# Linear-time Sorting

# Review

- We have now introduced several algorithms that can sort n numbers in O(nlogn) time.
  - Merge sort and heapsort achieve this upper bound in the worst case;
  - Quicksort achieves it on average.
  - Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(nlogn)$ time.
- we showed that $\Omega(nlogn)$ time is necessary, in the worst case, to sort an n-element sequence with a comparison-based sorting algorithm.

# Linear-time Sorting (integer sort)

To achieve linear-time sorting of *n* elements:

- Assume **keys** are **integers** in the range **[0, *N*-1]**
- We can use other operations instead of comparisons.
- We can sort in linear time when *N* is small enough.

# Counting sort

- ***Counting sort*** assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When k = O(n), the sort runs in O(n) time.
- we assume that the input is an array A[1..n], and the array B[1..n] holds the sorted output, and the array C[1..k] provides temporary working storage.

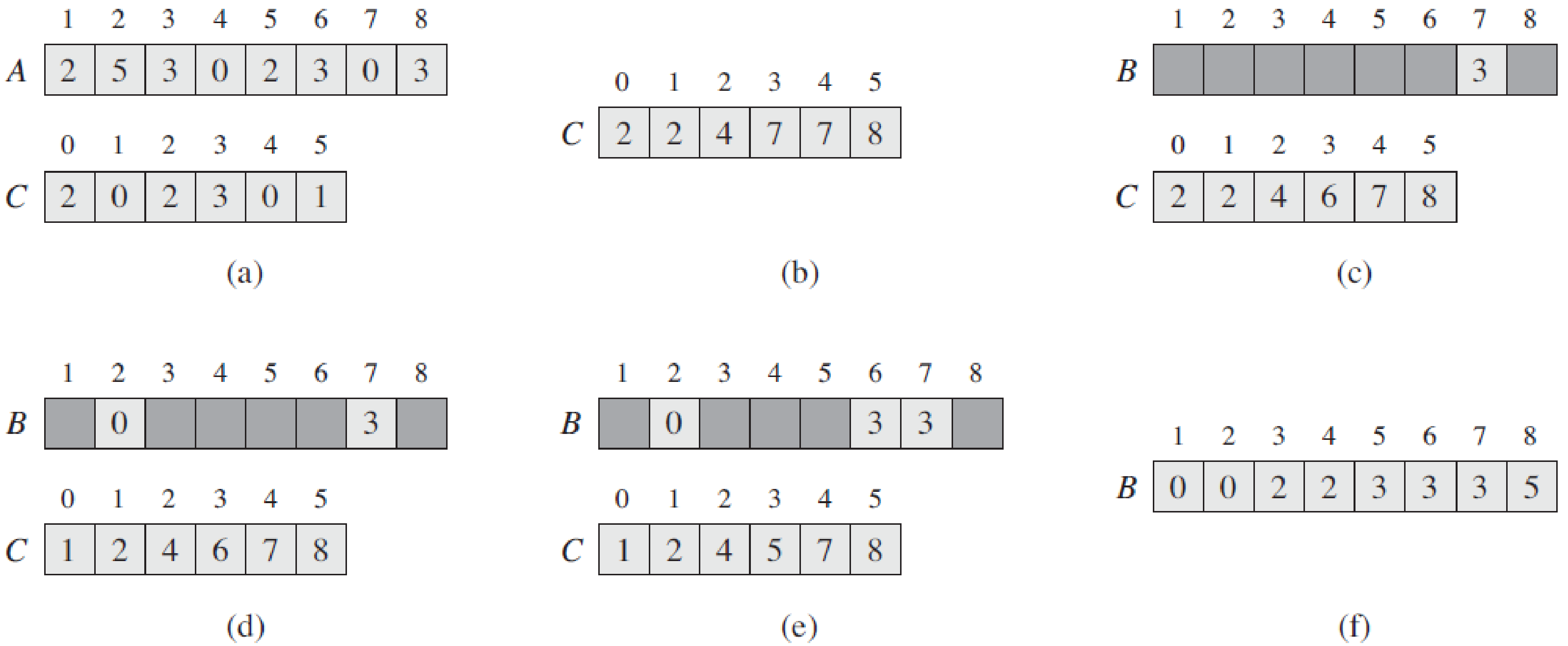
```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

# Counting sort-Run Time

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 $\Theta(n)$.

Counting sort beats the lower bound of $\Omega(n \lg n)$ proved : because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the elements to index into an array. The $\Omega(n \lg n)$ lower bound for sorting does not apply when we depart from the comparison sort model.

# Example




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 $B$.

# Exercises

- Using the example on slide 6 as a model, illustrate the operation of COUNTING-SORT on the array A={6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2}
- Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range [a .. B] in O(1) time. Your algorithm should use O(n+k) preprocessing time.

# Lexicographic Order

- A $\boldsymbol{d}$-tuple is a sequence of $\boldsymbol{d}$ keys $(\boldsymbol{k}_1, \boldsymbol{k}_2, \ldots, \boldsymbol{k}_{\boldsymbol{d}})$, where key $\boldsymbol{k}_{\boldsymbol{i}}$ is said to be the $\boldsymbol{i}$-th dimension of the tuple

- The lexicographic order of two $\boldsymbol{d}$-tuples is recursively defined as follows

$$(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_{\boldsymbol{d}}) < (\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_{\boldsymbol{d}})$$
$$\Leftrightarrow$$
$$(\boldsymbol{x}_1 < \boldsymbol{y}_1) \vee (\boldsymbol{x}_1 = \boldsymbol{y}_1 \wedge (\boldsymbol{x}_2, \ldots, \boldsymbol{x}_{\boldsymbol{d}}) < (\boldsymbol{y}_2, \ldots, \boldsymbol{y}_{\boldsymbol{d}}))$$

that is, tuples are compared by the first dimension, then by the second, etc.

# Lexicographic-Sort

Let ***stableSort***(***S***, ***C***) be a stable sorting algorithm that uses comparator ***C***

- $C_i$ is the comparator that compares two tuples by their ***i***-th dimension

Lexicographic-sort sorts a sequence of ***d***-tuples in lexicographic order by executing ***d*** times algorithm ***stableSort***, (one per dimension)

- runs in $O(dT(n))$ time, where $T(n)$ is the running time of ***stableSort***

**Algorithm** ***lexicographicSort***(***S***)
  **Input** sequence ***S*** of ***d***-tuples
  **Output** sequence ***S*** sorted in lexicographic order

  **for** $i \leftarrow d$ **downto** 1
    $stableSort(S, C_i)$

Example:

(7,4,6) (5,1,5) (2,4,6) (2,1,4) (3,2,4)

(2,1,4) (3,2,4) (5,1,5) (7,4,6) (2,4,6)

(2,1,4) (5,1,5) (3,2,4) (7,4,6) (2,4,6)

(2,1,4) (2,4,6) (3,2,4) (5,1,5) (7,4,6)

# Radix Sort

- A specialization of lexicographic-sort that uses count-sort as the stable sorting algorithm in each dimension
- Radix-sort is applicable to tuples where the keys in each dimension are integers in the range $[0, N - 1]$
- Radix-sort runs in time $O(d(n + N))$

**Algorithm** ***radixSort***$(S, N)$

**Input** sequence $S$ of $d$-tuples such that $(0, \ldots, 0) \leq (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \leq (N - 1, \ldots, N$ □ $1)$ for each tuple $(x_1, \ldots, x_d)$ in $S$
**Output** sequence $S$ sorted in lexicographic order

**for** $i \leftarrow d$ **downto** 1

*CountSort*$(S, N)$

# Radix Sort for Binary Numbers

- Consider a sequence of $n$ $b$-bit integers
  $x = x_{b-1} \ldots x_1 x_0$
- We represent each element as a $b$-tuple of integers in the range [0, 1] and apply radix-sort with $N = 2$
- This application of the radix-sort algorithm runs in $O(bn)$ time
- For example, we can sort a sequence of 32-bit integers in linear time

**Algorithm** *binaryRadixSort*($S$)

    **Input** sequence $S$ of $b$-bit integers
    **Output** sequence $S$ sorted

    replace each element $x$ of $S$ with the item $(0, x)$

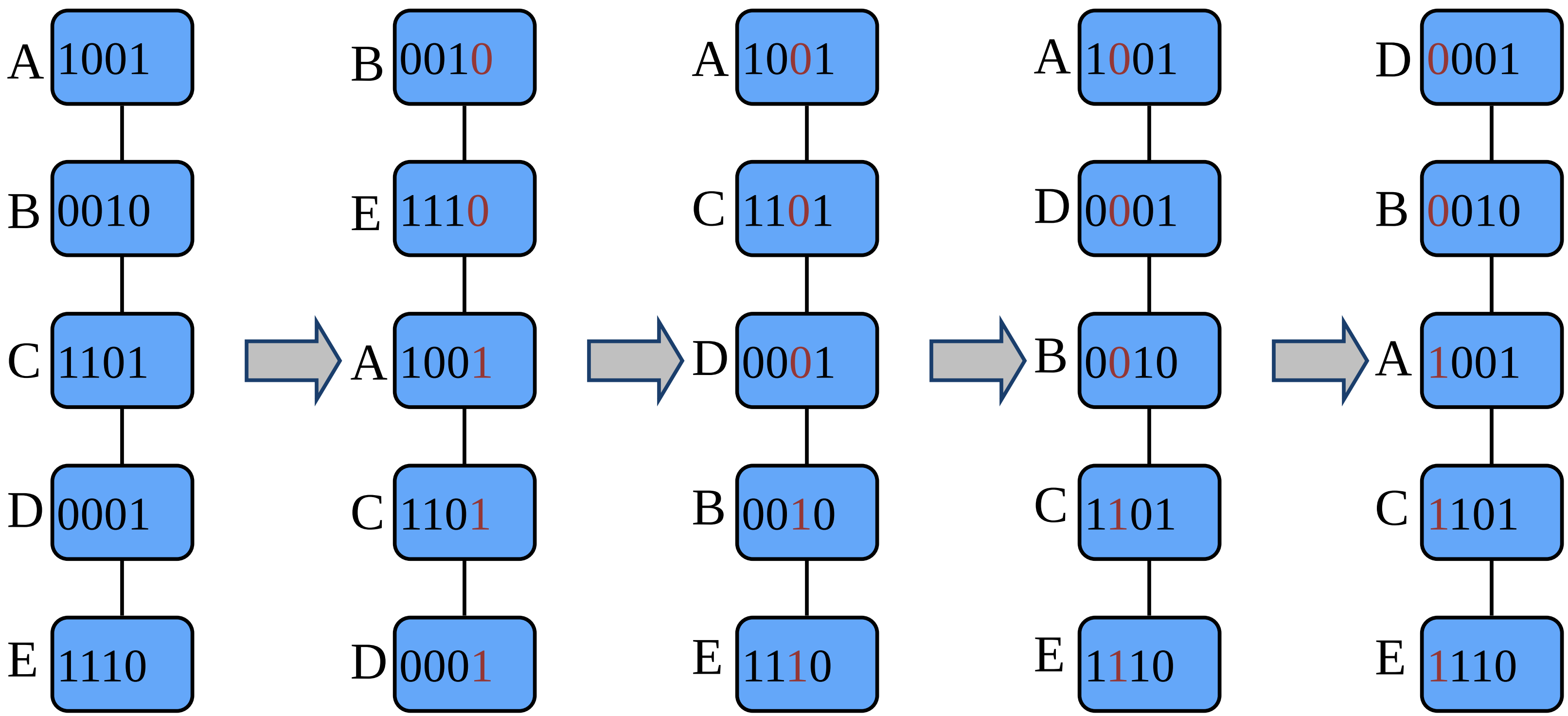
    **for** $i \leftarrow 0$ **to** $b - 1$

        replace the key $k$ of
            each item $(k, x)$ of $S$ with bit $x_i$ of $x$

        *CountSort*($S$, 2)

# Example

Use radix sort to sort sequence of 4-bit integers



# Exercises

Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

--------------------------------------------------------------------------------

Show how to sort $n$ integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

# Bucket Sort

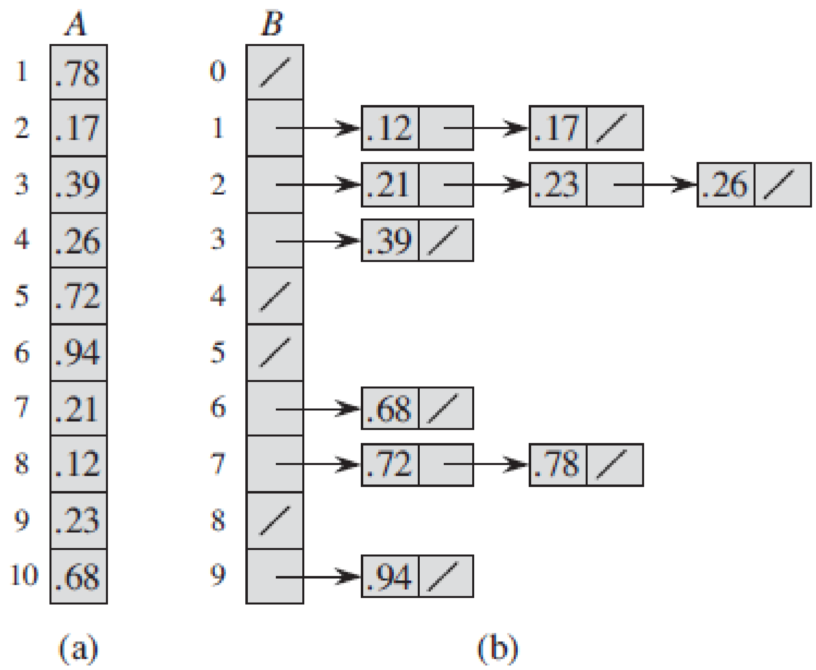
- Counting sort assumes that the input consists of integers in a small range,
- bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval [0,1).
- Bucket sort divides the interval [0,1) into n equal-sized subintervals, or ***buckets***, and then distributes the n input numbers into the buckets.
- To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.
- The worst-case running time for bucket sort is $O(n^2)$ if we like insertion sort or it will be $O(nlong)$ if we use merge sort.

# Bucket Sort

- With the bucket sort, we assumes that the input is an n-element array A and that each element A[i] in the array satisfies $0 \leq A[i] < 1$.
- There is an auxiliary array B[1 .. n-1] of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists.

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

# Example


A
B
1 .78
2 .17
3 .39
4 .26
5 .72
6 .94
7 .21
8 .12
9 .23
10 .68
0
1 .12 .17
2 .21 .23 .26
3 .39
4
5
6 .68
7 .72 .78
8
9 .94
(a)
(b)