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

Sorting Lower Bound
- limitation: only ordering 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)

Model algorithm operation as Decision Tree:
- represents all comparisons for input of size $n$
  with elements $<a_1, a_2, \ldots, a_n>$
- control, data movement, and other aspects of algorithm ignored (for purposes of lower bound).

Decision Tree
- 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.

Decision Tree

Ex: 3 element sort $<a_1, a_2, a_3>$
Note: all comparisons necessary!
(Adversary argument.)
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 $\Omega(n \log n)$.

**Proof:**

- The tree must have $\geq n!$ leaves
  - (# of permutations = $n!$: $n$ choices for 1st element, $n - 1$ for the 2nd, etc.)
- 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?

E.g., how to sort $A[1 \ldots n]$ containing distinct elements from $\{1 \ldots n\}$?

```plaintext
for $i \leftarrow 1$ to $k$ do $A[i] = i$
```

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:**

- 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

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

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 \ldots 100,000)
- If \(k = O(n)\), then \(O(n)\) time.
  (Example: sort 100,000 numbers ranging from 1 \ldots 10.
- Counting sort is not a comparison sort,
  since we use the \textit{values} of sorted elements
- Useful property: sort is \textbf{stable}, i.e.,
  Input order is maintained among “equal” keys.

Radix Sort

- IBM card sorting algorithm run by
  machine \textbf{and} human operator
- multi-pass, \textbf{stable}, digit-by-digit sort
- each pass distributes one card stack into ten bins
- \(d\) passes required to sort numbers with \(d\) digits

\textbf{Idea:} Sort on \textit{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 \rightarrow</td>
<td>457 \rightarrow</td>
<td>839 \rightarrow</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 \( \Theta(n + k) \)
- There are \( d \) passes, so total running time of radix sort is \( \Theta(d(n + k)) \)
- When \( d \) is constant and \( k = O(n) \), radix sort runs in time linear in \( n \).

Radix Sort: Analysis

- Sort \( n \) words of \( b \) bits each.
- View each word as having \( \frac{b}{r} \) digits of \( r \) bits each.

Ex: 32-bit word: \[
\begin{array}{cccc}
\text{8 bits} & \text{8 bits} & \text{8 bits} & \text{8 bits}
\end{array}
\]

\[
b = 32, \quad r = 8, \quad \frac{b}{r} = 4 \text{ digits}
\]
Therefore, radix sort makes

\[
\frac{b}{r} = 4 \text{ passes.}
\]

I.e., it processes each number 4 times.

Radix Sort: Analysis

- Each pass of radix sort takes \( \Theta(n + 2^r) \) time, since \( r \) bit values \( \rightarrow \) keys in range \( 0, 1, \ldots, 2^r - 1 \)
- \( T(n, b) = \Theta(b \frac{\log n}{\log 2}) \)

- So, how should we choose \( r \)?
  - As we increase \( r \) \( \rightarrow \) fewer passes are needed
  - But, as \( r \gg \log n \), counting sort time grows exponentially!
- Choose \( r = \log n \)
  \[
  \Rightarrow T(n, b) = \Theta(b \frac{\log n}{\log n})
  \]
Again, where:

\( n \) is input size
\( b \) is number of bits in each word
\( r \) is number of bits in each digit
Radix Sort: Analysis

- Sorting numbers in range:
  - $0$ to $n - 1$ ($b = \lg n$) in $\Theta(n)$ time.
  - $0$ to $n^2 - 1$ ($b = 2 \lg n$) in $2 \cdot \Theta(n)$ time.
  - $0$ to $n^d - 1$ ($b = d \lg n$) in $d \cdot \Theta(n)$ time.

Radix Sort: Example

- Sort $10^6$ 64-bit numbers:
  Radix sort: choose $r = \lg n = 20$; 4 passes (4 ops/#) Merge-, quicksort do $\Theta(n \lg n)$ work, $\Rightarrow$ 20 ops/#

- 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)