Chap8.Sorting in Linear Time

- Lower bounds for sorting
- Sorting in Linear Time
  - Counting sort
  - Radix sort
  - Bucket sort

Lower Bounds for Sorting

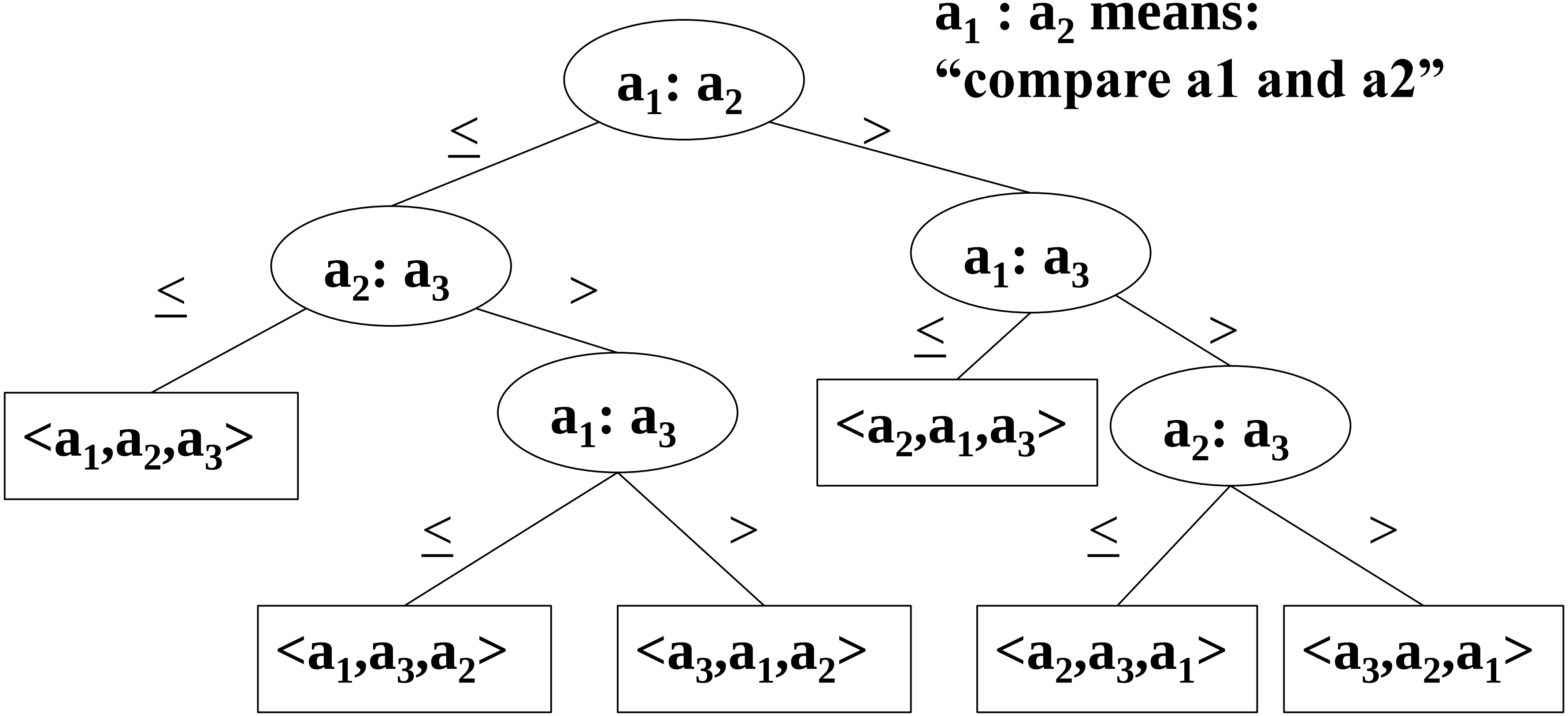
- All the sorts we have examined so far work by “key comparisons”. Elements are put in the correct place by comparing the values of the key used for sorting.
- Mergesort and Heapsort both have running time Θ (n log n)
- This is a lower bound on sorting by key comparisons

Decision Tree Model

- We can view a comparison sort abstractly by using a *decision tree.*
- The decision tree represents all possible comparisons made when sorting a list using a particular sorting algorithm
- Assume:
  - All elements are distinct.
  - All comparisons are of the form ai ≤ aj

Decision Tree for Insertion Sort (n = 3)

$a_1 : a_2$ means: “compare a1 and a2”



$< a_2,a_1,a_3 >$ means: $a2 \leq a1 \leq a3$

Lower Bounds for Comparison Sorts

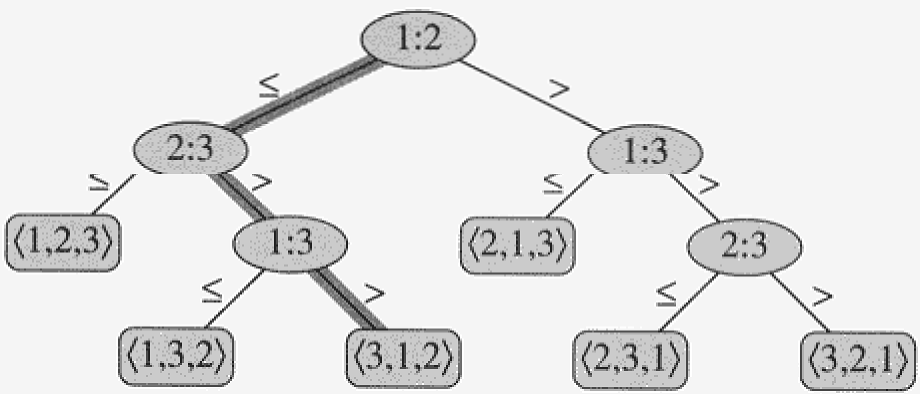


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i:j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, so the decision tree must have at least 6 leaves.

Permutations of n elements

- There are <u>n! permutations</u> of n elements
- That means that the decision tree which results in all possible permutations of elements must have n! leaves
- The longest path from the root to a leaf represents the worst case performance of the algorithm
- So the worst case performance is the height of the decision tree

Properties of decision trees

- Lemma Any binary tree of height h has $\leq 2^h$ leaves.
- Proof: By induction on h.
  Basis: $h = 0$. Tree is just one node, which is a leaf. $2^h = 1$.
  Inductive hypothesis: Assume true for height $= h-1$.
  i.e. # of leaves for height $h-1 \leq 2^{h-1}$
  Inductive step:
  - Extend tree of height $h-1$ by making as many new leaves as possible.
  - Each leaf becomes parent to two new leaves.
  - # of leaves for height h = 2 *(# of leaves for height $h-1$) $= 2 * 2^{h-1} = 2^h$.

A Lower Bound for Worst Case

- ***Theorem* 8.1:**
  Any comparison sort algorithm requires $\Omega(n \log n)$ c omparisons in the worst case.
- **Proof:**

  Consider a decision tree of height *h* with *l* leaves that sorts *n* elements.

  There are *n!* permutations of *n* elements.

  The tree must have at least *n!* leaves since each permutation of input must be a leaf.

A Lower Bound for Worst Case

A binary tree of height h has no more than 2^h leaves.

Therefore the decision tree has no more than 2^h leaves.

Thus: $n! \leq l \leq 2^h$

Therefore: $n! \leq 2^h$

Take the logarithm of both sides:

$\log(n!) \leq h$, or, equivalently, $h \geq \log(n!)$

$\boldsymbol{n! > (n/e)^n}$ by using Stirling's approximation of n!

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

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

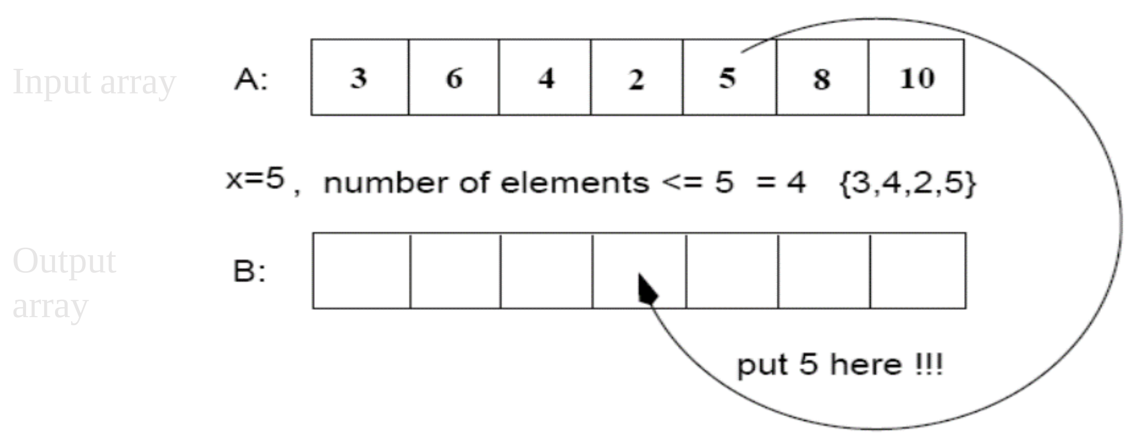
$$= n \log n - n \log e = \Omega(n \log n)$$

Can we do better?

- Linear sorting algorithms
  - Counting Sort
  - Radix Sort
  - Bucket sort
- Make certain assumptions about the data
- Linear sorts are *<u>not comparison sorts</u>*

Counting Sort

- Assumptions:
  - n integers which are in the range [0..k]
- Idea:
  - For each element x, find the number of elements $\leq$ x
  - Place x into its correct position in the output array



Counting Sort

Step 1:Find the number of times A[i] appears in A (i.e. frequencies)

input array A:

3	6	4	1	3	4	1	4

allocate C

1	2	3	4	5	6
0	0	0	0	0	0

i=1, A[1]=3

1	2	3	4	5	6
0	0	1	0	0	0

C[A[1]]=C[3]=1

i=2, A[2]=6

1	2	3	4	5	6
0	0	1	0	0	1

C[A[2]]=C[6]=1

i=3, A[3]=4

1	2	3	4	5	6
0	0	1	1	0	1

C[A[3]]=C[4]=1

⋮

i=8, A[8]=4

1	2	3	4	5	6
2	0	2	3	0	1

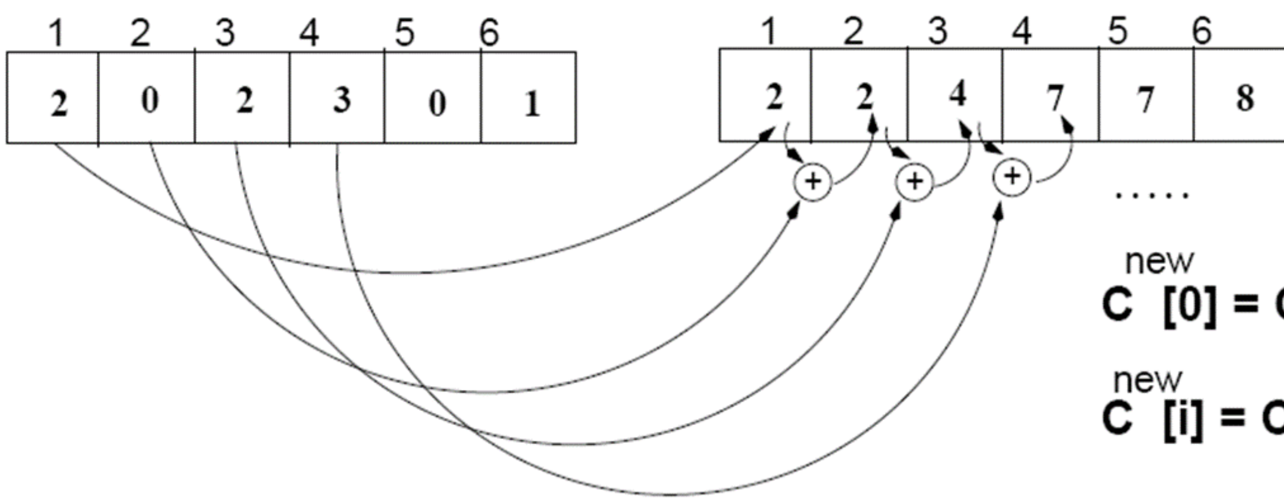
C[A[8]]=C[4]=3

C[i] = number of times element i appears in A

Counting Sort

Step 2: Find the number of elements $\leq$ A[i]

C (frequencies) C^{new} (cumulative sums)



$C^{new}[0] = C^{old}[0]$

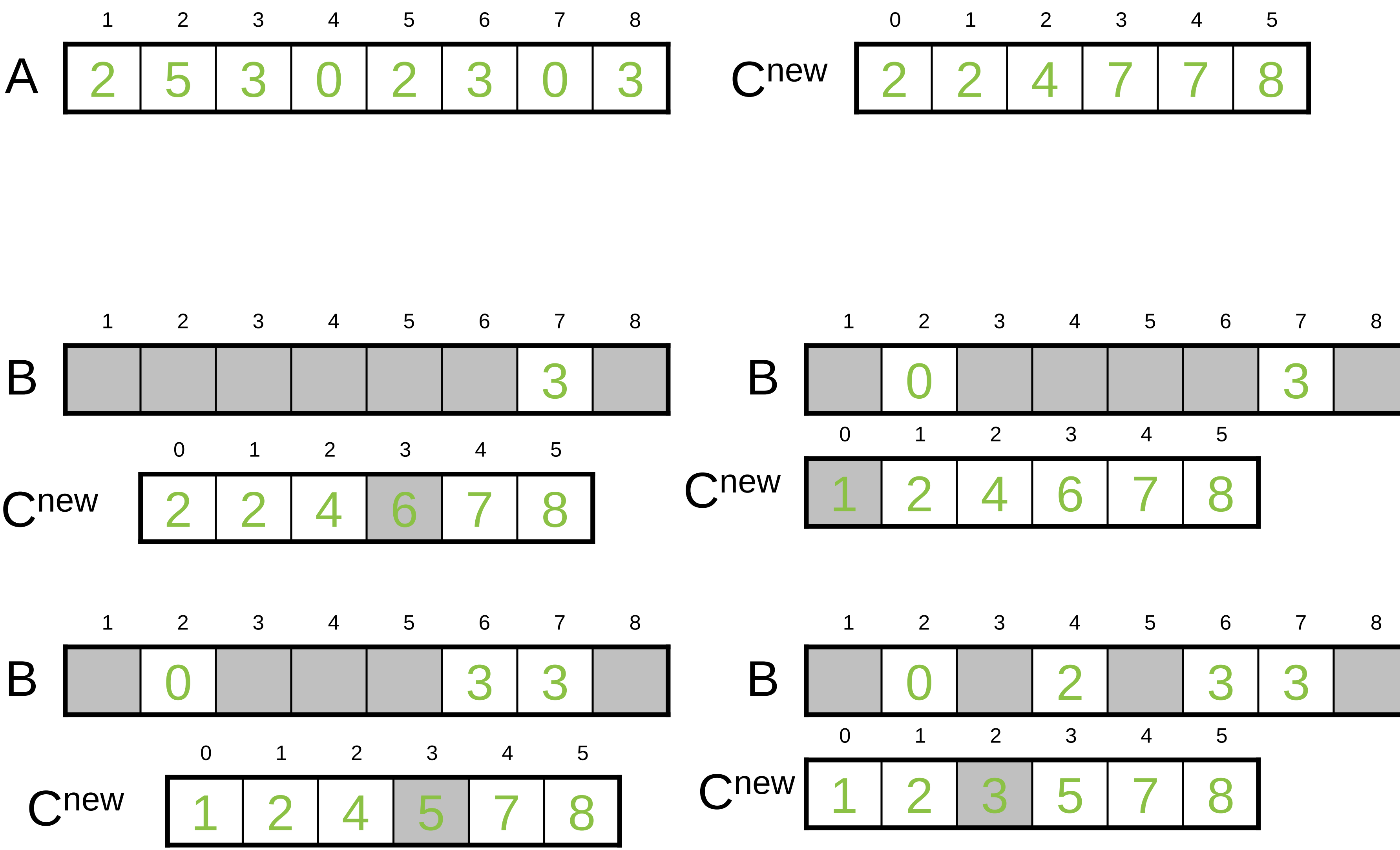
$C^{new}[i] = C^{new}[i-1] + C^{old}[i]$

C[i] = # elements <= i

Counting Sort

- Step 3.
  - Start from the last element of A
  - Place A[i] at its correct place in the output array
  - Decrease C[A[i]] by one

Example


A
1 2 3 4 5 6 7 8
2 5 3 0 2 3 0 3
C^new
0 1 2 3 4 5
2 2 4 7 7 8
B
1 2 3 4 5 6 7 8
3
C^new
0 1 2 3 4 5
2 2 4 6 7 8
B
1 2 3 4 5 6 7 8
0 3
C^new
0 1 2 3 4 5
1 2 4 6 7 8
B
1 2 3 4 5 6 7 8
0 3 3
C^new
0 1 2 3 4 5
1 2 4 5 7 8
B
1 2 3 4 5 6 7 8
0 2 3 3
C^new
0 1 2 3 4 5
1 2 3 5 7 8

Example (cont.)

A

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

B

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

C

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

B

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

C

0	1	2	3	4	5
0	2	3	4	7	7

B

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

C

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

B

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

Sorted!

Counting Sort

- Assumes each of the n input elements is an integer in the range 0 to k, for some integer k
- For each element x, determine the number of values $\leq$ x
- Requires three arrays
  - An input array A[1..n]
  - An array B[1..n] for the sorted output
  - An array C[0..k] for counting the number of times each element occurs (temporary working storage)

Counting Sort

<u>CountingSort(*A*, *B*, *k*)</u>

```
for i ← 0 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
/* C[i] now contains the number of elements equal to i. */
for i ← 1 to k
    do C[i] ← C[i] + C[i –1]
/* C[i] now contains the number of elements less than or equal to i. */
for j ← length[A] downto 1
    do B[C[A[ j ]]] ← A[j]
       C[A[j]] ← C[A[j]]–1
```

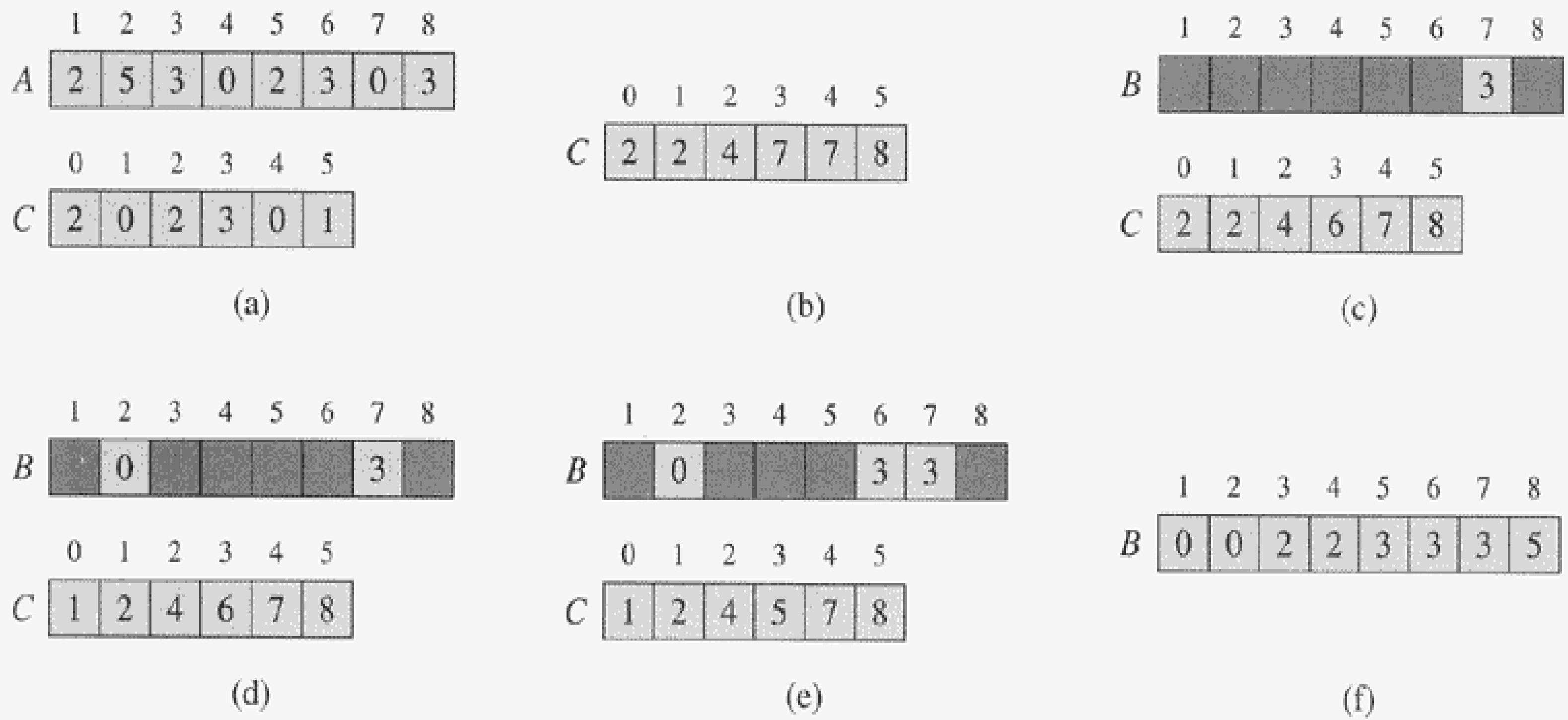


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Complexity of Counting Sort

<u>**CountingSort(*A, B, k*)**</u>

1. **for** $i \leftarrow 0$ to k
2. **do** $C[i] \leftarrow 0$
3. **for** $j \leftarrow 1$ to $length[A]$
4. **do** $C[A[j]] \leftarrow C[A[j]] + 1$
5. **for** $i \leftarrow 1$ to k
6. **do** $C[i] \leftarrow C[i] + C[i-1]$
7. **for** $j \leftarrow length[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]]-1$

Lines 1–2: $\Theta(k)$

Lines 3–4: $\Theta(n)$

Lines 5–6: $\Theta(k)$

Lines 7–9: $\Theta(n)$

Total $\boldsymbol{\Theta(n+k)}$

Complexity of Counting Sort

- The overall time is $O(n+k)$.
  When we have $k=O(n)$, the worst case is $O(n)$.
- Beats the lower bound of ($n \log n$) because it is not a comparison sort
- No comparisons made: it uses actual values of the elements to index into an array.