Announcements

- Weekly Reading Assignment: Chapter 7 and 8 (CLRS)
- Homework #3 is due on 10/6/05
- Extra help session today after class

Lower Bounds for Sorting

- Sorting methods that determine sorted order based only on comparisons between input elements must take $\Omega(n \lg n)$ comparisons in the worst case to sort. Thus, merge sort and heapsort are asymptotically optimal.
- Other sorting methods (counting sort, radix sort, bucket sort) use operations other than comparisons to determine the order can do better -- run in linear time.

Decision Tree

- **Each internal node is annotated by a_i: a_j for some i and j in range $1 \le i,j \le n$. Each leave is annotated by a permutation $\pi(i)$.**

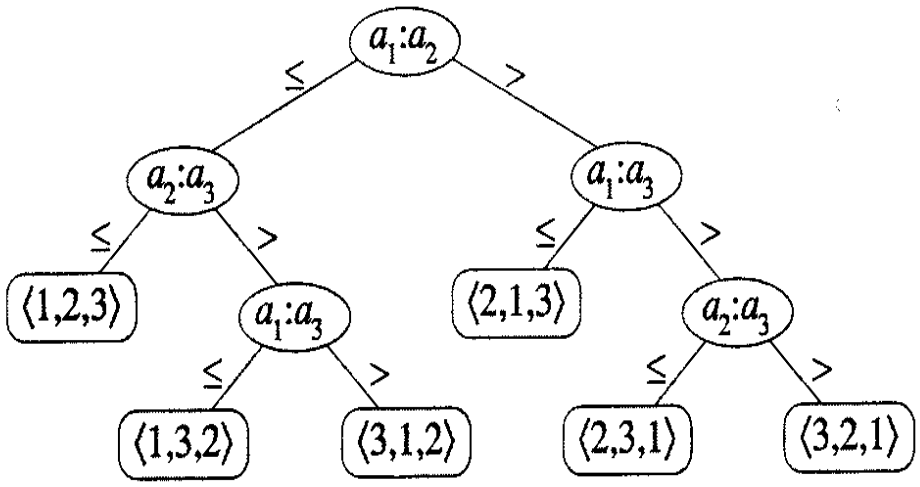

Figure 9.1 The decision tree for insertion sort operating on three elements. There are $3! = 6$ possible permutations of the input elements, so the decision tree must have at least 6 leaves.

Lower Bound for Worst Case

- **Any decision tree that sorts n elements has height $\Omega(n \lg n)$.**

***Proof*: There are $n!$ permutations of n elements, each permutation representing a distinct sorted order, the tree must have at least $n!$ leaves. Since a binary tree of height h has no more than 2^h leaves, we have**

$$n! \leq 2^h \Rightarrow h \geq lg(n!)$$

By Stirling's approximation: $n! > (n/e)^n$

$$h \geq lg(n!) \geq lg(n/e)^n = n \lg n - n \lg e = \Omega(n \lg n)$$

Counting Sort

- Assuming each of *n* input elements is an integer ranging 1 to *k*, when *k* = *O(n)* sort runs in *O(n)* time.

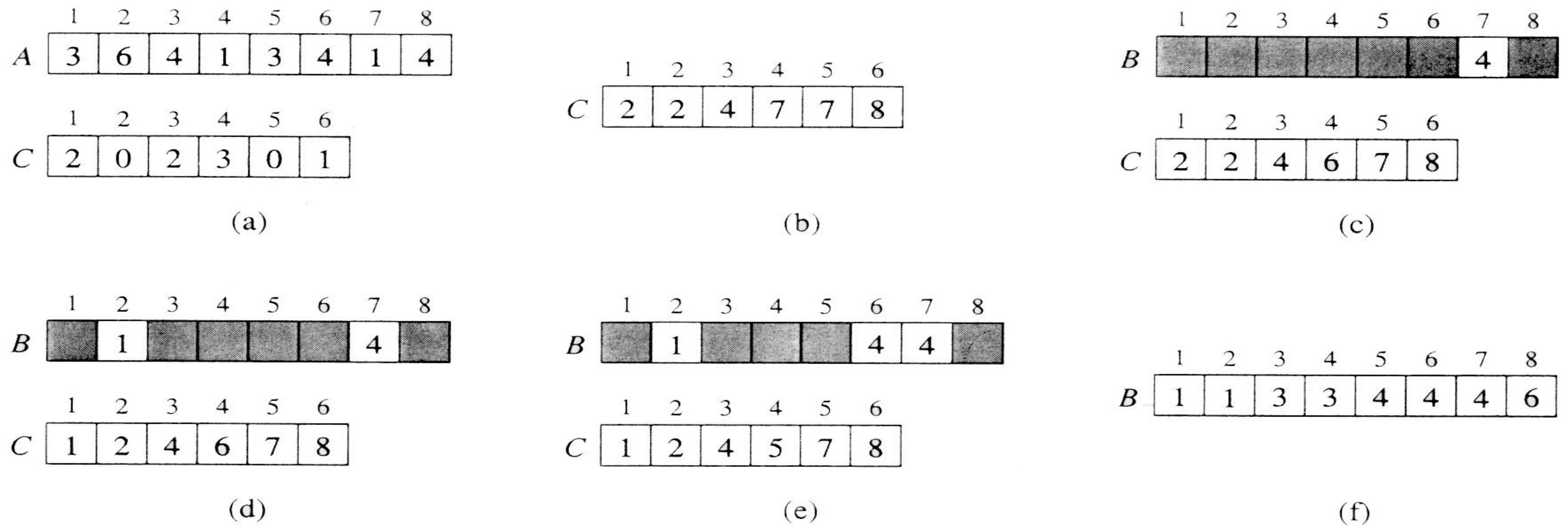

Figure 9.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a positive integer no larger than $k = 6$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, 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.

Counting-Sort (*A*, *B*, *k*)

```
for i ← 1 to k
        do C[i] ← 0
for j ← 1 to length[A]
        do C[A[j]] ← C[A[j]] + 1
for i ← 2 to k
        do C[i] ← C[i] + C[i-1]
for j ← length[A] downto 1
        do B[C[A[ j ]]] ← A[j]
               C[A[j]] ← C[A[j]] - 1
```

Algorithm Analysis

- **The overall time is $O(n+k)$. When we have $k=O(n)$, the worst case is $O(n)$.**
  - **for-loop of lines 1-2 takes time $O(k)$**
  - **for-loop of lines 3-4 takes time $O(n)$**
  - **for-loop of lines 5-6 takes time $O(k)$**
  - **for-loop of lines 7-9 takes time $O(n)$**
- **Stable, but not in place.**
- **No comparisons made: it uses actual values of the elements to index into an array.**

Radix Sort

- It was used by the card-sorting machines to read the punch cards.
- The key is sort the "least significant digit" first and the remaining digits in sequential order. The sorting method used to sort each digit must be "stable".
  - If we start with the "most significant digit", we'll need extra storage.

An Example

392		631		928		356
356		392		631		392
446		532		532		446
928	⇒	495	⇒	446	⇒	495
631		356		356		532
532		446		392		631
495		928		495		928
		↑		↑		↑

Radix-Sort(*A*, *d*)

1. for $i \leftarrow 1$ to d
2. do *use a stable sort to sort array A on digit i*

** To prove the correctness of this algorithm by induction on the column being sorted:

Proof: Assuming that radix sort works for d-1 digits, we'll show that it works for d digits.

Radix sort sorts each digit separately, starting from digit 1. Thus radix sort of d digits is equivalent to radix sort of the low-order d -1 digits followed by a sort on digit d .

Correctness of Radix Sort

By our induction hypothesis, the sort of the low-order *d*-1 digits works, so just before the sort on digit *d* , the elements are in order according to their low-order *d*-1 digits. The sort on digit *d* will order the elements by their *d*th digit.

Consider two elements, *a* and *b*, with *d*th digits a_d and b_d:

- **If $a_d < b_d$, the sort will put *a* before *b*, since $a < b$ regardless of the low-order digits.**
- **If $a_d > b_d$, the sort will put *a* after *b*, since $a > b$ regardless of the low-order digits.**
- **If $a_d = b_d$, the sort will leave *a* and *b* in the same order, since the sort is stable. But that order is already correct, since the correct order of is determined by the low-order digits when their *d*th digits are equal.**

Algorithm Analysis

- Each pass over n d-digit numbers then takes time $\Theta(n+k)$.
- There are d passes, so the total time for radix sort is $\Theta(d\,n+d\,k)$.
- When d is a constant and $k = O(n)$, radix sort runs in linear time.
- Radix sort, if uses counting sort as the intermediate stable sort, does not sort in place.
  - If primary memory storage is an issue, quicksort or other sorting methods may be preferable.

Bucket Sort

- Counting sort and radix sort are good for integers. For floating point numbers, try bucket sort or other comparison-based methods.
- Assume that input is generated by a random process that distributes the elements uniformly over interval [0,1). (Other ranges can be scaled accordingly.)
- The basic idea is to divide the interval into n equal-sized subintervals, or “buckets”, then insert the n input numbers into the buckets. The elements in each bucket are then sorted; lists from all buckets are concatenated in sequential order to generate output.

An Example

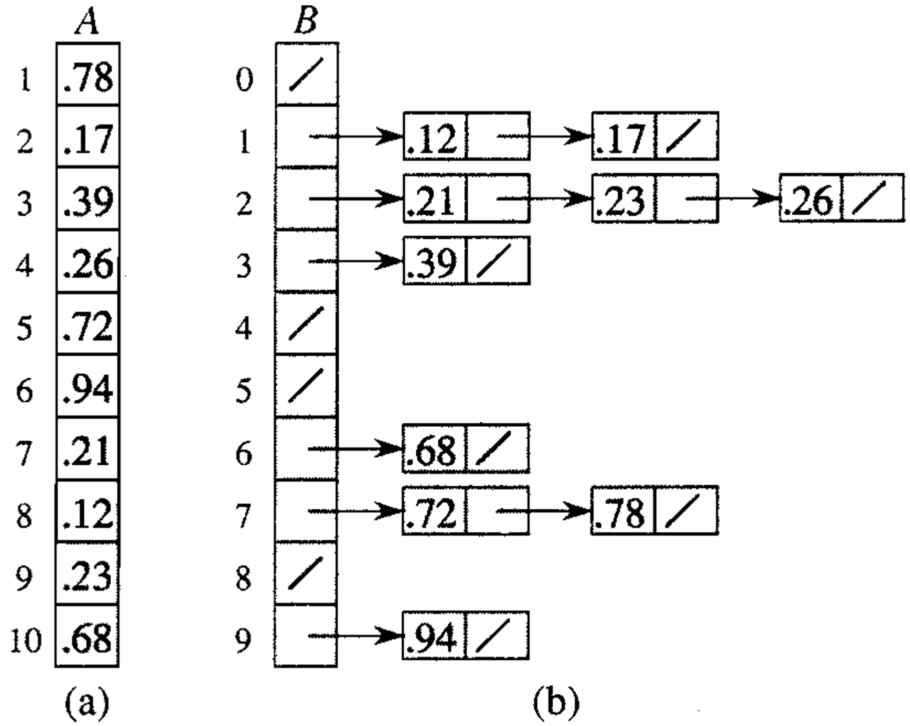

Figure 9.4 The operation of BUCKET-SORT. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Bucket-Sort (A)

$n \leftarrow length[A]$
for $i \leftarrow 1$ to n
    do insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
for $i \leftarrow 0$ to $n-1$
    do sort list $B[i]$ with insertion sort
Concatenate the lists $B[i]$s together in order

Algorithm Analysis

- All lines except line 5 take $O(n)$ time in the worst case. Total time to examine all buckets in line 5 is $O(n)$, without the sorting time.
- To analyze sorting time, let n_i be a random variable denoting the number of elements placed in bucket $B[i]$. The total time to sort is

$$\sum_{i = 0 \text{ to } n-1} O(\mathrm{E}[n_i^2]) = O\left(\sum_{i = 0 \text{ to } n-1} \mathrm{E}[n_i^2] \right) = O(n)$$

$$\begin{aligned} \mathrm{E}[n_i^2] &= \mathrm{Var}[n_i] + \mathrm{E}^2[n_i] \\ &= n\,p\,(1 - p) + 1^2 = 1 - (1/n) + 1 \\ &= 2 - 1/n = \Theta(1) \end{aligned}$$

Review: Binomial Distribution

- **Given n independent trials, each trial has two possible outcomes. Such trials are called "*Bernoulli trials*". If p is the probability of getting a head, then the probability of getting k heads in n tosses is given by (CLRS p.1113)**

$$P(X=k) = (n!/(k!(n-k)!))\, p^k\, (1-p)^{n-k} = b(k;n,p)$$

- **This probability distribution is called the "*binomial distribution*". p^k is the probability of tossing k heads and $(1-p)^{n-k}$ is the probability of tossing n-k tails. $(n!/(k!(n-k)!))$ is the total number of different ways that the k heads could be distributed among n tosses.**

Review: Binomial Distribution

See p. 1113-1116 for the derivations.

- $E[x] = n\,p$
- $Var[x] = E[X^2] - E^2[X] = n\,p\,(1-p)$
- $E[X^2] = Var[X] + E^2[X]$

$$= n\,p\,(1-p) + (n\,p)^2 = 1(1-p) + 1^2$$