Non Comparison based Sorting

- The sorting algorithms that we discussed share an interesting property:

the sorted order they determine is based only on comparisons between the input elements.

  - We call such sorting algorithms ***comparison sorts***.
  - All the sorting algorithms introduced thus far are comparison sorts.

(a)

	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

(b)

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

(c)

	1	2	3	4	5	6	7	8
B							3	

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

(d)

	1	2	3	4	5	6	7	8
B		0					3	

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

(e)

	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

(f)

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

COUNTING-SORT(A, B, k)

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

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

	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)

```
Count: [0, 0, 0, 0, 0, 0]
Output: [0, 0, 0, 0, 0, 0, 0, 0]
Count1: [2, 0, 2, 3, 0, 1]
Count: [2, 2, 2, 3, 0, 1]
Count: [2, 2, 4, 3, 0, 1]
Count: [2, 2, 4, 7, 0, 1]
Count: [2, 2, 4, 7, 7, 1]
Count: [2, 2, 4, 7, 7, 8]
```

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

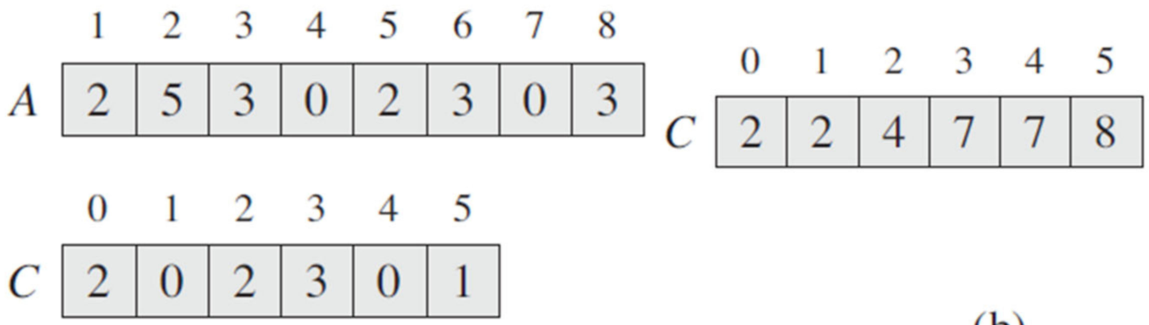
(b)

COUNTING-SORT(A, B, k)

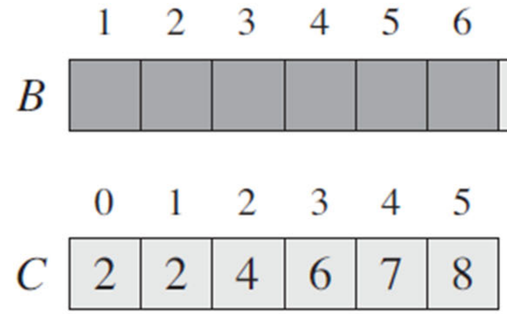
1 let $C[0..k]$ be a new array
2 **for** $i = 0$ **to** k
3 $\quad C[i] = 0$
4 **for** $j = 1$ **to** $A.length$
5 $\quad C[A[j]] = C[A[j]] + 1$
6 // $C[i]$ now contains the number of elements equal to i.
7 **for** $i = 1$ **to** k
8 $\quad C[i] = C[i] + C[i-1]$
9 // $C[i]$ now contains the number of elements less than or equal to i.
10 **for** $j = A.length$ **downto** 1
11 $\quad B[C[A[j]]] = A[j]$
12 $\quad C[A[j]] = C[A[j]] - 1$

Lines 7–8 determine for each i (0 to k) how many input elements are less than or equal to i by keeping a running sum of the array C.

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

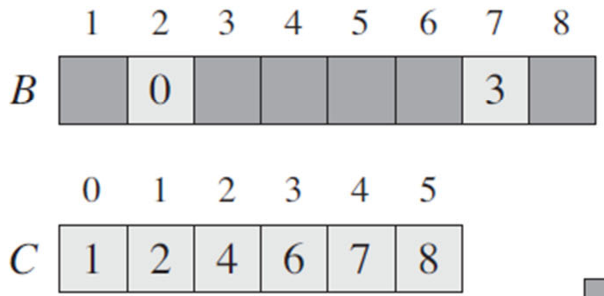


(a) (b)

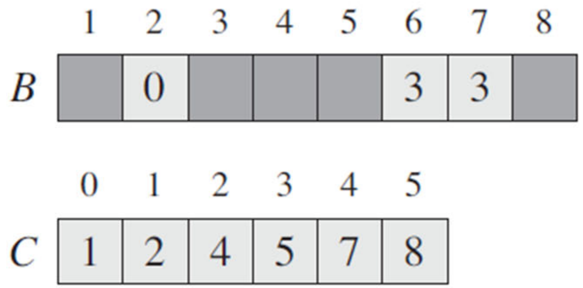


(c)

```
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```



(d)



(e)

```
C: [0, 0, 0, 0, 0, 0]
B: [0, 0, 0, 0, 0, 0, 0, 0]

B: [0, 0, 0, 0, 0, 0, 0, 0]
C: [2, 2, 4, 7, 7, 8]

B: [0, 0, 0, 0, 0, 0, 3, 0]
C: [2, 2, 4, 6, 7, 8]
B: [0, 0, 0, 0, 0, 0, 3, 0]
C: [1, 2, 4, 6, 7, 8]
B: [0, 0, 0, 0, 0, 3, 3, 0]
C: [1, 2, 4, 5, 7, 8]
B: [0, 0, 0, 2, 0, 3, 3, 0]
C: [1, 2, 3, 5, 7, 8]
B: [0, 0, 0, 2, 0, 3, 3, 0]
C: [0, 2, 3, 5, 7, 8]
B: [0, 0, 0, 2, 3, 3, 3, 0]
C: [0, 2, 3, 4, 7, 8]
B: [0, 0, 0, 2, 3, 3, 3, 5]
C: [0, 2, 3, 4, 7, 7]
B: [0, 0, 2, 2, 3, 3, 3, 5]
C: [0, 2, 2, 4, 7, 7]
```

```
[0, 0, 2, 2, 3, 3, 3, 5]
```

COUNTING-SORT(A, B, k)

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

How much time does counting sort require? The **for** loop of lines 2–3 takes time $\Theta(k)$, the **for** loop of lines 4–5 takes time $\Theta(n)$, the **for** loop of lines 7–8 takes time $\Theta(k)$, and the **for** loop of lines 10–12 takes time $\Theta(n)$. Thus, the overall time is $\Theta(k + n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.

Radix Sort Algorithm

- The process of radix sort works similar to the sorting of student names, according to the alphabetical order.

329		720		720		329
457		355		329		355
657		436		436		436
839	→	457	→	839	→	457
436		657		355		657
720		329		457		720
355		839		657		839

- Shading indicates the digit position sorted on to produce each list from the previous one.

Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n + k))$ time if the stable sort it uses takes $\Theta(n + k)$ time.

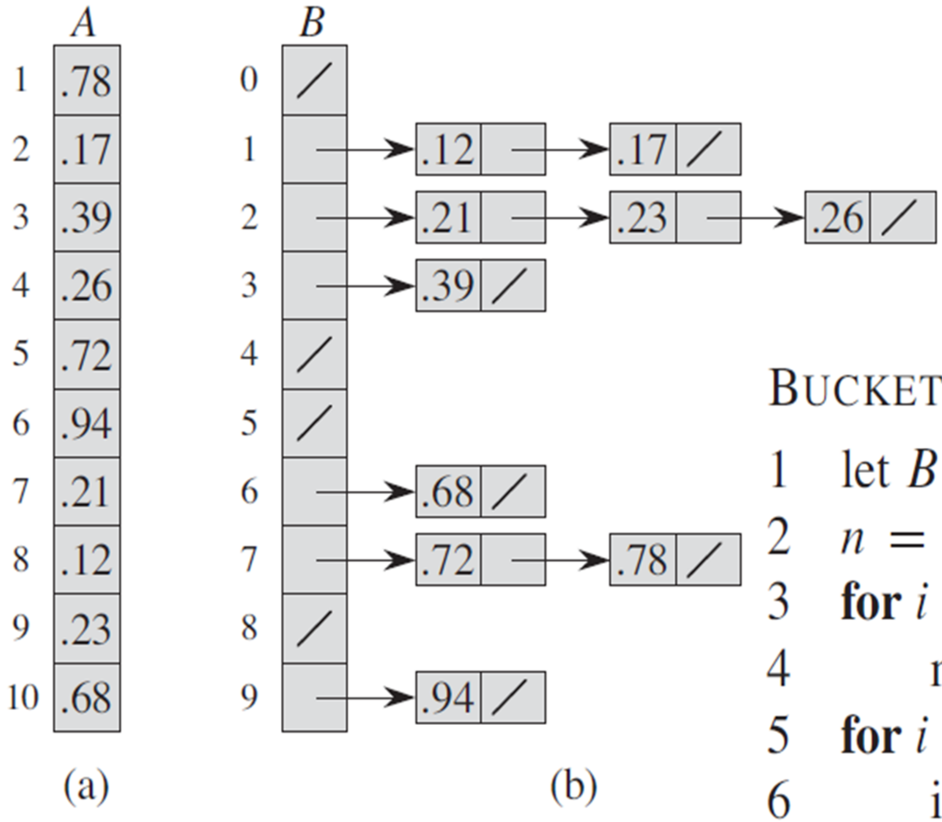
Radix Sort Algorithm

- Counting sort is often used as a subroutine in radix sort
- In order for radix sort to work correctly, counting sort must be stable.

Bucket Sort Algorithm

- Bucket sort is a sorting algorithm that works by distributing the elements of an array into a number of "buckets", and then sorting the elements within each bucket.
- It is mainly useful when the input is uniformly distributed over a range.
- The algorithm starts by creating an empty array of "buckets", where each bucket is a linked list.
- Then it iterates over the input array and places each element into the appropriate bucket based on its value.
- Once all the elements have been placed into their respective buckets, the algorithm iterates over the buckets and sorts the elements within each bucket using a different sorting algorithm.
- Finally, it concatenates all the sorted buckets to obtain the final sorted array.
- Bucket sort has an average and best-case time complexity of $O(n)$ and worst case is $O(n^2)$ but it is efficient for large number of small range of inputs.

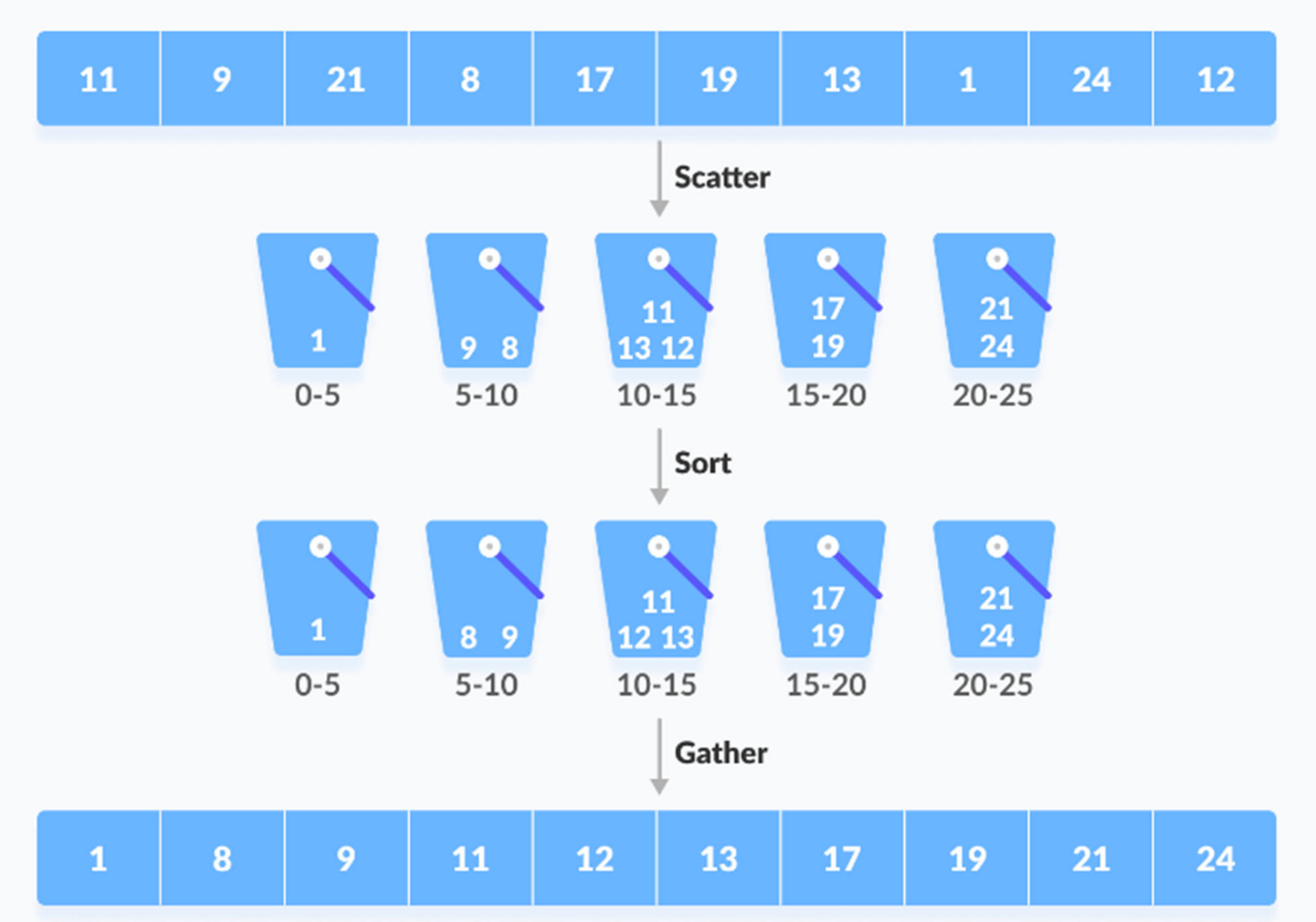
Bucket Sort Algorithm




```
BUCKET-SORT(A)
1  let B[0 .. n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4      make B[i] an empty list
5  for i = 1 to n
6      insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8      sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1 \,..\, 10]$. **(b)** The array $B[0 \,..\, 9]$ of sorted lists (buckets) after line 8 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 Algorithm

Bucket sort is used when:

- input is uniformly distributed over a range.
- there are floating point values

Bucket Sort Complexity

Time Complexity	
Best	O(n+k)
Worst	$O(n^2)$
Average	O(n)
Space Complexity	O(n+k)
Stability	Yes