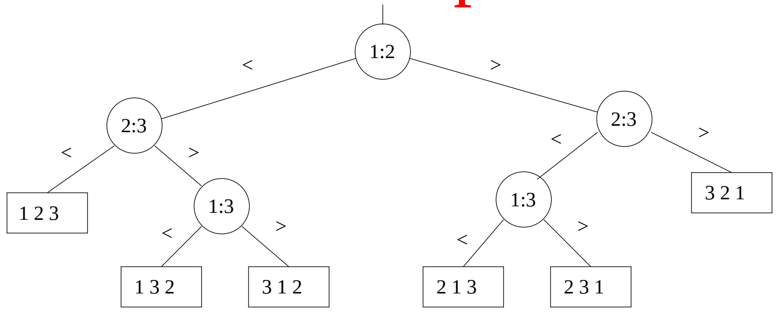
LOWER BOUND FOR COMPARISON-BASED SORTING

In this handout we consider the question: How efficiently can we sort? Such questions (determining how to best carry out a task) are among the most difficult and intellectually challenging problems of theoretical computer science. It is generally much more difficult to show that an algorithm is ‘‘best possible’’ for a certain task, than to come up with a ‘‘good’’ algorithm for that task. In the latter case, one need only argue about *one* algorithm — the one that is supposedly ‘‘good’’; in the former case, one must argue about *all possible* algorithms — even those which one knows nothing about!

To answer the question ‘‘what’s the best way to do a job’’ we must start by fixing the set of *tools* that may be used to do the job. For the problem at hand, sorting, we will take *pairwise comparisons of keys* as the available tools. Of course, this leaves out of consideration Bin sorting (which is not based on key comparisons). But, in a sense, we can justify this on the ground that Bin sorting is not a ‘‘general’’ sorting method: it is applicable only if keys have a particular form. At any rate, it's important to keep in mind that *the result we are about to present only applies to sorting algorithms based on comparisons*.

Suppose we want to sort n keys $K_1, K_2, \cdots, K_n$. Let's assume that all keys are distinct, so that for any K_i, K_j where $i \neq j$, either $K_i < K_j$ or $K_i > K_j$. The main theoretical device we'll use to analyze our problem is a *decision* (or *comparison*) *tree*. This is a useful way of representing *any* comparison-based algorithm that sorts n keys (for any given n). Before introducing the formal definition, let's develop some intuition about decision trees by studying an example.

Example 1: Below is a decision tree that corresponds to one possible algorithm for sorting 3 keys, K_1, K_2, K_3.



The internal nodes of this tree correspond to comparisons the algorithm makes. The labels in those nodes, specify which keys are to be compared. For example the label "1:2" of the root indicates that K_1 is to be compared with K_2. Depending on the outcome of the comparison in a node, the left or the right branch out of that node is taken. The label of each leaf is the permutation specifying how to rearrange the keys to get them sorted. For example, consider the node with label "2,3,1". This means that $K_2<K_3<K_1$. Note that the fact this is the order of the three keys is implied by the outcomes of the comparisons made in the path from the root to that leaf: To get to that leaf we must go from the root to the right (signifying that $K_1>K_2$); then from node "2:3" to the left (signifying that $K_2<K_3$) and from node "1:3" to the right (signifying that $K_1>K_3$). These three comparisons imply that $K_2<K_3<K_1$. This is the case for the path from the root to any leaf.
Thus the decision tree specifies the sequence of comparisons that the algorithm will perform to

sort 3 keys. An execution of the algorithm for a specific input (of 3 keys) corresponds to a path from root to a leaf. For example, suppose that the input keys are $K_1 = c$, $K_2 = a$, $K_3 = b$ (where $a < b < c$). The execution starts at the root of the tree. The node there specifies that K_1 and K_2 are to be compared. Since $K_1 > K_2$, we take the right branch from the root and arrive at the "2:3" node. This indicates we must compare K_2 to K_3 and since $K_2 < K_3$, we take the left branch and arrive at the "1:3" node. By comparing K_1 and K_3 we discover that $K_1 > K_3$ and therefore we take the right branch which leads us to the leaf labeled "2,3,1". This indicates that the keys in sorted order are $K_2 < K_3 < K_1$, as indeed is the case.
A more programming language-like description of the algorithm corresponding to the above decision tree is:

```
if $K_1 < K_2$ then
    if $K_2 < K_3$ then sorted order of keys is $K_1, K_2, K_3$
    else {$K_2 > K_3$}
        if $K_1 < K_3$ then sorted order of keys is $K_1, K_3, K_2$
        else {$K_1 > K_3$} sorted order of keys is $K_3, K_1, K_2$
else {$K_1 > K_2$}
    if $K_2 < K_3$ then
        if $K_1 < K_3$ then sorted order of keys is $K_2, K_1, K_3$
        else {$K_1 > K_3$} sorted order of keys is $K_2, K_3, K_1$
    else {$K_2 > K_3$} sorted order of keys is $K_3, K_2, K_1$    □
```

Using the intuition we have gained by studying this example, let us now give the formal definition of a decision tree.

Definition 1: A *decision tree of order n* is a binary tree so that

(1) It has $n!$ leaves, each labeled by a different permutation of $1, 2, \cdots, n$.

(2) Internal nodes are labeled by pairs of indices of the form "$i : j$", $1 \le i \ne j \le n$.

(3) In the path from the root to a leaf $\pi(1)\,\pi(2) \cdots \pi(n)$ (π is a permutation of $1, 2, \cdots, n$) there is either a node "$\pi(i) : \pi(i+1)$" — in which case the path goes from that node to its *left* child — or "$\pi(i+1) : \pi(i)$" — in which case the path goes from that node to its *right* child — for each $1 \le i < n$. (The path may contain other nodes too but must contain *at least* these.)

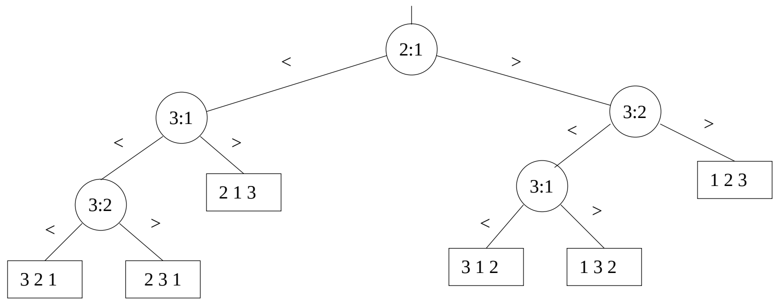
The idea is that leaves correspond to the possible outcomes of sorting n distinct keys. Internal node "$i : j$" corresponds to the comparison of K_i and K_j. If the outcome is $K_i < K_j$ then the left subtree of the node "$i : j$" contains the subsequent comparisons made until the order of the keys is determined. Symmetrically, if the outcome is $K_i > K_j$, the right subtree of "$i : j$" contains the subsequent comparisons. Part (3) of the definition essentially requires that every relationship determined by the algorithm must have been established by actual comparisons: the algorithm cannot "guess".

Any comparison-based algorithm for sorting n keys corresponds to a decision tree of order n. The execution of the algorithm on input sequence of keys $K_1, K_2, \cdots, K_n$ follows the path from the root to the leaf labeled by the permutation π such that $K_{\pi(1)} < K_{\pi(2)} < \cdots < K_{\pi(n)}$. Note that for a given sorting algorithm we need a different decision tree for each n to represent all possible executions of that algorithm when the input consists of n keys.

Example 2: Consider the algorithm for insertion sort:

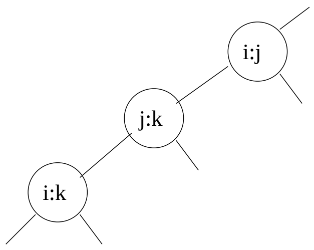
```
$INSERTION\text{-}SORT(A)$
begin
  for $i := 2$ to $n$ do
    for $j :=$ i down to 2 do
      if $A[j] < A[j-1]$ then $A[j] \longleftrightarrow A[j-1]$
      else break
end
```

The corresponding decision tree for sorting 3 keys is shown below:



It is very important to realize that the node labeled "i: j" in the decision tree specifies the comparison between keys K_i and K_j (as given in the initial array), and *not* the keys that presently happen to be in the i^{th} and j^{th} positions of the array. Note that the decision tree does not mention swaps of array elements: it simply determines the permutation according to which the elements of the array must be rearranged for the keys to be in sorted order. □

A node is called *redundant* in a decision tree, if the outcome of the comparison it represents can be inferred from previous comparisons in the path from the root to that node. A trivial example of a redundant node is one with the same label as one of its proper ancestors (this corresponds to repeating a comparison already made). A less trivial example is illustrated below



The “$i:k$” node is redundant since the outcome of the comparison it represents (K_i vs. K_k) can be inferred from the fact that $K_i < K_j$ (since the path goes from “$i: j$” to the left) and the fact that $K_j < K_k$ (since the path goes from “$j:k$” to the left). Thus the path from “$i:k$” to the right can never be followed.

Definition 2: A decision tree is *minimal* if it contains no redundant nodes.

For instance the trees in Examples 1 and 2 are both minimal. As we just remarked, only one path out of a redundant node could be followed, namely the one corresponding to the single possible outcome of the comparison represented by that node. Therefore we immediately get that:

Fact 1: A minimal decision tree must be a full binary tree. □

WORST CASE LOWER BOUND ON THE NUMBER OF COMPARISONS

Let C_n denote the minimum number of comparisons required to sort n keys. For any decision tree, the worst case number of comparisons required by the algorithm represented by that tree is precisely the height of the tree.

Lemma: Any binary tree of height h has at most 2^h leaves.

Proof: Trivial induction on h. □

By definition, any decision tree of order n is a binary tree with $n!$ leaves. Thus, by the lemma, it must have height at least $\lceil \log n! \rceil$. Thus, $C_n \geq \lceil \log n! \rceil$. Stirling's approximation says that $n! \approx \sqrt{2\pi n}\,(n/e)^n$. Thus,

$$C_n \geq \log \sqrt{2\pi n}\,(n/e)^n \geq \log(n/e)^n = n \log n - n \log e$$

from which it follows that C_n is $\Omega(n \log n)$.

AVERAGE CASE LOWER BOUND ON THE NUMBER OF COMPARISONS

Consider a decision tree of order n representing some sorting algorithm. If we assume that all initial arrangements of the keys to be sorted are equally likely, the average case number of comparisons performed by that algorithm is equal to the external path length of the decision tree divided by the number of leaves in the tree.† The following fact can be easily proved:

Fact 2: The tree that minimizes the external path length has all leaves in at most two depths d and $d-1$, for some d.□

By Facts 1 and 2 we may assume that in the decision tree that minimizes the average number of comparisons, all leaves have depth d or $d-1$ for some d. Let $\bar{C}_n$ be the minimum average case number of comparisons needed to sort n keys. Also, let N_d and N_{d-1} be the number of leaves in the corresponding decision tree at depth d and $d-1$ respectively. Recall that the number of leaves in T is $n!$. Thus we have

$$\bar{C}_n = [(d-1)\,N_{d-1} + d\,N_d]/n! \qquad (1)$$

But we have:

$$N_d + N_{d-1} = n! \qquad (2)$$

and

$$N_d + 2\,N_{d-1} = 2^d \qquad (3)$$

Equation (2) says that the total number of leaves in T is $n!$. For equation (3) note that if we gave 2 children to each leaf of depth $d-1$, we would get the maximum possible number of nodes at depth d, which is 2^d.

Solving (2) and (3) for N_d and N_{d-1}, we get:

$$N_d = 2\,n! - 2^d \qquad (4)$$

and

$$N_{d-1} = 2^d - n! \qquad (5)$$

Substituting (4) and (5) in (1) we obtain:

† Recall that the external path length of the tree is the sum of the depths of all leaves.

$$\bar{C}_n = [(d-1)\,N_{d-1} + d\,N_d]/n!$$

$$= (d\,n! + n! - 2^d)/n!$$

But $d = \lceil \log n! \rceil = \log n! + \varepsilon$, for some $0 \le \varepsilon < 1$. Therefore,

$$\bar{C}_n = (n! \log n! + n!\,\varepsilon + n! - n!\,2^{\varepsilon})/n!$$

$$= \log n! + (1 + \varepsilon - 2^{\varepsilon}), \quad \text{where} \;\; 0 \le \varepsilon < 1.$$

The quantity $1 + \varepsilon - 2^{\varepsilon}$ is non-negative for $0 \le \varepsilon < 1$. Thus,

$$\bar{C}_n \ge \log n! \approx \log \sqrt{2\pi n}(n/e)^n \ge \log(n/e)^n = n \log n - n \log e.$$

Hence, $\bar{C}_n$ is $\Omega(n \log n)$.

Therefore, any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons both in the worst and in the average case.