 1 2 3 4 5 6 7 8 9
C
2 3 1 1 2 1
0.12 0.17
0.21 0.23 0.26
0.39
0.68
0.72 0.78
0.94
B[10,5]
0 1 2 3 4 5 6 7 8 9
0 0.12 0.21 0.39 0.68 0.72 0.94
1 0.17 0.23 0.78
2 0.26
3
4
Bucket 2D Array [10,5]


1 2 3 4 5 6 7 8 9 10
A
0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68
Sort
0 1 2 3 4 5 6 7 8 9
C
2 3 1 1 2 1
0.12
0.17
0.21
0.23
0.26
0.39
0.68
0.72
0.78
0.94
0 1 2 3 4 5 6 7 8 9
B
null null null null
0.12
0.17
0.21
0.23
0.26
0.39
0.68
0.72
0.78
0.94
Scatter table (Array + Linked List)

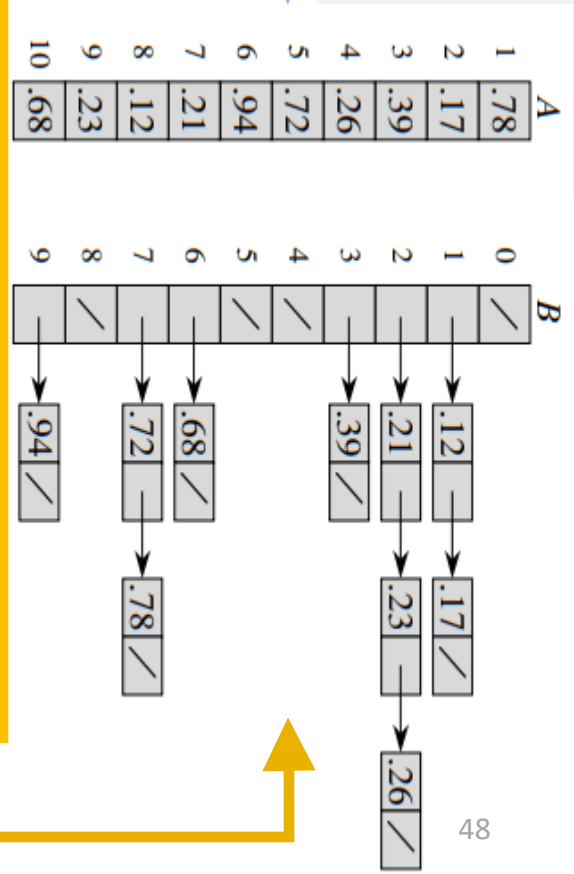
8.4

	1	2	3	4	5	6	7	8	9	10
A	0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

Sort

C	0	1	2	3	4	5	6	7	8	9
Bucket		2	3	1			1	2		1
		0.12 0.17	0.21 0.23 0.26	0.39			0.68	0.72 0.78		0.94

B	0	1	2	3	4	5	6	7	8	9
	null	0.12 0.17	0.21 0.23 0.26	0.39	null	null	0.68	0.72 0.78	null	0.94



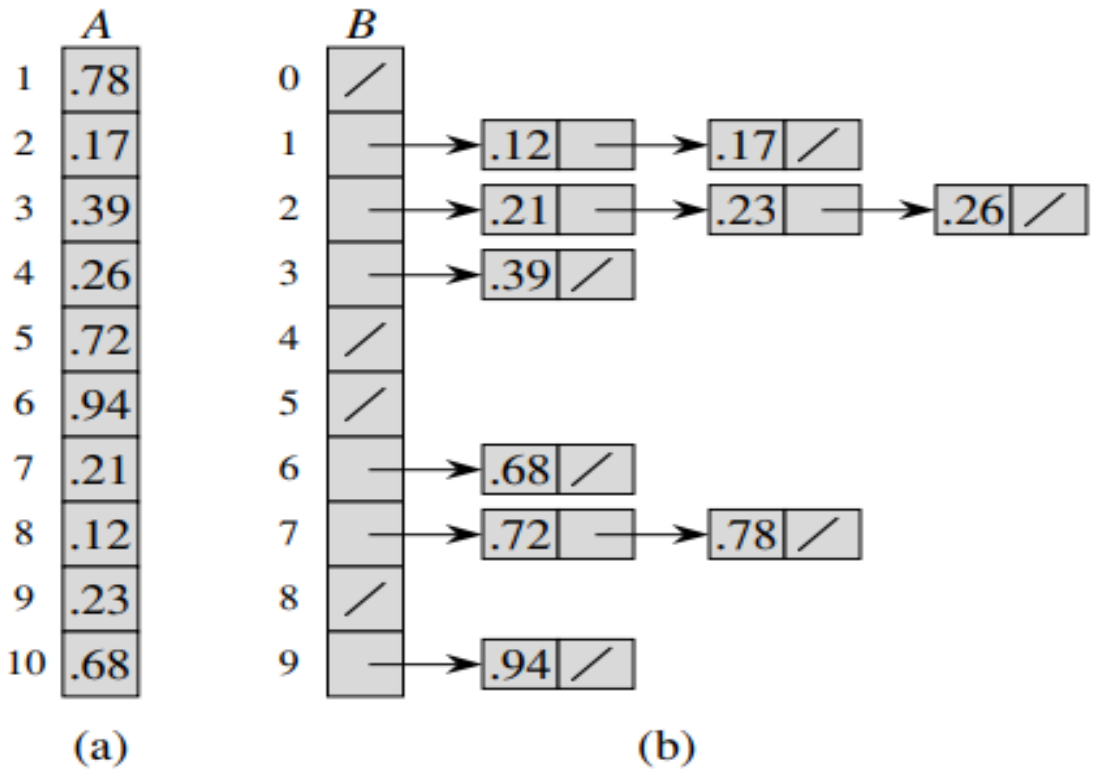


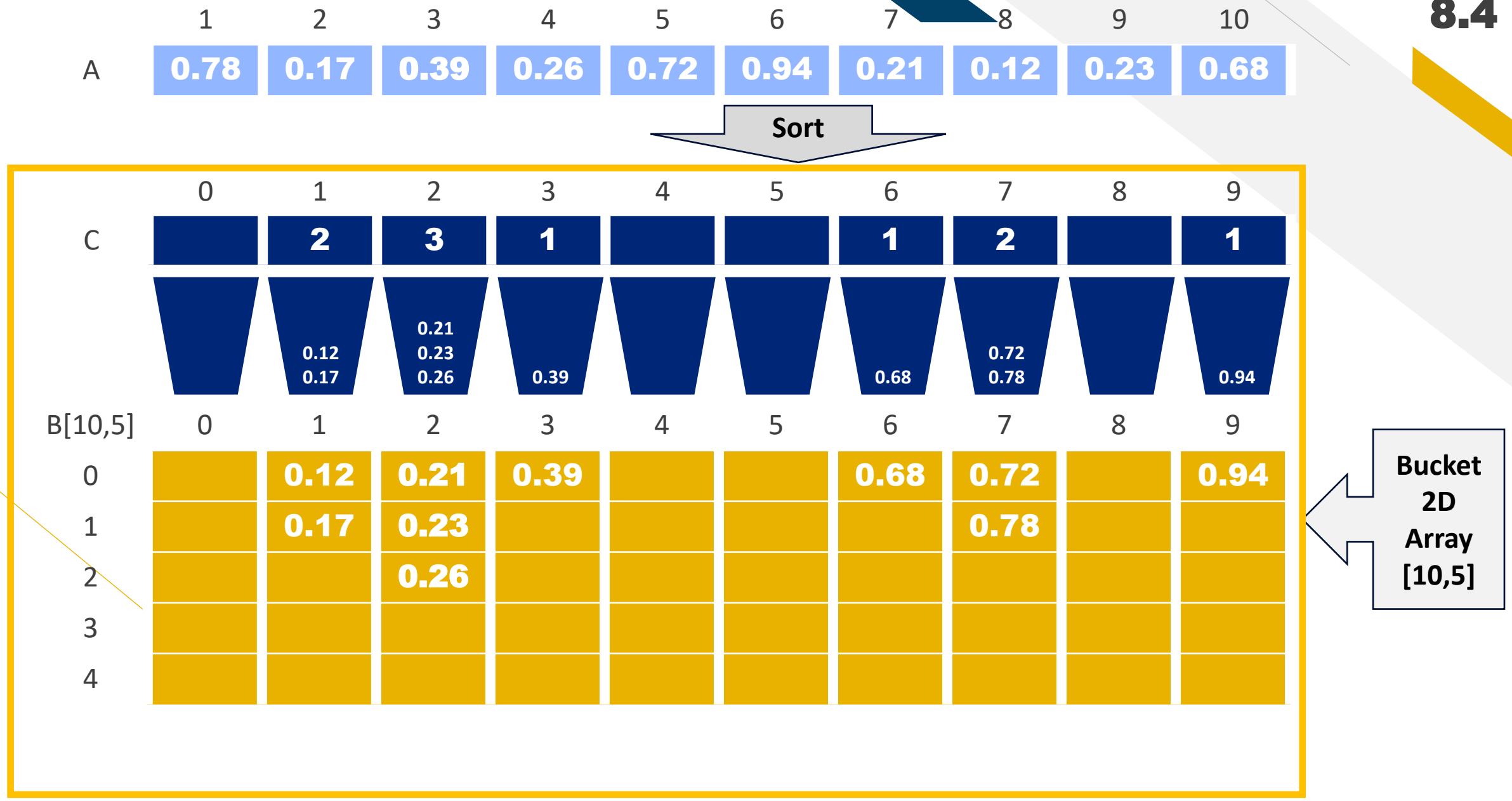
Figure 8.4 The operation of BUCKET-SORT. **(a)** The input array $A[1 \ldots 10]$. **(b)** The array $B[0 \ldots 9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i + 1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Bucket sort algorithm

Bucket sort

[1]

```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n
3        do insert A[i] into list B[⌊n A[i]⌋]
4  for i ← 0 to n − 1
5        do sort list B[i] with insertion sort
6  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```


1 2 3 4 5 6 7 8 9 10
A
0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68
Sort
0 1 2 3 4 5 6 7 8 9
C
2 3 1 1 2 1
0.12 0.17
0.21 0.23 0.26
0.39
0.68
0.72 0.78
0.94
B[10,5]
0 1 2 3 4 5 6 7 8 9
0 0.12 0.21 0.39 0.68 0.72 0.94
1 0.17 0.23 0.78
2 0.26
3
4
Bucket 2D Array [10,5]

```python
def bucketSort(A):
    # Create empty buckets to represnt array C, B
    bucket = []
    for i in range(len(A)):
        bucket.append([])
    print("Bucket[10] is constructed as :", bucket)
    # Insert elements into their respective buckets B [10,5]
    for j in A:
        index_b = int(5 * j)
        bucket[index_b].append(j)
    # Sort the elements of each bucket
    for i in range(len(A)):
        bucket[i] = sorted(bucket[i])
        print("Bucket[10,5] =", bucket)
    # Get the sorted elements
    k = 0
    for i in range(len(A)):
        for j in range(len(bucket[i])):
            A[k] = bucket[i][j]
            k += 1
    return A

A = [.78, .17, .39, .26, .72, .94, .21, .12, .23, .68]
print("Input array =", A)
print("Sorted array =",bucketSort(A))
```

```
Input array = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]
Bucket[10] is constructed as : [[], [], [], [], [], [], [], [], [], []]
The thread 'MainThread' (0x1) has exited with code 0 (0x0).
Bucket[10,5] = [[0.12, 0.17], [0.39, 0.26, 0.21, 0.23], [], [0.78, 0.72, 0.68], [0.94], [], [], [], [], []]
Bucket[10,5] = [[0.12, 0.17], [0.21, 0.23, 0.26, 0.39], [], [0.78, 0.72, 0.68], [0.94], [], [], [], [], []]
Bucket[10,5] = [[0.12, 0.17], [0.21, 0.23, 0.26, 0.39], [], [0.78, 0.72, 0.68], [0.94], [], [], [], [], []]
Bucket[10,5] = [[0.12, 0.17], [0.21, 0.23, 0.26, 0.39], [], [0.68, 0.72, 0.78], [0.94], [], [], [], [], []]
Bucket[10,5] = [[0.12, 0.17], [0.21, 0.23, 0.26, 0.39], [], [0.68, 0.72, 0.78], [0.94], [], [], [], [], []]
Bucket[10,5] = [[0.12, 0.17], [0.21, 0.23, 0.26, 0.39], [], [0.68, 0.72, 0.78], [0.94], [], [], [], [], []]
Bucket[10,5] = [[0.12, 0.17], [0.21, 0.23, 0.26, 0.39], [], [0.68, 0.72, 0.78], [0.94], [], [], [], [], []]
Bucket[10,5] = [[0.12, 0.17], [0.21, 0.23, 0.26, 0.39], [], [0.68, 0.72, 0.78], [0.94], [], [], [], [], []]
Bucket[10,5] = [[0.12, 0.17], [0.21, 0.23, 0.26, 0.39], [], [0.68, 0.72, 0.78], [0.94], [], [], [], [], []]
Bucket[10,5] = [[0.12, 0.17], [0.21, 0.23, 0.26, 0.39], [], [0.68, 0.72, 0.78], [0.94], [], [], [], [], []]
Sorted array = [0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94]
The program 'python.exe' has exited with code 0 (0x0).
```

Bucket sort running time

Bucket sort

```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n
3        do insert A[i] into list B[⌊nA[i]⌋]
4  for i ← 0 to n − 1
5        do sort list B[i] with insertion sort
6  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

- All lines except line 5 take O(n) time in the worst case!
- Line 5, total time is taken by the n calls to insertion sort -> take $O(n^2)$ time.
- Line 6, just put them into the proper order.

The expected time of bucket sort runs in linear time as:

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

Bucket sort running time

Bucket sort

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

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

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

$$\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] \quad \text{(by linearity of expectation)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) \quad \text{(by equation (C.21))} . \qquad (8.1)
\end{aligned}$$

Bucket sort running time

Bucket sort

We claim that

$$\mathrm{E}[n_i^2] = 2 - 1/n \tag{8.2}$$

for $i = 0, 1, \ldots, n-1$. It is no surprise that each bucket i has the same value of $\mathrm{E}[n_i^2]$, since each value in the input array A is equally likely to fall in any bucket. To prove equation (8.2), we define indicator random variables

$$X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$$

for $i = 0, 1, \ldots, n-1$ and $j = 1, 2, \ldots, n$. Thus, $n_i = \sum_{j=1}^{n} X_{ij}$.

Bucket sort running time

Bucket sort

To compute $\mathrm{E}[n_i^2]$, we expand the square and regroup terms:

$$\begin{aligned}
\mathrm{E}[n_i^2] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n}\mathrm{E}\left[X_{ij}^2\right] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}}\mathrm{E}\left[X_{ij}X_{ik}\right], \qquad (8.3)
\end{aligned}$$

Bucket sort running time

Bucket sort

where the last line follows by linearity of expectation. We evaluate the two summations separately. Indicator random variable X_{ij} is 1 with probability $1/n$ and 0 otherwise, and therefore

$$\begin{aligned} \mathrm{E}[X_{ij}^2] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) \\ &= \frac{1}{n} . \end{aligned}$$

When $k \neq j$, the variables X_{ij} and X_{ik} are independent, and hence

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

References

Texts | Integrated Development Environment (IDE)

[1] Introduction to Algorithms, Second Edition, Thomas H. C., Charles E. L., Ronald L. R., Clifford S., The MIT Press, McGraw-Hill Book Company, Second Edition 2001.

[2] https://visualstudio.microsoft.com/