Sorting in Linear Time

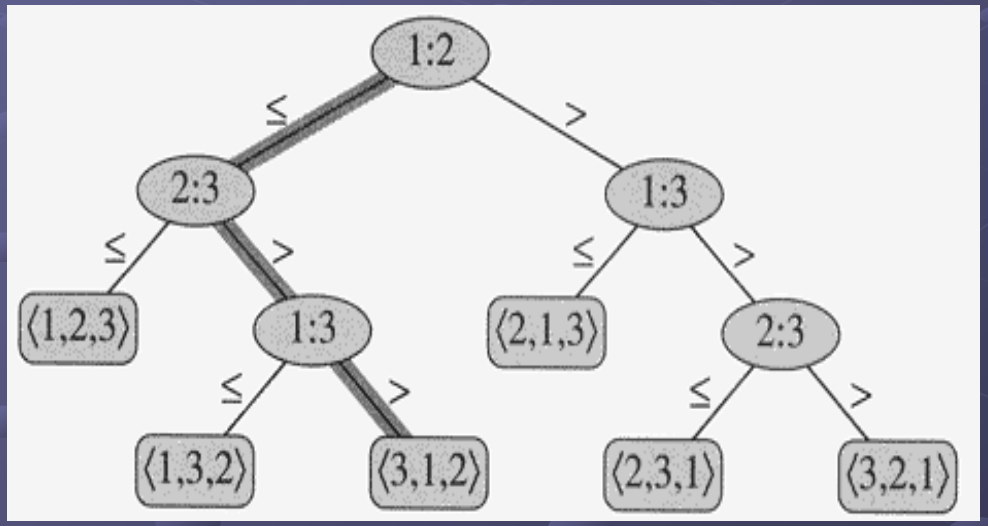
交通大學 資訊工程系

Lower Bounds for Sorting[1]

- ***Comparison sorts***
  - The sorted order they determine is based only on comparisons between the input elements.
  - Given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, or $a_i > a_j$ to determine their relative order.
- **Assumption:**
  - Without loss of generality, all of the input elements are distinct.
  - All the comparisons have the form $a_i \le a_j$.

Lower Bounds for Sorting[2]

- The *decision-tree model*
  - A full binary tree represents the comparisons between elements that are performed by a particular sorting algorithm on an input of a given size.



(Let $a_1 = 6, a_2 = 8, a_3 = 5$)

Lower Bounds for Sorting[3]

- Each internal node is annotated by $i : j$.
- Each leaf is annotated by a permutation $(\pi(1), \pi(2), \ldots, \pi(n))$.
- The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf.
- Necessary condition for a comparison sort to be correct:
  - Each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree.
  - Each of these leaves must be reachable from the root.

Lower Bounds for Sorting[4]

- **A lower bound for the worst case**
  - **The *length of the longest path* from the root of a decision tree to any of its reachable leaves.**
  - **The worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree.**
- ***Theorem 8.1***
  - **Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.**

Lower Bounds for Sorting[5]

- ***Proof of Theorem 8.1:***
  - **Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n numbers.**

    $n! \le l \le 2^h$,

    $h \ge \lg(n!) = \Omega(n \lg n)$.

- ***Corollary 8.2***
  - **Heapsort and merge sort are asymptotically optimal comparison sorts.**

Counting Sort[1]

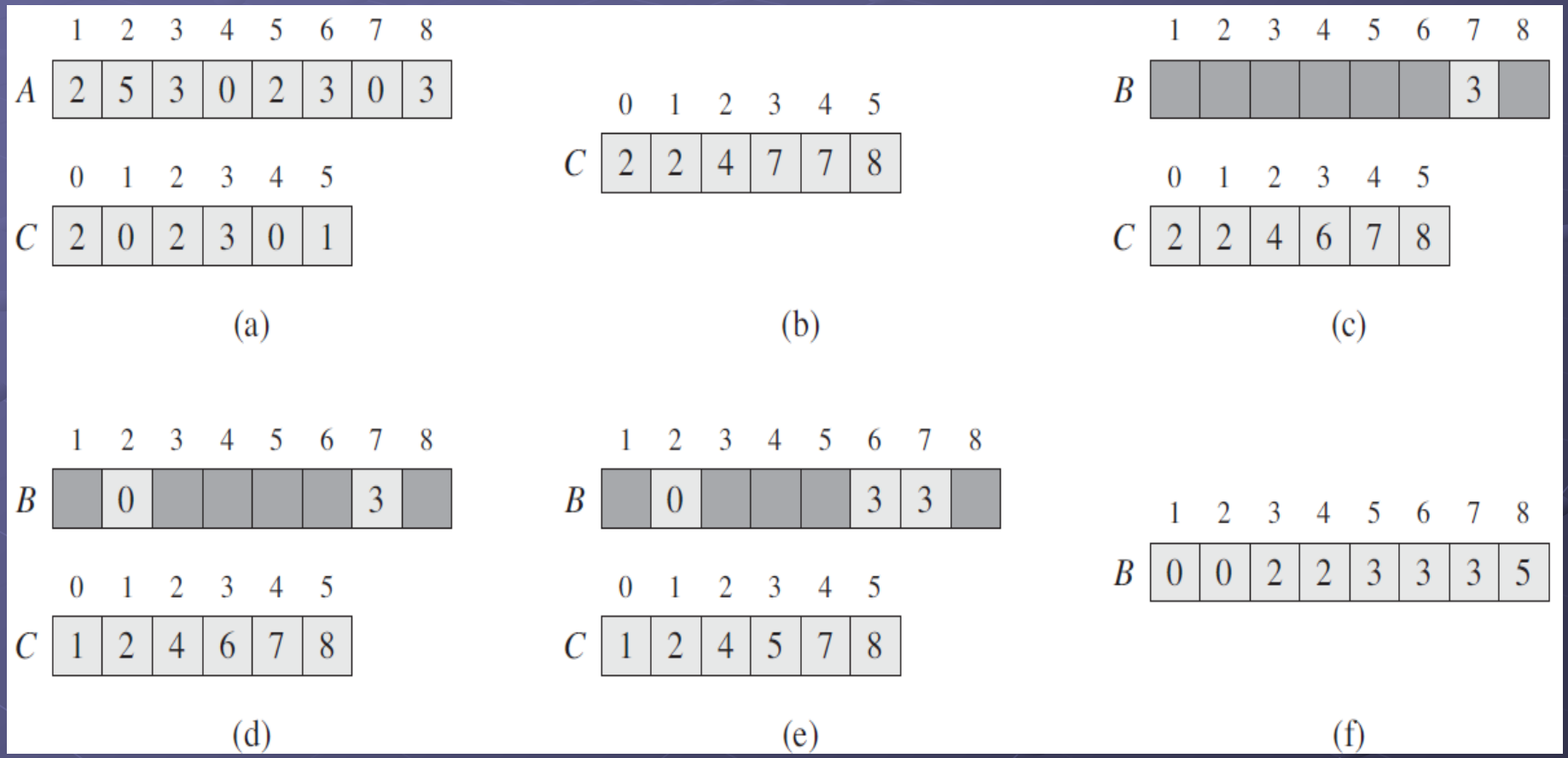
- **Assumptions**
  - Each of the n input elements is an integer in the range 0 to k, for some integer k.
- **Basic idea**
  - To determine, for each input element x, the number of elements less than x.
- **Running time: $\Theta(n)$.**
  - The $\Omega(n \lg n)$ lower bound for sorting does not apply when we depart from the comparison-sort model.

Counting Sort[2]

Algorithm

```
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
```

Counting Sort[3]

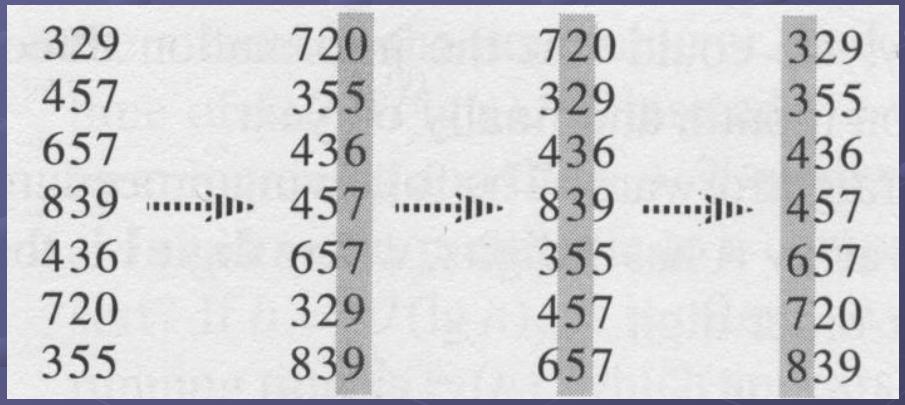


Counting Sort[4]

- ***Stable*** **property:**
  - **Numbers with the same value appear in the output array in the same order as they do in the input array.**
- **The reasons for the importance of stability**
  - **When satellite data are carried around with the element being sorted.**
  - **Counting sort is often used as a subroutine in radix sort.**

Radix Sort[1]

- Used by the card-sorting machines you can now find only in computer museum.



RADIX-SORT(A, d)

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

2 **do** use a stable sort to sort array A on digit i

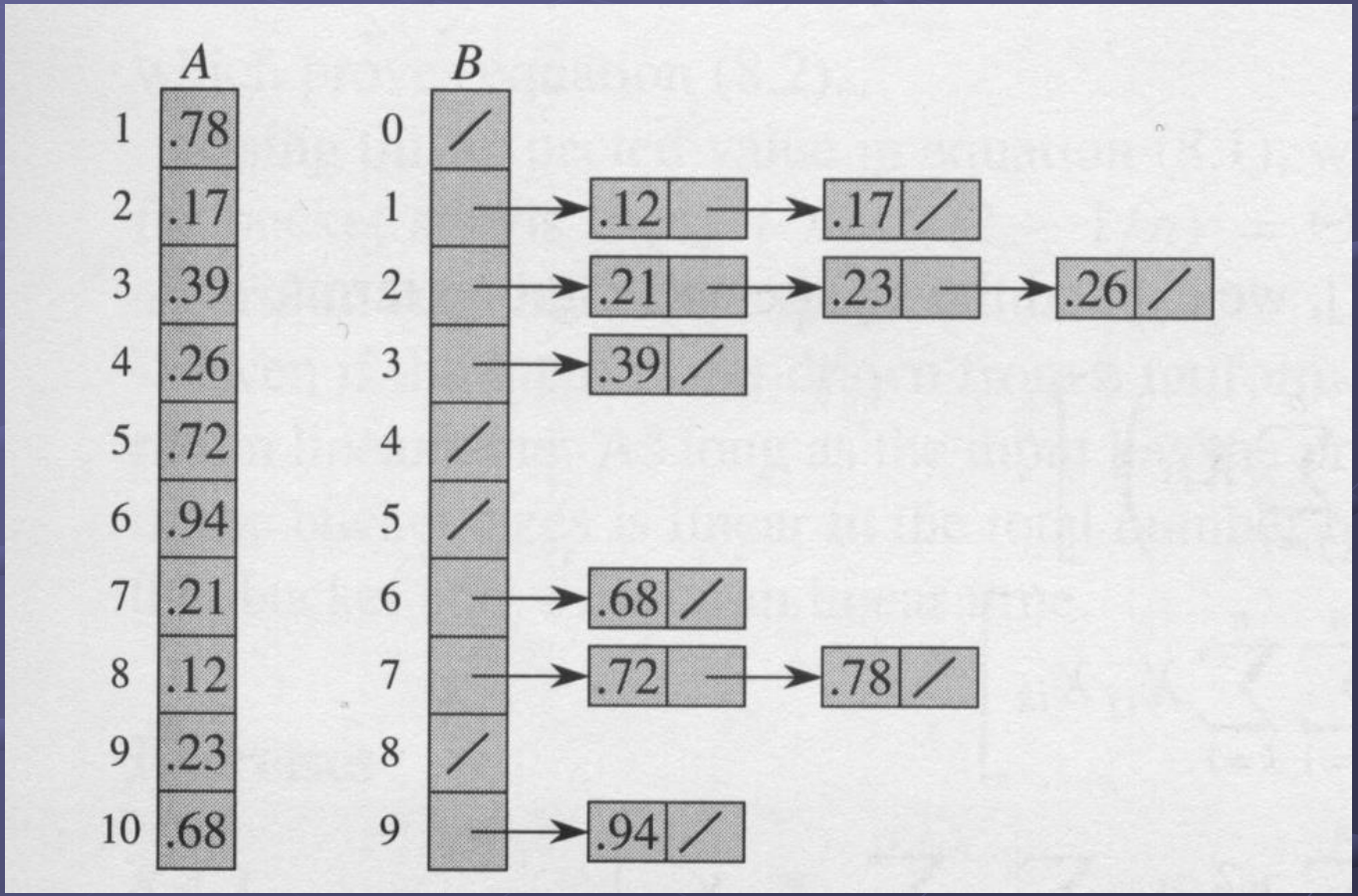
Radix Sort^2

- ***Lemma 8.3***
  - Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these number in $\Theta(d(n + k))$ time.
- When d is constant and $k = O(n)$, radix sort runs in linear time.
- ***Lemma 8.4***
  - Given n b-bit numbers and any positive integer $r \leq b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n+2^r))$ time.

Bucket Sort[1]

- **Bucket sort**
  - **Assumes that the input is generated by a random process that distributes elements uniformly over the interval [0, 1).**
  - **Runs in linear expected time when the input is drawn from a uniform distribution.**
  - **Divide the interval [0, 1) into *n* equal-sized sub-intervals, or buckets, and then distribute the *n* input numbers into the buckets.**

Bucket Sort[2]



Bucket Sort[3]

```
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[⌊n A[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
```

Bucket Sort[4]

- The running time of bucket sort is

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

- Expected running time: $\Theta(n)$.

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

We claim that

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