All sorting algorithms we've seen so far are based on comparing elements. E.g., Insertion sort, Selection sort, Mergesort, Heapsort, and Quicksort. Insertion sort, Selection sort, and Quicksort have worst-case running times \( \Theta(n^2) \), while the others have worst-case running time \( \Theta(n \log n) \).

**Question**
Can we do better?

**Goal**
We will prove that any comparison-based sorting algorithm has a worst-case running time \( \Omega(n \log n) \).
Decision-tree Example

Sort $< a_1, a_2, \ldots, a_n >$

Each internal node is labeled $a_i : a_j$ for $\{1, 2, \ldots, n\}$
- The left subtree shows subsequent comparisons if $a_i \leq a_j$
- The right subtree shows subsequent comparisons if $a_i > a_j$

Each leaf corresponds to an input ordering
Each internal node is labeled $a_i : a_j$ for \{1, 2, \ldots, n\}
- The left subtree shows subsequent comparisons if $a_i \leq a_j$
- The right subtree shows subsequent comparisons if $a_i > a_j$

Each leaf corresponds to an input ordering
Sort $< a_1, a_2, a_3 >$
$= < 6, 8, 5 >$:

Each internal node is labeled $a_i : a_j$ for $\{1, 2, \ldots, n\}$
- The left subtree shows subsequent comparisons if $a_i \leq a_j$
- The right subtree shows subsequent comparisons if $a_i > a_j$

Each leaf corresponds to an input ordering
Decision-tree Example

Sort \(< a_1, a_2, a_3 >\)
\(\leq < 6, 8, 5 >:\)

Each internal node is labeled \(a_i : a_j\) for \(\{1, 2, \ldots, n\}\)
- The left subtree shows subsequent comparisons if \(a_i \leq a_j\)
- The right subtree shows subsequent comparisons if \(a_i > a_j\)

Each leaf corresponds to an input ordering
Decision-tree Example

Sort $< a_1, a_2, a_3 >$

$\leq 6, 8, 5 >$

Each internal node is labeled $a_i : a_j$ for $\{1, 2, \ldots, n\}$

- The left subtree shows subsequent comparisons if $a_i \leq a_j$
- The right subtree shows subsequent comparisons if $a_i > a_j$

Each leaf corresponds to an input ordering
A decision tree can model the execution of any comparison-based sorting algorithm

- One tree for each input size $n$
- Worst-case running time = height of tree
Theorem

Any comparison-based sorting algorithm requires Ω(n \log n) comparisons in the worst case.

Proof.

- A decision tree to sort n elements must have at least n! leaves, since each of the n! orderings is a possible answer.
- A binary tree of height h has at most 2^h leaves
- Thus, n! ≤ 2^h
- \Rightarrow h ≥ \log n! = \Omega(n \log n)  \quad \text{(Stirling’s approximation)}

Corollary

Heapsort and merge sort are asymptotically optimal comparison-based sorting algorithms.
We just proved that worst case number of comparisons used is \( \Omega(n \log n) \).

Suppose that each of the \( n! \) input permutations is equally likely. What can be said about the average case running time?

Note: Average is taken by adding up individual running time of algorithm on each possible input and dividing total by \( n! \).

**Theorem**

When all input permutations are equally likely, any comparison-based sorting algorithm requires \( \Omega(n \log n) \) comparisons on average.
Theorem

When all input permutations are equally likely, any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons on average.

Proof.

- The External Path Length (EPL) of a tree is the sum over all leaves of the tree, of the length of the paths from the root to the leaves.
- Average number of comparisons used by a sorting algorithm is EPL of its associated comparison tree divided by $n!$.
- The EPL of a binary tree with $m$ leaves is at least $m \log_2 m + O(m)$.
- The comparison tree has $m = n!$ leaves
  $\Rightarrow$ its external path length is $n! \log_2 n! + O(n!)$
  $\Rightarrow$ average number of comparisons used is $\log_2 n! + O(1)$.
- We already saw $\log_2 n! = \Omega(n \log n)$. 
Are there sorting algorithms which are not comparison-based? Can they beat the $\Omega(n \log n)$ lower bound?

- **Counting sort**
  - Assumes items are in set \{1, 2, \ldots, k\}.
  - Is a *stable* sort (defined soon).

- **Radix sort**
  - Assumes items are stored in fixed size words using finite alphabet.
Counting-Sort

Counting-sort($A, B, k$)

Input: $A[1 \ldots n]$, where $A[j] \in \{1, 2, \ldots, k\}$
Output: $B[1 \ldots n]$, sorted
let $C[1 \ldots k]$ be a new array;
for $i \leftarrow 1$ to $k$ do
    $C[i] \leftarrow 0$;
end
for $j \leftarrow 1$ to $n$ do
    $C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$
end
for $i \leftarrow 2$ to $k$ do
    $C[i] \leftarrow C[i] + C[i - 1]; // C[i] = |\{key \leq i\}|$
end
for $j \leftarrow n$ to $1$ do
    $B[C[A[j]]] \leftarrow A[j];$
    $C[A[j]] \leftarrow C[A[j]] - 1;$
end
Example: Counting Sort

\[ A \begin{array}{c|c|c|c|c|c} 1 & 2 & 3 & 4 & 5 \\ \hline 4 & 2 & 1 & 4 & 2 \end{array} \]

\[ B \begin{array}{c|c|c|c} \hline \hline \hline \end{array} \]

\[ C \begin{array}{c|c|c|c|c} 1 & 2 & 3 & 4 \\ \hline \hline \hline \end{array} \]
Example: Counting Sort

\[
\begin{array}{c|c|c|c|c|c}
A & 1 & 2 & 3 & 4 & 5 \\
& 4 & 2 & 1 & 4 & 2 \\
B & \\
\end{array}
\quad
\begin{array}{c|c|c|c|c|c}
C & 1 & 2 & 3 & 4 \\
& 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\text{for } i \leftarrow 1 \text{ to } k \text{ do} \\
\quad C[i] \leftarrow 0; \\
\text{end}
\]
Example: Counting Sort

\[
\begin{array}{cccccc}
  & 1 & 2 & 3 & 4 & 5 \\
A & 4 & 2 & 1 & 4 & 2 \\
\end{array}
\]

\[
\begin{array}{c}
B \\
\end{array}
\]

\[
\begin{array}{cccc}
  & 1 & 2 & 3 & 4 \\
C & 0 & 0 & 0 & 1 \\
\end{array}
\]

for \( j \leftarrow 1 \) to \( n \) do
\[
C[A[j]] \leftarrow C[A[j]] + 1; \quad \text{\hrulefill} \quad C[i] = |\{\text{key} = i\}|
\]
end

Last Revision: September 11, 2014
Example: Counting Sort

\[
A = \begin{bmatrix}
4 & 2 & 1 & 4 & 2 \\
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
0 & 1 & 0 & 1 \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
\hline
\end{bmatrix}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \text{ do} \\
\quad C[A[j]] \leftarrow C[A[j]] + 1; \quad // \quad C[i] = |\{\text{key} = i\}|
\]

end
Example: Counting Sort

\[
\begin{array}{cccccc}
A & 1 & 2 & 3 & 4 & 5 \\
& 4 & 2 & 1 & 4 & 2 \\
\end{array}
\quad
\begin{array}{cccc}
C & 1 & 2 & 3 & 4 \\
& 1 & 1 & 0 & 1 \\
\end{array}
\]

\[
B
\]

for \( j \leftarrow 1 \) to \( n \) do
\[
C[A[j]] \leftarrow C[A[j]] + 1; \quad // \quad C[i] = |\{\text{key} = i\}|
\]
end
Example: Counting Sort

\[
\begin{array}{cccccc}
A & 1 & 2 & 3 & 4 & 5 \\
 4 & 2 & 1 & 4 & 2
\end{array}
\quad
C & 1 & 2 & 3 & 4 \\
 1 & 1 & 0 & 2
\]

\[
\begin{array}{c}
B \\
\end{array}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \text{ do}
\]
\[
\quad C[A[j]] \leftarrow C[A[j]] + 1; \quad // \quad C[i] = |\{\text{key} = i\}|
\]
\[
\text{end}
\]
### Example: Counting Sort

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

**B**

```plaintext
for j ← 1 to n do
    C[A[j]] ← C[A[j]] + 1; // C[i] = |{key = i}|
end
```
Example: Counting Sort

\[
\begin{array}{cccccc}
A & 1 & 2 & 3 & 4 & 5 \\
& 4 & 2 & 1 & 4 & 2 \\
\end{array}
\ 
\begin{array}{cccccc}
C & 1 & 2 & 3 & 4 \\
& 1 & 2 & 0 & 2 \\
\end{array}
\ 
\begin{array}{cccc}
B & & & \\
& & & \\
\end{array}
\ 
\begin{array}{cccc}
C' & 1 & 3 & 0 \\
& 1 & 3 & 0 \\
\end{array}
\]

For \( i \leftarrow 2 \) to \( k \) do
\[
C[i] \leftarrow C[i] + C[i - 1]; \quad // C[i] = |\{\text{key} \leq i\}|
\]
end
Example: Counting Sort

\[
\begin{array}{c}
\text{A} & 1 & 2 & 3 & 4 & 5 \\
& 4 & 2 & 1 & 4 & 2 \\
\hline
\text{B} & & & & & \\
& & & & & \\
\hline
\text{C} & 1 & 2 & 3 & 4 \\
& 1 & 2 & 0 & 2 \\
\end{array}
\]

\[
\begin{array}{c}
\text{C'} & 1 & 3 & 3 & 2 \\
& 1 & 3 & 3 & 2 \\
\end{array}
\]

for \( i \leftarrow 2 \) to \( k \) do
\[
C[i] \leftarrow C[i] + C[i - 1]; \quad // \quad C[i] = |\{\text{key} \leq i\}|
\]
end
Example: Counting Sort

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

|   |   |   |   |   |
|---|---|---|---|
| B |   |   |   |   |

|   |   |   |   |   |
|---|---|---|---|
| C' | 1 | 3 | 3 | 5 |

\[
\text{for } i \leftarrow 2 \text{ to } k \text{ do } \\
\quad C[i] \leftarrow C[i] + C[i - 1]; \quad // \quad C[i] = |\{\text{key } \leq i\}| \\
\text{end}
\]
Example: Counting Sort

```
for j ← n to 1 do
    B[C[A[j]]] ← A[j];
    C[A[j]] ← C[A[j]] − 1;
end
```
Example: Counting Sort

\[
\begin{array}{ccccc}
A & 1 & 2 & 3 & 4 & 5 \\
\hline
& 4 & 2 & 1 & 4 & 2 \\
B & 2 & 4 & & & \\
& & & & & \\
C & 1 & 3 & 3 & 5 & \\
C' & 1 & 2 & 3 & 4 & \\
\end{array}
\]

\[
\text{for } j \gets n \text{ to } 1 \text{ do}
\begin{align*}
B[C[A[j]]] & \leftarrow A[j]; \\
C[A[j]] & \leftarrow C[A[j]] - 1;
\end{align*}
\text{end}
\]
### Example: Counting Sort

#### Algorithm

```plaintext
for j ← n to 1 do
    B[C[A[j]]] ← A[j];
    C[A[j]] ← C[A[j]] − 1;
end
```

#### Arrays

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>C'</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
Example: Counting Sort

for \( j \leftarrow n \) to 1 do
\[
\begin{align*}
B[C[A[j]]] & \leftarrow A[j]; \\
C[A[j]] & \leftarrow C[A[j]] - 1;
\end{align*}
\]
end
Example: Counting Sort

\[
\begin{align*}
\text{for } j \leftarrow n \text{ to } 1 \text{ do} \\
& B[C[A[j]]] \leftarrow A[j]; \\
& C[A[j]] \leftarrow C[A[j]] - 1;
\end{align*}
\]
Input: $A[1\ldots n]$, where $A[j] \in \{1, 2, \ldots, k\}$

Output: $B[1\ldots n]$, sorted

let $C[1\ldots k]$ be a new array;

for $i \leftarrow 1$ to $k$ do
  $C[i] \leftarrow 0$; // $O(k)$
end

for $j \leftarrow 1$ to $n$ do
  $C[A[j]] \leftarrow C[A[j]] + 1$; // $O(n)$
end

for $i \leftarrow 2$ to $k$ do
  $C[i] \leftarrow C[i] + C[i - 1]$; // $O(k)$
end

for $j \leftarrow n$ to $1$ do
  $C[A[j]] \leftarrow C[A[j]] - 1$; // $O(n)$
end

Total: $O(n + k)$
If $k = O(n)$, then counting sort takes $O(n)$ time.

- But didn’t we prove that sorting must take $\Omega(n \log n)$ time?
- No, actually we proved that any comparison-based sorting algorithm takes $\Omega(n \log n)$ time.
- Note that counting sort is not a comparison-based sorting algorithm.
- In fact, it makes no comparisons at all!
Counting sort is a **stable** sort
- it preserves the input order among equal elements.

Exercise
What other sorts have this property?
Radix Sort

- Sort on *least significant* digit first using stable sort

<table>
<thead>
<tr>
<th>2 3 2 9</th>
<th>2 7 2 0</th>
<th>2 7 2 0</th>
<th>2 3 2 9</th>
<th>2 3 2 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 4 5 7</td>
<td>5 3 5 5</td>
<td>2 3 2 9</td>
<td>5 3 5 5</td>
<td>2 7 2 0</td>
</tr>
<tr>
<td>3 6 5 7</td>
<td>3 4 3 6</td>
<td>3 4 3 6</td>
<td>3 4 3 6</td>
<td>3 4 3 6</td>
</tr>
<tr>
<td>5 8 3 9</td>
<td>5 4 5 7</td>
<td>5 8 3 9</td>
<td>5 4 5 7</td>
<td>3 6 5 7</td>
</tr>
<tr>
<td>3 4 3 6</td>
<td>3 6 5 7</td>
<td>5 3 5 5</td>
<td>3 6 5 7</td>
<td>5 3 5 5</td>
</tr>
<tr>
<td>2 7 2 0</td>
<td>2 3 2 9</td>
<td>5 4 5 7</td>
<td>2 7 2 0</td>
<td>5 4 5 7</td>
</tr>
<tr>
<td>5 3 5 5</td>
<td>5 8 3 9</td>
<td>3 6 5 7</td>
<td>5 8 3 9</td>
<td>5 8 3 9</td>
</tr>
</tbody>
</table>
Radix Sort: Correctness

**Induction on digit position**

- Assume that the numbers are sorted by their low-order \(i-1\) digits.
- Sort on digit \(i\).

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2720</td>
<td>2329</td>
<td>2329</td>
<td>3436</td>
<td>3436</td>
<td>5839</td>
<td>5839</td>
<td>5839</td>
</tr>
<tr>
<td>1</td>
<td>3657</td>
<td>3657</td>
<td>5457</td>
<td>5457</td>
<td>5457</td>
<td>5457</td>
<td>5457</td>
<td>5457</td>
</tr>
</tbody>
</table>
Radix Sort: Correctness

**Induction on digit position**

- Assume that the numbers are sorted by their low-order $i - 1$ digits

- Sort on digit $i$
  - Two numbers that differ on digit $i$ are correctly sorted by their low-order $i$ digits
Induction on digit position

- Assume that the numbers are sorted by their low-order $i-1$ digits

- Sort on digit $i$
  - Two numbers that differ on digit $i$ are correctly sorted by their low-order $i$ digits
  - Two numbers equal on digit $i$ are put in the same order as the input $\Rightarrow$ correctly sorted by their low-order $i$ digits
Radix Sort: Running Time & Application

Lemma

Given \( n \) \( d \)-digit numbers in which each digit can take on up to \( k \) possible values, radix sort correctly sorts these numbers in \( O(d(n + k)) \) time if the stable sort it uses takes \( O(n + k) \) time.

Application:
Sorting numbers in the range from 0 to \( n^b - 1 \), where \( b \) is a constant

- \( b \log n \) bits for each number
- each number can be viewed as having \( O(b) \) digits of \( \log n \) bits each
- running time is \( O(d(n + k)) = O(b(n + 2^{\log n})) = O(bn) \)
- since \( b \) is a constant, the running time is \( O(n) \).