Linear Time Sorting
Problem complexity

- Merge Sort, Heap Sort – W.C. $n \lg n$
- Quick Sort – Avg. Case $n \lg n$
- All these algorithms – *Comparison Sorting*
- Sorted order determined by comparisons between input elements
- Can a comparison sort algorithm do better than $n \lg n$?
Lower Bounds for sorting

- Comparison operations - $\succ, \succeq, \preceq, \prec, =$
- Assumption – all elements are distinct
- Implication – “$=$” comparison not necessary and all other comparison yield the same information
The decision-tree model

• Abstract representation of comparison sort algorithms
• Decision tree – full binary tree representing comparisons between elements by a comparison sort on an input of a fixed size
• \( n! \) leaves because any sorting algorithms should be able to produce any permutation of the input
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.
Lower Bound for worst case

• Length of the longest path from the root of a decision tree to any of its leaves – W.C.
• Worst case – height of the decision tree for that algorithm

Theorem: Any comparison sort requires $\Omega(n \log n)$ comparisons in the worst case

Corollary: Heap Sort and Merge Sort are asymptotically optimal comparison sorts
Counting Sort

• Assumption $n$ inputs are in a range 0 to $k$
• When $k=O(n)$, Counting Sort runs in $\Theta(n)$
• Goal – for each element $x$ determine how many elements $< x$
• Lets us place $x$ in its right position
• Input: $A[1..n]$
• Sorted Output: $B[1..n]$
• $C[0..k]$ temporary working storage
COUNTING-SORT(A, B, k)

1. for i ← 0 to k
2. do C[i] ← 0
3. for j ← 1 to length[A]
4. do C[A[j]] ← C[A[j]] + 1
5. ▷ C[i] now contains the number of elements equal to i.
6. for i ← 1 to k
7. do C[i] ← C[i] + C[i - 1]
8. ▷ C[i] now contains the number of elements less than or equal to i.
9. for j ← length[A] downto 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

- for loop 1-2 – $k$
- for loop 3-4 – $n$
- Total – $\Theta(k+n)$
- In practice, counting sort is used when $k=O(n)$
- This gives a running time $\Theta(n)$
- No comparisons
- Note how we traded space for time
- **Stable** – Elements appear in output in same order as they do in the input