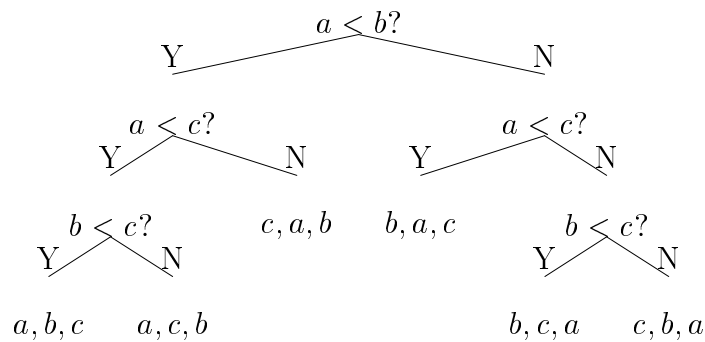


## 6 Lower Bounds on Sorting

### 6.1 Lower Bounds in General

- Consider an algorithm to sort three elements  $\{a, b, c\}$  by comparison of keys as a decision tree, as in the tree below.



- There are  $n!$  total permutations of  $n$  elements, so a decision tree for sorting  $n$  elements must have at least  $n!$  leaves.
- For a binary tree of  $k$  leaves and height  $h$  we have  $k \leq 2^h$ , that is,  $h \geq \lg k$ .
- So a decision tree, with  $n!$  leaves, must have height at least  $\lceil \lg(n!) \rceil$
- And the lower bound for number of comparisons is the lower bound for the height of the decision tree.
- So the lower bound for sorting by comparing keys is  $\lceil \lg(n!) \rceil$

## 6.2 Lower Bounds for the Average Case

- The worst case requires looking for the maximal path down a decision tree.
- The average case requires looking at the average path length.
- Consider the average case, that is,

$$\frac{\text{sum of all path lengths to leaves}}{n!}$$

**Proposition 6.1.** *The sum of all path lengths to leaves is minimized when the tree is completely balanced.*

*Proof.* Disconnecting any subtree from a balanced binary tree, and moving that subtree so as to connect it somewhere else in a binary tree, necessarily increases the sum of the path lengths.  $\square$

**Theorem 6.2.** *The average case of any sort by comparison of keys is at least*

$$\lg(n!)$$

*comparisons.*

*Proof.* The minimal sum of all path lengths to trees is

$$(n!) \cdot (\lg(n!))$$

so the average path length is bounded below by  $\lg(n!)$   $\square$

## 7 Linear Time Sorting Algorithms

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

### 7.1 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 value  $k$ , then we can sort in time  $\Theta(n)$  using the counting sort.

Note that the algorithm as presented in the text uses what is in fact a prefix sum calculation in the middle of the algorithm.

### 7.2 Bucket Sort

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.

### 7.3 Radix Sort

If we have  $b$ -bit numbers, then we can sort on least significant bit, then next least significant, and so forth. In  $b$  passes, the elements are sorted.

In general, we can do more than one bit at a time provided we sort on *digits*, least significant to most significant, and we use a stable sort in every pass.

**Definition 7.1.** *A sort is **stable** if two data items  $a_i$  and  $a_j$  with equal keys and  $i < j$  always terminates with  $\text{Sort}(a_i)$  appearing before  $\text{Sort}(a_j)$  in the resulting array.*

## 8 Selection Algorithms

**Definition 8.1.** *The selection problem is the problem of finding, in an array of  $n$  items, the  $i$ -th smallest item for any chosen  $i$ .*

**Definition 8.2.** *The median element in a list of  $n$  elements for  $n$  odd is that element for which  $(n - 1)/2$  are smaller and  $(n - 1)/2$  are larger. (For  $n$  even, we fudge and use the average of the middle two elements.)*

### 8.1 Max and Min

**Theorem 8.3.** *Any algorithm to find the max and min of  $n$  elements by comparison of keys must make at least  $3n/2 - 2$  comparisons in the worst case.*

*Proof.* First we comment that it's possible to do both max and min simultaneously in  $3n/2 - 2$  comparisons, so this bound is sharp.

The totally naive, and slow, method would be to find the max first, taking  $n - 1$  comparisons, and then to find the min, taking  $n - 2$  further comparisons.

We can do better, indeed  $3n/2$ , by imitating the CREW PRAM algorithm and by using some comparisons twice. Compare the odd subscript elements (say) with the even subscript elements to get  $n/2$  winners. Compare the winners to get  $n/4$  new winners. Continue until we get the max in  $n/2 + n/4 + \dots = n - 1$  steps.

Now go back and use the  $n/2$  smaller elements of the pairs from the first step, and find the  $n/4$  smaller pairs of these. Then the  $n/8$  smaller pairs. Instead of  $2n - 3$  steps in the naive approach, we get to use the first set of  $n/2$

comparisons twice, so that the min only takes us  $n/4 + n/8 + \dots = n/2 - 1$  comparisons extra beyond those needed for the max.

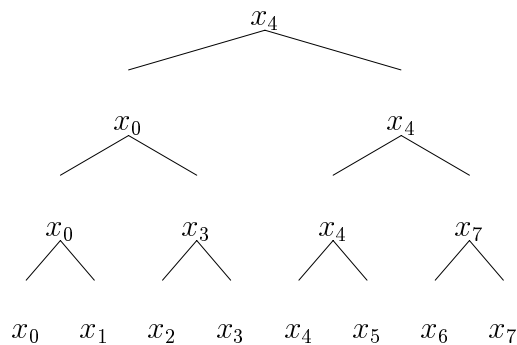
We aren't going to go the effort of proving the last extra minus 2.  $\square$

## 8.2 The Second Largest Key

We can clearly find the second largest key by doing two passes of max and taking  $(n - 1) + (n - 2)$  comparisons. We would like to do better.

The second largest key is the key that loses a comparison only to the max. If there were a way to keep track of this, and use it, we might be able to do with fewer comparisons.

Arrange the  $n$  elements as leaves in a binary tree, and then have them compete in a comparison tournament. For example, we might have:



We have  $x_4$  as the max. The only possible candidates for the **second** largest element are those that lost to  $x_4$ , namely,  $x_0, x_7, x_5$ . In general, then, we can find the second largest element in  $n + \lg n - 2$  comparisons. Actually,  $n + \lceil \lg n \rceil - 2$

This proves the following theorem.

**Theorem 8.4.** *In the worst case, an algorithm that finds the second largest element of  $n$  keys by comparison of keys needs  $n + \lceil \lg n \rceil - 2$  comparisons.*

**Theorem 8.5.** *In the worst case, an algorithm that finds the second largest element of  $n$  keys by comparison of keys must make at least  $n + \lceil \lg n \rceil - 2$  comparisons.*

*Proof.* We won't prove this. □

### 8.3 Selection in Worst-Case Linear Time

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

*Proof.* Read carefully pages 189-192 of the text. □

### 8.4 Finding the Median

**Theorem 8.7.** *Any algorithm to find, by comparison of keys, the median of  $n$  keys for  $n$  odd must do at least  $3(n - 1)/2$  comparisons in the worst case.*

*Proof.* We assume keys are distinct.

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.

**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  $y \geq \text{median}$ , or*

$x < y$ , for some  $y \leq \text{median}$ . A comparison is **noncrucial** if  $x > \text{median}$  and  $y < \text{median}$  (or symmetrically).

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

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 number of keys larger than and the number smaller than the median will be balanced.

Case	Adversary strategy
$N, N$	One smaller, one larger
$L, N$	Assign a value smaller than median to the $N$ key
$N, L$	Assign a value smaller than median to the $N$ key
$S, N$	Assign a value larger than median to the $N$ key
$N, S$	Assign a value larger than median to the $N$ key

All of these are noncrucial.

We can force the algorithm to make  $(n - 1)/2$  noncrucial comparisons using the above table.

Hence  $(n - 1) + (n - 1)/2$  total. □