Lecture 5: February 21, 2002

Today:

- Lower bounds of $\Omega(n \log n)$ for sorting
- Sorting in $O(n)$ time
  - Counting Sort
  - Radix Sort
Model of Computation

- Need to specify exactly what we can and what we cannot do
- Model needs to be realistic w.r.t.
  - what we can do (e.g., consider a SORT operation)
  - what we cannot do (e.g., we need to read the input)
Sorting Lower Bound

Comparison model:

- Limitation: only operation allowed is comparing two elements, i.e. cannot look at elements’ values!
  (covers all sorting algorithms so far)
- Freebie: ignore all other operations (control, data movement etc)
Algorithm as a Decision Tree

- Represents all comparisons for input of size $n$ with elements $< a_1, a_2, ..., a_n >$
- Each internal node holds $i : j$ for $i, j \in \{1, 2, \ldots, n\}$
  - Left subtree indicates subsequent comparisons if $a_i \leq a_j$
  - Right subtree indicates subsequent comparisons if $a_i > a_j$
- Each leaf holds a permutation $\langle \Pi(1), \Pi(2), \ldots, \Pi(n) \rangle$ indicating that the ordering $\langle a_{\Pi(1)}, a_{\Pi(2)}, \ldots, a_{\Pi(n)} \rangle$ has been established.
Algorithm as a Decision Tree

Decision tree can model a comparison sort:

- one tree for each distinct input size $n$
- view tree as if algorithm splits after each compare
- tree represents all possible execution traces
- algorithm running time = length of path

Worst-case running time = height of tree.
So... what’s the height of the tree?
Lower Bound for Comparison Sorting

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

**Proof:**

- The tree must have $\geq n!$ leaves  
  (# of permutations = $n!$)
- A height $h$ binary tree has at most $2^h$ leaves

Thus,

$$2^h \geq n! \geq (n/2)^{n/2}$$

and therefore

$$h = \Omega(n \log n)$$
Can we sort in $o(n \log n)$ time?

Answer: Yes, sometimes.
Sorting in linear time

- Counting sort - **no** key comparisons
- However, allowed to use key **values**!

**Input:** $A[1..n]$, where $A[j] \in 1, 2, \ldots, k$
(i.e., $k$ is max. value found in $A[\cdot]$)

**Output:** $B[1..n]$, sorted

**Uses:** $C[1..k]$ auxiliary storage, size $O(k)$

**Algorithm idea:**

- count the number of times each element occurs (using $C[\cdot]$)
- find the position of each element in the sorted array (using $C[\cdot]$ again)
- permute accordingly (from $A[\cdot]$ to $B[\cdot]$)
Counting Sort

\[ \text{COUNTING-SORT}(A, B, k) \]

1. for \( i \leftarrow 1 \) to \( k \)
2. do \( C[i] \leftarrow 0 \)
3. for \( j \leftarrow 1 \) to \( \text{length}[A] \)
4. do \( C[A[j]] \leftarrow C[A[j]] + 1 \)
5. \( \triangleright \) \( C[i] \) now contains \# of elements \( = \) to \( i \).
6. for \( i \leftarrow 2 \) to \( k \)
7. do \( C[i] \leftarrow C[i] + C[i - 1] \)
8. \( \triangleright \) \( C[i] \) now contains \# of elements \( \leq i \).
9. for \( j \leftarrow \text{length}[A] \) down to 1
11. \( C[A[j]] \leftarrow C[A[j]] - 1 \)
Counting Sort: Example

A: \[4 \ 3 \ 5 \ 8 \ 1 \ 8\]

C: \[1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 2\]

C': \[1 \ 1 \ 2 \ 3 \ 4 \ 4 \ 4 \ 6\]

B: \[\ \ \ \ \ \ \ \ \ \] 

Why do we need line 11?
Counting Sort

Analysis

• $O(n + k)$ time

• If $n = o(k)$, then $O(k)$ total time.
  (Example: sort 10 numbers ranging from 1 . . . 100, 000.)

• If $k = O(n)$, then $O(n)$ time.
  (Example: sort 100,000 numbers ranging from 1 . . . 10.)

• Counting sort is not a comparison sort,
  since we use the values of sorted elements

• Useful property: sort is stable, i.e.,
  Input order is maintained among “equal” keys.

But what if we want to sort $n$ numbers from 1 . . . $n^2$ ?
Radix Sort

- IBM card sorting algorithm run by machine **and** human operator
- multi-pass, **stable**, digit-by-digit sort
- $d$ passes required to sort numbers with $d$ digits

**Idea:** Sort on *least* significant digit first:

<table>
<thead>
<tr>
<th>Unsorted</th>
<th>Ones</th>
<th>Tens</th>
<th>Hundreds</th>
</tr>
</thead>
<tbody>
<tr>
<td>329</td>
<td>720</td>
<td>720</td>
<td>329</td>
</tr>
<tr>
<td>457</td>
<td>355</td>
<td>329</td>
<td>355</td>
</tr>
<tr>
<td>657</td>
<td>436</td>
<td>436</td>
<td>436</td>
</tr>
<tr>
<td>839</td>
<td>457</td>
<td>839</td>
<td>457</td>
</tr>
<tr>
<td>436</td>
<td>657</td>
<td>355</td>
<td>657</td>
</tr>
<tr>
<td>720</td>
<td>329</td>
<td>457</td>
<td>720</td>
</tr>
<tr>
<td>355</td>
<td>839</td>
<td>657</td>
<td>839</td>
</tr>
</tbody>
</table>
Radix Sort

Correctness: Induction on digit position

Assume numbers are sorted by low-order $t-1$ digits. Then, when we sort on digit $t$:

- Two numbers that differ in $t^{th}$ digit are correctly sorted.
- Two numbers having same $t^{th}$ digit are put in same order as they occur in the input to this pass, i.e., the correct order.
Radix Sort: Analysis

Running time depends on auxiliary stable sort

- If each digit lies in range 1 to \( k \), and \( k \) is not too large, counting sort is obvious choice for auxiliary stable sort.
- Each pass over \( n \) \( d \)-digit numbers takes time \( \Theta(n + k) \Rightarrow k = O(n) \) is a natural choice
- There are \( d \) passes, so total running time of radix sort is \( \Theta(d(n + k)) = \Theta(dn) \)
- When \( d \) is constant and, radix sort runs in time **linear** in \( n \).
- Can handle numbers up to \( n^d \) for any constant \( d \)!
Radix Sort: Example

- Sort $2^{20}$ 64-bit numbers:
  Radix sort: $n = 2^{20}$, range $0 \ldots 2^{64} < n^4$
  $\Rightarrow$ 4 passes
  Merge sort: $> n \lg n$ work
  $\Rightarrow$ 20 passes

- In practice, radix sort is:
  - fast for large inputs
  - simple to code
Concluding thoughts

• Model of computation is crucial!

• “Can only compare keys” implies $\Omega(n \log n)$ lower bound for sorting.

• Can look at digits of key, and key in small range yields $O(n)$ algorithm for sorting.

• In fact, can sort in $O(n \log \log n)$ time for any range (algorithm fairly complex)