So the lower bound for sorting by comparing keys is \(\lceil \log n \rceil \), and the lower bound for number of comparisons in the lower bound for the height of the decision tree.

So a decision tree, with \( n \) leaves, must have height at least \( \lceil \log n \rceil \). For a binary tree of \( h \) leaves and height \( y \), we have \( h \geq 2^y \), that is, \( y \geq \log_2 h \). There are \( n! \) total permutations of \( n \) elements, so a decision tree for sorting \( n \) elements must have at least \( \log_2 n! \) leaves.

Consider an algorithm to sort three elements, as in the decision tree, as in the tree below.

6.1 Lower Bounds in General

6 Lower Bounds on Sorting
somewhere else in a binary tree, necessarily increases the sum of the path lengths.

\[ \text{Proposition 6.1. The sum of all path lengths to leaves is minimized when the tree is completely balanced.} \]

Consider the average case, that is,

The average case requires looking at the average path length.

The worst case requires looking for the maximal path down a decision tree.

6.2 Lower bounds for the average case.
\begin{proof}
The minimal sum of all path lengths to trees is

\[(i u)^{\leq} \cdot (i u)\]

so the average path length is bounded below by \((i u)^{\leq}\).
\end{proof}
Bucket sort, as presented in the text, permits something like the counting sort to be run on data that doesn't necessarily have to be just integers in a fixed range.

7.1 Counting Sort

These are all variations on relatively obvious ideas; we will not dwell on these algorithms, since all the following take time less than \( \Theta(n \log n) \), none of these can do a sort based on comparing elements.

7.2 Bucket Sort

of the algorithm.

Note that the algorithm as presented in the text uses what is in fact a prefix sum calculation in the middle value \( i \), then we can sort in time \( \Theta(n) \) using the counting sort.

If we have the additional information that the \( n \) elements to be sorted are in the range \( 0 \) to \( k \) for some known

Linear Time Sorting Algorithms
Heapsort is definitely not stable.

For example, merge sort is stable.

with \texttt{Sort(a)} appearing before \texttt{Sort(b)} in the resulting array.

\textbf{Definition 7.1.} A sort is \textbf{stable} if two data items \(a\) and \(b\) with equal keys and \(a > b\) always terminates

\textbf{significant}, and we use a stable sort in every pass.

In general, we can do more than one bit at a time provided we sort on \texttt{digits}, least significant to most.

In \(q\) passes, the elements are sorted.

If we have \(q\)-bit numbers, then we can sort on least significant digit, then next least significant, and so forth.
We can do better, indeed $\frac{3n}{2}$ by initializing the CREW PRAM algorithm and by using some comparisons.

Find the min, taking $n - 2$ further comparisons.

The totally naive, and slow, method would be to find the max first, taking $n - 1$ comparisons, and then to

so this bound is sharp.

Proof. First we comment that it is possible to do both max and min simultaneously in $\frac{3n}{2} - 2$ comparisons.

at least $\frac{3n}{2} - 2$ comparisons in the worst case.

Theorem 8.3. Any algorithm to find the max and min of $n$ elements by comparison of keys must make

8.1 Max and Min

(\begin{enumerate}
\item two elements.
\item For $n$ even, we fudge and use the average of the middle
\item For $n - 1$ or $n - 2$ are smaller and $(n - 1)/2$ are larger.
\item $n - 1$.
\end{enumerate}

Definition 8.2. The median element in a list of $n$ elements for $n$ odd is that element for which

smallest item for any chosen $i$.

Definition 8.1. The selection problem is the problem of finding, in an array of $n$ items, the $i$-th

Selection Algorithms
We aren't going to go the effort of proving the last extra minus 2.

\[ \frac{1}{2} - \frac{1}{u} = \cdots + \frac{1}{u} + \frac{1}{u + 1} \]

This set of \( u \) comparisons twice, so that the min only takes \( u \) comparisons twice. Instead of \( u \) smaller comparisons, we get to use the pairs of these. Then the \( u/g \) smaller pairs. Instead of \( u^2 - \frac{3}{2} \) steps in the naive approach, we get to use the \( u^2 \) smaller elements of the pairs from the first step, and find the \( u/4 \) smaller steps.

Now go back and use the \( u^2/2 \) smaller elements of the pairs from the first step, and find the \( u/4 \) smaller steps.

Compare the winners to get \( u/4 \) new winners. Continuing until we get the max in \( u \). Continue with the even subscript elements (say) with the even subscript elements to get \( u^2/2 \) winners.
Actually, \( u \geq \log n \) so \( n = u + \log n \) and \( u = 2 \) and \( u - 2 \) comparisons.

To find the second largest element in \( u = \log n \) comparisons, we can find the second largest element for the second largest element are those that lost.

We have \( u = \log n \) as the max. The only possible candidates for the second largest element.

For example, we might have:

Arrange the \( n \) elements as leaves in a binary tree, and then have them compete in a competition tournament.

track of this, and use it, we might be able to do with fewer comparisons.

If there were a way to keep

The second largest key is the key that loses a comparison only to the max. If there

We would like to do better.

We can clearly find the second largest key by doing two passes of \( n \) and taking

8.2 The Second Largest Key
Proof. Read carefully pages 189-192 of the text.

\( O\).

\section*{Theorem 8.6.} The \( i \)-th smallest element in a list of \( n \) items can be found in worst case time that is

\[8.3 \quad \text{Selection in Worst-Case Linear Time}\]

Proof. We won't prove this.

Comparison of keys must make at least \( \lceil \frac{n}{2} \rceil + 2 \) comparisons.

\section*{Theorem 8.5.} In the worst case, an algorithm that finds the second largest element of \( n \) keys by comparison of keys needs \( \lceil \frac{n}{2} \rceil + n - 2 \) comparisons.

\section*{Theorem 8.4.} In the worst case, an algorithm that finds the second largest element of \( n \) keys by this proves the following theorem.
number of keys larger than and the number smaller than the median will be balanced.

The "adversary" strategy is as follows. Choose a particular value to be the median. The first time that a key is used in a comparison, a value will be assigned to that key. For as long as it is possible to do so, the value in the comparison will be set to the median. If not possible, then a value will be assigned such that it is less than the chosen value or greater than it, whichever makes the median more balanced.

We have to do at least \( n - 1 \) crucial comparisons in order to find the median.

Definition 8.8. A comparison involving a key \( x \) is a crucial comparison for \( x \) if it is the first comparison where \( x < y \), for some \( y \), for some \( y \neq \text{median} \), or \( x > y \), for some \( y \neq \text{median} \). A comparison is noncrucial if \( x < \text{median} \) and \( y > \text{median} \) (or symmetrically).

We claim that any algorithm that outputs the median must also construct the information that relates all other keys to the median. Otherwise, the value of some key \( y \) whose relation relative to the median isn’t known could be changed so as to make the output incorrect.

Proof: We assume keys are distinct.

At least \( 3(n - 1)/2 \) comparisons in the worst case.

Theorem 8.7. Any algorithm to find, by comparison of keys, the median of \( n \) keys for \( n \) odd must do

8.4 Finding the Median
Hence \( \left( \frac{1}{2} \right) (1 - u) + (1 - u) \) \( \left( \frac{1}{2} \right) \) total.

We can force the algorithm to make \( u - 1 \) noncentral comparisons using the above table.

<table>
<thead>
<tr>
<th>All of these are noncentral.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign a value larger than median to the N Key</td>
</tr>
<tr>
<td>Assign a value larger than median to the N Key</td>
</tr>
<tr>
<td>Assign a value smaller than median to the N Key</td>
</tr>
<tr>
<td>Assign a value smaller than median to the N Key</td>
</tr>
<tr>
<td>One smaller, one larger</td>
</tr>
<tr>
<td>Adversary strategy</td>
</tr>
</tbody>
</table>